Fakultät für
Mathematik, Informatik
und Naturwissenschaften

Discrete Adjoint Optimization with OpenFOAM

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Ing. Markus Towara
aus Köln

Betreuer: Univ.-Prof. Dr. rer. nat. Uwe Naumann
Univ.-Prof. Dr.-Ing. Wolfgang Schröder

Tag der mündlichen Prüfung: 07. Dezember 2018

Contents

Abstract

English version

Computer simulations and computer aided design in the past decades have evolved into a valuable instrument, penetrating just about every branch of engineering in industry and academia. More specifically, computational fluid dynamics (CFD) simulations allow to inspect flow phenomena in a variety of applications. As simulation methods evolve, mature, and are adopted by a rising number of users, the demand for methods which not only predict the result of a specific configuration, but can give indications on how to improve the design, increases.

This thesis is concerned with the efficient calculation of sensitivity information of CFD algorithms, and their application to numerical optimization. The sensitivities are obtained by applying Algorithmic Differentiation (AD).

A specific emphasis of this thesis is placed on the efficient application of adjoint methods, including parallelism, for commonly used CFD finite volume methods (FVM) and their implementation in the open source framework *OpenFOAM*.

Deutsche Version

In den vergangenen Jahrzehnten haben sich Computersimulation und Computer gestützte Design Methoden zu einem wertvollen Instrument entwickelt, welches nahezu jeden Bereich der technisch- und naturwissenschaftlichen Forschung und Wirtschaft beeinflusst. CFD Simulationen erlauben es, Strömungsphänomene in einer Vielzahl von Anwendungen zu untersuchen. Je mehr numerische Simulationsmethoden sich fortentwickeln und Anwendung finden, umso größer wird der Bedarf an Methoden, die nicht nur das Ergebnis von spezifischen Konfigurationen vorhersagen können, sondern auch Hinweise zur Optimierung des Designs geben können.

Diese Arbeit beschäftigt sich mit der effizienten Berechnung von Ableitungsinformationen auf CFD Algorithmen und deren Anwendung zur numerischen Optimierung. Die Ableitungen werden mittels Algorithmischen Differenzierens (AD) generiert.

Ein besonderer Augenmerk dieser Arbeit liegt auf der effizienten Anwendung adjungierter Methoden, unter Berücksichtigung von Parallelismus, im Kontext von populären CFD Finite-Volumen Algorithmen, und deren Implementierung im Open-Source Strömungslöser *OpenFOAM*.

Acknowledgements

First and foremost I want to thank my thesis supervisor Prof. Dr. Uwe Naumann. Your supervision was a unique combination of personal freedom and guidance when needed. My thanks also go to my co-supervisor Prof. Dr. Wolfgang Schröder for your in depth comments and insightful input, especially on the CFD parts of this thesis.

I want to thank all of my current and former co-workers, HiWis and students at STCE. You made the long hours working on this thesis feel like less work and always provided inspiring ideas. Particular thanks go to Johannes Lotz for proof-reading and copious amounts of C++ hacking advice. Gratitude is also awarded to Andreas Wierz for proof-reading and excessive nitpicking about layout and typesetting.

Further, I want to express my gratitude to Dr. Carsten Othmer of Volkswagen Research for introducing me to the continuous adjoint method in OpenFOAM during my time at Volkswagen. Without this, this thesis probably would not have been written.

Also I would like to thank the broader AD community for being a welcoming group, filled with bright and friendly people.

Last, but in no way least, I want to thank my friends and family, you were always there for me offering encouragement, support, and distraction during easy and not so easy times.

I swear to god I had something for this.

— Sterling Archer

List of Abbreviations

AD	Algorithmic Differentiation
AMPI	Adjoint Message Passing Interface
AST	Abstract Syntax Tree
BFS	Breadth-First Search
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CFD	Computational Fluid Dynamics
CHT	Conjugate Heat Transfer
CRS	Compact Row Storage
DAG	Directed Acyclic Graph
DNS	Direct Numerical Simulation
DOF	Degrees of Freedom
FD	Finite Differences
FEM	Finite Element Method
FSI	Fluid Structure Interaction
FVM	Finite Volume Method
GPL	General Public License
HPC	High-Performance Computing
IO	Input/Output
LES	Large Eddy Simulation
LOC	Lines of Code
NACA	National Advisory Committee for Aeronautics
NURBS	Non-Uniform Rational Basis Spline
NVME	Non-Volatile Memory Express
ODE	Ordinary Differentiation Equation
OpenFOAM	Open Field Operation and Manipulation
OS	Operating System
PDE	Partial Differential Equation
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RANS	Reynolds Averaged Navier-Stokes Equations
RHS	Right-Hand Side
RWTH	Rheinisch-Westfälische Technische Hochschule
SAC	Single Assignment Code
SDLS	Symbolically Differentiated Linear Solvers
SSD	Solid State Drive

List of Symbols

$\mathbf{u}$	$[\mathbf{U}]$	Velocity [field]
$\bar{\mathbf{u}}$	$[\bar{\mathbf{U}}]$	Adjoint velocity [field]
p	$[\mathbf{p}]$	(Kinematic) pressure [field]
$\bar{p}$	$[\bar{\mathbf{p}}]$	Adjoint pressure [field]
k	$[\mathbf{k}]$	Turbulence kinetic energy [field]
ϵ	$[\boldsymbol{\epsilon}]$	Rate of turbulence energy dissipation [field]
ω	$[\boldsymbol{\omega}]$	Specific rate of dissipation [field]
ϕ	$[\boldsymbol{\phi}]$	Face flux [field]
ψ	$[\boldsymbol{\psi}]$	Generic scalar transport variable [field]
α	$[\boldsymbol{\alpha}]$	Topological flow resistance penalty [field]
β	$[\boldsymbol{\beta}]$	Surface sensitivity [field]
$\mathbf{q}$	$[\mathbf{Q}]$	Point [field]
$\mathbf{r}$	$[\mathbf{R}]$	Residual vector [field]
ν	$[\eta]$	Kinematic [dynamic] viscosity
$\dot{\bullet}$	$\bullet^{(1)}$	Tangent
$\bar{\bullet}$	$\bullet_{(1)}$	Adjoint
$\boldsymbol{\gamma}$		Optimization parameters
$\mathbf{x}$		Iteration state
m		Mass
ρ		Density
τ		Shear-Stress
y^+		Dimensionless wall distance
t		Time
V		Volume
M		Material volume
S		Surface
Ω		Domain
Γ		Boundary
$\mathbf{s}$		Surface area vector
$\mathbf{n}$		Surface normal vector
$\mathbf{e}_i$		Cartesian basis vector
Re		Reynolds number
φ		Elemental function
$\bullet$		Placeholder variable
P		Processor
$\mathcal{P}$		Pre-processing step
$\mathcal{F}$		Processing step
$\mathcal{J}$		Post-processing step
$\boldsymbol{l}, \boldsymbol{d}, \boldsymbol{u}$		LDU sparse matrix coefficients
$\boldsymbol{L}, \boldsymbol{U}$		LDU sparse matrix addressing

1 Introduction

1.1 Outline

The focus of this thesis lies on the efficient generation of sensitivity information for computational fluid dynamics (CFD) applications, using the discrete adjoint approach. Methods and algorithms are presented in a general CFD setting, and are implemented in a discrete adjoint OpenFOAM framework. The application of these sensitivities to specific optimization problems is demonstrated in several cases. The presented optimization cases are to demonstrate the flexibility of the sensitivities obtained by the discrete adjoint framework, and do not necessarily represent the state of the art in numerical optimization. A thorough discussion of state of the art constrained optimization techniques would be outside of the scope of this thesis and remain as an avenue for future research.

This thesis is structured as follows. Chapter 1 introduces notations and conventions used in this thesis. Related work and the most important contributions of this thesis are briefly summarized. Furthermore, a high level overview over numerical simulation and optimization is given. The most important theoretical foundations, as well as the motivation for this thesis, are laid out in Chapter 2. Basics of the CFD method, Algorithmic Differentiation (AD) and optimization are covered.

Building on the foundations, the application of AD to CFD algorithms is discussed in detail in Chapter 3. Starting from a black-box approach as a proof of concept, a variety of improvements are presented, which increase the efficiency of AD in the context of iterative CFD solvers. Parallel adjoint communication is incorporated into the CFD solvers to retain the primal parallelism, requiring adaptations of the linear solvers.

Using these results, Chapter 4 discusses different strategies implemented to more efficiently generate adjoints for steady state simulations. Furthermore, the generated adjoint sensitivities are verified against tangents and continuous adjoints. Alternative optimization methods, such as parametric optimization are discussed and implemented.

Extended case studies for both topology optimization and shape sensitivities are presented in Chapter 5. A scaling study on current HPC hardware, the RWTH compute cluster, was performed. The results showcase the scalability of both the primal and adjoint implementations.

This thesis closes with an overview over related work, carried out or supervised by the author, as well as a summary and outlook. A brief developer documentation for the discrete adjoint OpenFOAM implementation can be found in the appendices.

1.2 Notation

Here we introduce the most important notations and conventions, used throughout this thesis. Non-standard notations will be reintroduced once they first become relevant.

Vectors: Bold letters, individual entries are numbered starting from zero, e.g. $\mathbf{v} = [v_0, v_1, v_2]$.

For spatial information, alternatively alphabetic indices $\mathbf{v} = [v_x, v_y, v_z]$ might be used.

Matrices: Upper case letters, individual entries are denoted with lower case letter and numbered starting with zero. E.g.:

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}.$$

Scalar product: The scalar (inner) product between two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is denoted by $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y} = \sum_{i=0}^{n-1} x_i y_i$.

Outer product: The outer product between two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is denoted by $\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^T \in \mathbb{R}^{n \times n}$.

Functions: Lower case letters or in calligraphy, e.g. $f(\mathbf{x})$ or $\mathcal{J}(y)$.

Placeholder: $\bullet$ stands for a placeholder variable, to which super or subscripts and modifiers can be attached, e.g. $\bar{\bullet} = \bullet_{(1)}$.

Unit systems: If not otherwise specified SI units are used.

Unit system for bytes: For the designation of bytes, we use the binary multiple system, differing from the SI definition, also known as TiB, GiB, MiB, KiB: $1\,\text{TB} = 1024^1\,\text{GB} = 1024^2\,\text{MB} = 1024^3\,\text{KB} = 1024^4\,\text{bytes}$.

Spatial and temporal relations: In the context of FVM discretizations, spatial relations are indicated by sub indices, e.g. $\mathbf{x}_i$. Temporal (or pseudo temporal iteration) relations are indicated by upper indices, e.g. $\mathbf{x}_i^{j+1} = f(\mathbf{x}_{i+1}^j, \mathbf{x}_i^j)$.

Powers: To avoid confusion with the temporal indices, if not immediately obvious by context, we indicate powers by enclosing the target in brackets first, e.g. $(y)^2$.

Gradient operators: In the context of FVM discretization, the gradient operator is used as a spatial operator, ignoring temporal dimensions, if any:

Scalar Gradient: $\boldsymbol{\nabla} p = \begin{bmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \\ \frac{\partial p}{\partial z} \end{bmatrix}$

Vector Gradient: $\boldsymbol{\nabla} \mathbf{u} = \boldsymbol{\nabla} \otimes \mathbf{u} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} & \frac{\partial u_z}{\partial x} \\ \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} \\ \frac{\partial u_x}{\partial z} & \frac{\partial u_y}{\partial z} & \frac{\partial u_z}{\partial z} \end{bmatrix}$

Outer product: $\mathbf{u} \otimes \boldsymbol{\nabla} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} & \frac{\partial u_x}{\partial z} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} & \frac{\partial u_y}{\partial z} \\ \frac{\partial u_z}{\partial x} & \frac{\partial u_z}{\partial y} & \frac{\partial u_z}{\partial z} \end{bmatrix}$

Divergence: $\boldsymbol{\nabla} \cdot \mathbf{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$

$$\textbf{Laplace Operator: } \Delta \mathbf{u} = \boldsymbol{\nabla}^2 \mathbf{u} = \boldsymbol{\nabla}\left(\boldsymbol{\nabla} \otimes \mathbf{u}\right) = \begin{bmatrix} \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2} \\ \frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_y}{\partial z^2} \\ \frac{\partial^2 u_z}{\partial x^2} + \frac{\partial^2 u_z}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} \end{bmatrix}$$

1.3 Related Work & Contribution

In the field of CFD and numerical optimization, a wide range of prior knowledge exists. While the foundations of fluid mechanics research range back several centuries [And16], the application of computational methods to discretize and obtain solutions to the governing equations became popular in the 20th century, with the advent of general purpose computing hardware [Sha04]. Compared to the history of CFD, the widespread application of adjoint optimization techniques to CFD problems is more recent, however first applications still date back to the 1970s (see e.g. [Jam03; GP00] for a short overview of the history of adjoint methods in the context of CFD). Most researchers apply either a continuous adjoint formulation, or a discrete adjoint formulation on the residuals obtained from solutions of the FVM systems [Gil+03]. Algorithmic Differentiation (AD) is concerned with the generation of derivatives of algorithms, given as computer programs (for a historical discussion of AD see [GW08]). While the rules of AD can be applied by hand, for complex codes a tool driven approach is highly desirable.

The application of AD to complex CFD tools is often limited to specific numerical kernels, tailored to a predefined goal. A full differentiation of code frameworks is seldom achieved, be it for lack of applicable tools or too extreme memory requirements imposed by the data flow reversal of the adjoint mode. A recent applications of an AD tool driven approach to a complex CFD code includes SU2 [Eco18; ASG16]. Workflows based on source code transformation were used in [Zho+18] and [MHM18].

This thesis pursues the approach to initially cover as much as possible of the used CFD framework by AD. The advantage of such an approach is twofold: First, an initial differentiated version of the problem can be obtained very rapidly, without much analytical insight into the underlying problem. Starting from there, possible optimizations can then be identified, applied, and evaluated. Second, a full AD implementation gives the flexibility to pursue a wide range of optimization tasks, without needing to adapt the underlying CFD framework to each of these applications. Starting from a naive black-box implementation of AD, treating the CFD algorithms as a general computer program, different tactics are employed to improve performance and lower the memory footprint. This is achieved by exploiting prior knowledge, blurring the lines between fully discrete and continuous approaches. To the author's knowledge, this work is the broadest application of AD to the general purpose CFD tool OpenFOAM. Other adjoint solvers for OpenFOAM exist, in the form of continuous adjoint solvers and implementations of the discrete residual adjoint methods, possibly involving finite differences (FD) [OVW07; He+18].

OpenFOAM is particularly suited for this approach, due to its split architecture into a general CFD framework, (which was fully covered by AD), and individual solvers and utilities based on the framework (which were covered by AD as needed). The changes required for the introduction of AD into a complex CFD code base are discussed in detail in Section 3.2. Adjoint communication patterns ([Sch14]) were implemented, such that the parallel convergence of the primal is retained for adjoint calculations (Section 3.5), which is demonstrated on the RWTH compute cluster (Section 5.1). The symbolic differentiation of linear solvers ([Gil08], Section 3.4) required the careful inspection of the FVM discretization matrices involved, in order to correctly implement the symbolic adjoints of the parallel matrix vector products (Section 3.5.5).

The application of AD to iterative solvers poses challenges in terms of memory consumption. To overcome a memory bottleneck encountered, an existing adjoint vector compression technique was implemented ([NL18], Section 3.8.2). This induced a bigger than expected run time penalty. To overcome this, a novel optimization to the adjoint vector compression technique was developed, implemented, and studied (Section 3.8.3). This allowed to regain most of the performance of the unaltered adjoint vector implementation.

The developed discrete adjoint solvers are verified against adjoints obtained by the continuous adjoint method (Section 4.3). The implications of a frozen turbulence assumption versus a full differentiation of the turbulence models are studied. The availability of a fully differentiated Spalart-Allmaras turbulence model facilitates the calculation of shape sensitivities of an airfoil (Section 5.2). In addition to topological and shape sensitivities, a parametric optimization setting is presented (Section 4.6), demonstrating the differentiation of a solver directly coupled to a mesh generator. A similar parametric approach was chosen in [Aur+16], however differentiating an external CAD environment and mesher.

A novel connection between the FVM mesh formulation and the bipartite (partial) graph coloring formulation was developed, allowing to effectively obtain compressed Jacobians. The method and resulting colorings are presented in Section 4.5.4. The obtained colors are used to efficiently calculate the Jacobians of the FVM residuals using tangent or adjoint mode. A similar approach is presented in [He+18], also implemented in OpenFOAM, however in this work the authors used different coloring algorithms and FD to determine the Jacobians.

While many concepts are presented, such that they can be conveniently implemented in OpenFOAM, the methods developed and used in this thesis are applicable to a wide variety of CFD algorithms and optimization tasks.

Previous publications of the author, related to the topics of this thesis, include [TN13; TSN15], and [TN18]. Parts of Sections 3.2 and 3.3 (black-box differentiation of OpenFOAM and checkpointing) were discussed in [TN13]. Parts of Section 3.5 (parallel black-box adjoints using adjoint MPI) were discussed in [TSN15]. Furthermore, parts of Sections 3.4, 3.5 (symbolically differentiated linear solvers, embedded into SIMPLE algorithm, involving parallelism), and 5.1 (HPC study of 3D Pitz-Daily case) were previously presented in [TN18].

Prior Publications

[TN13] M. Towara and U. Naumann. “A Discrete Adjoint Model for OpenFOAM”. In: *Procedia Computer Science* 18.0 (2013). 2013 International Conference on Computational Science, pp. 429–438.

[TN18] M. Towara and U. Naumann. “SIMPLE Adjoint Message Passing”. In: *Optimization Methods and Software* (2018), pp. 1–18.

[TSN15] M. Towara, M. Schanen, and U. Naumann. “MPI-Parallel Discrete Adjoint OpenFOAM”. In: *Procedia Computer Science* 51 (2015). 2015 International Conference on Computational Science, pp. 19–28.

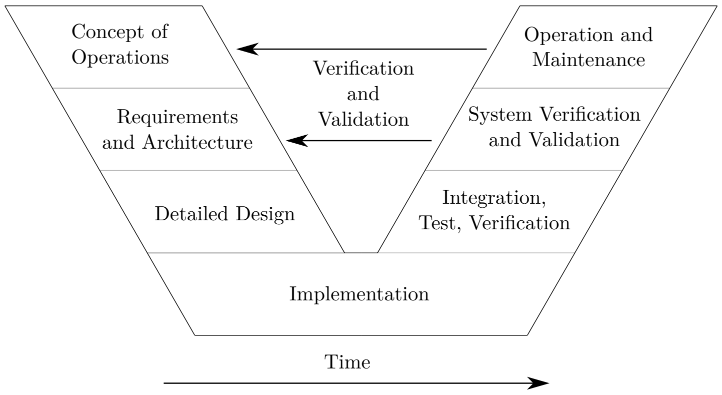

Concept of Operations
Verification and Validation
Operation and Maintenance
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, Verification
Implementation
Time

Figure 1.1: The *V-model* of product development, also known as *Vee-diagram*. Modeled after [Osb+05]. See also VDI 2206 [GM03].

1.4 Numerical Simulation

1.4.1 Introduction & Motivation

Numerical simulations of physical problems have become an integral part of many modern design processes [Ott+03]. Simulation is able to augment classical design verification processes, relying on physical experiments, with numerical experiments. The repeatability of numerical experiments allows the efficient exploration of the design space for many different configurations and boundary conditions. Virtual product development allows to move the test and evaluation of preliminary designs towards earlier in the development cycle, avoiding costs like expensive tooling for pre-production models. Model candidates can be evaluated in a shorter time period, allowing to iterate through the stages of development at a faster pace. With correctly chosen models, numerical simulations allow to create observations for conditions which might not be readily reproducible or observable in a lab environment, such as:

- reduced or zero gravity [CS93],
- extreme temperatures or pressures [Anc+97],
- extreme length scales, e.g. quantum level [HTK12] or astronomical scales [Aba+03],
- short (e.g. microseconds) [Hes+08] or long (e.g. centuries) [Cro00] time scales,
- hazardous or restricted processes, e.g. nuclear explosions [KGG18],
- or probabilistic quantum level effects [LW90].

Industry fields which heavily rely on numerical simulation to reduce product development times include the automotive [Tho98], aerospace [SV16], and defense [NW11] industries. Statistical simulation approaches are prevalent in the financial sector [GG06].

Product development, be it physical (mechanical) or virtual (software), is usually an iterative process [TN86]. Requirements are identified, incorporated into a design, and implemented. The implementation is tested and integrated into the systems context (e.g. a part into a car or a module into a software framework). During testing and integration, more likely than not, issues are identified, that feed back into the requirements, starting off another iteration of the development process. This iterative design process is often illustrated with the *V-model* of product development. Many different variants of this model exist, one of them is shown in Figure 1.1.

The incorporation of numerical simulations into product development is often called computer-aided engineering (CAE), in distinction, and as a supplement to, the more classical computer-aided design (CAD). Numerical simulation can help to reduce the time spent in each iteration of the product development cycle.

In civil engineering, the finite element method (FEM) [Hug12] is the dominant method of predicting the stresses in materials and the resulting deformations. In fluid mechanics, the finite volume method (FVM) [Pat80] is widely popular, but alternative methods, including FEM and Galerkin [CKS00] methods, are available.

Multiphysical simulations aim to bridge the divide between different fields of science, e.g. by coupling fluid simulation with structural analysis (fluid structure interaction, FSI) [Zim06], or by coupling fluid flow and heat transfer (conjugate heat transfer, CHT).

1.4.2 Computational Fluid Dynamics

Computational fluid dynamics (CFD) is the application of computational methods to the field of fluid mechanics. Early applications date back to the beginnings of the 20th century, where solutions were obtained by hand or mechanical computers [Hun98], however wide spread application has only been realized with the introduction of general purpose computing devices [Wen08]. Today CFD methods are an important application for current high performance computing (HPC) resources. With the ever increasing performance of current HPC infrastructure, more complex simulations (multi-scale, multi-physics) and solution methods, such as direct numerical simulation (DNS) [Ors70] have become feasible. This thesis focuses on applications of the FVM method (introduced in the next chapter) in the context of CFD applications. However, other discretization methods are available, such as finite element methods (FEM) or probabilistic methods. The most common application of CFD is the solution of the Navier-Stokes partial differential equations, which allow to predict the flow of viscous fluids. For fluids, that exhibit a negligible amount of viscosity, the less complex Euler equations can be solved instead. This is particularly useful for compressible flows in transonic flow.

In the context of FVM CFD simulations, a wide variety of commercial, academic, and open-source software codes exist. OpenFOAM (Open Source Field Operation and Manipulation) is an open-source software suite, which has a strong user base in both industry and academia, due to its versatility, good parallel scalability, and lack of licensing costs. Development of OpenFOAM started in the early 90s as a research project at Imperial College London [Wel+98; Jas96]. It was first released to the public as a commercial code, called FOAM. Later the code was released as open-source under the GNU GPL-v3 [GPL] in 2004, forming OpenFOAM. Development remains very active, however development is currently fragmented in three different forks. The rights to the OpenFOAM trademark currently belong to ESI Group, which develops and distributes the OpenFOAM-plus fork.

In this thesis the applicability of the discrete adjoint mode of AD to the CFD design optimization process is explored. As a demonstrator for these techniques a discrete adjoint version of OpenFOAM-plus, developed by the author, is used.

For numerical optimization, gradient based methods are popular, as they offer better convergence compared to gradient free methods, requiring less evaluations of the underlying simulation models. The efficient computation of derivatives is its own research field, with different strategies for obtaining the derivatives. This will be discussed in detail in later sections.

1.4.3 Topology Optimization

As an example for numerical optimization in the context of CFD simulation, we will use the topology optimization technique throughout this thesis.

The concept of topology optimization was first introduced in FEM analysis of mechanical stresses [Ben89]. Topology optimization aims to find an optimal structure, relative to some cost function, within a given design space and boundary conditions. The difference between topology optimization and a classical shape optimization is that the topology optimization does not start from and adapt an initial design, which for a non-global optimization might introduce a strong bias toward a specific design, but is allowed to explore the full design domain. For example, a truss structure might be optimized to be as light as possible, while still withstanding a set of load conditions.

Topology optimized designs often appear rather organic in shape and, if optimized without design constraints, might be hard to manufacture. However, with the increasing sophistication of additive manufacturing methods [Nin+15] and advanced CAD systems, highly optimized complex structures are increasingly common, e.g. in the aircraft industry [Emm+11]. An example for such a structure, manufactured with additive manufacturing, is given in Figure 1.2. Topology optimization methods generally use many optimization parameters, therefore efficient calculation methods are needed. Ideally, the computational complexity should be independent of the number of parameters. This motivates the usage of adjoint methods, introduced in later sections.

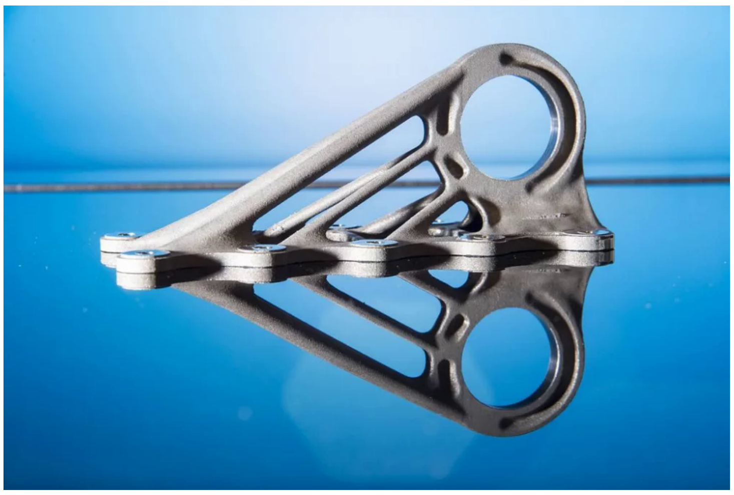


Figure 1.2: Structural part of an Airbus A350, optimized for weight. Part sintered from metal by additive manufacturing. Source: Airbus [Air18].

In the field of structural FEM, different flavors of topology optimization have emerged, such as level set methods [SM13].

In contrast to the history of applications in FEM, the application of topology optimization to CFD problems is comparatively recent [BP03]. Topology optimization immerses a geometry into the available design domain, by selectively blocking cells not located inside the geometry for the flow, e.g. by some penalization. A big advantage of CFD topology optimization is that the same mesh representation can be used for all optimization stages, eliminating the need for expensive remeshing or mesh morphing. If needed, the domain can still be refined near the topology boundaries. A disadvantage of the naive implementation is that the immersed geometry is not separated from the rest of the design space by real walls, to which wall boundary conditions could be easily applied. This complicates the application of wall boundary conditions (e.g. heat transfer) and the evaluation of turbulent wall functions. Furthermore it reduces the accuracy of the solution. If higher physical accuracy near the walls is required, the immersed boundary method [Pes02] can be combined with topology optimization [Mit+08].

Topology optimization for ducted flows was introduced to OpenFOAM using continuous adjoints [OVW07; Oth08]. The introduction of penalty terms to the Navier-Stokes equations, as well as the derivation of the continuous adjoint equations are discussed in detail in Section 2.5.

2 Foundations

In this chapter the necessary foundations, needed to comprehend the later chapters, will be laid. A brief introduction to (computational) fluid dynamics is given, before the discretization with FVM is introduced. Algorithmic differentiation and general optimization methods will be discussed. A brief introduction to the AD tool `dco/c++` will be given, which will be used later on to illustrate abstract concepts with code examples. Finally, the discrete adjoint residual approach is introduced. Additionally graph coloring techniques are presented, which will later be applied to more efficiently evaluate the discrete adjoint residuals.

2.1 Navier-Stokes Equations

The Navier-Stokes [Nav23; Sto51] equations are the most prevalent equations in CFD. They describe the conservation of momentum and mass for viscous fluids. In the following, the basics of general conservation laws are presented. They are subsequently used to derive the Navier-Stokes equations for incompressible Newtonian flow.

2.1.1 Derivation from Conservation Laws

The fundamental laws of physics dictate, that certain physical quantities in a system remain constant, as the system evolves in time. Important conservation laws include the conservation of energy, momentum, and mass.

Reynolds Transport Theorem

In the context of fluid flow, the conservation laws specify the conservation of physical quantities over moving material volumes M, which travel along the fluid. However, for practical computation it is desirable to express these laws for control volumes V. Mass can travel over the system boundaries through inlets and outlets. An observed mass volume thus leaves the system after a relatively short period of time, making material volumes inconvenient. The Reynolds transport theorem [RBM03] provides the required connection between a material volume M and corresponding control volumes V.

Let Ψ be a quantifiable attribute of a flow (e.g. mass, momentum, energy) and let $\psi = \frac{\mathrm{d}\Psi}{\mathrm{d}m}$ be the intensive value of Ψ (amount of ψ per unit mass m), that is

$$\Psi = \int_M \rho\psi \,\mathrm{d}M \,.$$

For a material volume M, the total change of quantity Ψ is determined by the change of Ψ in V plus the net flow of Ψ into and out of the control volume through its control surface S. Let ρ be the density of the fluid, $\mathbf{u} \in \mathbb{R}^3$ the velocity and $\mathbf{n} \in \mathbb{R}^3$ the outward facing normal to the control surface S. Then the Reynolds transport theorem states that

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_M \rho\psi \,\mathrm{d}M = \int_V \frac{\partial}{\partial t}(\rho\psi)\,\mathrm{d}V + \int_S \rho\psi\mathbf{u}\cdot\mathbf{n}\;\mathrm{d}S\,,$$

or by transforming the surface integral to a volume integral by applying the divergence theorem

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_M \rho\psi \,\mathrm{d}M = \int_V \left(\frac{\partial}{\partial t}(\rho\psi) + \boldsymbol{\nabla}\cdot(\rho\psi\mathbf{u})\right)\mathrm{d}V\,. \tag{2.1}$$

Derivation of Mass Conservation for Incompressible Flow

The principle of conservation of mass states that, without the presence of mass sources and sinks, the mass of fluid in a region M will be conserved, that is it does not change over time:

$$\frac{\mathrm{d}m}{\mathrm{d}t}\bigg|_M = 0\,.$$

The mass conservation for a fixed control volume V can be derived from Equation (2.1). Choosing $\Psi = m$ the corresponding intensive quality is $\psi = 1$ and

$$\int_V \left(\frac{\partial\rho}{\partial t} + \boldsymbol{\nabla}\cdot\rho\mathbf{u}\right)\mathrm{d}V = 0\,.$$

For this equation to hold for any control volume V, the integrand has to vanish at each point $v \in \Omega \subset \mathbb{R}^3$:

$$\frac{\partial\rho}{\partial t} + \boldsymbol{\nabla}\cdot\rho\mathbf{u} = 0\,. \tag{2.2}$$

For incompressible flows, the density is assumed to be constant throughout the domain in both space and time ($\mathrm{d}\rho(\mathbf{x},t)/\mathrm{d}t = 0$). Thus, Equation (2.2) can be simplified to the differential form

$$\boldsymbol{\nabla}\cdot\mathbf{u} = 0\,.$$

This equation is called mass conservation or mass continuity equation. From the equation directly follows, that the velocity field of an incompressible fluid is divergence free.

Applying the divergence theorem gives the integral form

$$\int_S \mathbf{u}\cdot\mathbf{n}\,\mathrm{d}S = 0\,,$$

which makes it obvious that all flow (mass) that enters a system must leave it again at some other point.

Derivation of Momentum Conservation for Incompressible Flow of Newtonian Fluids

Newton's second law states:

$$\frac{\mathrm{d}m\mathbf{u}}{\mathrm{d}t} = \sum \mathbf{f}\,,$$

where $\mathbf{u}$ is the velocity of an object, m is the mass, and $\mathbf{f}$ are forces acting on the object. From this, the momentum conservation equation can be derived by considering the velocity $\mathbf{u}$ as the conservation variable ψ in Equation (2.1)

$$\frac{\partial}{\partial t}\int_V \rho\mathbf{u}\;\mathrm{d}V + \int_S \rho\mathbf{u}\mathbf{u}\cdot\mathbf{n}\;\mathrm{d}S = \sum \mathbf{f}\,,$$

where $\mathbf{f}$ are the external and internal (viscous) forces acting on the system, replacing the material volume integration. The forces on the right hand side will now be expressed with intensive quantities, by making the assumption of Newtonian fluids. For Newtonian fluids, it is assumed that the shear stress τ inside the fluid can be expressed by the shear velocity $d\mathbf{u}/d\mathbf{n_u}$ as

$$\tau = \mu \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{n_u}} \,,$$

where $\mathbf{n_u}$ is the direction perpendicular to the flow and μ is the (dynamic) viscosity.

From this assumption the following relations for the stress tensor T and the deformation tensor D can be derived (see e.g. [BW97]). They only include intensive quantities and introduce the pressure p into the momentum equations.

$$\begin{aligned} T &= -\left(p + \frac{2}{3}\eta \boldsymbol{\nabla}\cdot\mathbf{u}\right) I + 2\eta D \,, \\ D &= \frac{1}{2}\left(\boldsymbol{\nabla}\mathbf{u} + (\boldsymbol{\nabla}\mathbf{u})^T\right) \,. \end{aligned}$$

With those tensors, the acting forces can be split into internal viscous forces and external forces

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_V \rho\mathbf{u}\,\mathrm{d}V + \int_S \rho\mathbf{u}\mathbf{u}\cdot\mathbf{n}\,\mathrm{d}S = \int_S T\cdot\mathbf{n}\,\mathrm{d}S + \int_V \rho\mathbf{b}\,\mathrm{d}V \,,$$

where $\mathbf{b}$ are the external body forces per unit mass. By introducing an infinitesimally small control volume V, one obtains the differential form of the momentum conservation equations

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \boldsymbol{\nabla}\cdot(\rho\mathbf{u}\mathbf{u}) = \boldsymbol{\nabla}\cdot T + \rho\mathbf{b} \,.$$

For isothermal and subsonic flows, the assumptions of constant fluid density and viscosity are usually valid. With those two assumptions, the derivatives of ρ and μ vanish in both space and time, allowing to simplify the equations into

$$\frac{\partial\mathbf{u}}{\partial t} + (\mathbf{u}\otimes\boldsymbol{\nabla})\,\mathbf{u} = \nu\boldsymbol{\nabla}^2\mathbf{u} - \frac{1}{\rho}\boldsymbol{\nabla}p + \mathbf{b} \,, \tag{2.3}$$

where $\nu = \mu/\rho$ is the kinematic viscosity.

Navier-Stokes Equations for Steady Incompressible Flow

Combining the conservation of momentum and conversation of mass into one system of partial differential equations, the governing Navier-Stokes equations for steady ($\partial\mathbf{u}/\partial t = 0$) incompressible laminar flow read as follows:

$$(\mathbf{u}\otimes\boldsymbol{\nabla})\,\mathbf{u} = \nu\boldsymbol{\nabla}^2\mathbf{u} - \frac{1}{\rho}\boldsymbol{\nabla}p + \mathbf{b} \tag{2.4}$$

$$\boldsymbol{\nabla}\cdot\mathbf{u} = 0 \,. \tag{2.5}$$

2.1.2 Navier-Stokes Equations for Topology Optimization

As mentioned in Section 1.4.3, topology optimization in a CFD context can be implemented by blocking the flow in specific cells of the discretization, corresponding to regions which should be excluded from the design domain. A commonly used approach is to block the flow by introducing an artificial resistance for the flow in the momentum Equation (2.4):

$$\begin{aligned} (\mathbf{u} \otimes \boldsymbol{\nabla})\,\mathbf{u} &= \nu \boldsymbol{\nabla}^2 \mathbf{u} - \frac{1}{\rho} \boldsymbol{\nabla} p + \mathbf{b} - \alpha \mathbf{u} \\ \boldsymbol{\nabla} \cdot \mathbf{u} &= 0\,. \end{aligned} \tag{2.6}$$

The new term $\alpha\mathbf{u}$ in (2.6) implements a momentum sink, which introduces a flow resistance and as such allows to penalize regions of cells considered counterproductive for the flow. Illustratively the resistance term $\alpha\mathbf{u}$ can be interpreted as a porosity. The pressure drop Δp over a porous medium of depth Δx is commonly modeled by Darcy's law [Whi86] as $\Delta p/\Delta x = -(\mu/\kappa)\mathbf{u}$, with permeability κ. Therefore the momentum source is an implementation of Darcy's law with $\mu/\kappa = \alpha$.

The question in which regions to penalize the flow, that is to find an optimal field α, motivates the need for (adjoint) sensitivities of a given objective with respect to the (discretized) parameters $\boldsymbol{\alpha}$.

In order to avoid momentum sources, which would accelerate the flow instead of penalizing it, the values of α should be constrained such that $\alpha \geq 0$. To avoid too stiff discretization matrices and to obtain a steady converged state for the field $\boldsymbol{\alpha}$, the parameter should also be capped below a certain maximum value $\alpha_{\max}$.

The constrained optimization problem to find a feasible and optimal field α can be stated as

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \mathcal{J}(\mathbf{u}(\alpha), p(\alpha), \alpha) \\ \text{subject to} \quad & \mathbf{r}\,(\mathbf{u}(\alpha), p(\alpha), \alpha) = 0 \\ & 0 \leq \alpha < \alpha_{\max}\,, \end{aligned}$$

where $\mathcal{J}$ is a scalar or multivariate objective and $\mathbf{r}$ are the residuals of the momentum and mass conservation equations.

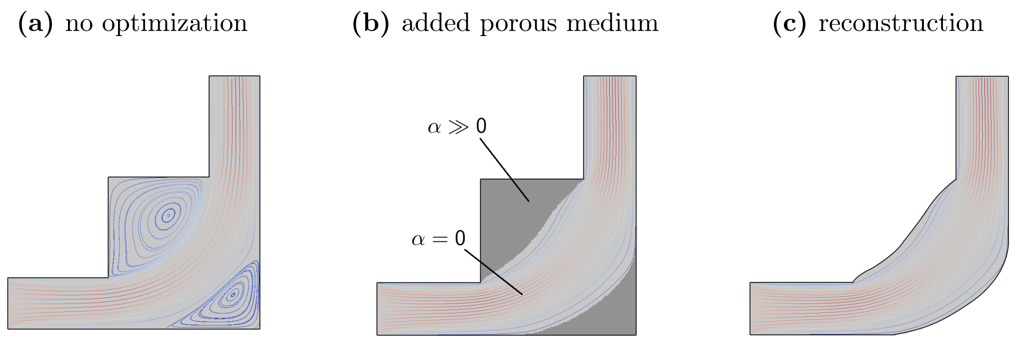


Figure 2.1: Typical topology optimization workflow, from baseline geometry (left) to final optimized geometry (right).

A typical three step topology optimization workflow is depicted in Figure 2.1. Starting from an initial configuration of the domain (left), cells are penalized to improve the flow according to some objective (middle). To reduce discretization errors and to make the design parametric again, the geometry is transformed to a CAD representation, and remeshed. On this geometry the final flow state is calculated (right).

2.1.3 Turbulence Models

In contrast to laminar flows, turbulent flows are characterized by strong irregularities of all their properties. Most importantly they exhibit strong changes of their velocity and pressure in space and time [Dur08]. Those irregularities are chaotic in nature and occur on a variety of turbulent length scales. Turbulent flows exhibit a high amount of dissipation due to the internal viscous shear stresses.

Broadly speaking, turbulent behavior occurs once the flow has passed a critical Reynolds number. The Reynolds number Re is defined as

$$\mathrm{Re} = \frac{\rho u L}{\mu} = \frac{uL}{\nu}\,,$$

where L is a characteristic length of the flow domain (e.g. diameter of a pipe, length of a car), and u is the scalar velocity magnitude of the fluid with respect to the obstacle. The somewhat arbitrary choice of characteristic length L makes the Reynolds number only a rough indicator for the flow properties of the flow. A pipe flow with Reynolds number $\mathrm{Re}_D > 4000$ is considered to be fully turbulent, while a flow with $\mathrm{Re}_D < 2000$ is considered laminar. Flows in the region in between 2000 and 4000 are in a transition area, and are consequently called transition flows.

The analysis of turbulent flows is a very relevant field, as most fluid flows occurring in nature or technical applications exhibit some degree of turbulence.

The most intuitive, yet very expensive, way of calculating solutions to turbulent flows is to utilize direct numerical simulation (DNS). With DNS the Navier-Stokes equations (2.5) are solved without any further modeling of the turbulent nature of the problem. For DNS to give reasonable results, all length scales of the physical problem, from the smallest turbulent length scales to the macroscopic scale, have to be resolved directly, both in space and time, by the chosen discretization. This resolution requirement makes DNS challenging and even on current HPC hardware it is rarely used for complex geometry. One of its main uses is to derive and understand turbulence models and to simulate flows which are very hard to model otherwise, e.g. the laminar/turbulent transition region. With the transition of HPC to exascale computing, DNS methods will likely become more prevalent, as the lower algorithmic complexity of DNS should improve parallel scalability [Che+09].

The minimization of the computational requirements, while still reasonably capturing the physical influence of turbulence, has been a very active field of research over the last decades (for a historical overview see e.g. [Wil+98]). In the following sections a brief introduction of the most common one- and two-equation turbulence models for steady flows will be given.

One Equation Model: Spalart-Allmaras

A popular one equation turbulence model is the Spalart-Allmaras model [SA92]. It is popular due to its comparatively modest computational effort, involving the solution of only one additional

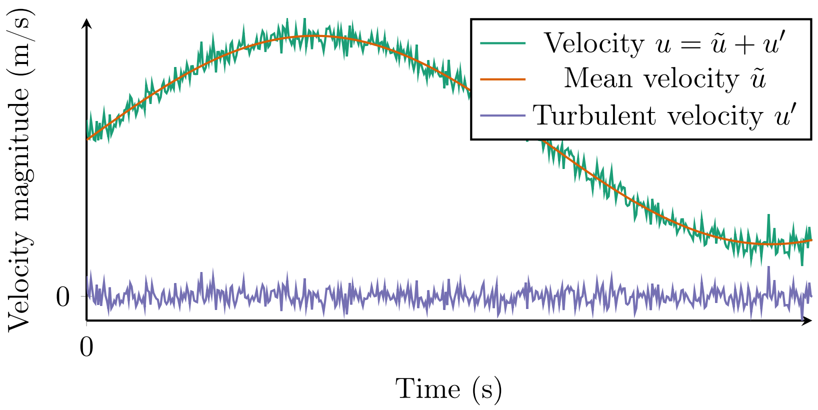


Figure 2.2: Visualization of the Reynolds decomposition. Function u is decomposed into a smooth (filtered) part $\bar{u}$ and chaotic part u'.

PDE. This model is particularly suitable for aerospace and turbomachinery applications, as it was developed for aerodynamic flows.

It models the transport of the kinematic eddy turbulent viscosity $\tilde{\nu}$ as a convection-diffusion equation:

$$\frac{\partial(\rho\tilde{\nu})}{\partial t} = \boldsymbol{\nabla}\cdot(\rho D_{\tilde{\nu}}\tilde{\nu}) + \frac{C_{b2}}{\sigma_{\nu_t}}\rho\|\boldsymbol{\nabla}\tilde{\nu}\|^2 + C_{b1}\rho\tilde{S}\tilde{\nu}\,(1-f_{t2}) - \left(C_{w1}f_w - \frac{C_{b1}}{\kappa^2}f_{t2}\right)\rho\frac{\tilde{\nu}^2}{\tilde{d}^2} + S_{\tilde{\nu}}\,.$$

The equation involves several empirically determined constants C, σ, κ, as well as other closure equations. For these definitions, we defer to the literature [SA92; AJ12]. From the complexity of this governing equation it can be figured, that the derivation of a continuous adjoint turbulent model (see Section 2.5), while possible [Zym+09], is laborious and the implementation error prone.

Two Equation Models: Reynolds-Averaged Navier-Stokes Equations

The Reynolds-averaged Navier-Stokes equations (RANS) are time averaged equations of motion. They are used to transform a flow field, that is transient only at the turbulent length scale to a steady flow where the amount of turbulence is modeled by an additional set of variables.

The velocity of a flow is split into a mean and a fluctuating part using Reynolds decomposition,

$$\mathbf{u} = \tilde{\mathbf{u}} + \mathbf{u}'\,,$$

where $\tilde{\mathbf{u}}$ is the mean (time averaged) velocity and $\mathbf{u}'$ the fluctuating turbulent part. (In the literature the mean velocity is usually denoted with $\bar{\mathbf{u}}$, to avoid collision with the adjoint notation we use the differing notation $\tilde{\mathbf{u}}$.) Figure 2.2 illustrates the superposition of a (synthetic) continuous differentiable function $\tilde{u}$, with randomized values from a standard distribution of lower length scale u'. This example mimics the characteristics of the Reynolds decomposition.

The effect of the turbulence can be modeled by an additional transport equation. The RANS momentum equation for steady incompressible flow can be given as follows (with $i = \{0, 1, 2\}$, Einstein summation over the indices j, and $\mathbf{x} = [x_0, x_1, x_2] = [x, y, z]$):

$$\tilde{u}_j\frac{\partial\tilde{u}_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\nu\frac{\partial\tilde{u}_i}{\partial x_j} - u_i'u_j'\right)\,.$$

For the solution of the RANS equations, additional closure terms are needed for the $u_i{}'u_j{}'$ term, which is commonly referred to as Reynolds stress tensor. It is the closure of this term, in which the different RANS turbulence models differ from one another. Using the Boussinesq approximation [Pop00], the Reynolds stress tensor can be related to a turbulent eddy viscosity ν_t. In the following, two common closure models are presented, which enable to calculate ν_t, namely the k-ϵ and the k-ω models.

k-ϵ model: The k-ϵ equations model the transport of turbulent energy k and turbulent dissipation (into heat) rate ϵ [LS83]:

$$\begin{aligned}\frac{\partial k}{\partial t} + \frac{\partial (k u_i)}{\partial x_i} &= \frac{\partial}{\partial x_j}\left[\frac{\nu_t}{\sigma_k}\frac{\partial k}{\partial x_j}\right] + 2\nu_t E_{ij} E_{ij} - \epsilon \\ \frac{\partial \epsilon}{\partial t} + \frac{\partial (\epsilon u_i)}{\partial x_i} &= \frac{\partial}{\partial x_j}\left[\frac{\nu_t}{\sigma_\epsilon}\frac{\partial \epsilon}{\partial x_j}\right] + C_{1\epsilon}\frac{\epsilon}{k} 2\nu_t E_{ij} E_{ij} - \epsilon - C_{2\epsilon}\frac{\epsilon^2}{k}\ .\end{aligned}$$

The eddy viscosity needed in the equations, as well as for the closure of the Reynolds stress tensor is calculated as $\nu_t = C_\nu k^2/\epsilon$.

The k-ϵ model is suited for the calculation of sheer free layer flows [Bar+97] and flows with low pressure gradients. However, it does not perform well for flows that exhibit large adverse pressure gradients [Wil+98].

k-ω model: The standard k-ω model [Wil+98], with default parameters reads as:

$$\begin{aligned}\frac{\partial k}{\partial t} + u_j\frac{\partial k}{\partial x_j} &= \tau_{ij}\frac{\partial u_i}{\partial x_j} - \frac{9}{100}k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{1}{2}\nu_T\right)\frac{\partial k}{\partial x_j}\right] \\ \frac{\partial \omega}{\partial t} + u_j\frac{\partial \omega}{\partial x_j} &= \frac{5}{9}\frac{\omega}{k}\tau_{ij}\frac{\partial u_i}{\partial x_j} - \frac{3}{40}\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{1}{2}\nu_T\right)\frac{\partial \omega}{\partial x_j}\right]\ .\end{aligned}$$

The eddy viscosity needed in the equations, as well as for the closure of the Reynolds stress tensor is calculated as $\nu_t = k/\omega$. The k-ω model is best suited for flows with wall effects. Extended versions like k-ω SST (shear stress transport) [Men93] exist, which switch between k-ω behavior near walls and k-ϵ behavior in the free stream.

Other Models

If one desires to capture the transient effects of the turbulence or calculates a case for which the mentioned RANS methods are not well suited, other more computationally expensive methods are available. One particularly useful method is the large eddy simulation (LES) model. The LES model is a filtering approach, which removes the information of the flow field on the lowest turbulent length scale frequencies and models the effect of that information on the flow field by various approaches. A flow field filtered in such a way allows to calculate on a coarser mesh, compared to a DNS case. LES simulations are inherently transient, they thus consume a lot of computational resources. The requirements are amplified when coupled with the calculation of the adjoint, as potentially very long iteration histories need to be reversed. The detached eddy simulation (DES) is a combination of the RANS and LES approaches, that switches between a RANS model in regions of the flow with very small turbulent length scales (especially near walls), which can not be covered by LES without a prohibitively fine mesh, and a LES model in regions where the spatial resolution of the mesh allows to resolve the turbulence by LES.

2.1.4 Computational Mesh

The calculation of solutions to the governing equations using FVM requires the discretization of the domain with finitely small control domains. They need to cover the full computational domain, without overlapping, follow the shape of the boundary, and fulfill certain quality criteria. This domain discretization is commonly called mesh or grid (for structured meshes). For general CFD applications, meshes can be categorized into one of two classes: Structured or unstructured. Structured meshes allow the efficient determination of the cell neighborhood of a given cell without costly memory lookups. For example, in a 2D mesh, where the cells are numbered in a matrix like $(row, column)$ fashion, the neighborhood of cell (away from any boundaries) (i, j) can be obtained by $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. In general this leads to good cache locality of the code implementing the discretization, and sparse (banded) discretization matrices with known sparsity pattern. Structured meshes do not necessarily have to be equidistant or orthogonal (see e.g. Figure 2.3). However, the number of neighbors and connectivity for each cell is fixed, severely limiting the possibility to refine the mesh and the adaptability to complex geometries. This downside can be mitigated by allowing hanging nodes, resulting in a mesh which retains the good numerical characteristics of a structured mesh, while still allowing to refine near important geometry features.

Unstructured meshes allow the decomposition of the computational domain into arbitrary polyhedral subvolumes. Thus, the number of neighbors of a cell is not fixed, and the neighborhood can not be trivially constructed. The mesh connectivity must be stored in a suitable data structure and looked up when the neighborhood needs to be constructed. This leads to more memory access and a non cache-local memory access pattern.

The convergence speed and solution quality of many numerical solution algorithms depends on the mesh quality. Mesh quality can be characterized by different metrics, the importance of which varies between different solution algorithms.

Cell aspect ratio: Ratio between the biggest and smallest area of a cells boundary box (best if ratio equals one).

Cell non-orthogonality: Angle between the vector connecting two cells and the face normal of the face connecting both cells (best if equals zero). Compare to vector $\mathbf{S}_f$ in Figure 2.4(a).

Cell non-conjunctionality: Distance from the intersection of the connection of two cell centers with the face connecting both cells (f') to the face center (f_c) (best if equals zero). Compare to Figure 2.4(b).

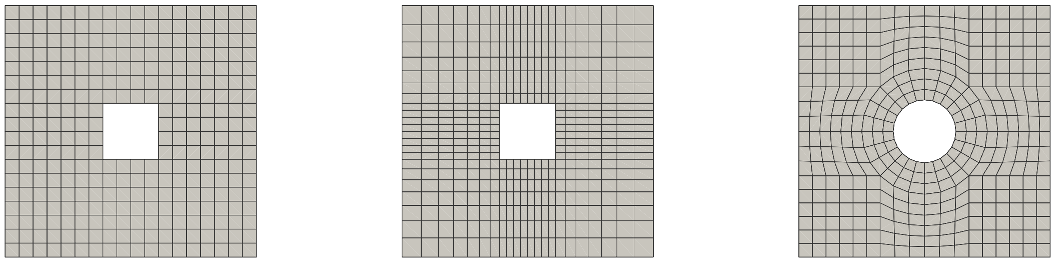
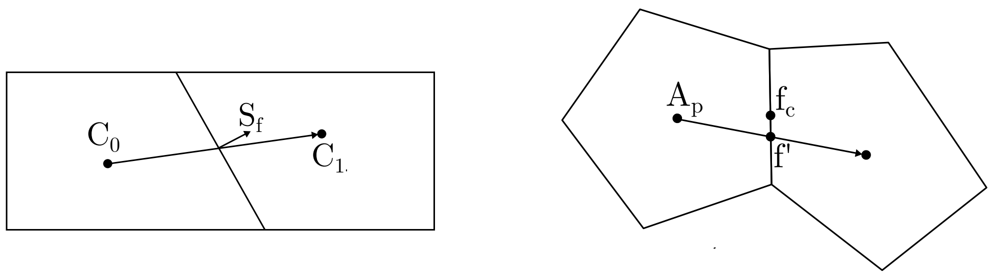


Figure 2.3: Three structured meshes. Structured equidistant mesh (left), structured mesh refined around region of interest (middle), structured non-Cartesian mesh (right).



(a) Mesh non-orthogonality **(b)** Mesh non-conjunctionality

Figure 2.4: Visualization of mesh non-orthogonality (left) and non-conjunctionality (right).

2.2 Finite Volume Methods

The finite volume method (FVM) was developed in the 1970s and 1980s as a way to discretize and solve partial differential equations (PDEs). The FVM is popular in CFD, as it allows to discretize directly on the physical domain without transforming to a computational domain first [MMD+16] (like the test function space required by FEM). It is able to discretize complex domains, granted that enough volumes are available to capture the features of the domain. It is related to the finite difference (FD) and finite element methods (FEM). For a comparison between FVM, FEM, and FD see e.g. [EGH00].

The FVM is based on the concept of balance equations on discrete control volumes. In a first step the PDEs, given in differential form, are integrated and transformed into balance equations over the discrete control volumes. The resulting volume and surface integrals are transformed into discrete expressions by approximating them with numeric quadratures. In a second approximation step the quantities from the cell centroids are interpolated onto the faces and are numerically integrated over the face area. The surface integrals describe the flux of conservation quantity in and out of the volumes through their respective faces.

The formulation of fluxes through the cell faces makes the method locally conservative (meaning that the flux leaving a volume is equal to the flux entering its adjacent volume). This makes the method well suited for problems where fluxes are of importance, such as fluid mechanics. It can be applied to both structured and unstructured meshes in arbitrary 2D and 3D domains.

2.2.1 FVM on Scalar Transport Equation

We will motivate the FVM with the discretization of the general convection diffusion equation. It models the transport of a scalar physical quantity (e.g. temperature) within a fluid field, which travels with given velocity $\mathbf{u}$. The quantity is convected downstream by the velocity field. In addition to convection, the quantity spreads in all directions, due to diffusivity.

The transport of scalar quantity ψ in a fluid flow can be modeled by the following partial differential equation [MMD+16]:

$$\underbrace{\frac{\partial \rho\psi}{\partial t}}_{\text{transient term}} + \underbrace{\nabla \cdot (\rho\mathbf{u}\psi)}_{\text{convective term}} = \underbrace{\nabla \cdot (\mu\nabla\psi)}_{\text{diffusive term}} + \underbrace{Q^{\psi}}_{\text{source term}} ,$$

where ρ and $\mathbf{u}$ are the density and velocity of the fluid field transporting ψ, and μ is the diffusivity constant for ψ. The equation is closely related to the Navier-Stokes momentum equation. The momentum can be stated as the transport equation, where $\psi = \mathbf{u}$. The non-linearity introduced in the convection complicates the formulation of the FVM, thus for now we will treat ψ and $\mathbf{u}$ as separate entities. Starting from the PDE formulation, the construction of the FVM follows along the lines of the derivation of the governing equation, but backwards.

Integrating the differential steady state form ($\partial(\rho\psi)/\partial t = 0$) over a (finite) volume $V_C \subset \Omega \subset \mathbb{R}^3$ gives the following integral form:

$$\int_{V_C} \boldsymbol{\nabla} \cdot (\rho \mathbf{u} \psi)\,\mathrm{d}V = \int_{V_C} \boldsymbol{\nabla} \cdot (\mu \boldsymbol{\nabla} \psi)\,\mathrm{d}V + \int_{V_C} Q^{\psi}\,\mathrm{d}V\,. \tag{2.7}$$

Transforming the volume integrals for convection and diffusion to surface integrals by applying the divergence theorem:

$$\oint_{\partial V_C} (\rho \mathbf{u} \psi) \cdot \mathbf{n}\,\mathrm{d}\mathbf{S} = \int_{\partial V_C} (\mu \boldsymbol{\nabla} \psi) \cdot \mathbf{n}\,\mathrm{d}\mathbf{S} + \int_{V_C} Q^{\psi}\,\mathrm{d}V\,.$$

This states, that, in absence of transient effects, the amount of quantity entering and exiting each finite volume V_C driven through the convective and diffusive fluxes, has to match the amount created/removed by the source term. From this integral formulation over cells and faces, we aim to introduce approximations, which transform the integrals on each volume V_C into sums. The summands of the individual elements can later be assembled into a global linear system, which allows to solve for the field of the unknown quantity $\boldsymbol{\psi}$.

The surface integrals of the fluxes are approximated by replacing them with numerical integrations on the faces. In the discretized domain the values of the quantity are only explicitly defined at the cell centers. The values thus need to be interpolated from the cell center to the cell faces. The most common option is to choose one integration point at the face center and multiplying it with the face area, yielding a second order accurate approximation (the integral of linear functions is evaluated exactly by this integration). We define the total flux $\mathbf{J}$ as the sum of the convective and diffusive flux as

$$\mathbf{J}^{\psi} = \rho \mathbf{u} \psi - \mu \boldsymbol{\nabla} \psi\,.$$

The integration over the flux through face f can then be approximated by

$$\oint_f \mathbf{J} \cdot \mathrm{d}\mathbf{S} \approx \mathbf{J}_f \cdot \mathbf{s}_f\,,$$

where $\mathbf{J}_f$ is the flux calculated at the midpoint of face f and $\mathbf{s}_f$ is the face area vector.

The per face calculation of the fluxes from values at cell centers of the adjacent cells makes the cell centered finite volume scheme inherently conservative. The flux leaving a control domain over one face exactly equals the flux entering the adjacent control domain (with opposite sign).

For orthogonal meshes, the vector connecting two cell centers does pass through the center of the face shared by both cells. In this case the value of ψ at the face center can be linearly interpolated from the neighboring cells as

$$\psi_f \approx \left(1 - \frac{\|\mathbf{d}_1\|}{\|\mathbf{d}\|}\right) \psi_C + \frac{\|\mathbf{d}_1\|}{\|\mathbf{d}\|} \psi_N\,,$$

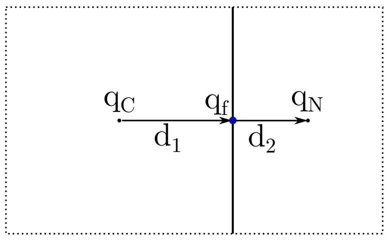


Figure 2.5: Two orthogonal cells, sharing one face. Cell centers are connected by $\mathbf{d} = \mathbf{d}_1 + \mathbf{d}_2$.

where $\mathbf{d}_1$ is the vector connecting the midpoint of cell C to the face center $\mathbf{q}_f$ and $\mathbf{d}$ the vector connecting $\mathbf{q}_C$ and $\mathbf{q}_N$ (see Figure 2.5).

The gradient in direction of the normal $\mathbf{n} = \mathbf{d}_2/\|\mathbf{d}_2\|$ can be approximated by the difference

$$\boldsymbol{\nabla}\psi_f \cdot \mathbf{n}_f \approx \frac{\psi_N - \psi_C}{\|\mathbf{d}\|} .$$

Thus, the flux integral of the face can be evaluated by a linear expression, depending only on the quantities of ψ at the cell centres of cell C and its neighbor cell N:

$$\mathbf{J}_f \cdot \mathbf{S}_f = a_C\psi_C + a_N\psi_N .$$

For non-orthogonal meshes, the vector connecting two cell centers does not necessarily pass through the center of the face shared by both cells (see Figure 2.6). If the intersection point is used instead of the actual face centerpoint, the numerical interpolation to the face loses its second order accuracy. Furthermore, the face normal direction, along which the gradient has to be evaluated, does not align with the vector connecting the cell centers anymore. The flux can be corrected to better match the flux which would be obtained by using the correct values at the midpoint. However, as this correction contains non-linear terms it can not be added to the system matrix but instead has to be treated as a source term for the linear equation system.

$$\psi_f = a_C\psi_C + a_N\psi_N + f(\psi_C, \psi_N) .$$

Discretizing the remaining volume integral of the source term from (2.7) by a numerical integration over the cell

$$\int_{V_C} Q^\psi \,\mathrm{d}V \approx Q^\psi_C \cdot V_C ,$$

we can express the balance equation at each cell by the expression

$$a_C\psi_C + \sum_{F\sim\mathrm{NB}(C)} a_F\psi_F = b_C ,$$

where $F \sim \mathrm{NB}(C)$ denotes all cells in the neighborhood of C. The source term, as well as the non-linear parts of the non-orthogonal corrections, are accumulated into the right-hand side b_C.

From those balance equations one can lump together all cells into a linear equation system:

$$A_\psi \boldsymbol{\psi} = \mathbf{b}_\psi ,$$

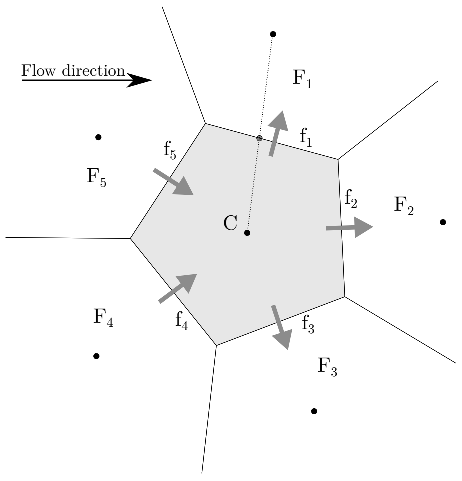


Figure 2.6: Cell centered finite volume. Flux is calculated through faces. Note the distance between face midpoint and the intersection between face and cell center vector due to non-orthogonality.

where the matrix coefficients a_C constitute the matrix diagonal A_{CC} and the coefficients a_F populate the off diagonal elements A_{FC} and A_{CF}. The matrix can be solved to obtain the desired quantities $\boldsymbol{\psi}$. The linear equation system needs to be assembled and solved multiple times, if non-linear effects are included in either the right-hand side (due to the non-orthogonal correction) or the assembly of the matrix (i.e. the matrix entries depend on ψ, which is commonly the case when discretizing the momentum equation, where the velocity is both the desired quantity and driving the convection).

The fluxes of each cell only directly depend on the values at the cell itself and the directly neighboring cells. Thus, the matrix A_ψ will in general be sparse. Figure 2.7 shows the sparsity pattern resulting from the discretization of the momentum equations for the motorbike tutorial case of OpenFOAM.

This case generates a $n_C = 352\,570$ cell unstructured mesh with $n_F = 1\,054\,817$ internal faces leading to a $352\,570 \times 352\,570$ sparse matrix with $n_{\mathrm{nz}} = n_C + 2 \cdot n_F = 2\,460\,120$ non-zero elements. Straight from the mesher the bandwidth of the matrix is 349 965 (left in the figure), a renumbering with the Cuthill–McKee algorithm [CM69] reduces the bandwidth to 20 875 (right in the figure). A reduced matrix bandwidth can lead to improved linear solver performance [Maf14]. For example, the incomplete LU factorization [CV97], utilized as a preconditioner in many iterative solvers, benefits from the lower fill-in generated by a mesh reordered with Cuthill–McKee.

2.2.2 Common FVM Boundary Conditions

In the FVM, fluxes over the domain boundaries are defined by boundary conditions defined on the boundary faces. A variety of boundary conditions exist to model diverse physical behavior. Here only the two most common choices are presented, namely the Dirichlet and Neumann boundary conditions. The former directly specifies the value of a transport quantity at the boundary face, the latter prescribes a flux.

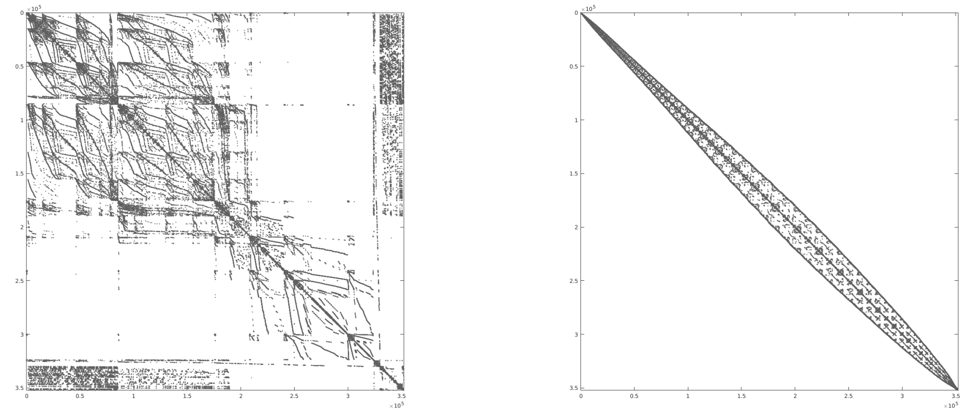


Figure 2.7: Sparse FVM discretization matrix of the momentum equations for a large unstructured mesh (motorbike tutorial case of OpenFOAM). Left in original node order from the `snappyHexMesh` mesher, right ordered with the Cuthill–McKee algorithm.

Dirichlet Boundary Condition

For a scalar quantity ψ, which is to be convected, e.g. at an inlet, the quantity on the boundary is set explicitly to a (possibly time and space dependent) specified value:

$$\psi_b = \psi_{\mathit{fix}}\,.$$

n A representation of a cell stencil in a structured Cartesian mesh, with a Dirichlet boundary condition applied to the west face, is shown in Figure 2.8. The flux through the boundary face can be explicitly calculated with the specified value as

$$\phi_b = \dot{m}_b \psi_b\,.$$

The flux only depends on the specified value and not on the cell central value ψ_C. This corresponds to an upwind interpolation of ψ_b from a virtual cell center $\mathbf{q}_W$.

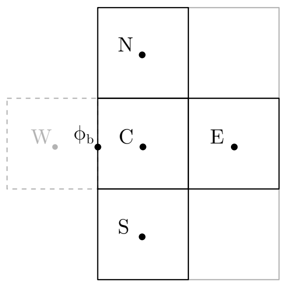


Figure 2.8: Finite volume stencil around a cell C with a boundary face to the west. Flux ψ_b is specified on the boundary face and enters the FVM discretization of the cell.

Examples for Dirichlet boundary conditions include:

- fixed velocity profile at an inlet, e.g. $\mathbf{u} = u_{in} \cdot \mathbf{n}_b$,
- velocity $\mathbf{u} = (0, 0, 0)$ at a no-slip wall boundary,
- and fixed pressure at outlet.

In OpenFOAM the Dirichlet boundary condition is specified by the `fixedValue` keyword, as shown in Listing 2.1 for a constant inlet velocity of 1 m/s.

```
dimensions       [0 1 -1 0 0 0 0];
boundaryField{
  inlet{
    type    fixedValue;
    value   uniform (1 0 0);
  }
}
```

Listing 2.1: Fixed velocity of 1 m/s at patch `inlet`, specified in case configuration file `0/U`.

Neumann Boundary Condition

The Neumann boundary condition fixes the flux through a face to a specified value:

$$\phi_b = q_b \|\mathbf{s}_b\| \,,$$

where q_b is the flux per unit-area and $\|\mathbf{s}_b\|$ is the face area of the boundary face. The prescription of the face flux is equivalent to fixing the gradient in face normal direction $\boldsymbol{\nabla}\psi_b \cdot \mathbf{n}_b$ to a fixed value. As the mass flux $\dot{m}$ is considered to be constant, the quantity ψ_b must change accordingly to realize the desired flux.

Common applications of the Neumann boundary condition are:

- zero gradient pressure condition at an inlet $\boldsymbol{\nabla}p \cdot \mathbf{n}_b = 0$,
- zero gradient velocity condition at an outlet $\boldsymbol{\nabla}\mathbf{u} \cdot \mathbf{n}_b = 0$,
- and symmetry plane.

In OpenFOAM the Neumann boundary condition is specified by the `fixedGradient` keyword. The by far most often used application of the Neumann boundary condition is to prescribe a zero gradient condition, which can also be set by using the `zeroGradient` keyword. This is illustrated for a zero gradient pressure condition on the inlet and domain walls in Listing 2.2

```
dimensions       [0 2 -2 0 0 0 0];
boundaryField{
  inlet{ type  zeroGradient; }
  walls{
    type  fixedGradient;
    value 0;
  }
}
```

Listing 2.2: Zero gradient pressure condition at patch `inlet`, specified in configuration file `0/p`.

2.3 Semi-Implicit Solution Algorithms

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) Algorithm [PS72] is one example for a class of solvers for systems of nonlinear partial differential equations (PDEs). The algorithm solves the steady incompressible Navier-Stokes equations by linearizing the equations, and discretizing them according to the FVM. The linearization of the convection term makes an outer iteration loop necessary. The embedding of linear equation system solvers into a more general outer iteration loop is a common occurrence in CFD and more general simulation codes which involve nonlinear (differential) equations [Nau+15].

The SIMPLE Algorithm decouples the momentum equations from the mass conservation equations. This allows to build and solve the linear equation systems for velocity and pressure independently, making the resulting linear systems considerably smaller and easier to solve. The Navier-Stokes equations can also be solved fully coupled which makes the inner iterations more expensive but considerably speeds up the convergence of the outer iteration.

For the solution of the momentum equation, the pressure is assumed to be known and accordingly only enters the equations on the right hand side. The momentum equations are solved component wise, giving a velocity field $\mathbf{U}^* = (\mathbf{u}_0^*, \dots, \mathbf{u}_{nC-1}^*)$ which fulfills them. For the velocity components outside of the implicit direction, the newest solutions from previous SIMPLE iterations are used, the coupling between the spatial directions is only introduced during the correction steps,

$$\begin{aligned} A_{u_x}\mathbf{U}_x^{*,i+1} &= \mathbf{b}_{u_x}(\mathbf{U}^i) \\ A_{u_y}\mathbf{U}_y^{*,i+1} &= \mathbf{b}_{u_y}(\mathbf{U}^i) \\ A_{u_z}\mathbf{U}_z^{*,i+1} &= \mathbf{b}_{u_z}(\mathbf{U}^i)\,, \end{aligned}$$

where the right hand side vector $\mathbf{b}_\mathbf{U}$ is a function of the velocities in the previous iteration step. At the beginning of the algorithm, the pressure, driving the velocity field, is only a guess. The velocities will in general not fulfill the (discretized) mass conservation equation. Therefore additional corrections are required.

When solving the momentum and mass conversation equations separately, there is no equation which allows to solve for the pressure field $\mathbf{p}$ directly, as the momentum equation is already used to uniquely determine the velocities, while the mass conversation equation only directly depends on the mass fluxes and hence the velocities. To close the equations, two correction terms for the pressure and velocities are defined. If the corrections can be related to one another, they can be used to determine a new pressure field.

A velocity correction is wanted which corrects the velocity field, such that it fulfills the mass conversation at every cell:

$$\mathbf{U} = \mathbf{U}^* + \mathbf{U}'\,,$$

where $\mathbf{U}^*$ are the velocities fulfilling the momentum equations and $\mathbf{U}'$ are the corrections needed to obtain velocities $\mathbf{u}$ consistent with the mass conversation equation.

We assume that the velocity corrections can be uniquely determined from a separate pressure correction

$$\mathbf{p} = \mathbf{p}^* + \mathbf{p}'\,,$$

where $\mathbf{p}^*$ is the current guess for the pressure, and $\mathbf{p}'$ is the correction required such that the pressures $\mathbf{p}$ drive the velocities to fulfill the mass conservation equation.

Tying the velocity corrections to the pressure corrections allows to assemble a linear equation for the pressure corrections

$$A_p \mathbf{p}' = \mathbf{b}_p \,,$$

which can be solved implicitly. The right hand side vector $\mathbf{b}_p$ bundles the explicit contribution of the pressure correction equation. The obtained pressure corrections $\mathbf{p}'$ can then be used to correct the pressures and also to explicitly correct the velocities to better solve the mass conservation equation (the latter correction is optional, but improves convergence). The alternating solution of the momentum and pressure correction equations is repeated until the flow field has converged.

In the discretization of the governing equations OpenFOAM introduces the scalar quantity ϕ, that describes the mass flux of fluid through a cell face (the convective part of the total flux $\mathbf{J}$ introduced in Section 2.2.1). The mass flux is required in the discretization of the pressure correction equation. It is defined as:

$$\phi = \rho A \left(\mathbf{u} \cdot \mathbf{n}\right) = \rho \left(\mathbf{u} \cdot \mathbf{s}\right) \,.$$

For incompressible problems, the normalization of the Navier-Stokes equations with ρ leads to a normalized mass face flux $\phi = \mathbf{u} \cdot \mathbf{s}$. As a face flux, it is defined on the cell faces, as opposed to the cell centered quantities like velocity and pressure, giving the discrete mass flux field $\boldsymbol{\phi} \in \mathbb{R}^{n_F}$. Intuitively the mass fluxes could be interpolated from the cell centered velocities onto the faces, however this introduces numerical issues, commonly referred to as checkerboarding. Instead the following iterative procedure is chosen to update ϕ in each iteration of the SIMPLE algorithm [RC83]:

$$\phi = \left(\frac{H(\mathbf{u})}{a_p} - \frac{1}{a_p}\boldsymbol{\nabla} p\right) \cdot \mathbf{s} \,,$$

where $H(\mathbf{u})$ is the product of the off-diagonal coefficients of a specific cell in the discretization of the momentum equations (see also Section 3.7.3) and a_p is the diagonal coefficient.

Summarizing, the SIMPLE Algorithm consists of the following steps:

1. Discretize the momentum equations and assemble linear equation system;
2. Solve discretized momentum equations to obtain intermediate velocity field $\tilde{\mathbf{U}}$;
3. Compute the uncorrected mass fluxes $\tilde{\boldsymbol{\phi}}$ at the cell faces;
4. Discretize the pressure correction equation and assemble linear equation system;
5. Solve discretized pressure correction equation to obtain pressure correction field $\mathbf{p}'$;
6. Calculate new pressure field $\mathbf{p}$ from $\mathbf{p}'$, if desired apply under-relaxation: $\mathbf{p}^{i+1} = \mathbf{p}^i + a(\mathbf{p} - \mathbf{p}^i)$, with $0 < a \leq 1$;
7. Correct the mass fluxes $\boldsymbol{\phi}$ at the cell faces with the calculated pressure corrections;
8. Correct velocities $\tilde{\mathbf{U}}$ to fulfill mass conservation, yielding $\mathbf{U}$
9. If desired, apply under-relaxation: $\mathbf{U}^{i+1} = \mathbf{U}^i + a(\mathbf{U} - \mathbf{U}^i)$, with $0 < a \leq 1$;
10. Repeat steps 1–9 until convergence of pressure and velocity fields is obtained or maximum number of iterations has been exceeded.

The SIMPLE algorithm is popular due to its rather straightforward implementation. For better convergence with less under-relaxation, the improved SIMPLEC algorithm can be implemented. Due to the decoupling of the PDEs, both algorithms produce linear equation systems which are significantly smaller and less stiff than a fully coupled system. Due to the steady growth of computing capabilities, a fully coupled approach becomes increasingly feasible for practical applications. Such solvers are available in the FOAM-extend project [Jas+18]. All applications of AD, shown on the SIMPLE algorithm in the later sections, are feasible for those types of solvers as well [STN].

2.4 Mesh & Field Conventions

2.4.1 Mesh Topology

OpenFOAM uses the FVM approach, as outlined in Section 2.2, to discretize ODEs. The computational domain is discretized into finitely small volumes, also called cells. We define the following terminology to construct the cells from basic entities.

- Let Ω be the computational domain $\Omega \subset \mathbb{R}^3$ and $\Gamma = \partial\Omega$ the boundary of that domain.
- Let Q be a set of distinct points $Q = \{\mathbf{q} \in (\Omega \cup \Gamma)\}$ (the obvious symbol choice of $\mathbf{p}$ is already taken by the pressure).
- Let E be a set of edges, each connecting two points, $E = \{(\mathbf{q}_i, \mathbf{q}_j) \mid \mathbf{q}_i, \mathbf{q}_j \in Q, i \neq j\}$.
- Let F be a set of faces. A face is made up of an edge cycle, containing at least three edges. The cycle is defined as a path $((\mathbf{q}_0, \mathbf{q}_1), (\mathbf{q}_1, \mathbf{q}_2), \ldots, (\mathbf{q}_n, \mathbf{q}_0))$, where the start and endpoint of the path are identical and each other point is unique. All points $\mathbf{q}_i$ of a face lie on a common plane in 3D space, that is the face is planar.
- Let F_Γ be the subset of F containing all boundary faces. A boundary face is a face of which all points indexed by its edges lie in Γ.
- Let F_P be a set of subsets of faces called patches. $F_P = \{F_{P,i} \subset F_\Gamma\}$. Boundary conditions can be applied per patch. Thus the main use for patches is to group boundary faces, to which a boundary condition should be applied.
- Let F_Ω be the set of interior faces $F \setminus F_\Gamma$. An interior face includes at least one point which lies in Ω.
- Let C be a set of cells. A cell $c \in C$ is a space $c \subset \Omega$ which is enclosed by at least four faces without any gaps. Every edge of a cells face is shared by another face of that cell and potentially also by faces of other cells.
- Let the number of cells $|C|$ be denoted as n_C, the number of interior faces $|F_\Omega|$ as n_F, and the number of boundary faces $|F_\Gamma|$ as n_{F_Γ}.

Definition 1 (Volume centerpoint).
We define the centerpoint $\boldsymbol{q}_{C_v} \in \mathbb{R}^3$ of a cell $c \in C$ as the point (centroid) which coincides with the arithmetic mean of all points within the volume. For a formal definition see e.g. [Cox+69].

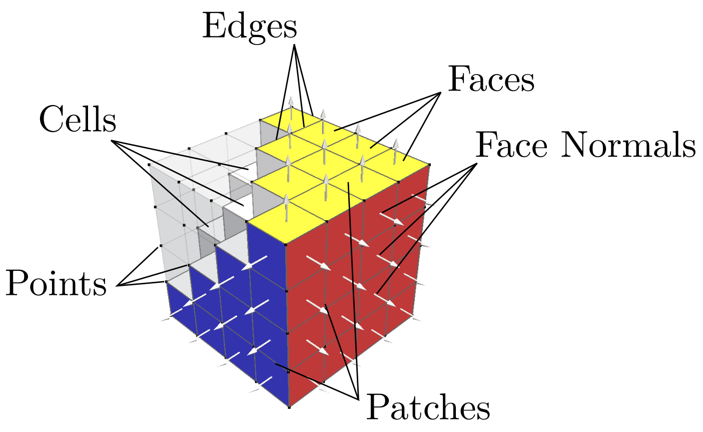


Figure 2.9: Unit cube meshed with 64 hexahedral elements. Some cells are made translucent to show the interior cells and faces. Three instances of points, edges, faces, face normals, patches, and cells are marked.

Definition 2 (Face centerpoint).
Similarly we define the centerpoint $\mathbf{q}_{C_f} \in \mathbb{R}^3$ *of a face* $f \in F$ *as the point (centroid) which coincides with the arithmetic mean of all points on the face. For a formal definition see e.g. [Cox+69].*

In practice the face centroid of complex shapes is computed by decomposing the face into triangles, calculating the areas and centroids of each triangle, and calculating the final centroid from the triangle centroids weighted by their area. Assuming uniform density throughout the domain, the face/volume centerpoints coincide with the center of mass.

Definition 3 (Face normal vector).
The face normal vector $\mathbf{n}_f \in \mathbb{R}^3$ *is defined as the vector, that is perpendicular to the plane containing all points of the planar face* $f \in F$*. By definition it points outside of the domain enclosed by the face, and is normalized to length one, that is* $\|\mathbf{n}_f\| = 1$*.*

Definition 4 (Face area vector).
The face area vector $\mathbf{s}_f \in \mathbb{R}^3$ *is defined as the product of the face normal vector with the face area* A_f*:* $\mathbf{s}_f = A_f \mathbf{n}_f$*, that is* $\|\mathbf{s}_f\| = A_f$*.*

The face area vector $\mathbf{s}$ can be used to efficiently calculate fluxes through a cell face.

Figure 2.9 demonstrates a very simple structured mesh of the unit cube. The points are spaced equidistantly in the Cartesian directions at a distance of $\frac{1}{4}$. Consequently the mesh consists of $5^3 = 125$ points. The points are connected by 300 edges to form 240 faces. Of those faces, $4 \cdot 4 \cdot 6 = 96$ are boundary faces, that can be naturally assigned to six boundary patches coinciding with the sides of the cube. Each face can only be part of one patch, however it can be assigned to an arbitrary patch, depending on the required boundary conditions (which are set on a per-patch level). Finally, the faces form $4^3 = 64$ hexahedral (cube) cells.

2.4.2 OpenFOAM Field Conventions

Table 2.1 lists the OpenFOAM fields corresponding to the most important physical quantities for flows. The correlation between the notation for physical quantity and the corresponding discretized field is straightforward for all fields, except the velocity. As the velocity $\mathbf{u} \in \mathbb{R}^3$ is already a vector quantity, we define the velocity field as $\mathbf{U} = (u_{0,x}, u_{0,y}, u_{0,z}, \ldots, u_{n_C-1,x}, u_{n_C-1,y}, u_{n_C-1,z}) \in \mathbb{R}^{3n_C}$. If the field of an individual velocity dimension is required, we denote it as $\mathbf{U}_x, \mathbf{U}_y, \mathbf{U}_z \in \mathbb{R}^{n_C}$ respectively. This notation allows to discern between physical quantity and the discretized field, and is also consistent with the OpenFOAM notation for the velocity field, which is `U`.

Table 2.1: Relation of physical quantities and their corresponding OpenFOAM fields for compressible and incompressible flows.

Physical Quantity	Symbol	Field	OpenFOAM field	Data type	Unit incompressible	Unit compressible
Velocity	$\mathbf{u}$	$\mathbf{U}$	`U`	`volVectorField`	m/s	m/s
Pressure	p	$\mathbf{p}$	`p`	`volScalarField`	$\mathrm{m^2/s^2}$	$\mathrm{kg\,m/s^2}$
Mass flux	ϕ	$\boldsymbol{\phi}$	`phi`	`surfaceScalarField`	$\mathrm{m^3/s}$	kg/s
Turb. kinetic energy	k	$\mathbf{k}$	`k`	`volScalarField`	$\mathrm{m^2/s^2}$	$\mathrm{m^2/s^2}$
Turb. dissipation rate	ϵ	$\boldsymbol{\epsilon}$	`epsilon`	`volScalarField`	$\mathrm{m^2/s^3}$	$\mathrm{m^2/s^3}$

For incompressible flows, OpenFOAM scales the pressure with the inverse of the constant density ρ, making the density disappear from the momentum equations (2.3). The unit of this scaled pressure, referred to as kinematic pressure, is consequently $\mathrm{m^2/s^2}$ instead of the more common $\mathrm{kg\,m/s^2}$. Also the face flux is calculated as $\mathbf{u} \cdot \mathbf{s}$ for the incompressible case while for the compressible case $\rho(\mathbf{u} \cdot \mathbf{s})$ is used.

Physical quantities in OpenFOAM have to be defined in a consistent unit system and arithmetic compatibility of different fields is checked at run-time. While different unit systems, such as imperial units are possible, SI units are most commonly used. To enable the run time check of units, the dimensions of a field has to be specified in its input dictionary by a vector specifying the cardinality of the individual units in a specific order. The order of the unit specification is shown in Table 2.2. Listing 2.3 gives an example for a velocity field with units length per time.

```
FoamFile{type volVectorField;}

dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (1 0 0);
```

Listing 2.3: Definition of a field storing vectors with unit *length over time*, i.e. velocities.

Table 2.2: Order of units in the OpenFOAM unit system and corresponding SI unit [OF18].

No.	Property	SI unit
1	Mass	Kilogram (kg)
2	Length	Metre (m)
3	Time	Second (s)
4	Temperature	Kelvin (K)
5	Quantity	Mole (mol)
6	Current	Ampere (A)
7	Luminous intensity	Candela (cd)

2.5 Continuous Adjoints

In the context of solving PDEs, the application of AD leads to a *discretize first — differentiate later* approach. That is the governing (primal) equations are first transformed from a continuous problem to a discrete one (defined on finitely small control volumes), optionally linearized and then solved. The derivatives of the governing equations are obtained by applying AD to the implementation of the discretization process. AD can be introduced at different levels of the discretization process, ranging from differentiating the whole discretization and solution process (discussed in Section 3.2.2) to only differentiating the calculation of residuals (see Section 4.5) and supplying additional analytical insight to obtain the full derivatives [Lot16]. We call this the discrete adjoint approach.

In contrast, the continuous adjoint approach yields a *differentiate first — discretize later* setting. That is from the primal PDEs a corresponding set of adjoint PDEs, along with adjoint boundary conditions, is derived symbolically (usually by variational calculus). The resulting adjoint PDEs are discretized and solved separately from the primal equations (however primal variables may appear in the adjoint equation, resulting in a coupling between primal and adjoint equations). Therefore, the discretization of the adjoint equations can be tailored to the physical properties of the adjoint, e.g. by employing upwinding of the adjoint convection equation. In the context of the adjoint Navier-Stokes equations, the adjoint convection direction is opposed to the primal (different sign, see derivation in following sections). A visual comparison of the different approaches to obtain the final sensitivities is given in Figure 2.10.

A drawback of the continuous adjoint method is that the obtained derivatives are not necessarily consistent to the primal, as implemented. For some PDEs, the adjoint equations are ill-conditioned, leading to bad convergence of the solution algorithms. Also the derivation process of the adjoint equations is complex and error prone. For complex, e.g. turbulence, equations, a closed symbolic derivation might not be possible at all [Car+10] without changes to the primal equations or boundary conditions [Kav+15].

While this thesis is focused on obtaining derivatives using AD, we will briefly introduce the continuous adjoint equations, specifically to obtain topology sensitivities. We will use these methods to verify the AD implementation in later sections. This introduction is based on the derivations in [Oth08].

The solution of the equations listed above, including topology optimization using fixed stepsize steepest descent, is implemented in the standard OpenFOAM solver `adjointShapeOptimization-`

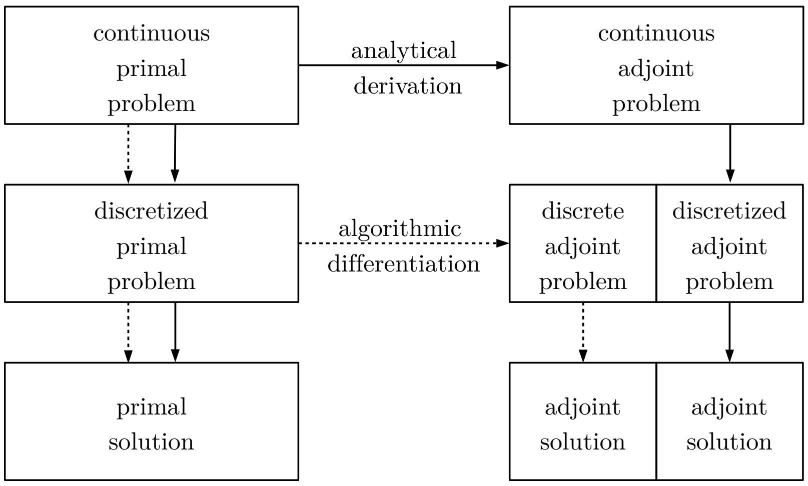

continuous primal problem
analytical derivation
continuous adjoint problem
discretized primal problem
algorithmic differentiation
discrete adjoint problem
discretized adjoint problem
primal solution
adjoint solution
adjoint solution

Figure 2.10: Comparison of solution procedure for the discrete and continuous adjoint. The continuous adjoint differentiates on the differential equation level and then discretizes and solves the obtained adjoint equations. In contrast the discrete adjoint discretizes the primal equations first and then obtains the adjoints by applying AD to its implementation. Solid arrows connect the building blocks of the continuous adjoint, dashed lines the ones of the discrete adjoint.

`Foam` [OVW07]. A comparison between the continuous and a corresponding discrete adjoint solver can be found in Section 4.3.3.

2.5.1 Derivation of the Topological Sensitivity

An optimization problem in CFD can be stated as the task to minimize an objective $\mathcal{J}$, e.g. pressure loss in a system, under the constraint that the physical laws of fluid flow are fulfilled.

Introducing the residual vector $\mathbf{r} = (r_1, r_2, r_3, r_4)$ as the residual of the Navier-Stokes momentum and mass conservation equations, we can state the (not yet discretized) optimization problem as:

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \mathcal{J}(\alpha, \mathbf{u}, p) \\ \text{subject to} \quad & \mathbf{r}(\alpha, \mathbf{u}, p) = 0 \,. \end{aligned} \tag{2.8}$$

In residual form and neglecting external body forces the Navier-Stokes equations read as

$$\begin{aligned} (r_1, r_2, r_3)^T &= (\mathbf{u} \otimes \boldsymbol{\nabla})\mathbf{u} + \boldsymbol{\nabla} p - \nu \boldsymbol{\nabla}^2 \mathbf{u} + \alpha \mathbf{u} \\ r_4 &= -\boldsymbol{\nabla} \cdot \mathbf{u} \,. \end{aligned}$$

Constraint optimization problems can be reformulated into algebraic equations without constraints by introducing Lagrange multipliers. Lagrange multipliers are additional unknown variables which are introduced for every constraint equation.

This allows to reformulate the general optimization problem $f(\mathbf{x})$ under constraint $g(\mathbf{x}) = 0$ from

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g(\mathbf{x}) = 0 \,. \end{aligned} \tag{2.9}$$

to the transformed problem

$$\Lambda(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda\left(g(\mathbf{x})\right) ,$$

by introducing the Lagrange multiplier λ.

A solution $(\mathbf{x}, \lambda)$, for which the derivatives of Λ w.r.t. both $\mathbf{x}$ and λ vanish, is a candidate for the solution of the constraint optimization problem (2.9)

$$\frac{\partial \Lambda(\mathbf{x}, \lambda)}{\partial \mathbf{x}} = 0$$
$$\frac{\partial \Lambda(\mathbf{x}, \lambda)}{\partial \lambda} = 0 .$$

By applying the Lagrangian multiplier approach to the constraint topology optimization problem (2.8) a modified cost function L is obtained.

$$\mathcal{L} := \mathcal{J} + \int_{\Omega} (\bar{\mathbf{u}}, \bar{p}) \cdot \mathbf{r} \, \mathrm{d}\Omega$$

Here the Lagrange multiplier is defined as $(\bar{\mathbf{u}}, \bar{p}) = (\bar{u}_x, \bar{u}_y, \bar{u}_z, \bar{p})$ and ensures that the residual $\mathbf{r}$ vanishes at each location inside the computational domain Ω. The multiplier $\bar{\mathbf{u}}$ is called adjoint velocity, the multiplier $\bar{p}$ adjoint pressure.

By applying variational calculus and assuming some restrictions on the feasible cost functions, the following relation for obtaining the desired sensitivities $\partial\mathcal{L}/\partial\boldsymbol{\alpha}$ can be found [Oth08]:

$$\frac{\partial \mathcal{L}}{\partial \alpha_i} = (\bar{\mathbf{u}}_i \cdot \mathbf{u}_i)\, V_i .$$

For this relation to hold, the Lagrangian multipliers must be specifically chosen, to ensure that the variations of $\mathcal{L}$ with respect to $\mathbf{u}$ and p vanish. The values for the adjoint multipliers $\bar{\mathbf{u}}$ and $\bar{p}$ can be found by solving the following additional PDEs:

$$-\left(\boldsymbol{\nabla}\bar{\mathbf{u}} + (\boldsymbol{\nabla}\bar{\mathbf{u}})^T\right)\mathbf{u} = \nu\boldsymbol{\nabla}^2\bar{\mathbf{u}} - \boldsymbol{\nabla}\bar{p} - \alpha\bar{\mathbf{u}}$$
$$\boldsymbol{\nabla} \cdot \bar{\mathbf{u}} = 0 .$$

Comparing to the primal Navier-Stokes equations, the convection of the adjoint velocity in the opposite direction of the primal flow can be clearly seen in the negative sign of the convective term $\left(\boldsymbol{\nabla}\bar{\mathbf{u}} + (\boldsymbol{\nabla}\bar{\mathbf{u}})^T\right)\mathbf{u}$. The equations are linear in $\bar{\mathbf{u}}$, however the matrix vector product (dot product on a per equation level) $\bar{\mathbf{u}}^T\mathbf{u}$ introduces mixed terms, which require an iterative solution if a segregated solver is used. That is, the convection is discretized as $\left(\boldsymbol{\nabla}\bar{\mathbf{u}}^{i+1} + \left(\boldsymbol{\nabla}\bar{\mathbf{u}}^i\right)^T\right)\mathbf{u}^i$.

2.5.2 Derivation of Adjoint Boundary Conditions

The boundary conditions for the continuous equations depend on the chosen cost function $\mathcal{J}$ and have to be individually derived for each desired cost function. The boundary conditions for the use with ducted flows and total power loss, as defined in (2.10), are given in Table 2.3.

$$\mathcal{J} = -\int_{\Gamma} \left(p + \frac{1}{2}\|\mathbf{u}\|^2\right) \mathbf{u} \cdot \mathbf{n} \, \mathrm{d}\Gamma \tag{2.10}$$

	Inlet	Wall	Outlet
$\mathbf{u}$	$u_n = u_{inlet}$	$u_n = 0,\ u_t = 0$	$\boldsymbol{\nabla}\mathbf{u} \cdot \mathbf{n} = 0$
p	$\boldsymbol{\nabla} p \cdot \mathbf{n} = 0$	$\boldsymbol{\nabla} p \cdot \mathbf{n} = 0$	$p = p_{outlet}$
$\bar{\mathbf{u}}$	$\bar{u}_t = 0,\ \bar{u}_n = u_n$	$\bar{u}_t = 0,\ \bar{u}_n = 0$	$u_n(\bar{u}_t - u_t) + \nu(\mathbf{n} \cdot \Delta)\bar{u}_t = 0$
$\bar{p}$	$\boldsymbol{\nabla}\bar{p} \cdot \mathbf{n} = 0$	$\boldsymbol{\nabla}\bar{p} \cdot \mathbf{n} = 0$	$\bar{p} = \bar{\mathbf{u}} \cdot \mathbf{u} + \bar{u}_n u_n + \nu(\mathbf{n} \cdot \Delta)\bar{u}_n + -\frac{1}{2}u^2 - u_n^2$

Table 2.3: Primal and adjoint boundary conditions for inlet, walls, and outlet in ducted flows. Primal/adjoint velocities normal/tangential to the patches are denoted by $u_n = \mathbf{u} \cdot \mathbf{n}$, $u_t = \mathbf{u} \cdot \mathbf{t}$, $\bar{u}_n = \bar{\mathbf{u}} \cdot \mathbf{n}$, $\bar{u}_n = \bar{\mathbf{u}} \cdot \mathbf{t}$.

2.5.3 Continuous Shape Sensitivity

By making an assumption connecting the adjoints to shear forces, the adjoint momentum equations can also be used to derive a continuous optimization procedure for surface shapes [Oth08]. For the detailed derivation of those relations, consult [Oth08]. More advanced derivations for different cost functions, as well as turbulence models are found in a variety of publications by Giannakoglou et al., e.g. [Zym+09]. For a specific amount of outward facing movement in surface normal direction β, at an arbitrary point on the surface, the sensitivity can be computed as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\beta} = -A\nu(\mathbf{n} \cdot \boldsymbol{\nabla})\bar{u}_t \cdot (\mathbf{n} \cdot \boldsymbol{\nabla})u_t\,, \tag{2.11}$$

with A representing the surface area affected by the move of the surface by β. The shape sensitivities thus depend on the gradients in normal direction of the primal and adjoint velocities tangential to the walls.

Shape sensitivities give an indication which nodes have to be moved inward or outward respective to their face normals. A comparison between this approach and a discrete adjoint solver which directly uses the individual points of the mesh as parameters is shown in Section 4.4.

2.6 Sparse Matrix Storage

2.6.1 General Sparse Storage Schemes

In a wide variety of (technical) applications, the matrices obtained by the discretization of nonlinear PDEs are sparse. For FVM, the matrices are sparse, because the stencils used to approximate the spatial and temporal derivatives are only influenced by a limited number of values in the direct neighborhood of the derivation point. A matrix is called sparse if the number of non-zero elements is low, compared to the number of zeroes in the matrix. That is for a matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$:

$$\frac{n_{nz}}{mn} \ll 1\,,$$

where $n_{nz} = \|\{a_{ij} \,|\, a_{ij} \neq 0\}\|$ is the number of non-zero elements in the matrix A.

One distinguishes between structurally zero elements of a matrix and zero valued elements, with the former being a subset of the latter. For a matrix which is generated by some fixed algorithm (code), structurally zero elements are elements which are zero regardless of the values used to calculate them (as long as changes in the inputs do not induce changes in the control flow of the algorithm). Additional zero valued elements can be created through numerical operations such as multiplication by zero. The sparsity pattern, that is the positions of all structural non-zero elements, of Jacobians/Hessians can be obtained by the propagation of pattern sets [Var11]. The determination of sparsity patterns in OpenFOAM is presented in detail in Section 4.5.2.

Exploiting the sparsity of such matrices is beneficial for a multitude of reasons (in the following we assume square matrices, or at least $\mathcal{O}(m) = \mathcal{O}(n)$):

- Storage of the dense matrix is costly at $\mathcal{O}(n^2)$, it is not uncommon for the dense matrix to not fit into memory at all. With sparse storage schemes (see below) the memory can be reduced to $\mathcal{O}(n_{nz})$.
- The multiplication of a sparse matrix with a dense vector can be sped up from $\mathcal{O}(n^2)$ to $\mathcal{O}(n_{nz})$ by eliminating memory access and operations on zero entries.
- Direct solution of linear equation systems (e.g. using LU-decomposition) are sped up by exploiting sparsity.

Sparse matrix storage schemes aim to reduce the memory required to store the matrix coefficients, while retaining all information of the matrix and providing efficient access routines to the individual entries. Popular choices for sparse matrix storage include the compact row storage (CRS) scheme and the coordinate format.

We will illustrate both schemes using an illustrative example. Let matrix $A \in \mathbb{R}^{4\times 4}$ with $n_{nz} = 8$ individual (structural) non-zero matrix entries a_{ij} be defined as

$$A = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 \\ 0 & a_{11} & a_{12} & 0 \\ a_{20} & 0 & a_{22} & 0 \\ a_{30} & 0 & 0 & a_{33} \end{bmatrix} .$$

The CRS scheme stores the non-zero elements of the matrix in left-to-right and top-to-bottom order in vector $\boldsymbol{v}$ (row-wise storage). Consequently, entries of the same row are always adjacent. The corresponding column indices of the nonzero elements are stored in vector $\boldsymbol{c_I}$. The vector $\boldsymbol{r_I}$ holds for each row the position of the first non-zero entry in this row in the vectors $\boldsymbol{v}$ and $\boldsymbol{c_I}$. Thus, the memory requirement depends on the number of non-zeroes and the number of rows: $MEM = n_{nz} \cdot sizeof(value\_type) + (n_{nz} + m) \cdot sizeof(index\_type)$.

$$\begin{aligned} \boldsymbol{v} &= [a_{00}, a_{01}, a_{11}, a_{12}, a_{20}, a_{22}, a_{30}, a_{33}]^T \\ \boldsymbol{c_I} &= [0, 1, 1, 2, 0, 2, 0, 3]^T \\ \boldsymbol{r_I} &= [0, 2, 4, 6]^T \ . \end{aligned}$$

The coordinate format stores the non-zero elements in vector $\boldsymbol{v}$, the row indices of the nonzero elements in vector $\boldsymbol{r_I}$ and the column indices in $\boldsymbol{c_I}$. Consequently, the memory requirement is $n_{nz} \cdot sizeof(value\_type) + 2 \cdot n_{nz} \cdot sizeof(index\_type)$.

$$\begin{aligned} \boldsymbol{v} &= [a_{00}, a_{01}, a_{11}, a_{12}, a_{20}, a_{22}, a_{30}, a_{33}]^T \\ \boldsymbol{r_I} &= [0, 0, 1, 1, 2, 2, 3, 3]^T \\ \boldsymbol{c_I} &= [0, 1, 1, 2, 0, 2, 0, 3]^T \ . \end{aligned}$$

The coordinate format is not unique, in that the entries in $\boldsymbol{v}$ can be stored in arbitrary order. In implementation it is advisable to store the entries of $\boldsymbol{v}$ in a cache efficient order. I.e., if matrix vector products are required, $\boldsymbol{v}$ should be stored row-wise, such that subsequent entries can be efficiently read from cache.

For $n_{nz} > m$ (for practical applications usually $n_{nz} \gg m$), the CRS scheme occupies less memory than the coordinate format. The adjacency of the nonzero entries of a row and the explicit availability of the column indices makes the CRS format well suited for performing sparse matrix vector multiplications. However, it adds a layer of complexity to the retrieval of individual matrix entries. A matrix stored in CRS format can not be trivially transposed. A matrix stored in coordinate format can be transposed by simply switching the row and column vectors $\boldsymbol{r}_I$ and $\boldsymbol{c}_I$.

2.6.2 OpenFOAM LDU Format

OpenFOAM uses a variation of the coordinate format to store its matrix coefficients. The specifics of the storage scheme will become important when symbolic adjoints are applied to the embedded linear solvers. In the context of CFD simulation, the diagonal of the FVM discretization matrix is always dense, as the discretization using finite volumes always yields a central coefficient. Furthermore, the discretization matrix $A \in \mathbb{R}^{n_C \times n_C}$ is structurally symmetric and square. The diagonal coefficients of the matrix are stored in a dense vector $\boldsymbol{d} = [a_{ii} \mid 0 \leq i < n_C]$. A diagonal entry a_{ii} can be looked up directly by accessing d_i; thus no additional row and columns indices need to be stored. All other non-zero entries are stored in vectors $\boldsymbol{l}$ and $\boldsymbol{u}$, where $\boldsymbol{l} = [a_{ij} \mid a_{ij} \neq 0,\ i > j]$ are the coefficients below the diagonal and $\boldsymbol{u} = [a_{ij} \mid a_{ij} \neq 0,\ i < j]$ are the coefficients above the diagonal. In the implementation, the vectors $\boldsymbol{l}, \boldsymbol{d}$, and $\boldsymbol{u}$ can be obtained by calling the access functions `lower()`, `diag()` and `upper()` respectively. Note, that to resolve the symbol clash between velocity $\mathbf{u}$ and upper entries $\boldsymbol{u}$, all vectors corresponding to the LDU format are typeset in bold italics.

For symmetric matrices, only one of the vectors $\boldsymbol{l}$ and $\boldsymbol{u}$ needs to be stored. In addition to the non-zero values stored in $\boldsymbol{l}$ and/or $\boldsymbol{u}$, the row and columns indices of the matrix entries need to be stored. The indices are stored in addressing arrays $\boldsymbol{L} \in \mathbb{N}^{n_F}$ and $\boldsymbol{U} \in \mathbb{N}^{n_F}$. The indices are ordered, such that $\boldsymbol{L}$ is monotonously increasing and subsets of $\boldsymbol{U}$ with identical $\boldsymbol{L}$ are also monotonously increasing (see following example). This ensures good caching performance for the evaluation of matrix vector products. The addressing arrays can be obtained from the `lduMatrix` class by calling `lowerAddr()` and `upperAddr()` respectively.

Due to the structural symmetry of the finite volume discretization, entries of the lower part are given by

$$a_{U_i,L_i} = l_i\,,$$

and the entries of the upper part by

$$a_{L_i,U_i} = u_i\,.$$

As an example let

$$A = \begin{pmatrix} d_0 & u_0 & u_1 & 0 & 0 & 0 \\ l_0 & d_1 & u_2 & u_3 & 0 & 0 \\ l_1 & l_2 & d_2 & u_4 & u_5 & 0 \\ 0 & l_3 & l_4 & d_3 & u_6 & u_7 \\ 0 & 0 & l_5 & l_6 & d_4 & u_8 \\ 0 & 0 & 0 & l_7 & l_8 & d_5 \end{pmatrix}$$

be a 6×6 banded matrix storing 24 non-zero entries. Then the LDU format of A is given by

$$\begin{aligned}
\boldsymbol{l} &= \begin{bmatrix} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 \end{bmatrix} \\
\boldsymbol{d} &= \begin{bmatrix} d_0 & d_1 & d_2 & d_3 & d_4 & d_5 \end{bmatrix} \\
\boldsymbol{u} &= \begin{bmatrix} u_0 & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 & u_8 \end{bmatrix} \\
\boldsymbol{L} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 \end{bmatrix} \\
\boldsymbol{U} &= \begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5 \end{bmatrix} .
\end{aligned}$$

For the solution of linear systems arising from the FVM discretization, boundary conditions need to be applied to faces which only connect to one cell. The boundary coefficients are stored in two additional vectors $\boldsymbol{B}$ and $\boldsymbol{I}$, called boundary components and internal components. The coefficients in $\boldsymbol{I}$ are the coefficients which correspond to the influence of the boundary conditions onto the central coefficients of the matrix stored on the diagonal. In contrast to the $\boldsymbol{l}, \boldsymbol{d}$ and $\boldsymbol{u}$ coefficients, those coefficients are not necessarily identical for all dimensions of a `fvVectorMatrix`. The coefficients of $\boldsymbol{I}$ for the correct dimension are only added to $\boldsymbol{d}$ on demand, before calling the linear system solver, and removed again after the solver has finished. The coefficients in $\boldsymbol{B}$ can be imagined as virtual cells outside of the computational domain, which arise from the boundary conditions. They are added to the right hand side of the equation system on demand.

For later reference, Listing 2.4 gives a truncated overview of the relevant `lduMatrix.H` and `lduAddressing.H` header files.

```
class lduMatrix {
private:
  //- LDU mesh reference
  const lduMesh& lduMesh_;
  //- Coefficients (not including interfaces)
  scalarField *lowerPtr_, *diagPtr_, *upperPtr_;
public:
  //- Abstract base-class for lduMatrix solvers
  class solver {
  protected:
    const FieldField<Field, scalar>& interfaceBouCoeffs_;
    const FieldField<Field, scalar>& interfaceIntCoeffs_;
    lduInterfaceFieldPtrsList interfaces_;
    [...]
  public:
    const FieldField<Field, scalar>& interfaceBouCoeffs() const;
    const FieldField<Field, scalar>& interfaceIntCoeffs() const;
    const lduInterfaceFieldPtrsList& interfaces() const;
    [...]
  };

  //- Return the LDU addressing, access to coefficients
  const lduAddressing& lduAddr() const;
  scalarField& lower();
  scalarField& diag();
  scalarField& upper();

  bool hasDiag() const;
  bool hasUpper() const;
  bool hasLower() const;
  bool diagonal() const;
  bool symmetric() const;

  //- Init the update of interfaced interfaces for matrix operations
  void initMatrixInterfaces([...]) const;

  //- Update interfaced interfaces for matrix operations
  void updateMatrixInterfaces([...]) const;
  [...]
};

class fvMeshLduAddressing : public lduAddressing {
private:
  labelList::subList lowerAddr_;
  const labelList& upperAddr_;
public:
  //- Return lower addressing (i.e. lower label = upper triangle)
  const labelUList& lowerAddr() const;

  //- Return upper addressing (i.e. upper label)
  const labelUList& upperAddr() const;
};
```

Listing 2.4: Truncated LDU description of class `lduMatrix`.

2.7 Differentiation of Computer Programs

In this section the fundamental methods to obtain derivatives of arbitrary numerical code are presented. After the derivation of finite differences, first- and higher-order models of AD are introduced.

2.7.1 Finite Differences

Finite differences (FD) are a popular method to approximate the derivatives of uni- or multivariate functions. FD introduces approximation errors and, when implemented with floating-point precision, also additional numerical truncation and rounding errors.

The derivative of a continuous function $f : \mathbb{R} \to \mathbb{R}$ at the location x_0 is defined as

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) = \lim_{h\to 0} \frac{f(x_0 + h) - f(x_0)}{h} \,.$$

If this limit exists, the function is called differentiable at the location x_0. For finite values of h, this definition can be used to approximate the derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \,. \tag{2.12}$$

This finite difference is called forward difference. Analogously the backward difference is defined as

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h} \,.$$

Both approximations introduce an approximation error of $\mathcal{O}(h)$, as will be shown in Theorem 1. A more accurate approximation, scaling with $\mathcal{O}(h^2)$, can be found at the cost of one additional function evaluation (assuming that $f(x_0)$ is already known and therefore evaluation at x_0 does not incur additional cost).

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h} \,. \tag{2.13}$$

The accuracy of the FD schemes can easily be proven using Taylor expansion, as shown in the following three proofs. The convergence properties will be used in a later section to verify the AD implementation.

Theorem 1 (Onesided FD).
Let f be a function $\mathbb{R} \to \mathbb{R}$ which is differentiable at all points in the interval $[x, x_0]$. Then the approximation error of the forward difference (2.12) *scales with $\mathcal{O}(h)$ for $h \to 0$.*

Proof. The Taylor expansion of function f at $x = x_0 + h$ truncated after the third term of the infinite sum evaluates to

$$f(x_0 + h) = f(x_0) + h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^3) \,.$$

Subtracting $f(x_0)$ on both sides of the equation and dividing by h gives

$$\frac{f(x_0 + h) - f(x_0)}{h} = \frac{\mathrm{d}f}{\mathrm{d}x} + \frac{h}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^2)$$

$$\Leftrightarrow \frac{\mathrm{d}f}{\mathrm{d}x}(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h) \,.$$

$\square$

The proof for the backwards difference directly follows from the Taylor expansion

$$f(x_0 - h) = f(x_0) - h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^3)\,.$$

Theorem 2 (Central FD).
Let f be a function $\mathbb{R} \to \mathbb{R}$ which is differentiable at all points in the interval $[x_0 - h, x_0 + h]$. Then the approximation error of the central difference (2.13) *scales with $\mathcal{O}(h^2)$ for $h \to 0$.*

Proof. The Taylor expansions of function f at $x = x_0 + h$ and $x = x_0 - h$, truncated after the third term of the sum evaluate to

$$f(x_0 + h) = f(x_0) + h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^3)$$
$$f(x_0 - h) = f(x_0) - h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^3)\,.$$

Subtracting the second equation from the first, the second order term vanishes and leads to the desired approximation:

$$f(x_0 + h) - f(x_0 - h) = 2h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \mathcal{O}(h^3)$$

$$\Leftrightarrow \frac{\mathrm{d}f}{\mathrm{d}x}(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + \mathcal{O}(h^2)\,.$$

□

The definition of the finite difference can be straightforwardly extended to the multivariate case $f : \mathbb{R}^n \to \mathbb{R}$, giving approximations for the i-th directional derivative as

$$\frac{\mathrm{d}f}{\mathrm{d}x_i}(\mathbf{x}_0) = \frac{f(\mathbf{x}_0 + h\mathbf{e}_i) - f(\mathbf{x}_0)}{h} + \mathcal{O}(h)$$
$$\frac{\mathrm{d}f}{\mathrm{d}x_i}(\mathbf{x}_0) = \frac{f(\mathbf{x}_0) - f(\mathbf{x}_0 - h\mathbf{e}_i)}{h} + \mathcal{O}(h)$$
$$\frac{\mathrm{d}f}{\mathrm{d}x_i}(\mathbf{x}_0) = \frac{f(\mathbf{x}_0 + h\mathbf{e}_i) - f(\mathbf{x}_0 - h\mathbf{e}_i)}{2h} + \mathcal{O}(h^2)\,.$$

FD can be applied to obtain second and higher order derivatives, by eliminating the first order derivative term from the Taylor expansion. This can be interpreted as reapplying the first order FD model to both evaluation points of the first order central difference.

Theorem 3 (Second order FD).
The second derivative f of a univariate function $f(x) : \mathbb{R} \to \mathbb{R}$, which is twice differentiable at all points in the interval $[x_0 - h, x_0 + h]$ can be approximated as

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) \approx \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2}\,.$$

The approximation error scales with $\mathcal{O}(h^2)$.

Proof. As before the Taylor series of $f(x_0+h)$ and $f(x_0-h)$ are

$$\begin{aligned} f(x_0+h) &= f(x_0) + h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \frac{h^3}{6}\frac{\mathrm{d}^3 f}{\mathrm{d}x^3} + \mathcal{O}(h^4) \\ f(x_0-h) &= f(x_0) - h\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) + \frac{h^2}{2}\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) - \frac{h^3}{6}\frac{\mathrm{d}^3 f}{\mathrm{d}x^3} + \mathcal{O}(h^4)\,. \end{aligned}$$

Adding both equations and subtracting $2f(x_0)$ from both sides yields the desired result:

$$f(x_0+h) + f(x_0-h) - 2f(x_0) = h^2\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) + \mathcal{O}(h^4)$$

$$\Leftrightarrow \quad \frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x_0) = \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2} + \mathcal{O}(h^2)\,.$$

□

FD allows the evaluation of the Jacobian of a multivariate function $f : R^n \to \mathbb{R}^m$ at cost $\mathcal{O}(n) \cdot$ *cost*(f), irrespective of the output dimension m. The truncation error can introduce numerical noise. The stepsize h needs to be tuned, such that it is low enough to give a reasonable approximation, but high enough not to encounter numerical issues due to machine precision. This is challenging, especially for multivariate functions with partial derivatives which differ in order of magnitude. Advantages are the comparatively simple implementation and the ability to differentiate complex models, to which the source code must not necessarily be available.

2.7.2 Algorithmic Differentiation

Algorithmic Differentiation [GW08; Nau12] (AD), sometimes also called Automatic Differentiation [Bar+00], names the process of generating derivatives of a given (numerical) computer program, calculating the sensitivity of one or several outputs w.r.t. a set of inputs. Conceptually AD implements the evaluation of the chain rule on the sequence of operations connecting the inputs to the outputs.

Let $f(\mathbf{x})$ be a function which maps a vector to a scalar $f : \mathbb{R}^n \to \mathbb{R}$, and which is at least twice continuously differentiable (C^2). Then the gradient $\boldsymbol{\nabla} f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^n$ of that function is defined as

$$\boldsymbol{\nabla} f = \sum_{i=0}^{n-1} \frac{\partial y}{\partial x_i}\mathbf{e}_i = \left(\frac{\partial y}{\partial x_0}, \ldots, \frac{\partial y}{\partial x_{n-1}}\right)^T ,$$

where $\mathbf{e}_i$ denotes the i-th Cartesian unit vector.

The Hessian of f is given by the symmetric matrix $H \in \mathbb{R}^{n\times n}$ of second order partial derivatives, where each entry h_{ij} is given by:

$$H|_{ij} = h_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}\,.$$

Let $g(\mathbf{x}) \in C^1$ be a different function which maps from a vector to a vector $g : \mathbb{R}^n \to \mathbb{R}^m$. Then the Jacobian J of g is a $m \times n$ matrix where each entry is given by:

$$J|_{ij} = \frac{\partial g_i}{\partial x_j}\,.$$

As can be seen from the above definition, the gradient is the special case of a single column Jacobian.

AD allows to evaluate the derivatives of functions implemented as computer programs with machine precision accuracy. First order AD assumes that the function is differentiable at least once at all points of interest. AD relies on the fact, that each computer program can be decomposed at run time into a *single assignment code* (SAC), which we define below.

Definition 5 (SAC).
Each computer program implementing numerical functions can be decomposed into a sequence of elemental functions φ_j and assignments, mapping n independent inputs to m dependent outputs with p intermediate variables:

$$\begin{aligned}&\textit{for } j = n, \dots, n+p+m-1 :\\ &\quad v_j = \varphi_j(v_i)_{i \prec j},\end{aligned}$$

where $i \prec j$ denotes a direct dependence of the variable v_j on v_i. The result of each elemental function *φ_j is assigned to a unique auxiliary variable v_j. The n* independent inputs *$x_i = v_i$, for $i = 0, \dots, n-1$, are mapped onto m* dependent outputs *$y_j = v_{n+p+j}$, for $j = 0, \dots, m-1$. The values of p* intermediate variables *v_k are computed for $k = n, \dots, n+p-1$. If not otherwise specified we restrict the functions admittable as* elemental functions *to unary or binary functions, limiting the number of arguments for each elemental function to at most two.*

Definition 6 (DAG).
A directed acyclic graph $G = (V, E)$ is a directed graph which contains no cycles, i.e. there is no directed path starting and terminating at the same node [TS11].

A SAC can be conveniently represented as a DAG, where the $n+p+m$ nodes are the inputs, output, and intermediate variables, uniquely defined by the elemental functions. The edges model the dependence of the elemental functions on intermediate values and the inputs:

$$(v_i, v_j) \in E \quad \Leftrightarrow \quad \frac{\partial v_j}{\partial v_i} \neq 0 \,.$$

We label the edges with the partial derivatives, to aid the calculation of the full derivatives by executing the chain rule on the paths in the graph. An example program, its transformation to a SAC, and the corresponding DAG are shown in Figure 2.11.

Tangent Mode of AD

In this and the following definitions for AD models the notation from [Nau12] is used. To declutter the indices, later also the notations from [GW08] are employed for first order derivative models.

Definition 7 (First order tangent model).
Let $f : \mathbb{R}^n \to \mathbb{R}^m$ with $\boldsymbol{y} = f(\boldsymbol{x})$, $\boldsymbol{x}, \boldsymbol{x}^{(1)} \in \mathbb{R}^n$ and $\boldsymbol{y}, \boldsymbol{y}^{(1)} \in \mathbb{R}^m$. Then the first order tangent model *$f^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m$ of f calculates the primal as well as the Jacobian, evaluated in direction $\boldsymbol{x}^{(1)}$:*

$$\begin{pmatrix} \boldsymbol{y}^{(1)} \\ \boldsymbol{y} \end{pmatrix} = f^{(1)}\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}\right) = \begin{pmatrix} \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)} \\ f(\boldsymbol{x}) \end{pmatrix} .$$

The superscript $\bullet^{(1)}$ denotes the first tangent direction. The motivation for this notation will become apparent when higher derivatives are introduced. To make notation more compact, and to broaden compatibility to existing literature [GW08], for first order tangents we define the following equivalent notation:

$$\dot{\bullet} := \bullet^{(1)} .$$

With the gradient in direction of a tangent $\dot{\mathbf{x}}$, the full Jacobian can be assembled at cost $\mathcal{O}(n) \cdot \mathit{cost}(f)$, by evaluating the tangent model with all unit vectors $\dot{\mathbf{x}} = \mathbf{e}_i \in \mathbb{R}^n$, giving one column of the Jacobian for each evaluation. Here $\mathcal{O}(n)$ includes the overhead of the tangent model evaluation $\dot{f}$, compared to a passive primal evaluation of f.

The tangent model is now applied to each assignment of the SAC. Assuming differentiability of all elemental functions at their respective evaluation points, the tangent model of AD augments each elemental assignment with its tangent as follows:

$$\begin{aligned} &\text{for } j = n, \dots, n+p+m-1 \\ &\dot{v}_j = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot \dot{v}_i \\ &v_j = \varphi_j(v_i)_{i \prec j} . \end{aligned} \tag{2.14}$$

Here the variables $\dot{\mathbf{v}}$ are the tangents associated with the primal values $\mathbf{v}$. The directional derivatives are evaluated alongside the primal elemental functions, propagating them from the inputs to the outputs.

As an illustration the tangent model for the function

$$y = (x_0 \cdot x_1)(1 + x_2)$$

is shown in Figure 2.11. Transforming this function to a SAC gives the following sets of inputs, output, and auxiliary variables: $n = \|\{v_0, v_1\}\| = 2$, $m = \|\{v_4\}\| = 1$, $p = \|\{v_2, v_3\}\| = 2$. This SAC can alternatively be represented as the DAG shown in the figure. An auxiliary variable t is inserted into the DAG. Its partial derivatives are defined, such that the tangent model

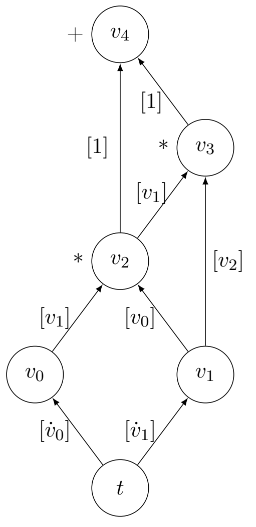
$$\dot{v}_4 = \frac{\mathrm{d}v_4}{\mathrm{d}t} = \boldsymbol{\nabla} v_4(v_0, v_1) \cdot [\dot{v}_0, \dot{v}_1]^T$$

is created by multiplying the partials along the paths connecting t to v_4. The SAC is transformed according to the rules of Equation 2.14, resulting in a program augmented by the tangent statements. The resulting calculation is shown on the right of the figure. It calculates the tangent $\dot{y} = \dot{v}_4$ as well as the primal result $y = v_4$.

Adjoint Mode of AD

Definition 8 (First order adjoint model).
Let $f : \mathbb{R}^n \to \mathbb{R}^m$ with $\boldsymbol{y} = f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{y} \in \mathbb{R}^m$ be an at least once differentiable function. Further let $\boldsymbol{x}_{(1)} \in \mathbb{R}^n$ and $\boldsymbol{y}_{(1)} \in \mathbb{R}^m$ be adjoint variables corresponding to the primal variables $\boldsymbol{x}, \boldsymbol{y}$. Then the first order adjoint model *$f_{(1)} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^m$ of f calculates the primal as well as the product of the transposed Jacobian with $\boldsymbol{y}_{(1)}$:*

$$\begin{pmatrix} \boldsymbol{x}_{(1)} \\ \boldsymbol{y} \end{pmatrix} = f_{(1)} \left(\boldsymbol{y}_{(1)}, \boldsymbol{x} \right) = \begin{pmatrix} \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot \boldsymbol{y}_{(1)} \\ f(\boldsymbol{x}) \end{pmatrix} .$$



(a) DAG

$$\begin{aligned} v_2 &= v_0 * v_1 \\ v_3 &= v_2 * v_1 \\ v_4 &= v_3 + v_2 \end{aligned}$$

(b) SAC

$$\begin{aligned} \dot{v}_2 &= v_1 * \dot{v}_0 + v_0 * \dot{v}_1 \\ v_2 &= v_0 * v_1 \\ \dot{v}_3 &= v_2 * \dot{v}_1 + v_1 * \dot{v}_2 \\ v_3 &= v_2 * v_1 \\ \dot{v}_4 &= 1 * \dot{v}_2 + 1 * \dot{v}_3 \\ v_4 &= v_3 + v_2 \end{aligned}$$

(c) Augmented SAC

Figure 2.11: Illustration of the tangent model for $y = (x_0 \cdot x_1)(1 + x_2)$, with DAG annotated with partial derivatives on the edges (left), primal SAC (middle), and SAC augmented to calculate the first order tangent model (right).

The full Jacobian can therefore be calculated at cost of $\mathcal{O}(m) \cdot \mathit{cost}(f)$ relative to the primal function evaluation. For the common case of a scalar output y, the gradient can be obtained at cost $\mathcal{O}(1) \cdot \mathit{cost}(f)$. Compared to the factor $\mathcal{O}(n) \cdot \mathit{cost}(f)$ of tangent mode this lower complexity is the prime motivating feature of the adjoint mode.

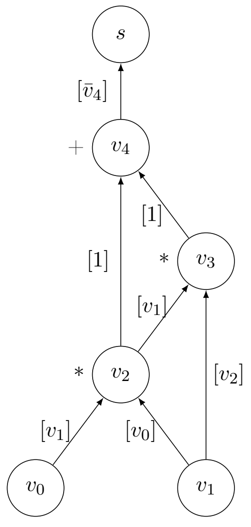
The subscript $\bullet_{(1)}$ denotes the first adjoint direction. Similar to the tangent mode we define the following equivalent notation for first order adjoint models:

$$\bar{\bullet} := \bullet_{(1)} \,.$$

Again, the adjoint model is now applied to each assignment of the SAC. In adjoint mode, a forward evaluation of the original program is succeeded by the propagation of adjoints for all v_i in reverse order, that is, for $i = n + p - 1, \ldots, 0$:

$$\left.\begin{aligned} \text{for } j =& n, \ldots, n+p+m-1 \\ v_j =& \varphi_j(v_i)_{i \prec j} \end{aligned}\right\} \text{forward section,}$$
$$\left.\begin{aligned} \text{for } i =& n+p-1, \ldots, 0 \\ \bar{v}_i =& \sum_{j : i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot \bar{v}_j \end{aligned}\right\} \text{reverse section.} \tag{2.15}$$

Here the variables $\bar{\mathbf{v}}$ are adjoint variables associated with the primal values $\mathbf{v}$. The adjoint sensitivities are evaluated only after the primal elemental functions have been evaluated, propagating them from the outputs back to the inputs. In practice the sum resulting in $\bar{v}_i$ is not evaluated all at once, but $\bar{v}_i$ is incremented by the incoming partial derivatives one at a time, motivating the incremental nature of adjoint code [GW08].



(a) DAG

$$\begin{aligned} v_2 &= v_0 * v_1 \\ v_3 &= v_2 * v_1 \\ v_4 &= v_3 + v_2 \end{aligned}$$

(b) SAC

$$\begin{aligned} & v_2 = v_0 * v_1 \\ & v_3 = v_2 * v_1 \\ & v_4 = v_3 + v_2 \\ & \bar{v}_3 = \bar{v}_4; && \bar{v}_2 = \bar{v}_4 \\ & \bar{v}_2 += v_1 * \bar{v}_3; && \bar{v}_1 = v_2 * \bar{v}_3 \\ & \bar{v}_0 = v_1 * \bar{v}_2; && \bar{v}_1 += v_0 * \bar{v}_2 \end{aligned}$$

(c) Augmented SAC

Figure 2.12: Illustration of the adjoint model for $y = (x_0 \cdot x_1)(1 + x_2)$. Left: DAG with adjoint extension s, annotated with partial derivatives on the edges. Center: SAC. Right: SAC augmented to compute the first order adjoint model.

Note that the v_j computed in the forward section are potentially required as arguments of local partial derivatives within the reverse section. They are read in reverse with respect to the original order of their evaluation. The additional persistent memory requirement of the adjoint code is $\mathcal{O}(n + p + m)$. This data flow reversal is the main challenge in adjoint AD. It is responsible for a naive implementation of AD typically not being applicable to large-scale numerical simulations. The available persistent memory may simply not be large enough [Nau12].

In Figure 2.12 we show the application of adjoint mode to the function already considered for tangent mode. An auxiliary variable s is inserted into the DAG. Its partial derivative is defined, such that the adjoint model

$$[\bar{v}_0, \bar{v}_1] = \frac{\mathrm{d}s}{\mathrm{d}\,[v_0, v_1]} = \boldsymbol{\nabla} v_4\,[v_0, v_1]^T \cdot \bar{v}_4$$

is created by multiplying the partials along the paths connecting $[\bar{v}_0, \bar{v}_1]$ to s. Note how the forward evaluation of the program is now spatially separated from the calculation of the adjoints. The adjoints are evaluated in reverse order, propagating the adjoints of the outputs $\bar{y} = \bar{v}_5$ back to the inputs $\bar{x}_0 = \bar{v}_0$ and $\bar{x}_1 = \bar{v}_1$. The intermediate v_2 is required to calculate $\bar{v}_1$ after the primal evaluation has already ended, highlighting the additional memory cost added by the adjoint method. As variables v_1 and v_2 have more than one outgoing edge, their adjoint value is influenced by multiple paths of the data flow reversal, motivating the incremental nature of the adjoint propagation. The auxiliary variable v_3 is not required in the reverse section as it is only used linearly by other assignments and thus vanishes when evaluating Equation 2.15.

Higher Order Derivative Models

Derivatives of second and higher order can be obtained by recursively applying the first order models onto models obtained by either tangent or adjoint mode. In the following, only the second order models are presented, as they can be used for a variety of optimization tasks, and can further be used to verify adjoints versus tangents, as presented in Section 4.4. For third and higher order models, please refer to literature, e.g. [Nau12].

Definition 9 (Tangent over Tangent Model).
Let $f : \mathbb{R}^n \to \mathbb{R}$ with $y = f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^n$, and $y \in \mathbb{R}$ be an at least twice differentiable function. Applying the first order tangent model to $f^{(1)}\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}\right)$ yields the second order tangent model $f^{(1,2)}\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1,2)}, \boldsymbol{x}\right) : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$, which calculates the following relations:

$$\begin{pmatrix} y^{(1,2)} \\ y^{(1)} \\ y^{(2)} \\ y \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}^{(1)^T} \cdot \boldsymbol{\nabla}^2 f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(2)} + \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1,2)} \\ \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)} \\ \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(2)} \\ f(\boldsymbol{x}) \end{pmatrix}.$$

One entry h_{ij} of the Hessian can thus be obtained from $y^{(1,2)}$ by seeding $\mathbf{x}_{(1)} = e_i$, $\mathbf{x}_{(2)} = e_j$, and $\mathbf{x}_{(1,2)} = 0$. The full Hessian can be obtained at cost $n(n+1)/2 \cdot \mathit{cost}\left(f^{(1,2)}\right)$ by exploiting the symmetry of the Hessian. For sparse matrices, the cost can be further lowered by coloring approaches (see Section 2.11).

Definition 10 (Tangent over Adjoint Model).
Applying the first order adjoint model to $f_{(1)}\left(y_{(1)}, \boldsymbol{x}\right)$ yields the second order tangent over adjoint model $f_{(1)}^{(2)}\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}, y_{(1)}, y_{(1)}^{(2)}\right) : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}$, which calculates the following relations:

$$\begin{pmatrix} \boldsymbol{x}_{(1)}^{(2)} \\ \boldsymbol{x}_{(1)} \\ y^{(2)} \\ y \end{pmatrix} = \begin{pmatrix} y_{(1)} \cdot \boldsymbol{\nabla}^2 f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(2)} + \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y_{(1)}^{(2)} \\ \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y_{(1)} \\ \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(2)} \\ f(\boldsymbol{x}) \end{pmatrix}.$$

The i-th row/column of the Hessian can thus be obtained from $\mathbf{x}_{(1)}^{(2)}$ by seeding $\mathbf{x}^{(2)} = e_i$, $y_{(1)} = 1$ and $y_{(1)}^{(2)} = 0$. The full Hessian can be obtained at cost $n \cdot \mathit{cost}\left(f_{(1)}^{(2)}\right)$. Again the number of evaluations of $f_{(1)}^{(2)}$ can be improved by coloring, seeding multiple directions of $\mathbf{x}^{(2)}$ at once.

Definition 11 (Adjoint over Adjoint Model).
Applying the first order adjoint model to $f_{(1)}\left(y_{(1)}, \boldsymbol{x}\right)$ yields the second order adjoint over adjoint model $f_{(1,2)}\left(\boldsymbol{x}_{(1,2)}, \boldsymbol{x}, y_{(1)}, y_{(2)}\right) : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}$, which calculates the following relations:

$$\begin{pmatrix} y_{(1,2)} \\ \boldsymbol{x}_{(1)} \\ \boldsymbol{x}_{(2)} \\ y \end{pmatrix} = \begin{pmatrix} \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}_{(1,2)} \\ \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y_{(1)} \\ y_{(1)} \cdot \boldsymbol{\nabla}^2 f(\boldsymbol{x}) \cdot \boldsymbol{x}_{(1,2)} + \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y_{(2)} \\ f(\boldsymbol{x}) \end{pmatrix}.$$

The i-th row/column of the Hessian can be obtained from $\mathbf{x}_{(2)}$ by seeding $\mathbf{x}_{(1)} = e_i$, $y_{(1)} = 1$, and $y_{(2)} = 0$. The full Hessian can be obtained at cost $n \cdot \mathit{cost}\left(f_{(1,2)}\right)$. The cost is thus identical to the tangent over adjoint model, and no further gain in complexity is realized by applying the adjoint model twice. In practice the choice of which adjoint model to use depends on the implementation of the AD tool and caching considerations on the executing machine [Lot16].

Definition 12 (Adjoint over Tangent Model).
Applying the first order adjoint model to $f^{(1)}\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}\right)$ *yields the second order adjoint over tangent model* $f^{(1)}_{(2)}\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}, y_{(1)}, y^{(2)}_{(1)}\right) : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}$, *which calculates the following relations:*

$$\begin{pmatrix} \boldsymbol{x}_{(2)} \\ \boldsymbol{x}_{(1,2)} \\ y^{(1)} \\ y \end{pmatrix} = \begin{pmatrix} y^{(1)}_{(2)} \cdot \boldsymbol{\nabla}^2 f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)} + \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y_{(2)} \\ \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot y^{(1)}_{(2)} \\ \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)} \\ f(\boldsymbol{x}) \end{pmatrix}.$$

The i-th row/column of the Hessian can thus be obtained from $\mathbf{x}_{(2)}$ by seeding $\mathbf{x}^{(1)} = \mathbf{e}_i$, $y^{(1)}_{(2)} = 1$, and $y_{(2)} = 0$. The full Hessian can be obtained at cost $n \cdot \mathit{cost}\left(f^{(1)}_{(2)}\right)$.

The adjoint over tangent model is rarely used in practice, due to the increased storage costs required for the backward propagation compared to the tangent over adjoint model.

2.7.3 Tool Driven Derivative Generation

Broadly speaking AD tools follow one of two approaches, *source code transformation* or *operator overloading*. Both approaches are briefly presented below.

Source Code Transformation

Source code transformation tools parse the primal source code and transform it by applying the tangent or adjoint models, Equations (2.14) or (2.15), onto the individual statements, producing a new source code which can then be compiled/executed and optimized by a regular compiler/interpreter. The generated source code looks and performs much like a code differentiated statement by statement by hand, but allows greater flexibility and more rapid code development.

The source code transformation approach requires that the tool has access, and is able to parse, the source code. This makes the code transformation of programs written in complex, object oriented languages like `C++` challenging. Especially templated codes, where substantial parts of the code are only instantiated at compile time, are not well suited to differentiation with source code transformation tools. Hybrid approaches are possible. For example, source code transformation can be applied to numerical kernels that are limited to `C` syntax and some subset of the `C++` standard. The remaining program wrapping the numerical kernels can then be differentiated by operator overloading. The resulting partial derivatives of the different code blocks can then be multiplied together using the chain rule.

General purpose AD tools which use the source code transformation approach include TAPENADE [HP13] (C and Fortran), dcc [För14] (C) and Tangent [Goo17] (python). Furthermore domain specific tools exist, e.g. DolfinAdjoint [Far+13] for the Dolfin [LW10] / FEniCS [Aln+15] packages for solving differential equations using FEM.

```cpp
#include <iostream>

struct ADtype{
  double v; // value component
  double t; // tangent component
  ADtype(const double& v, const double& t = 0.0) : v(v),t(t) {};
};

ADtype operator*(const ADtype& x1,const ADtype& x2){
  return ADtype(x1.v*x2.v, x1.v*x2.t + x1.t*x2.v);
}
ADtype operator+(const ADtype& x1,const ADtype& x2){
  return ADtype(x1.v+x2.v, x1.t+x2.t);
}

int main(){
  ADtype x = 2;
  x.t=1; // seeding
  ADtype y = x*x+x;
  std::cout << "v: " << y.v << " t: " << y.t << std::endl; // "v: 6 t: 5"
}
```

Listing 2.5: Implementation of a basic operator overloading tool. A custom type calculates tangents of programs containing additions and multiplications. Application is demonstrated by calculating the tangent $\dot{y} = 2x + 1 = 6$ of $y = x \cdot x + x$ at location $x = 2$.

Operator Overloading

To circumvent the issues faced by the source code transformation approach, the operator overloading approach uses the operator overloading features present in many modern programming languages. We will focus on the implementation, as it applies to `C++`. Operator overloading allows to replace the intrinsic implementations of the basic numerical operators $(+, -, \ldots)$ and functions (`sin`, `exp`, `pow`), implementing custom behavior. An AD tool implements one or multiple custom data types that are used to replace the floating point data types of the primal calculation. The custom data type implements the necessary operators to, in addition to the primal calculation, propagate tangents/adjoints through all (unary or binary) elemental functions occurring in the code.

As an illustration of this concept a very basic `C++` operator overloading tool, that is able to calculate tangents of programs containing assignments, additions, and multiplications is shown in Listing 2.5. It is applied to the expression $y = x \cdot x + x$.

Operator overloading tools are able to, with minor exceptions, cover the whole language standard of `C++` and are thus applicable to heavily templated code bases. Codes treated by those tools require only minimal code changes (some of which are mentioned in Section 3.2.3) and are by definition always up to date with the primal code base, as the instructions calculating the derivatives are generated at compile and run time alongside the primal.

Operator overloading introduces a run time penalty, which is more pronounced in some languages than in others. For compiled languages, most of the overhead can be offset by compile time optimization, most significantly function inlining. The memory overhead of the adjoint is generally more pronounced than for codes generated with source code transformation, as a representation of the SAC has to be stored at run time. The performance and memory consumption can be improved by (statement level) preaccumulation, briefly discussed in the next Section.

Operator overloading tools include ADOL-C [WG12] (C,C++), CoDi-Pack [SAG17] (C++), dco/c++ [LLN16] and AdiMat [BBV06] (Matlab).

Novel approaches are currently in development, further reducing the run time overhead by exploiting advanced templating features of C++, at the cost of requiring more code alterations [Lep+17]. This allows to extend preaccumulation from the statement level to whole code blocks.

2.8 Introduction to dco/c++

In this section we will give a brief overview of the architecture and interface of the operator overloading AD tool dco/c++ [LLN16]. Brief example drivers, usable to obtain gradients, Jacobians, and Hessians are given.

2.8.1 Scalar and vector tangent mode

The AD tool dco/c++ implements a generalized tangent scalar data type

```
template<class T> dco::gt1s<T>::type;
```

which carries a tangent component of type T alongside the value component of the same type. For a first order tangent model, this type is instantiated with a floating type data type (i.e. float or double). It can be initialized with a zero tangent by assigning a primal value.

```
dco::gt1s<double>::type x = 42;
```

The tangent type carries no further data; thus sizeof(dco::gt1s<T>::type) == 2*sizeof(T).

The derivative components of all dco/c++ types can be accessed by the interface routine dco::derivative(). Here it returns a reference to the tangent component of the passed variable.

```
dco::gt1s<double>::type x;
double t = dco::derivative(x); // const access
dco::derivative(x) = 1.0;      // non const access
```

The value component of a dco/c++ type can be accessed by the dco::value() routine. This allows to alter the value without changing the tangent/adjoint and to convert a dco/c++ type to a passive type, discarding its derivative information.

```
dco::gt1s<double>::type x;
double v = dco::value(x); // const access to value component
dco::value(x) = 42.0;     // non const access, tangent unmodified
x = 42.0;                 // overwrites tangent with 0.0!
```

Complementing the tangent scalar type gt1s<T>::type dco/c++ also defines a vector type gt1v<T,d>::type. This vector type carries a vector of tangents $\mathbf{t} \in \mathbb{R}^d$ alongside the primal value. The tangent model is applied to each vector entry individually, allowing to evaluate d seed directions at once. The vector mode reduces the number of elemental function evaluations.

```
gt1v<double,d> sin(const gt1v<double,d>& x){
  gt1v<double,d> y;
  dco::value(y) = sin(x);
  double partial = cos(dco::value(x));
  for(int i=0; i<d; i++)
    dco::derivative(y)[i] = partial*dco::derivative[i];
  return y;
}
```

Listing 2.6: Implementation of sin operation for tangent vectors of length d.

To evaluate n seed directions, instead of n calls to the scalar tangent model, only $\lceil n/d \rceil$ calls to the vector tangent model are needed. Further, it improves floating point performance, due to the increased cache locality introduced by the loop evaluating the individual tangents. The benefit due to caching varies from code to code. For maximum performance, the vector size is fixed at compile time and implemented as a plain array. A sensible vector size is e.g. 16. Tangent vectors which are too long consume excessive amounts of memory, and may decrease cache efficiency due to the vector not fully fitting into one cache line. The size of a vector type in memory is `sizeof(dco::gt1v<T>::type) == (d+1)*sizeof(T)`.

Accessing the values of the tangents follows the same interface as the scalar tangent type, with the difference that `dco::derivative` returns a reference to the tangent vector, from which the desired element can be accessed using the usual vector operations.

```
dco::gt1v<double>::type x = 42; // init tangents to zero
double t = dco::derivative(x)[0]; // const access
dco::derivative(x)[1] = 1.0;      // non const access
```

An exemplary implementation of the sin operation, using the interface introduced above, is shown in Listing 2.6.

The general purpose driver shown in Listing 2.7 calculates the gradient of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, assumed to be implemented (externally) as `T f (std::vector<T> x)`, requiring n calls to the function `f`.

To find a sensible vector length d, the driver is tested for a problem size of $n = 2^{13}$ and tangent vector sizes ranging from 2 to 1024. The functions considered are

$$y = \sum_{i=0}^{n-1} x_i \cdot x_i \quad \text{and} \quad y = \sum_{i=0}^{n-1} \sin x \,.$$

The results are shown in Figure 2.13. The results shown for $d = 1$ are obtained by the scalar tangent types and the curves are normalized, such that the run time of scalar execution equals one. The products $x_i \cdot x_i$ are computed cheaply. Thus, the run time is dominated by memory lookup of x_i, and caching effects are clearly observable. Here the optimal tangent vector size is 32, giving a speed up of approximately factor 2.5, compared to the scalar tangent version. Trigonometric functions are much more expensive to compute (here by a factor of roughly 25), therefore the saved calls of the sin function for the primal and cos function for the partial derivative have a bigger impact. Again, a vector size of 32 is a good choice, however the minimum time is achieved by choosing a vector size of 256, resulting in a speed up factor of 30.

```cpp
#include "dco_cpp_dev/src/dco.hpp"
#include <vector>
using namespace std;

typedef dco::gt1s<double>::type t1s_type;
const int vector_size = 16;
typedef dco::gt1v<double,vector_size>::type t1v_type;

template<typename T>
T f(vector<T> x);

// scalar driver
vector<double> calc_grad_f_t1s(const vector<double>& xd){
  const int n = xd.size();
  vector<double> grad(n);
  vector<t1s_type> x(xd.begin(),xd.end()); // copy passive values
  for(int i=0;i<n;i++){
    dco::derivative(x[i]) = 1.0; // seed i-th unit vector
    t1s_type y = f(x); // augmented primal calculation
    grad[i] = dco::derivative(y);
    dco::derivative(x[i]) = 0.0; // reset seed vector to 0
  }
  return grad;
}

// vector driver with vector size 16
vector<double> calc_grad_f_t1v(const vector<double>& xd){
  const int n = xd.size();
  const int d = vector_size;
  vector<double> grad(n);
  vector<t1v_type> x(xd.begin(),xd.end()); // copy passive values
  for(int i = 0; i < ceil(n/d); i++){
    // increment through local and global indices
    for(int j=i*d; j < min((i+1)*d,n); j++)
      dco::derivative(x[j])[j%d] = 1.0; // seed g-th unit vector
    t1v_type y = f(x); // augmented primal calculation
    for(int j=i*d; j < min((i+1)*d,n); j++){
      grad[j] = dco::derivative(y)[j%d]; // extract tangents
      dco::derivative(x[j])[j%d] = 0.0; // reset seed vectors to 0
    }
  }
  return grad;
}
```

Listing 2.7: Driver calculating the full gradient of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ using tangent mode of AD.

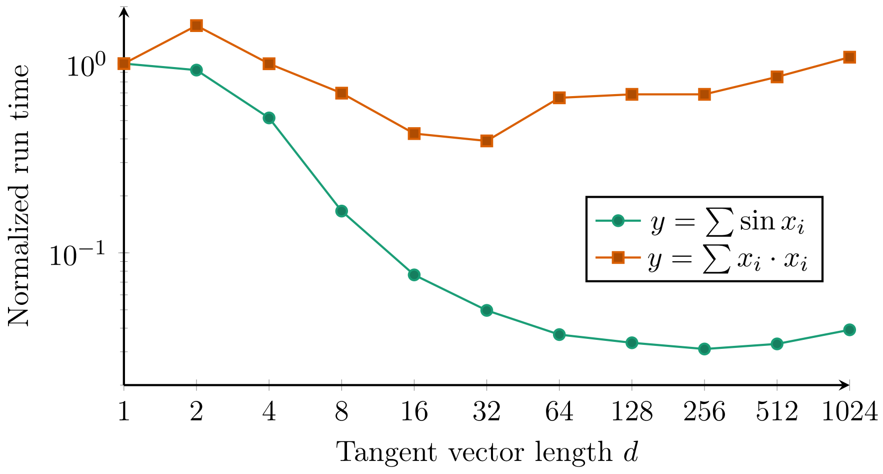


Figure 2.13: Run time of the tangent vector benchmark for varying tangent vector size d. Run time is normalized to the execution time of the scalar tangent.

The tangent mode of AD does not need any AD specific interface routines, other than the access functions `dco::value()` and `dco::derivative()`.

2.8.2 Adjoint mode

Concepts

For the calculation of adjoints, `dco/c++` uses efficient data structures to store the information required for data flow reversal. First, we introduce the graph data structure, with which the data flow reversal can be described. Second, we show how `dco/c++` stores this information in its internal representation. This will become important for the optimizations applied in Sections 3.4, 3.7, and 3.8.

Every elemental assignment in the code can be expressed as a DAG, modeling the SAC generated by the assignment. For an elemental assignment, the output is by definition a scalar, connected to one or multiple inputs. The SAC, consisting of only basic unary or binary operations, can be conveniently differentiated with the chain rule by overloading the operators. A traditional operator overloading approach, e.g. applied in `ADOL-C` [WG12], operates on data structures similar to the DAG with all intermediate nodes included in the graph representation. By implementing operator overloading via a template expression engine, the elemental gradients of a single assignment in the code can be assembled during the augmented forward run by multiplying together the partial derivatives created by the interior edges of the DAG. This technique is commonly referred to as *statement level preaccumulation* [GW08]. For the specific implementation in `dco/c++`, refer to [LLN16].

On the graph level, preaccumulation transforms the DAG into a bipartite graph connecting the inputs to the scalar output. This significantly reduces the storage requirements for the data flow reversal, as no intermediate nodes and edges of the SAC need to be stored.

An edge $e \in E$ in the bipartite graph structure (V, E), pictured in Figure 2.14, corresponds to a preaccumulated derivative of an output with respect to a specific input. The values of the edge labels can be calculated during the augmented forward section, as they do not depend on any

$$v = x_1 \cdot x_2 \cdot x_3^2$$
$$y = (x_2 + x_3) \cdot v$$

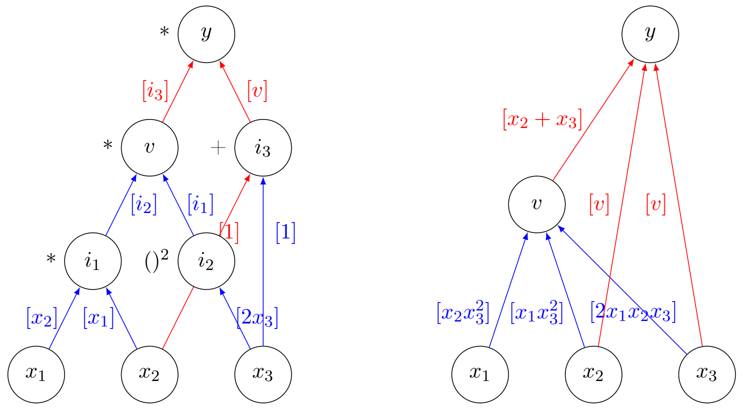


Figure 2.14: Illustration of the adjoint model for code consisting of two elemental assignments. For each assignment in the original code, the individual edges created by the corresponding SAC can be transformed into a bipartite graph using preaccumulation.

values created in the adjoint propagation phase. A vertex $v \in V$ represents an adjoint variable.

During the reverse propagation phase, also called interpretation phase, the adjoint information is propagated through the bipartite graphs modeling individual assignments $y = f(\mathbf{x})$. The edge labels storing the preaccumulated partials $\partial y / \partial x_i$ are multiplied with the corresponding adjoints $\bar{\mathbf{y}}$ of the left hand side of the primal assignment. The resulting products are used to increment the adjoints of the inputs $\bar{x}_i$ which are accessible via the edges of the graph:

$$\bar{x}_i = \frac{\partial y}{\partial x_i} \cdot \bar{y} \quad \forall i \in \{0, \ldots, n\} .$$

The values of the adjoint variables are stored in a contiguous vector in RAM, called the *adjoint vector*. The preaccumulated partial derivatives are stored in a graph like structure, storing the partials as well as the position of the element in the adjoint vector which needs to be incremented during the reverse propagation phase. We call this data structure the *adjoint stack*, as it continuously grows during the augmented forward run and gets evaluated sequentially in opposite order during the reverse propagation phase (last in, first out). In actual implementation, the stack is implemented as a vector instead, to allow reinterpretation. We call the combination of the *adjoint stack* and *adjoint vector* the *tape*.

```cpp
typedef std::vector<double> adjoint_vector;
struct partial_edge{
  double partial_val;
  int target_idx; // entry of the adjoint vector which will be incremented
};
typedef std::vector< std::vector<partial_edge> > partial_edges;

void interpret_tape(const partial_edges& s, adjoint_vector& v){
  for(int i = s.size()-1; i>=0;i--)
    for(const partial_edge& p : s[i])
      v[p.target_idx] += p.partial_val * v[i];
}
```

Listing 2.8: Conceptual implementation of the tape interpretation.

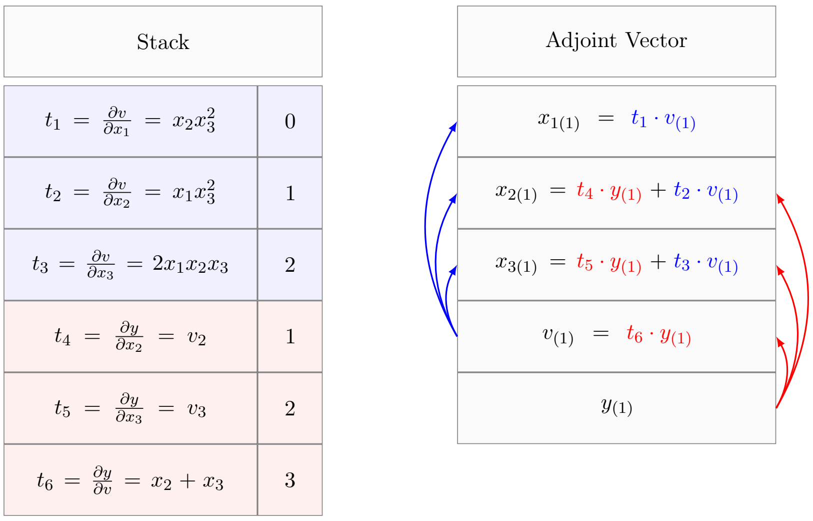

Stack
$t_1 = \frac{\partial v}{\partial x_1} = x_2 x_3^2$ 0
$t_2 = \frac{\partial v}{\partial x_2} = x_1 x_3^2$ 1
$t_3 = \frac{\partial v}{\partial x_3} = 2x_1 x_2 x_3$ 2
$t_4 = \frac{\partial y}{\partial x_2} = v_2$ 1
$t_5 = \frac{\partial y}{\partial x_3} = v_3$ 2
$t_6 = \frac{\partial y}{\partial v} = x_2 + x_3$ 3
Adjoint Vector
$x_{1(1)} = t_1 \cdot v_{(1)}$
$x_{2(1)} = t_4 \cdot y_{(1)} + t_2 \cdot v_{(1)}$
$x_{3(1)} = t_5 \cdot y_{(1)} + t_3 \cdot v_{(1)}$
$v_{(1)} = t_6 \cdot y_{(1)}$
$y_{(1)}$

Figure 2.15: Storage of preaccumulated partial derivatives and the target index in the adjoint vector (left), and adjoint vector (right). The preaccumulated partials are used to increment the adjoints during the reverse sweep.

For each assignment in the code (or in the SAC without preaccumulation), an entry on the stack will be generated, storing all partial edges corresponding to this assignment. A conceptual implementation of the tape interpretation is given in Listing 2.8, the corresponding implementation of the operators generating the tape entries is given in Appendix D.

For a graphical representation of the tape created by the SAC from Figure 2.14, as well as the incrementation process, see Figure 2.15. Note that the position of the variables which need to be loaded from the adjoint vector in order to increment a specific adjoint are not necessarily close together. For example, a simple iterative program dependent on a static parameter would need to increment the adjoint of the parameter, stored at the first position of the adjoint vector, for each iteration of the loop. Furthermore, one adjoint of the iteration variable x needs to be incremented, which will be stored at some (decreasing) distance to the adjoint of the parameter. This leads to random access memory patterns (which for this simple example could be handled by multiple cache lines, but will fail for more complex iterations) and also makes it difficult to offload chunks of the adjoint vector to secondary storage, as entries in multiple chunks may need to be incremented for the adjoint of a single expression.

The tape only stores the preaccumulated partials for a specific state of the primals. To evaluate the adjoints for different primal values, the tape has to be newly recorded, requiring a full re-evaluation of the program. Tools which store the full SAC (e.g. ADOL-C or optionally in CoDiPack) will only evaluate the Jacobians during the reverse interpretation sweep, allowing to evaluate the adjoint model for different primal states without re-recording. This approach comes at the expense of significantly higher memory consumption and lower performance for a single evaluation of the adjoint model. In contrast, different adjoint seeds can be evaluated with an already recorded tape.

An example tape representation directly exported from the internal dco/c++ data structure is shown in Figure 2.16. This tape is created by three iterations of the Babylonian iterative root finding algorithm. It calculates $\sqrt{a}$ with $a = 2$ from an initial guess of $x = 2$ using the iteration

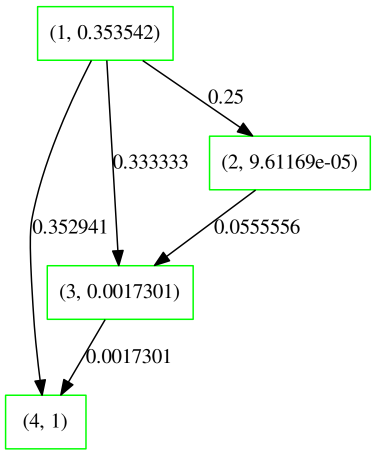


Figure 2.16: Internal representation of the tape, generated by Babylonian root finding algorithm.

procedure `x = 0.5*(a/x+x)`. The edge labels are the partial derivatives stored in the tape, the vertex labels are a pair of the location of the adjoint in the adjoint vector and the value of the adjoint after the reverse propagation has finished. The adjoint of the parameter $\bar{a}$ is located at the first entry in the adjoint vector and evaluates to $0.353542 \approx \frac{\sqrt{2}}{4}$. The Babylonian root finding algorithm is discussed in further detail in Section 3.8.2.

dco/c++ Adjoint Interface

The basic operations `dco::derivative()` and `dco::value()`, already introduced for the tangent mode, still apply for the adjoint mode. Additional routines, corresponding to the management and interpretation of the tape, are needed. dco/c++ provides a global tape, which is accessible through the `global_tape` pointer at any point in the program. Specialized tapes with local scope can also be created, however this feature is not needed in the context of this thesis. The global tape needs to be allocated at the beginning of the program and is alive until it is explicitly destroyed or the program exits.

```
#include <dco.hpp>
int main(){
  // allocation
  dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();

  some_calculation();

  // destruction
  dco::ga1s<double>::tape_t::remove( dco::ga1s<double>::global_tape );
}
```

If not otherwise specified, the tape is allocated as a chunk tape, meaning that the adjoint stack does not have a fixed size and grows on demand. New space for the stack is allocated in chunks, that are not necessarily adjacent in memory. An alternative is the usage of a fixed size tape, which yields slightly higher performance at the expense of less flexibility.

Variables corresponding to inputs, for which derivatives are desired, need to be *registered* in the tape. At this point they will be assigned a positive tape index, uniquely identifying them to avoid name aliasing. The tape index can be used to look up the corresponding adjoint from the adjoint vector.

The registration process helps to reduce the run time and memory overhead of the adjoint by performing an *activity analysis* [HNP05]. That is, by default variables are considered to be passive and do not enter the DAG with any partials. A variable becomes active, once it gets assigned the result of an elemental function involving an active variable.

```
dco::ga1s<double>::type x = 21; // x still passive, dco::tape_index(x)=0
x = 2*x; // activity analysis: x=42, dco::tape_index(x)=0
dco::ga1s<double>::tape->register_variable(x); // x active, dco::tape_index(x)=1
x = 2*x; // aliasing of x, x=84, dco::tape_index(x)=2
```

Once the independent variables have been registered, the primal calculation can be executed. The overloaded operators will populate the adjoint stack with the preaccumulated gradients and update the tape indices of the variables involved in the computations.

The aforementioned concepts are illustrated in the general purpose driver shown in Listing 2.9, which calculates the gradient of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, equivalent to the tangent driver discussed earlier.

```
#include "dco_cpp_dev/src/dco.hpp"
#include <vector>
using namespace std;

typedef dco::ga1s<double>       a1s_mode;
typedef dco::ga1s<double>::type a1s_type;

template<typename T> T f(vector<T> x);

vector<double> calc_grad_f_a1s(const vector<double>& xd){
  a1s_mode::global_tape = a1s_mode::tape_t::create();
  const int n = xd.size();
  vector<double> grad(n);
  vector<a1s_type> x(xd.begin(),xd.end()); // copy passive values
  a1s_mode::global_tape->register_variable(x.begin(),x.end());

  a1s_type y = f(x); // augmented primal calculation
  dco::derivative(y)=1; // seeding
  a1s_mode::global_tape->interpret_adjoint(); // reverse propagation

  grad = dco::derivative(x); // extract derivatives of vector x
  a1s_mode::tape_t::remove(a1s_mode::global_tape);
  return grad;
}
```

Listing 2.9: Driver calculating the full gradient of a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, using adjoint mode of AD.

The tape persists the interpretation and can be re-evaluated with different adjoint seeds. Re-evaluating a previously recorded tape allows to calculate different adjoints for the same primal evaluation path, without the need to execute the augmented primal calculation. Due to the incremental nature of the adjoint propagation, it is usually necessary to zero all elements of the adjoint vector prior to seeding and propagating new adjoints.

```
// zero all entries of the adjoint vector
dco::ga1s<double>::global_tape->zero_adjoints();
```

The contents of the tape can be reset without destroying the tape, which is more efficient than to destroy and re-allocate the tape repeatedly.

```
// delete tape entries and corresponding adjoint vector
dco::ga1s<double>::global_tape->reset_adjoints();
```

For the tape routines `interpret_adjoint`, `zero_adjoint` and `reset`, corresponding routines that allow to operate only on a subset of the tape, exist. The current position in the tape can be queried with `get_position`.

```
dco::ga1s<double>::position_t to
  = dco::ga1s<double>::global_tape->get_position();
// do sth.
dco::ga1s<double>::position_t from
  = dco::ga1s<double>::global_tape->get_position();

dco::ga1s<double>::global_tape->interpret_adjoint_to(to);
dco::ga1s<double>::global_tape->interpret_adjoint_from_to(from,to);
dco::ga1s<double>::global_tape->zero_adjoint_to(to);
dco::ga1s<double>::global_tape->zero_adjoint_from_to(from,to);
dco::ga1s<double>::global_tape->reset_adjoint_to(to);
```

Listing 2.10: Operations can be restricted to a part of the tape.

2.8.3 Higher Order Derivative Models

For higher order derivative models, the tangent and adjoint data types, as well as the access routines can be recursively nested. This will be demonstrated in Section 3.6, for now we will focus on first order data types.

2.9 Gradient Based Optimization

2.9.1 Steepest Descent

One of the most intuitive and popular choices for gradient based optimization is the method of steepest descent. Here we focus on our CFD optimization setting with parameters $\boldsymbol{\alpha} \in \mathbb{R}^{n_C}$, consisting of pre-processing, processing, and post-processing. The chaining of the processing functions requires the use of the total derivative. From a current parameter state $\boldsymbol{\alpha}^i$, the next state is determined by moving the state a distance λ^i in the (negative) direction of the gradient of $\mathcal{J}$ w.r.t. $\boldsymbol{\alpha}$:

$$\boldsymbol{\alpha}^{i+1} = \boldsymbol{\alpha}^i - \lambda^i \cdot \boldsymbol{\nabla}_{\boldsymbol{\alpha}} \mathcal{J}\left(\boldsymbol{\alpha}^i\right) .$$

As the gradient of a function points in the direction of steepest ascent, a reduction in the cost function can be achieved by moving the state in the opposite direction.

The optimal step size λ^i can be chosen by performing a line search along the gradient direction and finding the λ which minimizes the cost function:

$$\lambda^i = \min_{\lambda \in \mathbb{R}^+} \mathcal{J}\left(\boldsymbol{\alpha}^i - \lambda \cdot \boldsymbol{\nabla}_{\boldsymbol{\alpha}} \mathcal{J}\left(\boldsymbol{\alpha}^i\right)\right) .$$

This creates an additional optimization problem, albeit only for the scalar parameter λ, requiring additional cost function evaluations as well as derivatives $\mathrm{d}\mathcal{J}(\boldsymbol{\alpha}^i - \lambda \cdot \boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J})/\mathrm{d}\lambda$. As λ is a scalar, the derivatives can conveniently be calculated with tangent mode or FD.

To avoid the cost and complexity of calculating additional derivatives, a pragmatic approach is to choose a somewhat arbitrary value for λ, ensuring that it improves the cost functional, that is $\mathcal{J}(\boldsymbol{\alpha}^{i+1}) < \mathcal{J}(\boldsymbol{\alpha}^i)$. A popular implementation is the bisection algorithm. Starting from an initial guess, λ is iteratively divided, until an improvement in the cost function is achieved. The bisection approach is illustrated in Algorithm 1.

Algorithm 1: Bisection line search algorithm.

```
  Input: previous parameter state $\boldsymbol{\alpha}^i$
  Data: gradient $\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J}$, initial line search stepsize $\lambda_{start}$
  Output: new parameter state $\boldsymbol{\alpha}^{i+1}$
1 $\lambda \leftarrow \lambda_{start}$ ;
2 while $\mathcal{J}(\boldsymbol{\alpha}^i - \lambda\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J}) \geq \mathcal{J}(\boldsymbol{\alpha}^i)$ do
3   | $\lambda \leftarrow \lambda/2$ ;
4 end
5 $\boldsymbol{\alpha}^{i+1} \leftarrow \boldsymbol{\alpha}^i - \lambda\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J}$ ;
```

For $\|\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J}\| \neq 0$, the algorithm is guaranteed to terminate, as for $\lim_{\lambda \to 0^+}$ a reduction must be achieved, else $-\boldsymbol{\nabla}_{\boldsymbol{\alpha}}\mathcal{J}$ would not be a descent direction, contradicting the definition of the gradient. No additional derivatives are required for the bisection. Thus, the additional cost function evaluations can be performed in *passive* mode, as the gradient $\mathrm{d}\mathcal{J}/\mathrm{d}\boldsymbol{\alpha}^i$ is considered fixed during the line search.

To ensure convergence, the step size can be chosen such that the Wolfe conditions [Wol69] are fulfilled, adding further complexity to the line search algorithm.

As an example, showcasing the limitations of steepest descent, Figure 2.17 illustrates the convergence of the steepest descent algorithm for the Rosenbrock function [Ros60]

$$f(x, y) = (1 - x)^2 - 100(y - x^2)^2$$

with a fixed step size. Finding the optimum of the Rosenbrock function is notoriously difficult for gradient based optimization methods, due to the long valley with only a low gradient leading to the optimum. Consequently the steepest descent algorithm needs several thousand iterations to converge from the chosen starting position to the optimum at $(1, 1)$.

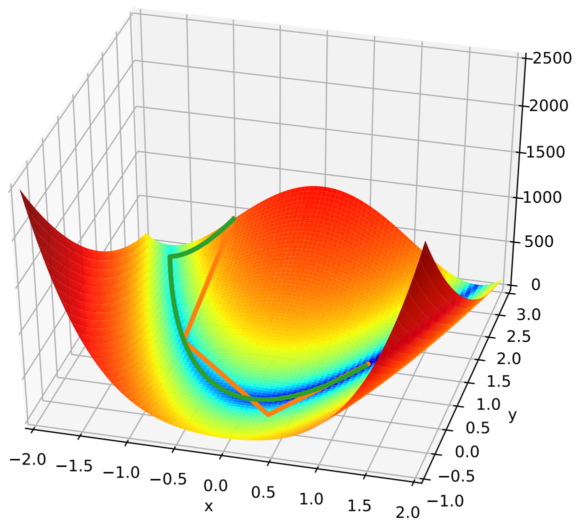


Figure 2.17: Iteration history of the steepest descent algorithm (green trajectory) and Newton's algorithm (red trajectory), from starting point $\mathbf{P} = (-1, 3)$ to the optimum at $\mathbf{O} = (1, 1)$ of the Rosenbrock function.

2.9.2 Newton's Method

With the additional information of curvature available from the Hessian $H = \boldsymbol{\nabla}^2 f(\mathbf{x})$, an improved iteration called *Newton's method* can be constructed.

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \underbrace{\left(\boldsymbol{\nabla}^2 f(\mathbf{x}^i)\right)^{-1} \cdot \boldsymbol{\nabla} f(\mathbf{x}^i)}_{=\mathbf{z}} .$$

This method allows to find (local) minima characterized by $\boldsymbol{\nabla} f(\mathbf{x}) = 0$. Due to the cost and stability issues involved in the explicit calculation of the inverse Hessian $\left(\boldsymbol{\nabla}^2 f(\mathbf{x}^i)\right)^{-1}$, in practice an equivalent two stage procedure is used, where the update step is calculated by the solution of a linear system.

$$\begin{aligned} \boldsymbol{\nabla}^2 f(\mathbf{x}^i) \cdot \mathbf{z} = \boldsymbol{\nabla} f(\mathbf{x}^i) \Rightarrow \mathbf{z} &= S\left(\boldsymbol{\nabla}^2 f(\mathbf{x}^i), \boldsymbol{\nabla} f(\mathbf{x}^i)\right) \\ \mathbf{x}^{i+1} &= \mathbf{x}^i - \mathbf{z} . \end{aligned}$$

For certain optimization problems, Newton's method can dramatically reduce the number of iterations needed, outweighing the additional complexity required to obtain the second order derivative information. For example, the Rosenbrock example in Figure 2.17 only needs four iterations of Newton's method to converge to machine precision, compared to thousands of iterations with steepest descent.

Newton's method can be shown to converge to the solution quadratically starting from a point within an interval to the solution. However, convergence is not necessarily guaranteed from all starting points [Deu11]. It is sometimes advisable to initialize the solution with a number of steepest descent iterations to get into the region of convergence of Newton's method and then switch to the faster converging Newton's method.

For problems of high dimension, where the assembly of the Hessian is considered too costly or complex, Quasi-Newton methods can be used. Quasi-Newton algorithms iteratively construct better approximations for the inverse Hessian, retaining some of the convergence properties of the Newton's method. A popular Quasi-Newton method is the BFGS algorithm which has been implemented in multiple variations [Bro70; Fle70; Gol70; Sha70].

2.10 Discrete Adjoint Residual Approach

Many researchers in the past already implemented an approach which only requires the residuals of the FVM discretization systems to be differentiated. Most just call it the *discrete adjoint approach* [Mav07; RU13; NJ07; He+18; Gil+03]. To avoid confusion with the algorithmic discrete adjoint, applied to the whole non-linear iteration step, we will call it the *discrete adjoint residual* approach.

For the usual (laminar) flow state in discretized form $\mathbf{x} = (\mathbf{U}, \mathbf{p})$, convergence of the governing equations implies that the residual $\mathbf{R}$ of the Navier-Stokes equations is (near) zero

$$\mathbf{R}\left(\mathbf{x}(\boldsymbol{\alpha}), \boldsymbol{\alpha}\right) = 0\,.$$

Thus, the total derivative of the residual can be expressed as

$$\frac{\mathrm{d}\mathbf{R}}{\mathrm{d}\boldsymbol{\alpha}} = \frac{\partial \mathbf{R}}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathbf{R}}{\partial \boldsymbol{\alpha}} = 0\,. \tag{2.16}$$

The sensitivity of the states w.r.t. the parameters can be calculated by transforming (2.16) into

$$\frac{\partial \mathbf{x}}{\partial \boldsymbol{\alpha}} = -\left(\frac{\partial \mathbf{R}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{R}}{\partial \boldsymbol{\alpha}}\,. \tag{2.17}$$

The desired sensitivities $\mathrm{d}\mathcal{J}/\mathrm{d}\boldsymbol{\alpha}$ can be calculated by inserting (2.17) into the total derivative for $\mathcal{J}$:

$$\frac{\mathrm{d}\mathcal{J}(\mathbf{x}(\boldsymbol{\alpha}), \boldsymbol{\alpha})}{\mathrm{d}\boldsymbol{\alpha}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial \boldsymbol{\alpha}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} - \frac{\partial \mathcal{J}}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{R}}{\partial \boldsymbol{\alpha}}\,. \tag{2.18}$$

Defining

$$\boldsymbol{\Lambda}_{\mathbf{x}}^{T} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{x}}\right)^{-1},$$

one can eliminate the inverse operation from (2.18) by solving the linear equation system

$$\left(\frac{\partial \mathbf{R}}{\partial \mathbf{x}}\right)^{T} \cdot \boldsymbol{\Lambda}_{\mathbf{x}} = \left(\frac{\partial \mathcal{J}}{\partial \mathbf{x}}\right)^{T}$$

for $\boldsymbol{\Lambda}_{\mathbf{x}}$ and substituting the result into (2.18)

$$\frac{\mathrm{d}\mathcal{J}(\mathbf{x}(\boldsymbol{\alpha}), \boldsymbol{\alpha})}{\mathrm{d}\boldsymbol{\alpha}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} - \left(\frac{\partial \mathbf{R}}{\partial \boldsymbol{\alpha}}\right)^{T} \cdot \boldsymbol{\Lambda}_{\mathbf{x}}\,. \tag{2.19}$$

When using the adjoint mode of AD, the (matrix-vector) product $(\partial \mathbf{R}/\partial \boldsymbol{\alpha})^{T} \cdot \boldsymbol{\Lambda}_{\mathbf{x}}$ can be evaluated without explicitly calculating the Jacobian $\partial \mathbf{R}/\partial \boldsymbol{\alpha}$, by exploiting that the adjoint model allows to calculate the projection of the transposed Jacobian in an arbitrary direction. Therefore, $\mathbf{\Lambda_x}$ can be chosen as a seed direction. The adjoint model then calculates the desired matrix-vector product. The explicit dependence $\partial\mathcal{J}/\partial\boldsymbol{\alpha}$ of the cost function on the parameters is often zero, or else can be calculated cheaply with one evaluation of the adjoint model. The costly operations are therefore the calculation of the Jacobian of the residual w.r.t. the state $\partial\mathbf{R}/\partial\mathbf{x}$, and the solution of the linear equation system.

The linear system can potentially be solved using a iterative matrix free solver (e.g. BiCG). Matrix free solvers, in contrast to regular solvers, only require a mean to evaluate matrix vector products of the matrix with arbitrary vectors. AD allows the efficient evaluation of the (transposed) Jacobian vector product. Using a matrix free solver the full Jacobian neither has to be evaluated, nor stored. However, due to the poor condition of the linear system, in practice matrix free solvers are usually not a feasible option. Using matrix free solvers, the problem can not be effectively preconditioned and will not converge well. Therefore, the full (sparse) Jacobian has to be calculated, either in adjoint or tangent mode, or using FD.

Note, that some authors calculate the Jacobian with FD and still call their implementation discrete adjoint, in reference to the adjoint equations (e.g. [He+18]).

To effectively calculate the sparse Jacobian $\partial\mathbf{R}/\partial\mathbf{x}$, coloring techniques can be used, lowering the complexity from $\mathcal{O}(n_\mathbf{x})$, to $\mathcal{O}(d)$, where d is the maximum number of cells which influence the discretization of a cell, that is

$$d = \max_{0\leq i<n} \sum_{j=0}^{n-1} \left| sgn\left(\frac{\partial \mathbf{R}_i}{\partial \mathbf{x}_j}\right)\right| .$$

Coloring techniques are presented in the following section.

The discrete adjoint residual approach has been implemented in the discrete adjoint OpenFOAM framework, along with the necessary coloring heuristics. Details are presented in Section 4.5.

2.11 Matrix Coloring

2.11.1 Jacobian Compression

The calculation of sparse Jacobians and Hessians can be considerably sped up by exploiting the sparsity and orthogonality of rows and/or columns.

Definition 13 (Structural Orthogonality).
Two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ are structurally orthogonal if and only if at each index at least one of the vectors is zero, that is

$$\sum_{i=0}^{n-1} (v_i w_i)^2 = 0 .$$

The tangent model computes the multiplication of the Jacobian $J = \boldsymbol{\nabla} f$ in a direction $\dot{\mathbf{x}}$, the adjoint model multiplications of the transposed Jacobian in direction $\bar{\mathbf{x}}$. If multiple columns of the Jacobians are pairwise structurally orthogonal, all non-zero entries of those columns can be computed with one evaluation of the tangent model, by superimposing the unit vectors which would be used to calculate the corresponding columns.

Similarly all non-zero entries of structurally orthogonal rows can be computed with one evaluation of the adjoint model.

Definition 14 (Jacobian column compression).
Let J be the Jacobian of a function $f : \mathbb{R}^n \to R^m$, with associated row and column index sets $\mathcal{I}$ and $\mathcal{J}$. Let $\mathcal{J}_\mathcal{O} \subset \mathcal{J}$ be a set of columns indices for which the columns of the Jacobians are pairwise structurally orthogonal, that is

$$J_{ij_1} \cdot J_{ij_2} = 0 \quad \forall i \in \mathcal{I},\ j_1, j_2 \in \mathcal{J}_\mathcal{O}, j_1 \neq j_2\,.$$

Then $\mathbf{v} = J \cdot \left(\sum_{i \in \mathcal{J}_\mathcal{O}} \mathbf{e}_i\right) = \sum_{i \in \mathcal{J}_\mathcal{O}} J\mathbf{e}_i$ contains all non-zero elements of the columns corresponding to the indices $\mathcal{J}_\mathcal{O}$.

Definition 15 (Jacobian coloring).
A Jacobian coloring groups rows/columns of the Jacobian into sets of structurally orthogonal rows/columns. We refer to those sets as colors of the Jacobian. Each row/column is assigned a color, and is thus included in exactly one set.

Definition 16 (Row Seed Matrix).
A row seed matrix is a matrix $S_\mathcal{R} \in \{0,1\}^{c \times n}$ with $\sum_{i=0}^{c-1} S_{ij} = 1$ for all $i = 0, \ldots, c$. The individual rows of the seed matrix $S_\mathcal{R}$ can be used as seed vectors for the adjoint model of AD, to compute a row of the compressed Jacobian $J_C = S_\mathcal{R} \cdot J \in \mathbb{R}^{c \times n}$.

Definition 17 (Column Seed Matrix).
A column seed matrix is a matrix $S_\mathcal{C} \in \{0,1\}^{m \times c}$ with $\sum_{j=0}^{c-1} S_{ij} = 1$ for all $j = 0, \ldots, c$. The individual columns of the seed matrix $S_\mathcal{C}$ can be used as seed vectors for the tangent model of AD, to compute a column of the compressed Jacobian $J_C = J \cdot S_\mathcal{C} \in \mathbb{R}^{m \times c}$.

Definition 18 (Bipartite row/column graph of a matrix).
The bipartite graph of a matrix A is defined as $G = ((R, C), E)$. Each row of the matrix A corresponds to a vertex r_i in the set R. Each column of the matrix A corresponds to a vertex c_j in the set C. An element r_i of the set R is connected to an element c_j in C by and edge $(r_i, c_j) \in E$ if and only if the matrix entry A_{ij} is non-zero.

The bipartite graph of a matrix can be partially or fully colored to obtain a feasible Jacobian coloring [GMP05].

We will briefly present the previous concepts on the following example matrix $A \in \mathbb{R}^{4 \times 4}$ with 8 non-zero entries.

$$A = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 \\ 0 & a_{11} & 0 & a_{13} \\ 0 & 0 & a_{22} & a_{23} \\ a_{30} & 0 & a_{32} & 0 \end{bmatrix}.$$

A possible column coloring for the example is $\mathcal{C} = \{\{c_0, c_3\}, \{c_1, c_2\}\}$, a possible row coloring is $\mathcal{R} = \{\{r_0, r_2\}, \{r_1, r_3\}\}$. The resulting seed matrices are

$$S_\mathcal{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad S_\mathcal{R} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

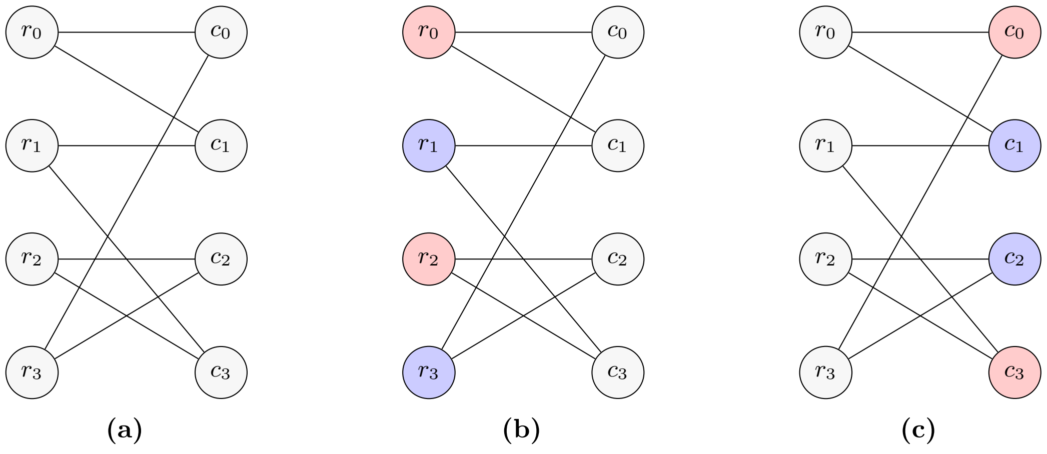


Figure 2.18: Bipartite graph representation of non-zero pattern of matrix A, partial distance-two row coloring, partial distance-two column coloring.

The resulting compressed Jacobians, from which all nonzero entries can be reconstructed, are:

$$A_{\mathcal{C}} = AS_{\mathcal{C}} = \begin{bmatrix} a_{00} & a_{01} \\ a_{13} & a_{11} \\ a_{23} & a_{22} \\ a_{30} & a_{32} \end{bmatrix} \quad \text{and} \quad A_{\mathcal{R}} = S_{\mathcal{R}}A = \begin{bmatrix} a_{00} & a_{01} & a_{22} & a_{23} \\ a_{30} & a_{11} & a_{32} & a_{13} \end{bmatrix}.$$

The bipartite graph for the matrix, as well as the feasible partial row and column colorings, given above, are shown in Figure 2.18.

Lemma 1.
A valid partial distance two row coloring on the bipartite graph solves the row compression problem, allowing to directly recover the nonzero entries from the compressed Jacobian [GMP05].

3 Differentiation of Complex Iterative CFD Algorithms

In this chapter the differentiation of complex iterative CFD algorithms is discussed. Particular emphasis is placed on the efficient implementation of the developed discrete adjoint methods in OpenFOAM. First we will focus on a black-box application of AD and then later implement various improvements.

3.1 Foundations

3.1.1 Optimization Problem

In the following we define the general optimization problem motivating most of this thesis. We consider a general set of parameters $\boldsymbol{\gamma} \in \mathbb{R}^p$. For topology optimization, we call this set $\boldsymbol{\gamma} = \boldsymbol{\alpha} \in \mathbb{R}^{n_C}$, for shape optimization $\boldsymbol{\gamma} = \boldsymbol{\beta} \in \mathbb{R}^{n_{F_\Gamma}}$.

Using these parameters, we define an optimization problem over a calculation consisting of three distinct phases:

Pre-processing $\mathcal{P}$: Initialize the solver and create an initial state $\mathbf{x}^0 \in \mathbb{R}^{n_\mathbf{x}}$.

Processing $\mathcal{F}$: Solve the underlying ODEs (iteratively), to obtain final state(s).

Post-Processing $\mathcal{J}$: Create a scalar output from the final state(s) by evaluating a cost function.

The full chain $y = \mathcal{J} \circ \mathcal{F} \circ \mathcal{P}$ needs to be differentiated in order to obtain the gradient $\frac{\mathrm{d}y}{\mathrm{d}\boldsymbol{\gamma}}$. Optimization can then be applied to $\boldsymbol{\gamma}$, in order to alter the solution and improve the cost function output, as calculated by the post-processor. To keep parameters in a feasible range, we assume box constraints for the individual parameters γ_i. In practice these can be commonly replaced by a global lower and upper bound.

$$\begin{aligned} \underset{\boldsymbol{\gamma}}{\text{minimize}} \quad & \mathcal{J}(\boldsymbol{\gamma}) \\ \text{subject to} \quad & l_i \leq \gamma_i \leq u_i,\ i = 0, \dots, p-1. \end{aligned}$$

Depending on the region of application, different combinations of unsteadiness are possible. We introduce feasible definitions for the functions $\mathcal{P}, \mathcal{F}$, and $\mathcal{J}$ for varying degrees of unsteadiness below. Ducted flows are often laminar and steady, while external aerodynamic flows often exhibit varying levels of transient effects.

Steady Data, Steady Parameters

This is the most common case for optimization of flows. It assumes that the simulation converges to a steady state. We denote the final state, reached after k iterations, as x^* to emphasize the fixed point nature of the iteration.

$$\begin{aligned}
&\mathcal{P}: \mathbb{R}^p \to \mathbb{R}^{n_\mathbf{x}} && \mathbf{x}^0 = \mathcal{P}(\boldsymbol{\gamma})\\
&\mathcal{F}: \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^p \to \mathbb{R}^{n_\mathbf{x}} && \mathbf{x}^* = \mathcal{F}(\mathbf{x}^0, \boldsymbol{\gamma}) = f^k(\mathbf{x}^{k-1}, \boldsymbol{\gamma}) \circ \ldots \circ f^2(\mathbf{x}^1, \boldsymbol{\gamma}) \circ f^1(\mathbf{x}^0, \boldsymbol{\gamma})\\
&\mathcal{J}: \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^p \to \mathbb{R} && y = \mathcal{J}(\mathbf{x}^*, \boldsymbol{\gamma})
\end{aligned}$$

For transient problems, the formulation of the parameters and cost functions can incorporate varying levels of unsteadiness.

Transient Data, Steady Parameters

The scalar cost function depends on states from potentially all k time steps, but the parameter set is kept constant during the time iteration. Such a formulation is e.g. useful to get a mean value of an oscillating phenomenon, using the parameters to reduce the fluctuations.

$$\begin{aligned}
&\mathcal{P}: \mathbb{R}^p \to \mathbb{R}^{n_\mathbf{x}} && \mathbf{x}^0 = \mathcal{P}(\boldsymbol{\gamma})\\
&\mathcal{F}: \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^p \to \mathbb{R}^{n_\mathbf{x} \times k} && (\mathbf{x}^k, \ldots, \mathbf{x}^1) = \mathcal{F}(\mathbf{x}^0, \boldsymbol{\gamma}) = f^k(\mathbf{x}^{k-1}, \boldsymbol{\gamma}) \circ \ldots \circ f^1(\mathbf{x}^0, \boldsymbol{\gamma})\\
&\mathcal{J}: \mathbb{R}^{n_\mathbf{x} \times k} \times \mathbb{R}^p \to \mathbb{R} && y = \mathcal{J}(\mathbf{x}^k, \ldots, \mathbf{x}^1, \boldsymbol{\gamma})
\end{aligned}$$

Transient Data, Transient Parameters

As before, the scalar cost function depends on states from potentially all k time steps, but the parameter set is allowed to change between time steps, expanding it from $\boldsymbol{\gamma} \in \mathbb{R}^p$ to $\mathbb{R}^{p \times k}$.

$$\begin{aligned}
&\mathcal{P}: \mathbb{R}^p \to \mathbb{R}^{n_\mathbf{x}} && \mathbf{x}^0 = \mathcal{P}(\boldsymbol{\gamma})\\
&\mathcal{F}: \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^{p \times k} \to \mathbb{R}^{n_\mathbf{x} \times k} && (\mathbf{x}^k, \ldots, \mathbf{x}^1) = f^k(\mathbf{x}^{k-1}, \boldsymbol{\gamma}^k) \circ \ldots \circ f^1(\mathbf{x}^0, \boldsymbol{\gamma}^1)\\
&\mathcal{J}: \mathbb{R}^{n_\mathbf{x} \times k} \times \mathbb{R}^{p \times k} \to \mathbb{R} && y = \mathcal{J}(\mathbf{x}^k, \ldots, \mathbf{x}^1, \boldsymbol{\gamma}^k, \ldots, \boldsymbol{\gamma}^1)
\end{aligned}$$

3.1.2 Reference Cases

In the following sections we will discuss the basic concept of AD in the context of CFD solvers. To illustrate the implementation of those concepts, they are introduced to the OpenFOAM CFD framework. Whenever we need a practical example we will refer to the following two test cases, which are well suited for topology optimization.

Angled Duct

The first test case is a 2D geometry of a 90 degree bend, depicted in Figure 3.1. The geometry is stated dimensionless with a characteristic length of L.

Two pipes of diameter L and length $2L$ are connected to a square of edge length $3L$. The flow enters from the lower left and leaves on the top right. A constant velocity profile is prescribed at the inlet, a zero gradient condition is applied at the outlet. The outlet pressure is fixed to zero with a zero gradient condition at the inlet. Walls are modeled as no-slip. A vortex forms in the upper left corner. The vortex is induced by the shear forces, created by the fluid which flows

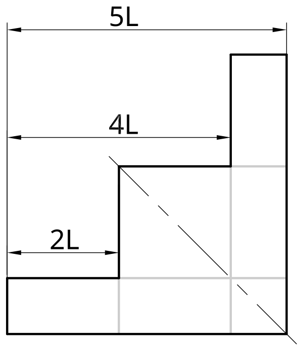


Figure 3.1: Geometry of the angled duct case. Inflow on the lower left, outflow on the top right. Blocks of the structured mesh are indicated in light gray. The geometry is symmetric along the dashed diagonal line.

from the lower pipe into the upward facing pipe. This vortex region is an obvious location for optimization, aiming to reduce the power loss in the system. For higher flow velocity, a second vortex region forms in the lower right corner. A structured mesh is created by `blockMesh` (see also Section 4.6). It can thus easily be scaled to different mesh resolutions.

At low Reynolds numbers, the use of a turbulence model is optional. For higher Reynolds numbers, the k-ω turbulence model is used.

Different mesh refinement levels, starting from the coarsest possible mesh for this geometry, up to over 200 000 cells, are listed in Table 3.1. The mesh levels below refinement level 15 are hardly useful to obtain a realistic solution. However, they might be used to inspect sparsity patterns and the discretization. The corresponding `blockMeshDict.m4` file is listed in Appendix C.10. The mesh resolution can be controlled by a `GNU m4` parameter, creating the final `blockMeshDict` for the mesher. If not otherwise specified a refinement level of 30, resulting in $n_C = 11\,700$ cells, is used.

Table 3.1: Refinement levels of the angled duct case. Level 1 is the coarsest possible mesh, finer meshes are obtained by uniformly refining the block edges.

Level	n_C	Level	n_C
1	13	15	2 925
2	52	30	11 700
5	325	60	46 800
10	1300	120	187 200

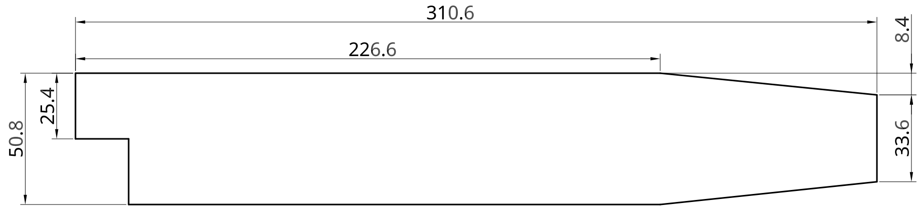


Figure 3.2: Dimensions of the Pitz-Daily example. Inflow on the left, outflow on the right. Dimensions in mm.

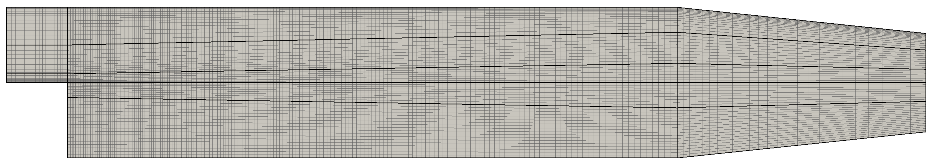


Figure 3.3: Geometry of the Pitz-Daily test case, consisting of 13 blocks (bold black lines) meshed with $n_C = 12\,250$ cells (gray lines).

Pitz-Daily Case

The second case is the flow over a backward facing step. This problem, or variations of it, have been extensively studied and used for verification in a variety of disciplines, in particular for the verification of turbulence models [Rum12]. This particular geometry was first introduced in [PD83] and is consequently referred to as the Pitz-Daily case. It is included as a standard tutorial and verification case in OpenFOAM. The geometry is shown in Figure 3.2. The flow enters from the left, passes the step and leaves the domain through the outlet at the end of the nozzle shaped exit.

At the inflow a constant velocity of 10 m/s is prescribed (Dirichlet condition) with a zero gradient condition on the pressure (Neumann condition). At the outflow a zero pressure, as well as a zero velocity gradient condition, are applied. At the walls a no-slip condition is used. The boundary conditions for the turbulence are calculated using the `kqRWallFunction` for the turbulent kinetic energy k, and `epsilonWallFunction` for the turbulence dissipation rate ϵ. The resulting Reynolds number of $Re = 25\,400$, calculated width the inlet width as the reference length, as well as the solution singularity expected at the step, makes the use of a turbulence model advisable. Else an accurate and steady solution will not be obtained on coarse meshes. The mesh is again obtained by `blockMesh`, using the parameters provided by OpenFOAM. The mesh is slightly graded towards the walls, to obtain better turbulence resolution. A planar view of the mesh for the configuration with $n_C = 12\,250$ cells is shown in Figure 3.3.

3.1.3 Power Loss Cost Function

For both reference cases, we utilize the power loss, induced by the pressure loss inside the system, as the objective. Other cost functions are possible, e.g. lift and drag are defined in Chapter 5.

The power loss is calculated as the difference between the (total) pressure integral on the inlet and outlet boundaries of the flow domain, multiplied by the flow velocity:

$$\mathcal{J} = \int_{\Gamma_{inlet}} -\left(p + \frac{1}{2}\|\mathbf{u}\|^2\right) \mathbf{u}\cdot\mathbf{n}\,\mathrm{d}\Gamma + \int_{\Gamma_{outlet}} -\left(p + \frac{1}{2}\|\mathbf{u}\|^2\right) \mathbf{u}\cdot\mathbf{n}\,\mathrm{d}\Gamma\,.$$

The integrals have opposite signs, as the scalar product $\mathbf{u}\cdot\mathbf{n}$ yields a positive flux for flow entering the domain and negative flux for flow leaving the domain (normals are defined to be facing inwards).

On the discretized result the integrals become sums over the boundary patches. As the flux ϕ is defined on the cell faces, they are used instead of the cell centered velocities, which would need to be interpolated to the faces first.

$$\mathcal{J} = \sum_{f\in F_{\Gamma_{inlet}}} -\phi_f\left(p_f + \frac{1}{2}\left(\frac{\phi_f}{A_f}\right)^2\right) + \sum_{f\in F_{\Gamma_{outlet}}} -\phi_f\left(p_f + \frac{1}{2}\left(\frac{\phi_f}{A_f}\right)^2\right)\,.$$

Like in the integral formulation, the results of the two sums have different signs, due to the different sign of the fluxes ϕ for inlet and outlet faces.

3.2 Differentiation of Complex Simulation Software

3.2.1 Motivation for Application of Tool Driven AD

Software in general and especially in computational engineering can be very complex. The growth of complexity in software systems is believed to be exponential in time [Leh96; Dvo09], corresponding to the exponentially growing capabilities of computing hardware, according to Moore's Law [Sch97]. Specifically, a general purpose tool like OpenFOAM, which is not tailored to a very specific and narrow domain and use case, will require a very large and complex code base. For illustration, refer to Figure 3.4, which shows the complex linkage pattern between the elemental OpenFOAM libraries. If one wants to retain the scope and broad applicability of the primal software, then as much as possible of the primal code base has to be covered by AD.

Table 3.2 shows the growth of the OpenFOAM code base during the last few releases. The developers of primal OpenFOAM have recently adopted a biannual release cycle. This makes an efficient upgrade path, which brings the adjoint version quickly up to date with the primal version, paramount. The code base is distributed over thousands of code files, the logic abstracted behind many layers of class inheritance, templatization, and function macros. This, as well as the usage of advanced `C++` language features, makes the application of current source code transformation tools only applicable to limited numerical kernels. In order to cover the whole package by AD, the application of an operator overloading tool is the only viable approach at this point.

The number of changes required to the code base for the application of operator overloading using `dco/c++` is summarized in Table 3.3. The changes are broken down to the sub libraries contained in the OpenFOAM `src/` directory. As can be expected, the number of changes roughly correlates to the amount of code included in the specific libraries, with the `OpenFOAM` core library and the `finiteVolume` library receiving most changes. The high number of changes in the `Pstream` library stems from the addition of the `AMPI` library sources and not from major changes in the existing code base. The specifics of the code changes required are addressed in Section 3.2.3.

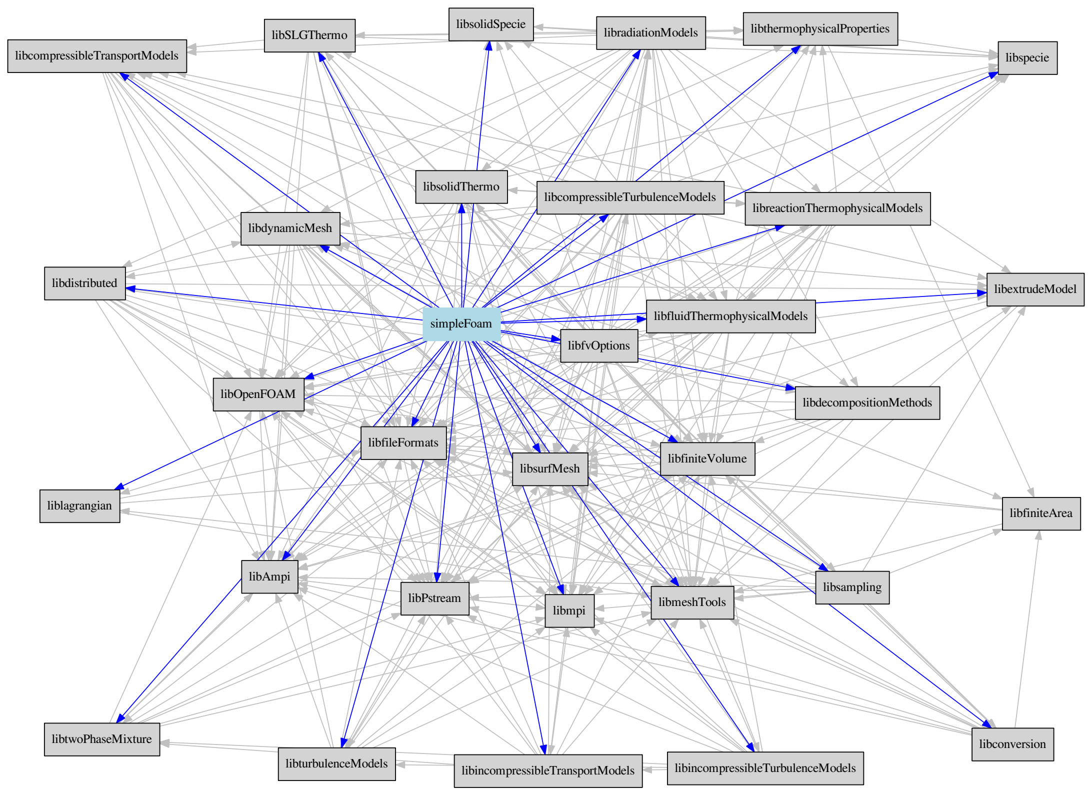


Figure 3.4: Library dependency within the OpenFOAM framework of the `simpleFoam` solver (blue node). Blue edges indicate libraries directly linked to the solver, gray edges linkages within the OpenFOAM framework.

Table 3.2: Growth of OpenFOAM code base from version `3.0` to `17.06-plus`. `src/` contains the OpenFOAM libraries, `applications/` contains the individual tools and solvers (e.g. `simpleFoam`).

		`src/`		`applications/`	
		files	LOC	files	LOC
`3.0`	C++	2878	439k	884	133k
	C++ Header	3118	201k	908	47k
`3.0-plus`	C++	2995	457k	909	141k
	C++ Header	3218	208k	949	48k
`16.12-plus`	C++	3188	498k	879	132k
	C++ Header	3423	225k	971	47k
`17.06-plus`	C++	3281	515k	895	134k
	C++ Header	3529	231k	1023	48k

Table 3.3: Comparison of number of files and lines of code (LOC) between stock OpenFOAM and discrete adjoint OpenFOAM.

Library Name	Identical Files	Changed Files	Identical LOC	Changed LOC
OpenFOAM	1501	102	173596	768
finiteVolume	1103	12	81077	355
meshTools	385	7	61060	71
lagrangian	673	22	58431	61
dynamicMesh	219	7	58421	17
thermophysicalModels	550	8	55950	30
mesh	148	4	37218	5
TurbulenceModels	236	19	28052	67
sampling	157	5	27274	30
functionObjects	254	5	26490	11
conversion	68	1	14763	1
regionModels	146	5	13845	8
surfMesh	95	1	9609	2
parallel	49	4	9287	10
fvOptions	98	2	7500	4
fileFormats	83	3	7100	11
fvMotionSolver	75	1	6227	3
edgeMesh	39	0	5463	0
rigidBodyDynamics	94	0	4773	0
sixDoFRigidBodyMotion	53	0	3684	0
triSurface	31	2	3526	10
OSspecific	34	0	3350	0
combustionModels	53	4	3337	5
ODE	35	1	2974	3
transportModels	44	1	2720	1
randomProcesses	28	2	2221	2
dynamicFvMesh	17	1	1802	1
renumber	19	0	1595	0
engine	25	1	1540	3
topoChangerFvMesh	15	0	1532	0
genericPatchFields	8	0	1469	0
Pstream	2	19	1208	3259
fvAgglomerationMethods	6	0	999	0
rigidBodyMeshMotion	4	0	654	0
regionCoupled	2	0	474	0

3.2.2 Differentiating a Complete Steady Iteration History

Without assuming any additional knowledge about the intrinsics of CFD and iterative solution methods, one can tackle the task of calculating the gradient of a cost function $\mathcal{J}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ w.r.t. the parameters $\boldsymbol{\gamma}$ by algorithmically differentiating the whole computer program, which implements the calculation of $\mathbf{x}$ and $\mathcal{J}$. This is commonly called *black-box* approach [Nau12], as no knowledge about the inner working of the model is required, as long as the inputs, outputs, and parameters of the model are well defined. The black-box approach assumes differentiability of the implementation at all locations of interest. With FD one can calculate such a gradient, even without having access to the source code implementing the model, by perturbing the parameters and observing the change of the outputs. This is especially useful when differentiating through functions which are only available as pre-compiled libraries (e.g. due to licensing and intellectual property concerns).

In the following, we assume the parameters $\boldsymbol{\gamma}$ to be the momentum penalty terms $\boldsymbol{\alpha}$ of topology optimization, as this will be used in the illustrative examples. The same statements apply virtually unchanged for the general case $\boldsymbol{\gamma} \in \mathbb{R}^p$. Further, we assume the case to converge to a steady solution, as introduced in Section 3.1.1. The final state of the problem is obtained by repeatedly applying functions $f^i(\mathbf{x}^{i-1}, \boldsymbol{\alpha}) : \mathbb{R}^{n_\mathbf{x} \times n_C} \to \mathbb{R}^{n_\mathbf{x}}$ to the initial state, which models the propagation of physical quantities in time.

For steady problems, the state converges to a fixed point $\mathbf{x}^*$ at which point the variation $\partial \mathbf{x}/\partial t$ of the solution over time is zero, even if calculated with a transient solver. The series of functions f^i also converges to a function $f^*(\mathbf{x}^*) = \lim_{k\to\infty} f^k(\mathbf{x}^*)$. For black-box differentiation, these fixed point properties are not exploited, however they will become important when using *reverse accumulation* and *piggybacking* in later sections.

The gradient of the cost function $y = \mathcal{J}(\mathbf{x}^k)$ after k iterations w.r.t $\boldsymbol{\alpha}$ is defined by the chain rule as:

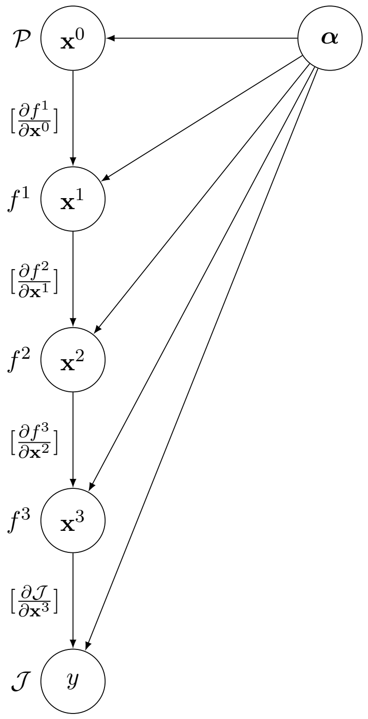
$$\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}} = \sum_{i=0}^{k} \frac{\partial \mathcal{J}}{\partial \mathbf{x}^k} \left(\prod_{j=i+1}^{k} \frac{\partial f^j}{\partial x^{j-1}} \right) \frac{\partial f^i}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}, \tag{3.1}$$

where $f^0 = P(\boldsymbol{\alpha})$ is the preprocessing step which creates the initial state $\mathbf{x}^0$. For $k \to \infty$, this gradient is the desired gradient of the fixed point:

$$\mathcal{J}(\mathbf{x}^*) = \lim_{k\to\infty} \mathcal{J}(\mathbf{x}^k) \Rightarrow \frac{\mathrm{d}\mathcal{J}(\mathbf{x}^*)}{\mathrm{d}\boldsymbol{\alpha}} = \lim_{k\to\infty} \frac{\mathrm{d}\mathcal{J}(\mathbf{x}^k)}{\mathrm{d}\boldsymbol{\alpha}}.$$

Figure 3.5 shows the DAG calculating y from $\mathbf{x}^0$ and $\boldsymbol{\alpha}$ by applying iteration steps f^1, f^2, and f^3. This iteration will more than likely not fully converge to a fixed point, however the derivatives of this partially converged state can still be evaluated according to Equation (3.1). The same expression can be constructed from the DAG by summing up the multiplication of partial derivatives along all paths in the DAG which connect $\boldsymbol{\alpha}$ to y. The summation on the right of Figure 3.5 illustrates this procedure for the given three iteration example. The sum is written to illustrate the pathwise summation, obviously it could be calculated with less operations by using distributivity.

For the converged solution $\mathbf{x}^*$ of a steady case, the value of the final state does not depend on the chosen starting point $\mathbf{x}^0$ anymore. (At least in the sense that $\partial y/\partial \mathbf{x}^0 = 0$. For non-convex spaces, the solution might converge to a local minimum, the location of which depends on the


$\mathcal{P}$
$\mathbf{x}^0$
$\boldsymbol{\alpha}$
$[\frac{\partial f^1}{\partial \mathbf{x}^0}]$
f^1
$\mathbf{x}^1$
$[\frac{\partial f^2}{\partial \mathbf{x}^1}]$
f^2
$\mathbf{x}^2$
$[\frac{\partial f^3}{\partial \mathbf{x}^2}]$
f^3
$\mathbf{x}^3$
$[\frac{\partial \mathcal{J}}{\partial \mathbf{x}^3}]$
$\mathcal{J}$
y

$$
\begin{aligned}
\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} \\
&+ \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3}\frac{\partial f^3}{\partial \boldsymbol{\alpha}} \\
&+ \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3}\frac{\partial f^3}{\partial \mathbf{x}^2}\frac{\partial f^2}{\partial \boldsymbol{\alpha}} \\
&+ \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3}\frac{\partial f^3}{\partial \mathbf{x}^2}\frac{\partial f^2}{\partial \mathbf{x}^1}\frac{\partial f^1}{\partial \boldsymbol{\alpha}} \\
&+ \frac{\partial \mathcal{J}}{\partial \mathbf{x}^3}\frac{\partial f^3}{\partial \mathbf{x}^2}\frac{\partial f^2}{\partial \mathbf{x}^1}\frac{\partial f^1}{\partial \mathbf{x}^0}\frac{\partial P}{\partial \boldsymbol{\alpha}}
\end{aligned}
$$

Figure 3.5: DAG of a three step iteration. The derivative $\mathrm{d}\mathcal{J}/\mathrm{d}\boldsymbol{\alpha}$ can be determined from the graph by multiplying and summing over paths from $\boldsymbol{\alpha}$ to y.

starting point $\mathbf{x}^0$. In the following we assume that a unique solution exists and that it can be found by solving the Navier-Stokes equations iteratively from an arbitrary starting point $\mathbf{x}^0$).

The vanishing influence of $\mathbf{x}^0$ can be derived from the following argument: If the series of functions f^i is contractive (it converges to a fixed point), then the norm of the individual Jacobians $\partial f^i/\partial \mathbf{x}^{i-1}$ is lower than one for any matrix norm: $\left\|\partial f^i/\partial \mathbf{x}^{i-1}\right\| < 1$. The norm of the full derivative accumulated by the chain rule is therefore zero in the limit case:

$$
\lim_{k\to\infty}\left\|\frac{\mathrm{d}f^k}{\mathrm{d}\mathbf{x}^0}\right\| = \left\|\prod_{i=1}^{k}\frac{\partial f^i}{\partial \mathbf{x}^{i-1}}\right\| \leq \underbrace{\left(\max_i \left\|\frac{\partial f^i}{\partial \mathbf{x}^{i-1}}\right\|\right)}_{<1}{}^{k} \overset{k\to\infty}{=\!=} 0\,.
$$

The limit still holds if not every function is contractive, but the number of non-contractive functions is finite.

With the same argument it can be shown that $\mathrm{d}J/\mathrm{d}\boldsymbol{\alpha}$ converges to a fixed point. Assuming convergence of the primal state, both terms $\partial\mathcal{J}/\partial \mathbf{x}^k$ and $\partial f^i/\partial \boldsymbol{\alpha}$ in (3.1) are bounded. Let us denote these bounded terms as $\mathbf{c}_\mathbf{x}$ and $\mathbf{c}_{\boldsymbol{\alpha}}$. Then $\mathrm{d}\mathcal{J}/\mathrm{d}\boldsymbol{\alpha}$ is bounded by

$$
\left\|\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}}\right\| \leq \left\|\mathbf{c}_\mathbf{x}\mathbf{c}_{\boldsymbol{\alpha}}\sum_{i=0}^{k}\prod_{j=i+1}^{k}\frac{\partial f^j}{\partial x^{j-1}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}\right\|.
$$

With $\max_i \left\|\partial f^i/\partial \mathbf{x}^{i-1}\right\| < 1$ the sum is bounded by a geometric series, which is known to converge to a finite limit for $k \to \infty$.

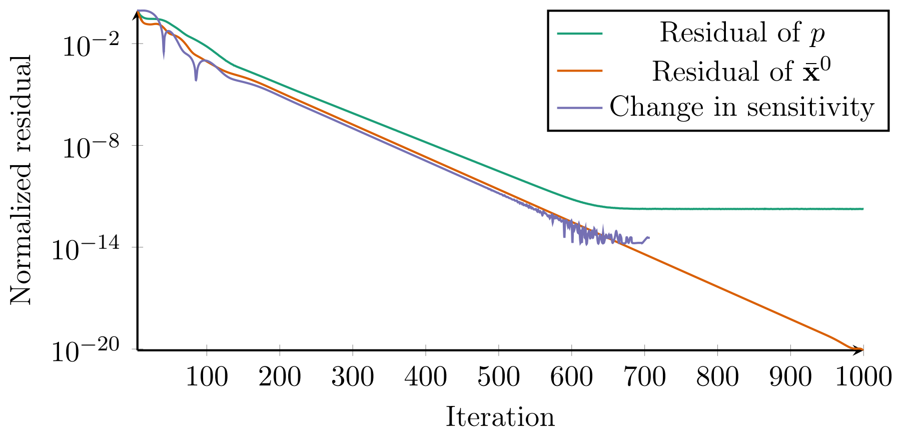


Figure 3.6: Iteration history of the primal and adjoint residuals for the fully converged angled duct case, evaluated over 1 000 iteration steps.

The independence on the starting point implies, that any errors made in the early stages of the procedure do not influence the outcome of the simulation $\mathbf{x}^*$, as long as the iteration scheme is robust enough to still converge to the correct solution $\mathbf{x}^*$ for the perturbed trajectory. The adjoints will accumulate some errors, due to the addition of the partial derivatives along *all* paths from $\boldsymbol{\alpha}$ to y, however those errors are minor, due to $\mathrm{d}y/\mathrm{d}\mathbf{x}^i \ll 1$ for the early stages of iteration. This allows to save some iteration time, by allowing the residuals of the inner linear systems to be higher for iterations which are still distant from the solution, therefore needing less linear solver iterations. In OpenFOAM this concept is called *relative tolerance*. Relative tolerance specifies that instead of solving to a specified absolute tolerance, the equations are solved such that the final residual is reduced by a specified factor from the initial residual. The initial residual will shrink as the outer iteration progresses towards the solution $\mathbf{x}^*$.

The evolution of adjoint $\bar{\mathbf{x}}$ can be used as a convergence criterion, which indicates that the adjoint $\bar{\boldsymbol{\alpha}}$ has converged (alternatively also the absolute change in $\bar{\boldsymbol{\alpha}}$ can be observed directly). The adjoint $\bar{\mathbf{x}}$ holds the derivative $\partial\mathcal{J}/\partial\mathbf{x}^i$ as the adjoint propagation steps backward through the iteration loop from $i = k$ to $i = 1$.

Figure 3.6 shows how $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{x}^i$ and the increments to $\bar{\boldsymbol{\alpha}}$ shrink, as the interpretation progresses backwards through the iteration history. For reference, the residual of the (forward) pressure equation is shown as well. The change in sensitivity after 700 iterations occasionally falls below the output precision and is rounded to zero, breaking the logarithmic scale of the figure. Thus, this curve is omitted after 700 iterations. The results were obtained on the angled duct case shown in Section 3.1.2.

In Figure 3.7 the iteration procedure is stopped with only partially converged primals. When stopping the iteration after only 50 iterations, the residual $\bar{\mathbf{x}}^0$ drops only by a factor of 20, indicating that the solution is not yet sufficiently independent of the starting value. This results in an offset in the adjoints compared to the fully converged case.

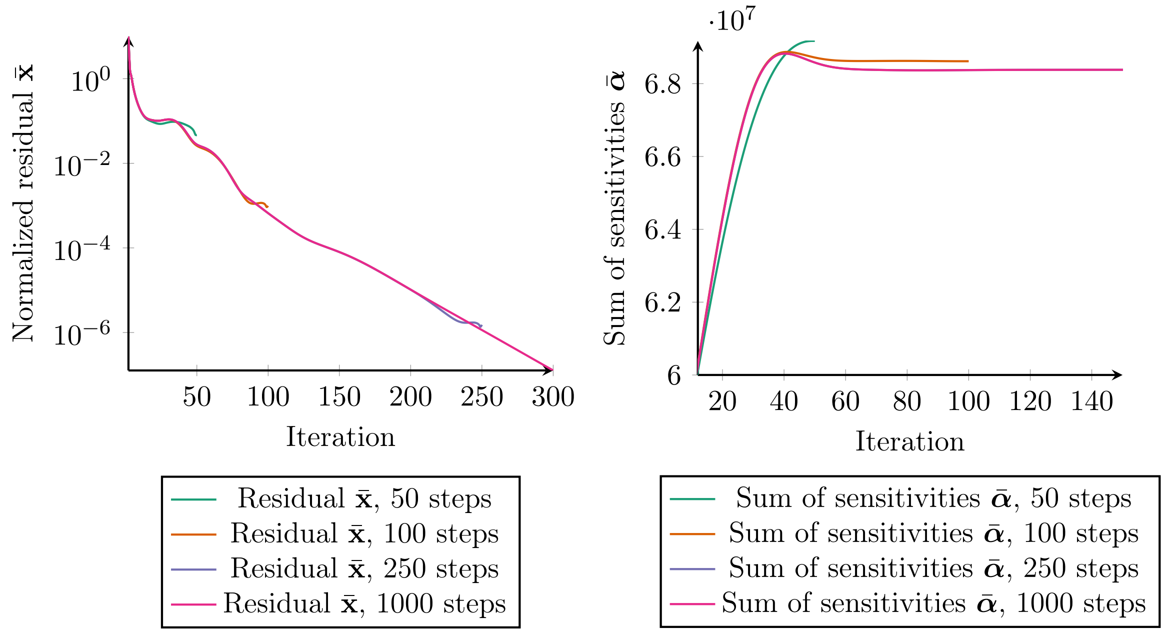


Figure 3.7: Iteration history of the normalized adjoint residual $\bar{\mathbf{x}}$ (left) and the corresponding sum of sensitivities (right). Early termination of the primal iteration leads to high adjoint residual and wrong adjoint sensitivities.

3.2.3 Introduction of AD into the OpenFOAM Code Base

In order to implement AD by operator overloading, all floating point variables which lie on a path in the computational graph between inputs and outputs of the program, have to be replaced with an instance of a different data type, that allows to track the derivatives as well as the primals (see Section 2.7.2). For our goal of differentiating a big software package, such as OpenFOAM, it is hard to know in advance if a variable will influence the derivative of the desired output variables in any way. Furthermore, OpenFOAM, despite being a highly sophisticated `C++` code, that heavily uses templating, is not designed for multiple scalar types to coexist at the same time. Thus, we chose to exchange the type of all floating point values with a `dco/c++` data type. OpenFOAM allows to choose the floating point datatype between `double` and `float`. This is enabled by the following central typedef in `src/OpenFOAM/primitives/Scalar/doubleScalar/doubleScalar.H`, which allows to exchange the data type for all floating point variables.

```
typedef double doubleScalar;
```

At this location we can insert the `dco/c++` data type, for example for first order adjoint mode:

```
typedef dco::ga1s<double>::type doubleScalar;
```

and for first order tangent mode:

```
typedef dco::gt1s<double>::type doubleScalar;
```

The definitions for higher order data types are listed in Section 3.6.

In addition to the type change, certain common modifications to the code are required to circumvent some limitations of the C++ language. One occurrence is the implicit casting of floating point values to integer types. Though considered to be bad style, this is permissible by the C++ standard and is used in the OpenFOAM code occasionally. There is no default implicit cast from the non-primitive dco/c++ data types to primitive types and dco/c++ also does not provide them by copy constructors or assignment operators, due to the potential of inadvertent (derivative) data loss. Thus, the passive floating point value needs to be extracted from the dco/c++ data type manually first, and can subsequently be implicitly or explicitly cast to an integer.

```
dco::ga1s<double>::type x = 42;
// fails: int i = x;
int i = static_cast<int>( dco::passive_value(x) );
```

The C++ implicit conversion rules allow for the following conversions in that order:

1. Zero or one standard conversion sequence,
2. zero or one user-defined conversion,
3. and zero or one standard conversion sequence.

This can become problematic, when the chain of type conversions becomes elongated by the additional complexity of the dco/c++ data types. For example, the conversion chain `double -> Foam::scalarField -> Foam::dimensionedScalarField` is permissible. However, the chain `double -> scalar -> Foam::scalarField -> Foam::dimensionedScalarField` is not, because the user-defined conversion is already used up by the implicit conversion from the double literal to the (dco/c++) scalar type and is not available for the conversion from `scalarField` to `dimensionedScalarField` anymore. Thus, the compiler has to be explicitly told that the double literal is supposed to be a scalar. This problem, and its solution, is also illustrated in the following listing. One can construct an instance of class `B` from a primitive `double` data type with implicit conversion `double -> A -> B`, however the conversion `double -> ScalarType -> A -> B` is not possible.

```
class ScalarType{
  double val;
  ScalarType(double val) : val(val){}
}; typedef ScalarType scalar;

class A{ A(scalar d){} };
class B{ B(A a){} };

int main(){
  // ok:    B(42.0); with scalar=double
  // fails: B(42.0); with scalar=ScalarType
  B(ScalarType(42.0)); // correct for both types
}
```

Listing 3.1: Example code for failing implicit type conversion. The literal 42.0 has to be passed as `ScalarType` to the constructor of `B` explicitly.

In our code base the transformation of implicit to explicit conversions was done by hand. There are tools which strive to identify and fix those issues by automatic source code transformation [HUB16]. However, the result should still be checked manually, such that no inadvertent loss of derivative information occurs. When deriving new software with AD in mind, explicit type conversions should be used as much as possible, to avoid type ambiguity and overlong type conversion chains.

Non-Differentiable Functions within OpenFOAM

The only function encountered within the OpenFOAM code base which triggers a floating point exception, due to evaluation at a non differentiable point, is the square root function at location zero. Evaluating the square root function at location zero leads to a division by zero when evaluating the derivative.

$$y = \sqrt{x} \quad \Rightarrow \quad \frac{\partial y}{\partial x} = \frac{1}{2\sqrt{x}}$$

This case is not commonly triggered but e.g. occurs if and only if calculating the L_2 norm of a vector of size zero. To sidestep this problem, we replace the derivative of the square root function by a version which adds a small $\epsilon > 0$ to the independent when the non differentiable point of the square root is hit.

$$y = \sqrt{x} \quad \Rightarrow \quad \frac{\partial y}{\partial x} := \begin{cases} \frac{1}{2\sqrt{x}} & x > 0 \\ \frac{1}{2\sqrt{x+\epsilon}} & x = 0 \end{cases}$$

Thus, at $x = 0$ the derivative evaluates to a very high number, mimicking the behavior of FD and approximating the right-sided limit of the square root function:

$$\lim_{x \to 0^+} \frac{1}{2\sqrt{x}} = \infty\,.$$

Similar approaches are also taken by the primal OpenFOAM implementation to sidestep issues in the evaluation of turbulence functions. We have not observed any influence of the final sensitivities on the choice of ϵ, leading us to believe that either the adjoints get propagated only into branches of the DAG which are not actually connected to the parameters, or that at least one of the partial derivatives on the path, connecting the parameters to the output, containing the calculation of $\sqrt{0}$, is zero.

3.2.4 Black-Box Differentiation of `simpleFoam` Solver

By introducing AD to the computational kernels of OpenFOAM, adjoint solvers can be developed on a high abstraction level. No particular insight into the low level implementation is required.

As an example the implementation of `adjointSimpleFoam` will be presented. This solver is based on the regular OpenFOAM steady, incompressible, `simpleFoam` solver. It enables the calculations of gradients w.r.t. the design parameters $\boldsymbol{\alpha}$ required for topology optimization. The `simpleFoam` solver implements the SIMPLE algorithm, as presented in Section 2.3. Further it also implements the faster converging SIMPLEC [VR84] algorithm, which can be enabled optionally. In the following a brief overview over the passive solver is given, so that the changes required for the adjoint versions become obvious.

The `simpleFoam` solver implements the iterations of the SIMPLE algorithm using a while loop, which will run until the maximum allowed iterations have been performed or previously specified convergence criteria have been met. Inside the loop body, the equation systems linearizing the Navier-Stokes equations are assembled and solved. The momentum equations are implemented in the external file `UEqn.H`, the pressure correction equations in `pEqn.H`. Those files are inlined by the `C`-preprocessor into the loop body at compile time (include directives inside function declarations are rarely used in `C/C++` code, but commonly occur within the OpenFOAM code).

```cpp
int main(){
  #include "createMesh.H"
  #include "createFields.H"

  while (simple.loop()){
    // --- Pressure-velocity SIMPLE corrector
    #include "UEqn.H"
    #include "pEqn.H"

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();
  }
}
```

Listing 3.2: Implementation of the main iteration loop of `simpleFoam`.

The implementation of the momentum equations (Listing 3.3) showcases the high abstraction level and object oriented design of OpenFOAM. The differential operators occurring in the Navier-Stokes equations (compare to Section 2.1) are directly visible in the assembly of the system matrix `UEqn`. The divergence operator $\nabla \cdot (\phi\, \mathbf{u})$ is discretized by `fvm::div(phi, U)`, the Laplacian $\nu \boldsymbol{\nabla}^2 \mathbf{u}$ by `fvm::laplacian(nu, U)`.

The right hand side consisting of the pressure gradient ∇p is build by `-fvc::grad(p)`. Note, that this is a simplified implementation for laminar flows. The actual implementation includes additional terms to model turbulence, which we omit here for clarity.

```cpp
  fvVectorMatrix UEqn
  (
      fvm::div(phi, U)
    - fvm::laplacian(nu, U)
    ==
    - fvc::grad(p)
  );
  UEqn().relax();
  fvOptions.constrain(UEqn());
  UEqn.solve();
  fvOptions.correct(U);
```

Listing 3.3: Implementation of the momentum equations in `UEqn.H`.

The pressure correction equations, as well as the momentum correction, are implemented in `pEqn.H`, shown in Listing 3.4. The notation is not as intuitive as for the momentum equations, however the general structure of the pressure correction equation is still visible in Line 11 of the listing.

```
volScalarField rAU(1.0/UEqn().A());
volVectorField HbyA("HbyA", U);
HbyA = rAU*UEqn().H();

surfaceScalarField phiHbyA("phiHbyA", fvc::interpolate(HbyA) & mesh.Sf());
MRF.makeRelative(phiHbyA);
adjustPhi(phiHbyA, U, p);

fvScalarMatrix pEqn
(
  fvm::laplacian(rAU(), p) == fvc::div(phiHbyA)
);

pEqn.setReference(pRefCell, pRefValue);
pEqn.solve();

phi = phiHbyA - pEqn.flux();

p.relax();
// Momentum corrector
U = HbyA - rAtU()*fvc::grad(p);
U.correctBoundaryConditions();
fvOptions.correct(U);
```

Listing 3.4: Implementation of the pressure correction equation in `pEqn.H`.

The penalty term for the topology design parameters is introduced to the solver by adding the component-wise product of $\boldsymbol{\alpha}$ with the velocities $\mathbf{U}$ as a source to the momentum equations. This is implemented by modifying the entries on the diagonal of the system matrix with `fvm::Sp(alpha, U)` (Listing 3.5). The function `fvm::Sp` accepts an implicit source term with positive contribution; thus the introduction of $\boldsymbol{\alpha}$ is not lowering the diagonal dominance of the system matrix. Alternatively, the resistance term could also be implemented by adding the term to the right hand side of the equation system as an explicit source term with `-fvc::Su(alpha, U)`.

```
fvVectorMatrix UEqn
(
    fvm::div(phi, U)
  - fvm::laplacian(nu, U)
  + fvm::Sp(alpha, U)
  ==
  - fvc::grad(p)
);
```

Listing 3.5: Implementation of the source term in `adjointSimpleFoam`.

For the black-box differentiation through the whole SIMPLE iteration history, no additional changes to the main iteration loop are required. Next, the steps required to seed and obtain the adjoints are added. Before entering the main loop, the tape data structure of `dco/c++` is initialized. The parameters $\boldsymbol{\alpha}$ are registered as inputs, allowing `dco/c++` to optimize the tape by applying varied analysis [HNP05]. The tape is then recorded while executing the iteration loop. After the loop finishes, the value of the objective $\mathcal{J}$ is calculated. The calculation of the objective function is implemented in a separate library which allows to calculate basic objective functions like power loss, drag, and lift. The adjoint of the objective is then seeded with one (the

1D unit vector, for parallel processing only on the root node) and the adjoint reverse propagation process is started. After the propagation has finished, the desired gradient of the objective w.r.t. the parameters $\boldsymbol{\alpha}$ is available in the adjoints of `alpha` and can be extracted and written to the sensitivity field `sens`.

```
int main(){
  #include "createMesh.H"
  #include "createFields.H"

  dco::a1s::global_tape = dco::a1s::tape_t::create();
  dco::a1s::global_tape->register_variable(alpha.begin(),alpha.end());

  while (simple.loop()){
    // Pressure-velocity SIMPLE corrector
    #include "UEqn.H"
    #include "pEqn.H"

    laminarTransport.correct();
    turbulence->correct();

    runTime.write();
  }

  scalar J = CostFunction(mesh).eval();

  if(Pstream::master())
    dco::derivative(J)=1;
  dco::a1s::global_tape->interpret_adjoint();

  forAll(alpha,i)
    sens[i] = dco::derivative(alpha[i]);
  sens.write()
}
```

Listing 3.6: Main iteration loop of `adjointSimpleFoam`

Note the usage of the `forAll` macro in the above code, defined by OpenFOAM as a helper to iterate through a generic list-like data structure. It only simplifies the creation of the loop counter ranging from zero to the size of the list, it does not create an iterator on the supplied list (for this the `forAllIters` macro exists, however this macro is rarely used). The straightforward definition of the macro is given below.

```
  //- Loop across all elements in a list
  #define forAll(list, i) \
    for (Foam::label i=0; i<(list).size(); ++i)
```

Listing 3.7: Definition of the `forAll` macro in `stdFoam.H`

The use of this macro is encouraged by the OpenFOAM coding style guide [SG18], and therefore it will appear in some of the coming listings.

The above implementation of a black-box solver is the simplest conceivable implementation of discrete adjoints using operator overloading, without the exploitation of further knowledge about the problem. Such optimizations will be discussed in future sections. Building on the calculation of sensitivities, an additional optimization loop can be placed on the outside of this code to optimize the parameters e.g. using steepest descent.

3.3 Checkpointing

The black-box differentiation of complex programs poses challenges, if the amount of RAM required to store the tape data structure outgrows the available physical memory. For iterative problems, an effective approach to lower the memory footprint is to incorporate recomputation, trading off lower memory usage against increased run time. A systematic approach to recomputation techniques is known as checkpointing [GW08] and is introduced in the following.

3.3.1 Introduction to Checkpointing

Checkpointing is a technique commonly used to lower the memory footprint needed to adjoin complex programs. It uses the deterministic nature of computer programs, which allows to restore the exact state of a program from the state at some earlier (or initial) point in the execution history.

Checkpointing involves a memory vs. run time trade off. The storage of all intermediate values needed for the reversal process is replaced by storing only selectively, and recomputing the missing values when they are needed. To avoid having to completely restart the program to generate those missing values, intermediate states of the program (*checkpoints*) are stored (either in RAM or on disk), which allow to restore the state of the program at an intermediate step and resume it from there. For the common occurrence of an iterative computation, embedded inside a main loop, it is practical to only record a small number of loop steps, adjoin them, and then resume the program from an earlier state in order to record the missing loop steps. This process can be repeated recursively, until all loop steps have been adjoined.

Figures 3.8 and 3.9 illustrate the basic checkpointing procedure for four iterations. In the first figure only one checkpoint is placed before the first iteration, saving the initial state $\mathbf{x}^0$. This allows to recreate state $\mathbf{x}^i$ of the simulation at any iteration i, by recalculating the state from the initial state $\mathbf{x}^0$. A single iteration step is adjoined at a time. Thus, only the partials of one step need to be stored at a time, instead of four. As the last iteration step is adjoined first, the previous (third) iteration step can not be immediately adjoined afterwards. The state $\mathbf{x}^2$, needed to execute the augmented primal iteration f^3, is not available in memory. State $\mathbf{x}^2$ must therefore be recomputed by restoring the only available state $\mathbf{x}^0$, and executing (in passive mode) iterations f^1 and f^2. Only then can iteration f^3 be executed in augmented forward mode. After the required intermediate values of this iteration step are available in memory, the adjoints obtained from adjoining the first iteration step (f^4) can be fed back as an input to the calculation of the adjoints for iteration step f^3. In the figure, this feedback of adjoints is indicated by right facing arrows The procedure is repeated for iteration steps two and one, requiring additional recomputations of the states $\mathbf{x}^2$ and $\mathbf{x}^1$. After all iteration steps have been adjoined, the data flow reversal can be finished by adjoining the pre-processor. The approach to only checkpoint the minimal amount of information necessary and recompute all other information is called *recompute all* approach.

The amount of passive recomputation can be minimized, by not only checkpointing the initial

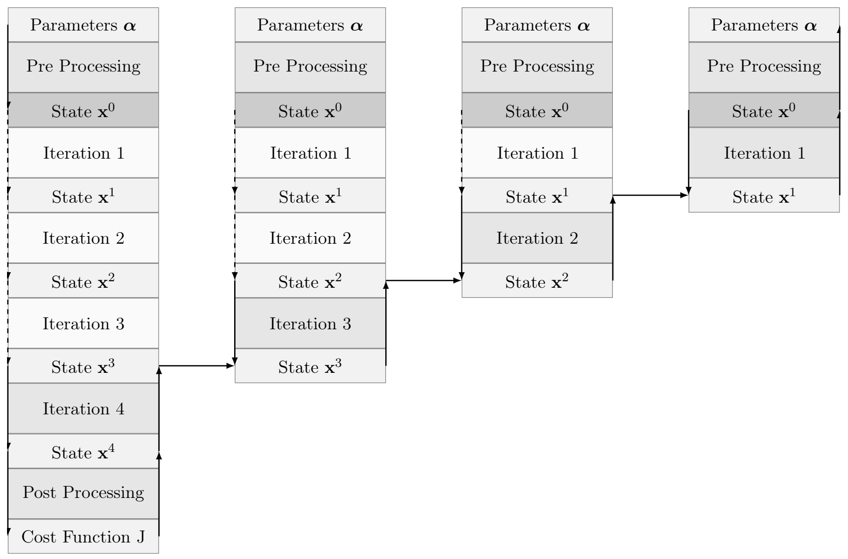


Figure 3.8: Reversal of iteration history with only one checkpoint for state $\mathbf{x}^0$. All other states have to be recomputed(*recompute all approach*).

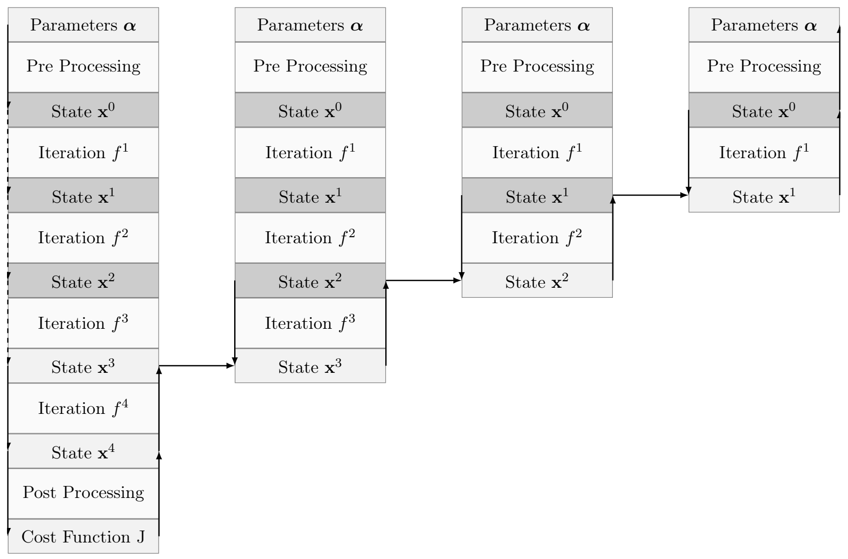


Figure 3.9: Reversal of iteration history with checkpoints for states $\mathbf{x}^0, \mathbf{x}^1$, and $\mathbf{x}^2$. This allows to reverse all iterations without any extra recomputation (*checkpoint all approach*).

state $\mathbf{x}^0$, but also all other states. This allows to immediately start the necessary forward evaluations of the iteration steps, without performing passive recomputations to advance the iteration state first. For the example in Figure 3.9, those states are $\mathbf{x}^0, \mathbf{x}^1$, and $\mathbf{x}^2$. Checkpoints for $\mathbf{x}^3$ and $\mathbf{x}^4$ are not required, as they would never be restored (the post processor is adjoined in conjunction with the last iteration step immediately after executing the three passive forward evaluations). The adjoints of state $\mathbf{x}^3$ and $\mathbf{x}^4$ are available after the first interpretation step and thus a checkpoint for $\mathbf{x}^3$ is not required. The approach to store all possible checkpoints is called *checkpoint all* approach.

For the general case of k iteration steps, the cost of adjoining the entire iteration history using the recompute all and checkpoint all schemes are as follows. Adjoining one iteration step at a time the *checkpoint all* approach requires the augmented forward and reverse evaluation of k iteration steps, as well as the (passive) forward computation of $k-1$ iteration steps to reach the first active step. The *recompute all* approach needs the same k active evaluations. In addition it needs

$$(k-1)+(k-2)+\ldots+1=\frac{k(k-1)}{2}$$

passive iterations to recover the state from the initial state for every reverse evaluation step. Therefore $((k-1)\cdot(k-2))/2$ additional passive iterations are needed, compared to the *checkpoint all* approach. It is clear that the behavior of *recompute all* with a run time factor of $\mathcal{O}(k)$ compared to the black-box evaluation is undesirable. *Checkpoint all* has an attractive run time factor of $\mathcal{O}(1)$, compared to black-box (assuming passive and active steps are of same run time cost, the factor is lower than two. In practice it will be lower), however the number of checkpoints which can be stored is limited by the available virtual or physical storage space. Thus, in practice a checkpointing scheme is chosen, that aims to minimize the calculation time, while limiting the number of checkpoints to still fulfill the memory constraints.

The question of how to optimally place a fixed number of checkpoints, in an evolution of arbitrary function calls with known cost, is a combinatorial NP-hard problem [Nau08]. A special case is the application to iterative methods, where checkpoints can be placed between each iteration, each iteration is assumed to have equal cost in terms of run time and memory (or at least the cost is quantifiable a priori), and where the number of iterations performed is known beforehand. For those assumptions, provably optimal spacings can be given without solving an optimization problem first. Such a provably optimal spacing is given by the *revolve*-algorithm [GW00; GW08].

For the reversal of the SIMPLE algorithm in discrete adjoint OpenFOAM, we first considered equidistant checkpointing. Intermediate steps of the problem are stored at a fixed distance and the checkpoint locations remain constant for the whole program execution. This simplifies the implementation significantly, but ignores the possibility to reuse checkpoints once they are no longer needed, because all iteration states reachable from the checkpoint have already been adjoined.

The discrete adjoint OpenFOAM package also implements binomial checkpointing in the form of *revolve* algorithm. The *revolve* algorithm places checkpoints with a logarithmic spacing, making sure that checkpoints are spaced more densely next to the current interpretation step, minimizing the number of recomputation steps. Checkpoint locations are reused once they are not needed at their former position anymore.

Figure 3.10 shows the application of *revolve* algorithm to 100 iteration steps of the SIMPLE algorithm for different numbers of checkpoints. For example, for three checkpoints, *revolve* places

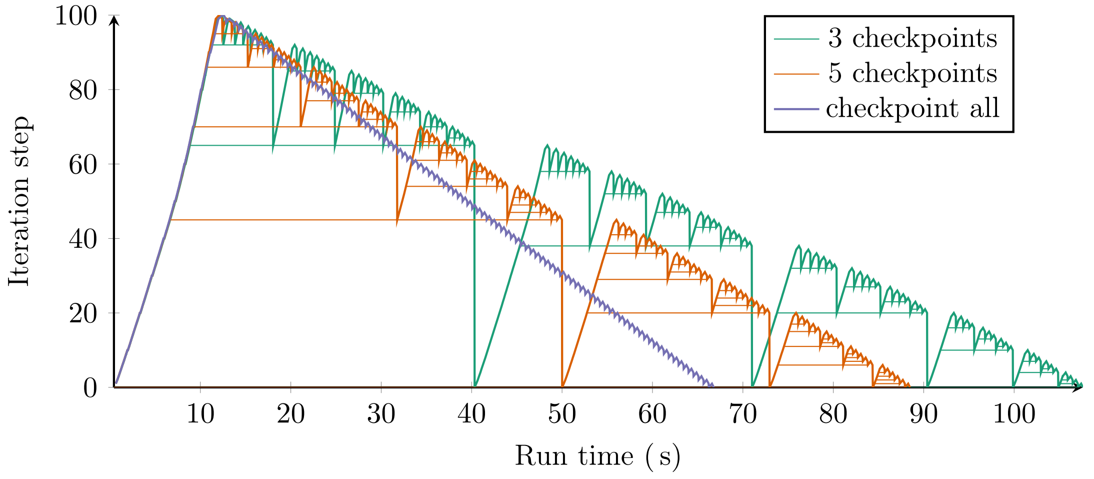


Figure 3.10: Calculation and reversal of 100 iteration steps with 100 (checkpoint all), five, and three *revolve* checkpoints. One iteration step is recorded at a time. Horizontal lines indicate the evolution of the checkpoint positions during the reversal procedure.

checkpoints at iteration steps $(0, 65, 92)$. The upper checkpoint is consecutively lowered until iteration step 65 has been reached and the center checkpoint can be replaced at a lower location. The checkpoints at iteration 0 is restored and is used to place the new checkpoints at $(0, 38, 58)$. This process is repeated until all iteration steps have been recorded and adjoined.

All checkpointing by default is done on the outer level of the iteration loop, therefore at least one complete iteration step has to fit into RAM. With some implementation effort, this can be transformed to capture checkpoints in between different PDEs (e.g. for the SIMPLE algorithm to place checkpoints between the momentum, mass conservation and turbulence equations). Further breaking it down to inside the PDE solver level is more challenging, due to the strict scoping of C++. The automatic placement of checkpoints at arbitrary positions in the program is currently a development target for dco/c++. Theoretical groundwork has already been laid in [Lot16]. In practice we have observed that problems which would require such high amounts of RAM profit immensely from being distributed to different MPI nodes (see Chapter 5), making more granular checkpoints rarely necessary.

For rising number of checkpoints, the difference between equidistant and binomial checkpointing vanishes. In the limit case, where every iteration step can be checkpointed, binomial checkpointing naturally cannot improve over an equidistant scheme anymore. The number of required recomputations for equidistant and revolve checkpointing over the number of placed checkpoints is illustrated in Figure 3.11.

3.3.2 Implementation of Checkpointing in Discrete Adjoint OpenFOAM

One requirement for the application of checkpointing is the separability of the main iteration loop, that is the solver has to be implemented such that the iteration procedure can be run for a specified number of iterations from an arbitrary state.

The stock OpenFOAM solvers are not structured to easily achieve this. All iterations are placed in a main loop and all flow fields are scoped locally to the main routine of the solver. Therefore a

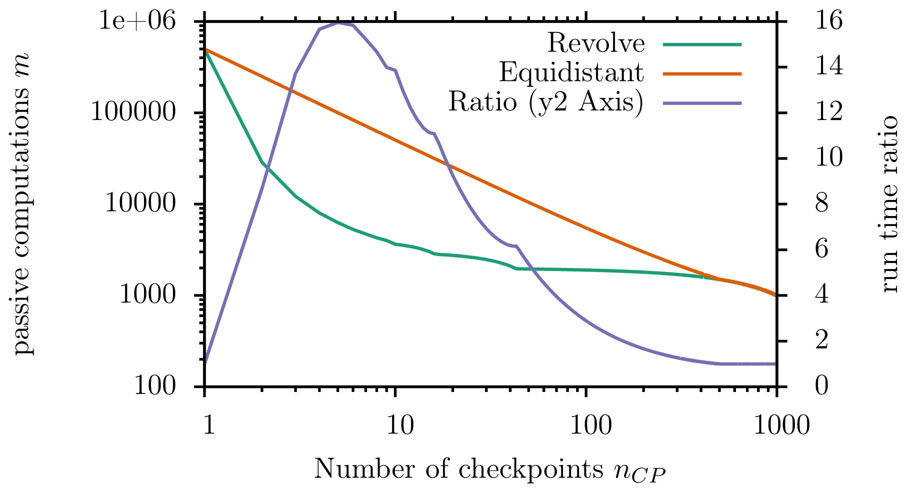


Figure 3.11: Number of required recomputations m for checkpointing 1000 iteration steps for increasing number of checkpoints n_{CP}. Run time ratio between equidistant and *revolve* in green on second y-axis.

wrapper around the `adjointSimpleFoam` solver was created, moving the flow fields into a class and separating the individual loop iterations into a member function. This function can be called to advance the simulation by a single iteration step. Additionally this class implements abstract methods from a `CheckController` class, which is designed to make the checkpointing interface applicable to a variety of solvers by defining a common set of routines to be implemented by every checkpointed solver. The separation of iteration steps is also useful for interfacing with external optimizer packages, which require the repeated evaluation of the primal simulation or gradients.

The checkpointing functionality is implemented in the discrete adjoint OpenFOAM framework by an abstract interface, from which the different checkpointing strategies are derived. The following functionality is provided by the checkpointing interface:

Store checkpoint: Store the primal values of the required flow fields $\mathbf{x}$ into a temporary buffer $\mathbf{B}_i \in \mathbb{R}^{n_\mathbf{x}}$.

Restore checkpoint: Overwrite the primal values of the flow fields $\mathbf{x}$ with the content of buffer $\mathbf{B}_i \in \mathbb{R}^{n_\mathbf{x}}$.

Register variables: Register the variables of the state $\mathbf{x}$ in the tape and remember the assigned tape indices in $\boldsymbol{\tau} \in \mathbb{N}^{n_\mathbf{x}}$.

Store adjoints: Extract the adjoints from the tapes adjoint vector (from the locations stored during the register variables step) and store them in a temporary buffer $\mathbf{T} \in \mathbb{R}^{n_\mathbf{x}}$.

Restore adjoints: Inject the stored adjoints from the buffer $\mathbf{T}$ back into the tape at the locations currently occupied by $\mathbf{x}$.

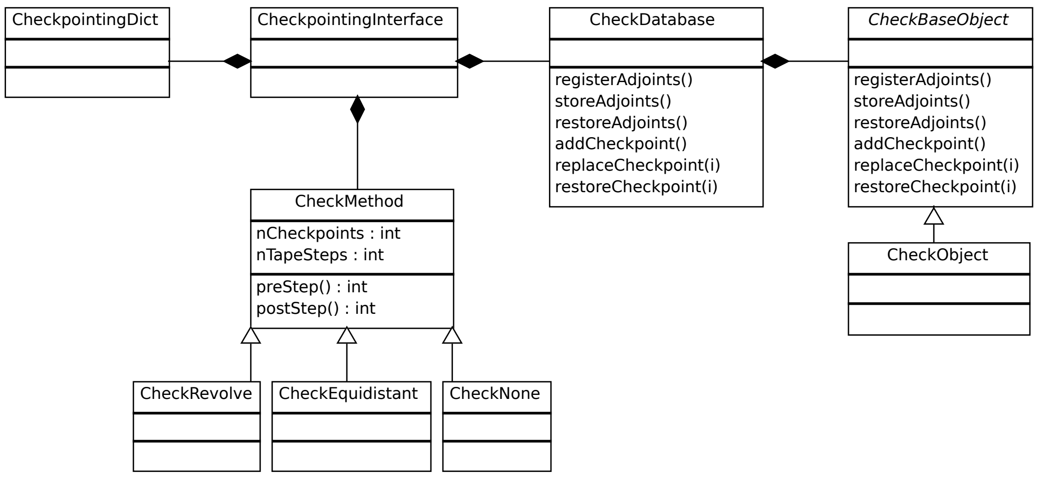


Figure 3.12: Class diagram of the checkpointing interface.

Figure 3.12 shows the class structure of the checkpointing interface, developed for discrete adjoint OpenFOAM. It allows to checkpoint instances of `geometricField`, which is a base class of `volScalarField` (e.g. used for pressure), `volVectorField` (e.g. for velocity), `surfaceScalarField` (e.g. face fluxes), and `surfaceVectorField` (e.g. surface normal vectors). In addition also generic scalar data can be checkpointed. Currently the checkpointing strategies equidistant (`CheckEquidistant`) and *revolve* (`CheckRevolve`) are implemented as specializations of the abstract `CheckMethod` base class. For verification, a dummy class (`CheckNone`) which implements black-box differentiation is implemented as well.

The memory size occupied by n_{CP} checkpoints is

$$M_{CP} = \underbrace{(n_{CP}+1)\cdot n_{\mathbf{x}}\cdot \texttt{sizeof(double)}}_{n_{CP}\text{ checkpoints + one adjoint buffer}} + \underbrace{n_{\mathbf{x}}\cdot \texttt{sizeof(Foam::label)}}_{\text{tape index store}}, \tag{3.2}$$

where `Foam::label` is a typedef for the data type used for integer data. Assuming that 64 bit `double` and `long` types are used, the memory requirement becomes

$$M_{CP} = (n_{CP}+2)\cdot n_{\mathbf{x}}\cdot 8\,\text{bytes}\,.$$

The size of the tape index storage can be reduced from $\mathcal{O}(n_{\mathbf{x}})$ to $\mathcal{O}(1)$ by using the fact that all variables of a field are registered consecutively, and thus the tape indices of a field are adjacent and deterministic.

Figure 3.13 shows the run time of the `adjointSimpleCheckpointingFoam` solver for adjoining all iteration steps of the SIMPLE algorithm on the angled duct example. Observed are two, five, and 100 checkpoints and iteration ranges from $[0,1]$ to $[0,100]$. The curves for 100 checkpoints, which correspond to a checkpoint all approach, show that the run time does not increase linearly. This is due to the decreasing cost of iteration steps near to the converged solution, as the embedded linear solvers need less iterations to converge to their tolerance limit.

In OpenFOAM, the ratio between tape cost and checkpoint cost is usually very high, making it feasible to store a high number of checkpoints. This is demonstrated by Table 3.4 and Figure 3.14.

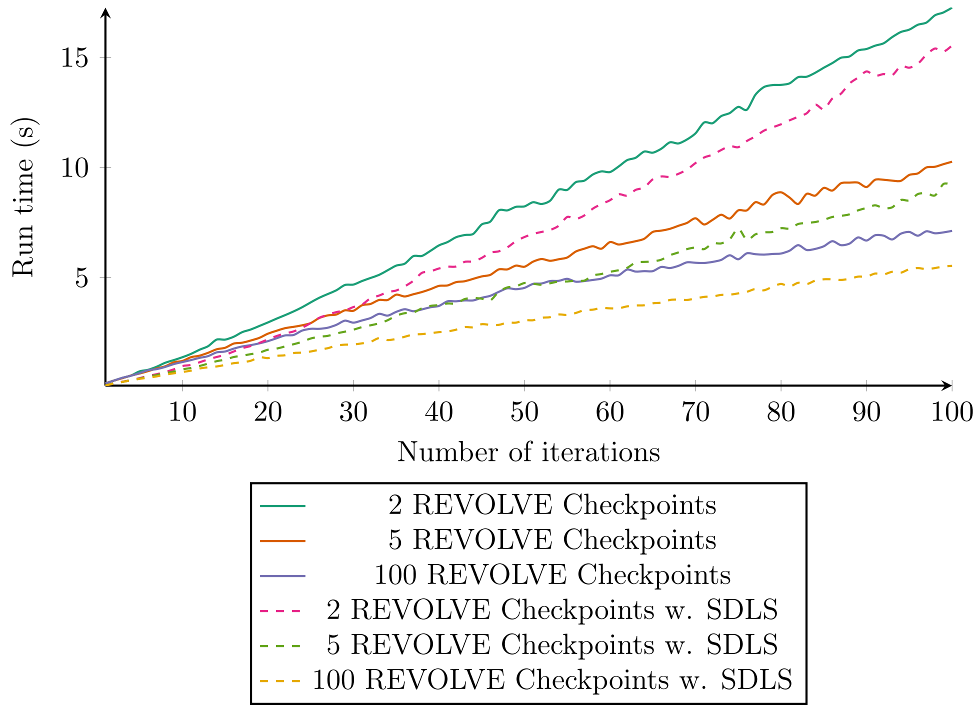


Figure 3.13: Run time over number of calculated iterations `adjointSimpleCheckpointingFoam` for varying density of checkpoints. Dashed lines indicate symbolically differentiated linear solvers.

Table 3.4: Checkpoint size and average tape size with standard deviation σ for one SIMPLE iteration step.

Case	Checkpoint size	Tape size SDLS		Tape size black-box	
	(MB)	Avg. (MB)	σ (MB)	Avg. (MB)	σ (MB)
Pitz-Daily	1.74	442.55	0.57	2282.19	167.127
Angled duct (Lvl. 6)	1.11	232.08	0.19	990.05	134.39
Angled duct (Lvl. 12)	4.36	923.51	0.81	3811.86	546.49

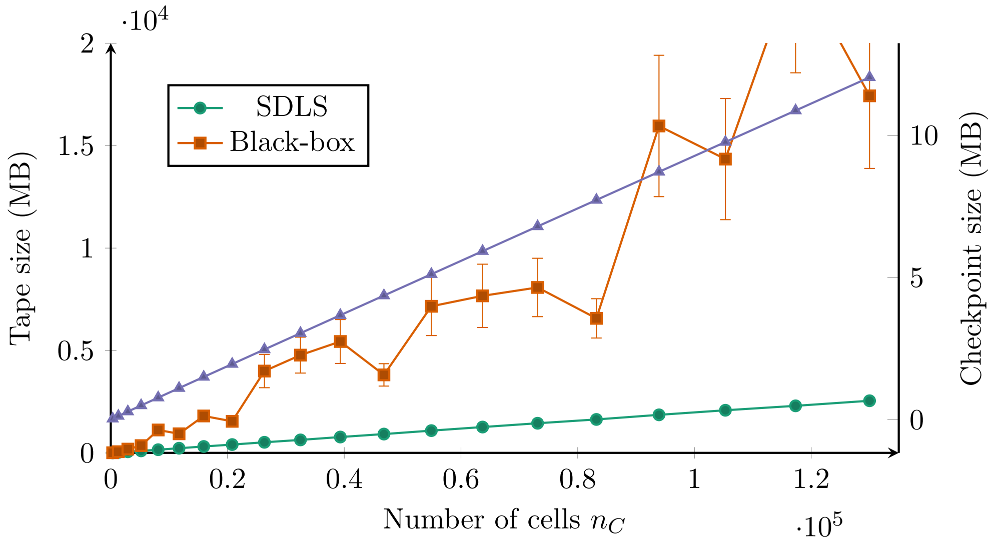


Figure 3.14: Average tape size (left y-axis) and checkpoint size (right y-axis, blue curve) for the angled duct case of different mesh resolutions.

The figure shows both the checkpoint and average tape size for 20 SIMPLE steps of the angled duct example, for varying mesh densities. The coarsest possible configuration for this case consists of 325 cells. From this configuration, 20 different levels of mesh refinement are created, ranging up to 130 000 cells.

The checkpoint size grows linearly with n_C, as predicted by Equation (3.2). With symbolically differentiated linear solvers (SDLS, see Section 3.4), the tape size also grows linearly. The tape size largely remains constant over all iterations of the SIMPLE algorithm. Using black-box differentiation of the linear solvers, the tape size becomes less predictable, due to the varying number of inner linear solver iterations. Thus, in the figure we also give error bars and the standard deviation of the tape sizes. On average, the tape size still grows roughly linearly with n_C, albeit with a much bigger factor.

The table includes the angled duct case for refinement levels 30 (11 700 cells) and 60 (46 800 cells). For both configurations, the ratio between tape size (utilizing SDLS) for one SIMPLE iteration step and one checkpoint is roughly 210. For the Pitz-Daily case with 12 225 cells, the ratio is even higher, with roughly 250. This is due to the additional complexity introduced by the differentiation of the k-ϵ turbulence model.

Thus, binomial checkpointing only reaches its full potential for cases with many iteration steps, that can not be covered well by equidistant checkpoints. Such cases are e.g. generated by transient flow simulations with high Reynolds numbers, which require a very small time step Δt to converge reliably.

3.3.3 Verification of the Checkpointing Implementation

One requisite for a successful application of checkpointing is that all variables belonging to the states of the iteration loop are correctly identified. All variables which

- are overwritten inside of the iteration,
- are scoped outside of the iteration,
- and have an influence on the final outcome of the calculation

need to be included in the checkpoint. The obvious first step to check the correctness of the checkpointing implementation is to compare the results to results obtained by black-box differentiation. However, if the results are incorrect, further insight into the process is needed, to determine which part of the calculation is incorrectly handled.

With only minor modifications to the solver and AD tool, it can be verified if all required state variables have been identified or if any are missing. Every iteration step has to be self contained, i.e. the operations inside of an iteration step are only allowed to depend on the primal values of the current state, variables which are local to the current iteration, or globally defined variables that are never overwritten (e.g. the parameters $\boldsymbol{\alpha}$). Any dependency to previous iterations has to be localized, that is added to the state, by the checkpointing routines. This makes sure that when resuming from a checkpoint, all required values are available (in scope) and have the right numeric value.
The AD tool can be adapted to identify edges in the tape, obtained by the black-box solver, which point to regions which will become illegal when checkpointing is applied and the order of loop execution becomes non-consecutive. A set of legal and illegal edges in the tape is illustrated in Figure 3.15. This technique was used to identify an issue in the checkpointing of shape adjoints. This will be discussed in Section 4.4.

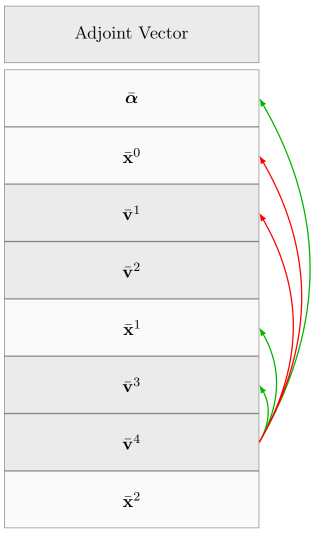


Figure 3.15: Legal (green) and illegal edges (red) for checkpointing. Edges are allowed to point to the global optimization parameters, to entries local to the iteration, and to entries of the directly preceding state. All other edges are illegal.

3.4 Symbolic Differentiation of Embedded Linear Solvers

Typically, the run time of PDE solvers is heavily dominated by the execution of linear equation solvers (compare to Chapter 5), which repeatedly solve the linearized and discretized differential equations. While differentiating all linear solvers with AD technically works, it is in practice undesirable, as the black-box differentiation of certain classes of linear solvers exhibit numerical instabilities [Chr18]. Furthermore, it is very costly to fully differentiate those solvers, especially in terms of memory required for the storage of the tape. For iterative solvers, the cost is strongly correlated with the condition number of the linear system. The cost grows linearly with the number of iterations required by the solver, which in turn grows with the condition number.

We will now outline how the adjoints of linear systems can be obtained, without fully differentiating the solution process by AD. When calculating the solution $\mathbf{x}$ to the linear system $A\mathbf{x} = \mathbf{b}$ without applying AD techniques, the adjoints $\bar{A}$ and $\bar{\mathbf{b}}$ are not propagated automatically. However, using the known adjoints of the solution $\bar{\mathbf{x}}$, the adjoints of A and $\mathbf{b}$ can be calculated symbolically [Gil08; Nau+15].

3.4.1 Symbolic Adjoint Relations for Linear Systems

Theorem 4 (Calculation of adjoints of RHS vector $\bar{\mathbf{b}}$).
For a regular matrix $A \in \mathbb{R}^{n\times n}$, vectors $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}^n$, and $\boldsymbol{x} = A^{-1}\boldsymbol{b}$, the adjoints of $\boldsymbol{b}$ are given by $\bar{\boldsymbol{b}} = A^{-T}\bar{\boldsymbol{x}}$.

Proof. See [Gil08; Nau+15]. □

Instead of explicitly calculating the transposed inverse A^{-T} of A, the adjoints of the right hand side can be calculated at cost $\mathcal{O}(n^3)$ (assuming dense matrices) by solving the equivalent linear equation system

$$A^T \cdot \bar{\mathbf{b}} = \bar{\mathbf{x}}\,. \tag{3.3}$$

This is particularly beneficial for sparse systems, as the inverse of a sparse matrix is in general not sparse. For the sub-case of symmetric matrices and direct linear system solvers, an existing (LU,QR or Cholesky) factorization of A from the primal solve $A\mathbf{x} = \mathbf{b}$ might be reused, at the cost of storing the possibly more dense factorization instead of A. This lowers the complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ for the forward and backward substitution of the factorization.

Theorem 5 (Calculation of matrix adjoints $\bar{A}$).
For a regular matrix $A \in \mathbb{R}^{n\times n}$, vectors $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}^n$ and $\boldsymbol{x} = A^{-1}\boldsymbol{b}$, the adjoints of the individual matrix entries A are given by the outer product $\bar{A} = -\bar{\boldsymbol{b}} \otimes \boldsymbol{x}^T$.

Proof. See [Gil08; Nau+15]. □

The adjoints $\bar{A}$ and $\bar{\mathbf{b}}$ can thus be calculated by solving the additional equation system (3.3) and evaluating an outer product. We assume that the memory locations of A and $\mathbf{b}$ are not aliased and thus A and $\mathbf{b}$ are truly independent.

To summarize, the discrete AD differentiation of the linear system solvers $\mathbf{x} = \mathcal{S}(A, \mathbf{b})$ is replaced by the following steps:

During primal solution:

$$A \cdot \mathbf{x} = \mathbf{b} \quad \rightarrow \quad \mathbf{x} := \mathcal{S}(A, \mathbf{b}) \; ;$$

During reverse propagation of adjoints:

$$A^T \cdot \bar{\mathbf{b}} = \bar{\mathbf{x}} \quad \rightarrow \quad \bar{\mathbf{b}} := \mathcal{S}(A^T, \bar{\mathbf{x}}) \tag{3.4}$$

$$\bar{A} := -\bar{\mathbf{b}} \otimes \mathbf{x}^T \, . \tag{3.5}$$

In the context of CFD, the linear solvers are embedded into a non-linear (iterative) calculation. The matrix A is usually stored in a sparse representation (e.g. compressed row storage or coordinate format). The outer vector product in Equation (3.5), forming $\bar{A}$, only needs to be applied to the adjoints corresponding to non-zero entries. While the outer product produces a dense matrix of adjoints, structurally zero matrix entries will by definition never influence any results, their adjoints are thus deemed irrelevant.

Note, that for the symbolic adjoint relation to hold, it is required that $\mathbf{x}$ is actually the solution to $A \cdot \mathbf{x} = \mathbf{b}$, that is $\mathbf{x} = A^{-1}\mathbf{b}$. When using iterative linear solvers, stronger accuracy limits may be required during the primal evaluation, in order to achieve the desired accuracy in the symbolic adjoint. Furthermore, the concept of relative tolerances (where one does not prescribe an absolute residual threshold, but a reduction of the residual in relation to the initial state by a certain factor) should not be applied when differentiating the linear system symbolically. It has been shown, that the concept of relative tolerance can still be applied to symbolically differentiated iterative linear solver schemes, when certain corrections are applied [AHM16].

A case study for the propagation of errors, due to the residual of the primal and adjoint linear systems is shown in Figure 3.16. Observed are 150 SIMPLE iteration steps of the angled duct case, introduced in Section 3.1.2. Linear solvers are GAMG for the pressure equation and Gauss-Seidel for the momentum equations. The linear systems are solved with a solver tolerance of ϵ_f for the primal solvers (for pressure and velocity) and ϵ_r for the linear solvers in the reverse propagation phase. The errors are obtained by comparing to a reference solution, obtained by $\epsilon_f = \epsilon_r = 10^{-12}$, which is at the limit of machine precision. The obtained results are not completely smooth, as the tolerances ϵ_f and ϵ_r are only the upper limit for the linear solver. The actual tolerances achieved by the discrete number of iteration steps of the linear solver might be lower, and do not scale linearly with the prescribed tolerance ϵ.

`dco/c++` allows to stop the recording of the adjoint stack (switch to passive mode) and restart the recording at a later time (switch to active mode). The resulting gap in the tape has to be filled when the adjoints are propagated from the outputs to the inputs, during the adjoint propagating phase. For this purpose, `dco/c++` allows to create functions to be called at a specific point in the adjoint propagating process, using a callback interface. Figure 3.17 shows a conceptual overview of the solution process of one velocity-pressure correction step with the corresponding calls to linear solvers (inside the gray boxes) and how adjoint information is propagated. The information missing from the tape due to the gaps must be supplied by evaluating Equations (3.4) –(3.5) inside the adjoint callback functions.

As the tape is switched off during the linear solver calls, passive versions (using regular floating point data types) of the linear solvers can be called, increasing performance and reducing memory overhead. However, for this the data of the matrix and vectors need to be copied into passive containers, introducing some additional code and data duplication.

The implementation of the symbolically adjoined linear solvers in the context of a sparse iterative CFD solver will be discussed in the following section.

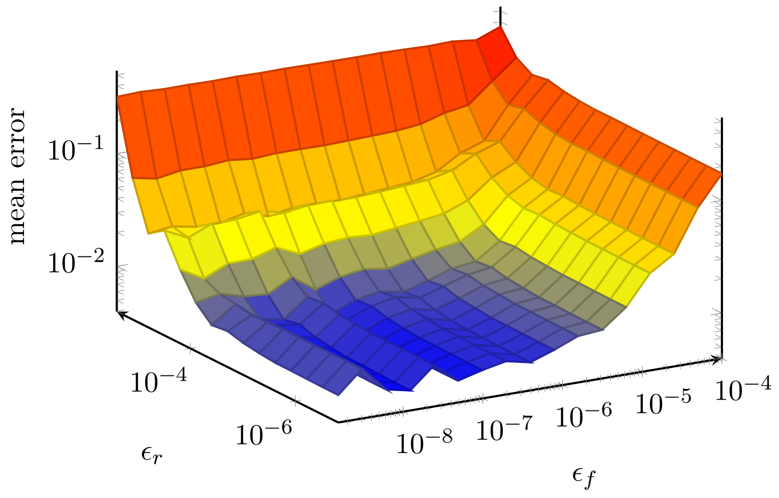


Figure 3.16: Mean error between reference result and result obtained by forward linear solver tolerance ϵ_f and reverse solver tolerance ϵ_r. Differentiated are 150 SIMPLE iteration steps of the angled duct case, starting from a solution of the potential equations.

3.4.2 Implementing SDLS in OpenFOAM

In OpenFOAM, the assembly and solution of linear equation systems is implemented in the `finiteVolume` library, using general linear equation system solvers implemented in the `OpenFOAM` library. The solvers for scalar and vector fields are implemented in

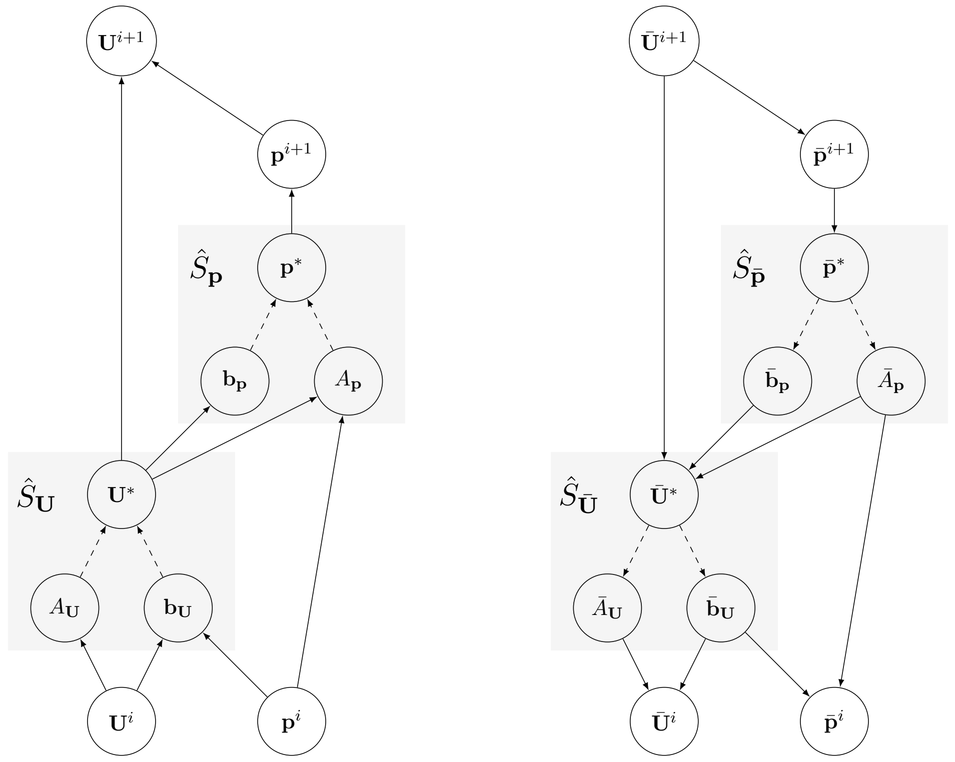
- `fvMatrices/fvScalarMatrix/fvScalarMatrix.C` and
- `fvMatrices/fvMatrix/fvMatrixSolve.C`

respectively. For vector fields, the matrix (FVM discretization) coefficients are scalars and identical for all dimensions. The coefficients differ only for the interfaces, which are not part of the LDU coefficients but are stored separately. Interfaces represent the boundary conditions and internal processor faces, if solving a parallel case. The equation systems are solved in a segregated fashion, i.e. the vector components are decoupled and solved for independently.

To solve a vector equation (e.g. from the discretized momentum equation), three (for the general 3D case) scalar equations with the same matrix coefficients but different interfaces and right hand sides are solved. The coupling between the equations is only introduced by the outer non-linear iteration scheme and the right hand side contributions. To compute the symbolic adjoints of the linear solvers, it is therefore sufficient to focus on the solution of scalar equations and let the AD tool automatically differentiate the assembly of the scalar subproblem from the vector problem.

The matrix classes in the `finiteVolume` library pass a `lduMatrix` (see Section 2.6), which is assembled from the chosen discretization schemes and boundary conditions, as well as a right hand side vector and an initial guess (usually the result of the previous non-linear iteration) to the linear equation system solvers defined in the `OpenFOAM` library.

Currently, the following iterative linear equation system solvers for `lduMatrices` are implemented in OpenFOAM:



(a) Augmented forward creating gaps (b) Adjoint reverse propagation filling gaps

Figure 3.17: Outline of the procedure for filling the gaps in the tape created during the augmented primal section by executing passive versions of the linear solvers only. New outer iteration values $\mathbf{U}^{i+1}$ and $\mathbf{p}^{i+1}$ are generated from $\mathbf{U}^i$ and $\mathbf{p}^i$. The shaded parts indicate the scope of the linear solvers. The upward facing dashed lines indicate the part of the solution process that is treated symbolically. Consequently this part is not stored in the tape and has to be supplied in an adjoint callback function in the (reverse) adjoint propagation step.

GAMG: Geometric Algebraic Multigrid solver with multiple choices for the smoother [Bra77],

PBiCG: Preconditioned Biconjugate gradient method [Fle76],

PBiCGStab: Preconditioned Biconjugate gradient stabilized method [Van92],

PCG: Preconditioned Conjugate Gradient method (for symmetric matrices) [HS52],

smoothSolver: Simple preconditioned Gauss-Seidel solver.

All solvers are coupled to the `lduMatrix` class with the same interface, and thus for the calculation of symbolic adjoints any of the above mentioned solvers can be used for the primal and adjoint callback linear systems. When using symbolic adjoints for the linear solvers, there is no particular need to use the same solver for the primal and adjoint linear systems. If e.g. the `smoothSolver` is used for the primal, to obtain a better stability of the primal convergence, it might be possible to use a solver with better convergence properties like `GAMG` for the adjoint.

We will highlight the major implementation points of the solver for scalar matrices implemented as `fvScalarMatrix::solveSegregated()` in `fvMatrices/fvScalarMatrix/fvScalarMatrix.C`. The code is simplified, in that it always assumes an asymmetric matrix. For symmetric matrices, the matrix transposal is not necessary, and not all matrices elements need to be incremented. However, the symmetric part of the matrix has to be incremented as well, making the code more complex. Furthermore, the code always assumes that the tape is switched on at the entry to the function, and thus symbolic differentiation is required. The full code, without those assumptions and the ability to switch off symbolic differentiation on demand, is included in Appendix B.

In Listing 3.8 the code for the assembly and solution of a scalar `lduMatrix` is shown. First the solution vector `psi` is initialized with the field of the last iteration `psi_` as an initial guess. Then the diagonal of the matrix is modified with coefficients of the boundary conditions (Line 5). Note that the matrix coefficients are stored in the `fvMatrix` object, and are accessible by dereferencing the `this` pointer. In order to not alter the matrix permanently, the diagonal is stored (Line 4) and restored at the end of the function (Line 21). The right hand side vector `totalSource` is assembled from the member field `source_` and the influence of boundary conditions (Lines 7,8). After those preparations, the `lduMatrix` solver object can be assembled from the matrix coefficients stored in `*this` (Line 13), the boundary and internal coefficients (Lines 14,15), the scalar interfaces of the solution vector `psi`, and the `solverControls` object. The newly constructed solver object is then used to solve the linear equation system with the right hand side `totalSource` for the unknowns `psi` (Line 18). Finally, the solution vector `psi` is corrected to comply with boundary conditions (Line 21).

```
 1 Foam::solverPerformance Foam::fvMatrix<Foam::scalar>::solveSegregated(const
       dictionary& solverControls){
 2   auto& psi = const_cast<GeometricField<scalar, fvPatchField, volMesh>&>(psi_);
 3
 4   scalarField saveDiag(diag());
 5   addBoundaryDiag(diag(), 0);
 6
 7   scalarField totalSource(source_);
 8   addBoundarySource(totalSource, false);
 9
10   // Solver call
11   solverPerformance solverPerf = lduMatrix::solver::New(
12     psi.name(),
13     *this,
14     boundaryCoeffs_,
15     internalCoeffs_,
16     psi_.boundaryField().scalarInterfaces(),
17     solverControls
18   )->solve(psi.primitiveFieldRef(), totalSource);
19
20   diag() = saveDiag;
21   psi.correctBoundaryConditions();
22 }
```

Listing 3.8: `fvMatrix` solve routine, can be differentiated by AD without change.

This code is taken straight from the OpenFOAM code base and can be used without any changes to compute the black-box derivatives of the linear solvers.

In Listing 3.9 the modifications that were made to incorporate the symbolic differentiation of the linear equation systems are shown.

The gap in the tape is opened by switching off the tape in Line 10 and closed by switching it back on in Line 43. This gap only wraps the creation and solution of the `lduMatrix` solver object. However, it does not include the code which modifies the matrix and source terms according to the boundary conditions. This initialization code is still handled by AD. The modified solver creates a callback object, which is inserted into the tape in Line 42. It will be called during the reverse propagation phase, when the gap in the tape is encountered.

A `dco/c++` callback object allows to store

Callback function: Function that will be called when the interpretation of the tape reaches the gap. Contents of the callback object are passed as argument to this function.

Output variables: Variables, the adjoints of which will be passed as an input to the callback function.

Input variables: Variables, the adjoints of which will be calculated by the callback function.

Data variables: Variables, the values of which are copied and made available to the callback function as auxiliary input variables.

For the symbolic differentiation of the linear solver, we store the following data (Lines 34–37):

- A `string`, which holds the field name (e.g. `"p"` for pressure), in order to look up the solver settings for the linear solver in the reverse section.
- An `int`, which holds a direction, $(0, 1, 2)$ indicating for which dimension should be solved. A scalar field is indicated by direction -1.
- A `fvMatrix`, which holds a copy of matrix (`*this`), used to solve the primal equations.
- A `scalarField`, which holds a copy of the solution `psi`, computed by the primal linear equation system solver.

The adjoints of the solver output `psi` are required as inputs to the callback function, and are thus registered as *output variables* in Line 40.

By passing the input adjoints and the data variables to the callback object, the desired adjoints can be computed during the execution of the callback function. The algorithms to compute the symbolic derivatives are shown in Listings 3.10 and 3.11. The first listing shows the retrieval of the output adjoints and auxiliary variables, the assembly of the adjoint system and its solution. The data variables, in particular `x` and the matrix coefficients corresponding to A are restored in Lines 3–8. The incoming adjoints $\bar{\texttt{x}}$ of the solver outputs are read in Lines 11,12. The internal and boundary coefficients of the matrix are set in Lines 20,21. These coefficients differ for the different components of a vector field. To extract the correct coefficients from the callback object, here the stored direction is needed. For a scalar field, `component(i)` always returns a reference to the scalar field, no matter which dimension is passed. Therefore, those lines both work for `Type=scalar` and `Type=vector`.

Next, the matrix A is transposed (here always assumed to be necessary). For the OpenFOAM sparse matrix format, this can be done by swapping the upper and lower coefficient vectors, as well as the boundary and interior coefficient vectors (Lines 24–31). As the matrix A stored

```
solverPerformance fvMatrix<scalar>::solveSegregated(const dictionary& solverControls){
  GeometricField<scalar, fvPatchField, volMesh>& psi =
    const_cast<GeometricField<scalar, fvPatchField, volMesh>&>(psi_);

  scalarField saveDiag(diag());
  addBoundaryDiag(diag(), 0);
  scalarField totalSource(source_);
  addBoundarySource(totalSource, false); // assemble RHS vector

  ADmode::global_tape->switch_to_passive(); // Gap in tape starts here

  ADmode::external_adjoint_object_t* D = ADmode::global_tape->create_callback_object();

  forAll(totalSource, i)
    D->register_input(totalSource[i]);
  for(int i = 0; i < this->upper().size(); i++)
    D->register_input(this->upper()[i]);
  for(int i = 0; i < this->lower().size(); i++)
    D->register_input(this->lower()[i]);
  for(int i = 0; i < this->diag().size(); i++)
    D->register_input(this->diag()[i]);

  // solve (*this) * psi = totalSource
  solverPerformance solverPerf = lduMatrix::solver::New
  (
    psi.name(),
    *this,
    boundaryCoeffs_,
    internalCoeffs_,
    psi_.boundaryField().scalarInterfaces(),
    solverControls
  )->solve(psi.primitiveFieldRef(), totalSource);

  D->write_data(psi.name());
  D->write_data(Foam::direction(-1)); // dummy direction for scalar fields
  D->write_data(*this); // copy of lduMatrix representation
  D->write_data(psi); // copy of solution to (*this) * psi = totalSource

  forAll(psi.primitiveField(),i)
    psi.primitiveFieldRef()[i] = D->register_output(dco::passive_value((psi.primitiveFieldRef()[i])));

  ADmode::global_tape->insert_callback<ADmode::external_adjoint_object_t>(symbolic::fillSolverGap<Foam::scalar>,D);
  ADmode::global_tape->switch_to_active(); // Gap in tape ends here

  diag() = saveDiag;
  psi.correctBoundaryConditions();
}
```

Listing 3.9: `fvMatrix` solve with creation of solver gap and checkpoints of the necessary data.

in the callback object is read only (and might be reused later), a copy is constructed for the transposed matrix.

The solver controls are read in from the `system/fvSolution` file, specifying the class of linear solver and required solution tolerance (Lines 33, 35). The adjoint equation linear systems

$$A^T\bar{\mathbf{b}} = \bar{\mathbf{x}}$$

is then assembled and solved in Lines 37–48. The result is available in `a1_b` after the solver has finished.

The second Listing 3.11 takes the adjoints computed from the adjoint linear equation system and writes them to the corresponding places in the adjoint vector of the tape. The adjoints of $\bar{\mathbf{b}}$ can be directly written to the tape via the registered input adjoints. For the adjoints of $\bar{A}$, the outer product is calculated as $\bar{a}_{ij} = \bar{b}_i \cdot x_j$ and written to the corresponding locations in the LDU addressing (obtained in Lines 8,9). For the diagonal part of the matrix, the indices directly corresponds to the index of the cells, therefore no addressing is required (Lines 23–25). For the off-diagonal entries, the indices have to be retrieved from the LDU addressing first. The incrementation of the off-diagonal coefficients is implemented in Lines 12–20.

It is important, that the input adjoints are incremented in exactly the order in which they were registered with the callback object, as else wrong elements of the adjoint vector are incremented.

Here we omitted the treatment of parallel boundaries and symmetry. The former will be introduced in a later section, the latter is detailed in Appendix B.

```
template<class Type>
inline void fillSolverGap(typename Foam::ADmode::external_adjoint_object_t *D){
  const Foam::word& fieldName = D->read_data<Foam::word>();
  const Foam::direction& cmpt = D->read_data<Foam::direction>();

  const Foam::fvMatrix<Type>& A = D->read_data<Foam::fvMatrix<Type> >();
  const Foam::volScalarField& x_ref =  D->read_data<Foam::volScalarField>();
  const Foam::scalarField& x = x_ref.primitiveField();

  Foam::scalarField a1_x(x);
  forAll(a1_x, i)
    a1_x[i] = D->get_output_adjoint(); // read incoming adjoints from tape

  Foam::fvMatrix<Type> A_T(A); // will hold transpose of A

  // component() will return scalarField for Type=scalar
  Foam::FieldField<Foam::Field,Foam::scalar> bcmpts = A_T.boundaryCoeffs().
      component(cmpt)();
  Foam::FieldField<Foam::Field,Foam::scalar> icmpts = A_T.internalCoeffs().
      component(cmpt)();

  // transpose matrix by swapping coefficient arrays
  A_T.lower() = A.upper();
  A_T.upper() = A.lower();
  icmpts = A_T.boundaryCoeffs().component(cmpt)();
  bcmpts = A_T.internalCoeffs().component(cmpt)();

  // Lookup solver controls
  Foam::word reverseFieldName = fieldName + Foam::word("Reverse");
  Foam::volScalarField a1_b(reverseFieldName,x_ref);
  const dictionary& reverseSolverControls = a1_b.mesh().solverDict([...]);

  // solve A^T * a1_b = a1_x
  Foam::lduMatrix::solver solver = Foam::lduMatrix::solver::New
  (
    fieldNameCmpt + Foam::word("Reverse"),
    A_T,
    bcmpts,
    icmpts,
    a1_b.boundaryField().scalarInterfaces(),
    reverseSolverControls
  );

  // solve for b_1
  solverPerformance solverPerf = solver->solve(a1_b.primitiveFieldRef(), a1_x);
  [...] // continued in next listing
```

Listing 3.10: Adjoint callback routine: Solution of the adjoint system.

```
  [...] // continuation of previous listing
  // increment input adjoint for b
  for(int i = 0; i < a1_b.size(); i++){
    D->increment_input_adjoint( dco::value(a1_b[i]) );
  }

  // Addressing for upper and lower half of matrix
  const labelUList& uAddr = A_T.lduAddr().upperAddr();
  const labelUList& lAddr = A_T.lduAddr().lowerAddr();

  // increment adjoints for A (upper part)
  double inc = 0;
  for(int i = 0; i < this->upper().size(); i++){
    D->increment_input_adjoint( dco::value( -a1_b[lAddr[i]]*x[uAddr[i]] );
  }

  // increment adjoints for A (lower part)
  for(int i = 0; i < this->lower().size(); i++){
    D->increment_input_adjoint( dco::value( -a1_b[uAddr[i]]*x[lAddr[i]] );
  }

  // increment adjoints for A (diag part)
  for(int i = 0; i < nd; i++){
    D->increment_input_adjoint( dco::value(-a1_b[i]*x[i]) );
  }
  // treatment for parallel boundaries will go here, see later Sections
  [...]
}
```

Listing 3.11: Adjoint callback routine: Incrementation of input adjoints.

3.5 Adjoints of Parallel Communication

In this section, we will investigate how to efficiently adjoin iterative MPI parallel programs, particularly in the presence of linear solvers. First the general concepts of MPI are introduced, then the calculation of adjoints via AMPI is discussed. Building on those foundations the calculation of adjoints in the CFD context is explored. Lastly, it is shown how the SDLS approach presented in Section 3.4 can be adapted for (A)MPI parallel calculation.

3.5.1 Message Passing Interface

The Message Passing Interface (MPI) [Mes94] is the de-facto standard for implementing parallel `C`, `C++` and Fortran codes on distributed memory machines. With MPI, each compute node runs one or multiple processes of the same executable, each calculating a subproblem of the serial problem (e.g. for CFD calculating the flow field on a subdomain of the original domain). All communication with other processes is wrapped into messages, which are passed from one process to another by the MPI libraries. A message is essentially a chunk of memory contiguous bytes of arbitrary length, that is accompanied by some meta data (e.g. tags). One process can not directly access the memory of another process, even if both share the same virtual memory space. All data which is not available in the local memory space has to be distributed using messages. Communication in MPI is either point-to-point or collective. Point-to-point messages originate from one process and are delivered to exactly one other process. An example for this is the `MPI_Send` and corresponding `MPI_Recv` construct for sending and receiving messages. The signatures for the `MPI_Send` and `MPI_Recv` routines are defined in the MPI standard as

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int
    tag, MPI_Comm comm);
```

and

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
     MPI_Comm comm, MPI_Status *status)
```

respectively. The `MPI_Send` command sends a message of length `count` and type `datatype` to the process specified by `dest`. The call to `MPI_Send` will block until the receiving process has called `MPI_Recv` with the process ID of the sending process as its `source` argument. On the other process `MPI_Recv` will block until the sending process has called `MPI_Send`. Thus, blocking MPI communication inherently leads to a synchronization of the involved processes. However, it can also lead to deadlocks, if the `MPI_Send` and `MPI_Recv` calls are not correctly paired.

Collective communication is used when a message is required to reach multiple processes at once, or when data is to be reduced. Collective communications can be grouped into different cases:

One to all communication: Data is sent from one process to all others. E.g. `MPI_Bcast`.

All to one: Data is sent from all processes to one root process, potentially involving a reduction operation. E.g. `MPI_Reduce`.

All to all: Data is sent from all processes to all other processes, potentially involving a reduction operation. E.g. `MPI_Allreduce`.

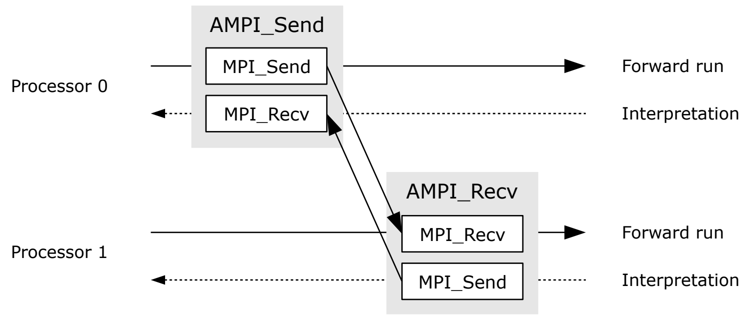


Figure 3.18: Illustration of an AMPI communication pattern. A primal `MPI_Send` generates an adjoint `MPI_Send` in opposite direction.

The specific communication patterns, used to implement the collective communications, are not specified by the standard and are implementation dependent (popular MPI implementations are OpenMPI, IntelMPI, and MpiCH).

3.5.2 Adjoint MPI

Adjoint MPI (AMPI) is a library developed at STCE [SN12], in cooperation with Argonne National Labs and the Institute for Research in Computer Science and Automation (Inria) [Utk+09]. It introduces the concept of adjoints to MPI. The challenge for adjoints in the context of parallel communication is the split between (augmented) primal evaluation and reverse adjoint propagation. Adjoint information needs to be propagated back from the outputs of the program to the inputs. For data that is received in the augmented primal evaluation, using regular MPI calls, the corresponding adjoints are not incremented on the remote process during the adjoint reverse propagation. Therefore, the adjoints on the remote process are incomplete.

In order to correctly evaluate the adjoints in presence of MPI communication, the MPI calls in the augmented primal section have to be accompanied by corresponding MPI calls in the adjoint reverse section, which distribute the adjoints back to the relevant processes. A basic case for this is illustrated in Figure 3.18. Data is sent from one process (P_0) to another process (P_1) with a pair of blocking `MPI_Send` and `MPI_Recv` calls. In the reverse section the direction of the calls is switched, transferring adjoint information from P_1 to P_0.

The AMPI library provides a set of wrapper functions for the regular MPI calls. In the augmented primal section, AMPI keeps track of the MPI calls and then passes the calls on to the MPI library implementation. To track the MPI calls, AMPI keeps a list of all executed calls including the required meta data, such as message source and destinations, tags, and message length. In order to execute the required MPI calls during the reverse interpretation phase, AMPI inserts callback objects (using the same mechanisms introduced in Section 3.4) into the `dco/c++` tape, which are called when adjoint data needs to be distributed using MPI during the adjoint reverse propagation.

For collective communications, the adjoint communication patterns are more complex. The adjoint of a variable x needs to be incremented for every statement which uses it nonlinear in x. Thus, if the local variable x is sent to multiple processes (e.g. using `MPI_Broadcast`) the adjoint $\bar{x}$ needs to (potentially) be incremented by adjoints sent back from all those processes. Hence, a primal call to `MPI_Broadcast` is augmented by a call to `MPI_Reduce` with reduce operation

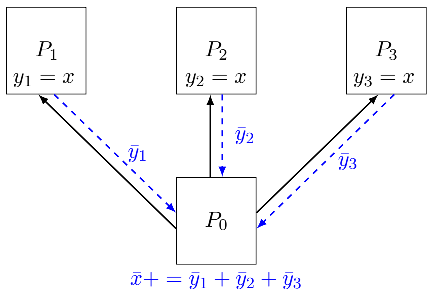


Figure 3.19: Primal (black) and adjoint (blue) communication for `MPI_Bcast`. The `MPI_Bcast` call in the primal section, distributing value x from root P_0 to all other processes, is transformed into a `MPI_Reduce` call in the reverse section, summing up the partial adjoints on the root P_0.

`MPI_SUM` in the reverse interpretation, which sums up all partial adjoints of $\bar{x}$ from the involved processes, and then increments the adjoint of the root process by this sum. This communication pattern is illustrated in Figure 3.19 for four processes and root process P_0.

Table 3.5 lists the primal MPI calls and the corresponding calls during reverse propagation for some common MPI calls. It has been shown that all common one-sided MPI operations can be adjoined with a constant overhead, compared to the primal MPI call, with the exception of the `MPI_Reduce` operation with `MPI_PROD` as the reduction operator [Sch14]. The number of adjoint increments required for a product reduction scales linearly with the number of involved processes, due to the product rule.

AMPI can be interfaced with different AD tools, requiring that these tools provide a set of interface functions. An interface for `dco/c++` is included in the open-source AMPI release.

3.5.3 Combining AMPI with OpenFOAM

OpenFOAM implements distributed messaging in a hierarchical fashion, by wrapping low level communication routines in a separate layer. This is organized as follows. On the high level, information is stored in data structures specific to the CFD domain. For example, `GeometricField<scalar, fvPatchField, volMesh>` stores a field of scalar typed values. It includes references to the boundary conditions and the underlying volume mesh. For convenience, the definition is abbreviated to `volScalarField` via a typedef. The communication of (field) data between different processes is abstracted into a library called `Pstream`, which allows the implementation of different parallelization strategies. However, at the moment only MPI communication is implemented in the regular OpenFOAM release.

Table 3.5: MPI routines used in OpenFOAM, their AMPI versions, and communication patterns specific to the augmented primal and adjoint sections of the adjoint code.

Primal MPI call	AMPI call	MPI call in reverse section
`MPI_Bsend`	`AMPI_Bsend`	`MPI_Recv`
`MPI_Recv`	`AMPI_Recv`	`MPI_Bsend`
`MPI_Allreduce`	`AMPI_Allreduce`	`MPI_Allreduce`
`MPI_Bcast`	`AMPI_Bcast`	`MPI_Reduce`

```
class AmpiTypeHelper{
private:
  const std::type_info& type;
  Foam::string caller;
  AmpiTypeHelper(const std::type_info& type, Foam::string caller)
    : type(type), caller(caller){}
  AmpiTypeHelper();
public:
  template<typename T>
  static const AmpiTypeHelper create(Foam::string caller=""){
    const AmpiTypeHelper t(typeid(T), caller);
    return t;
  }
  bool operator ==(const AmpiTypeHelper &b){
    return (b.type == this->type);
  }
  bool is_active_type() const {
    return type == typeid(dco::ga1s<double>::type)
    ||  type == typeid(Foam::Vector<dco::ga1s<double>::type >)
    ||  type == typeid(Foam::Tensor<dco::ga1s<double>::type >);
  }
};
```

Listing 3.12: Type helper, used to determine the type of data passed to the low level communication routines. For debug purposes, the caller function can also be passed along.

To simplify the low level communication routines, the OpenFOAM development team decided to not pass down type information from the high level data structures to the low level communication routines. All message data is casted to the `char` data type (which occupies one byte in C) before being passed to the low level routines (more specifically just the data pointer passed to the low level is cast to `char*`). This allows the use of the `MPI_BYTE` data type inside the MPI send and receive routines for all types of data, avoiding specializations of the low level communication routines for different data types. As OpenFOAM knows which data to expect on the high level, the type of the received data is not needed. However AMPI, which wraps around the MPI routines on the low level, needs to be able to separate floating point data from passive data (e.g. integers, strings). For passive data, the primal communication routines can be kept unchanged, as no derivative information is associated with them. For (active) floating point data, the communication routines have to be changed to accommodate the reverse propagation of adjoints.

To incorporate the type information into OpenFOAM, instead of rewriting the whole communication layer, a type helper is introduced. It is inserted into all calls to the low level MPI read and write routines. The code for the `AmpiTypeHelper` is shown in Listing 3.12. The type helper is constructed from an arbitrary variable and stores its `std::type_info` information. This allows to boolean compare the type of the variable to another type. It offers a routine which returns true if the corresponding type is or contains a `dco/c++` active type. With that information the low level communication routines can be augmented to decide whether to issue a MPI or AMPI call. An AMPI call only needs to be issued if active types are involved, and the augmented primal is executed (i.e. the tape is *active*) while the call to MPI is executed.

The procedure outlining the (adjoint) communication flow for standard OpenFOAM and

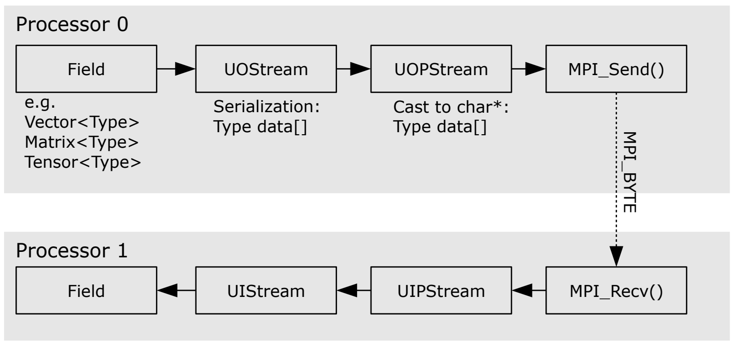


Figure 3.20: Unaugmented data flow from high level classes to communication layer.

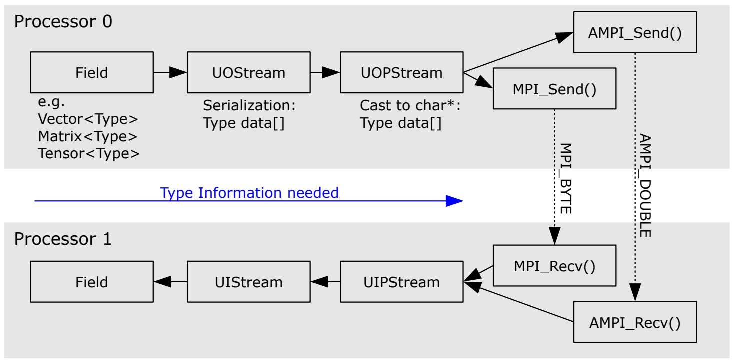


Figure 3.21: Type augmented data flow from high level classes to communication layer. Type information allows AMPI to determine whether adjoint communication patterns need to be applied.

discrete adjoint OpenFOAM using AMPI is illustrated in Figures 3.20 and 3.21.

To give an understanding of the amount of (adjoint) communication during a typical solver execution, Table 3.6 shows the total number of (A)MPI calls for 10 iteration steps of the `adjointSimpleFoam` solver on the Pitz-Daily case. The domain is decomposed onto 2, 4, and 8 processors. Listed are MPI calls, that is calls to the standard MPI routines without involvement of AMPI, AMPI calls, that is calls to MPI routines from inside the AMPI wrapper routines during the augmented primal evaluation, and AMPI_MPI calls, that is calls to MPI routines issued by AMPI during the reverse propagation sweep.

OpenFOAM provides its own implementations for collective operations (instead of relying on `MPI_Gather` and `MPI_Scatter`), which are implemented using the basic `MPI_Send` and `MPI_Recv` routines. Therefore, the only MPI operations present in significant number are blocking `MPI_Bsend` and `MPI_Recv` pairs, and `MPI_Allreduce` (with the `MPI_SUM` reduction operator). `MPI_Allreduce` is mainly used to calculate the residual of the linear solver iterations, while the send and receive calls are used to distribute data needed for the matrix vector products.

From the table it is obvious, that if the differentiation of the linear solvers is performed symbolically, only very few AMPI calls remain (only around 2% of all calls to MPI need to be passed through AMPI). For SDLS, the communication of data needed for the matrix vector products in the iterative linear solvers move from AMPI calls to MPI calls. This significantly

Table 3.6: Calls to different (A)MPI routines for 10 steps of `adjointSimpleFoam` on the 2D Pitz-Daily case. Tabulated are decompositions to $2, 4$ and 8 processors, with and without SDLS.

	no SDLS			SDLS		
	2	4	8	2	4	8
`MPI_Bsend`	0	0	0	45738	145578	517770
`MPI_Recv`	0	0	0	45738	145578	517770
`MPI_Allreduce`	10	20	40	40414	108592	333896
`AMPI_Bsend`	23758	74946	265298	640	1920	7040
`AMPI_Recv`	23758	74946	265298	640	1920	7040
`AMPI_Allreduce`	20718	55116	168112	166	332	664
`AMPI_Waitall`	20	36	56	20	36	56
`AMPI_MPI_Bsend`	23758	74946	265298	640	1920	7040
`AMPI_MPI_Recv`	23758	74946	265298	640	1920	7040
`AMPI_MPI_Allreduce`	20716	55112	168104	164	328	656
Total MPI calls	352	1010	2358	132232	400738	1371754
Total AMPI calls	136506	410120	1397716	2930	8448	29788
Total AMPI MPI calls	68232	205004	698700	1444	4168	14736
Total sum	205090	616134	2098774	136606	413354	1416278

reduces the number of calls to MPI routines during the reverse propagation phase and also lowers the number of adjoint callback objects in the `dco/c++` tape, reducing run time and memory.

The inclusion of SDLS comes at the cost of additional linear solver calls during the reverse propagation phase, which in turn issue additional MPI calls. Therefore, the number of `MPI_Send` and `MPI_Recv` calls for the SDLS case is higher than the corresponding `AMPI_Send` and `AMPI_Rev` calls for the non-SDLS case. However, for the studied case the increase is lower than a factor of two, meaning that the reverse propagation issues less MPI calls than the AD implementation does during reverse propagation. As the number of iterations during primal and reverse linear equation solves is not necessarily equal, the opposite could also be true for a different case.

The additional steps required to incorporate SDLS in an AMPI setting are discussed in detail in Section 3.5.5.

3.5.4 Tangent MPI Communication

The tangent data type of `dco/c++` can be used with MPI with only minor modifications. The memory layout of the (scalar or vector) tangent type always bundles together a value with its corresponding tangent(s). This contiguous memory layout allows to reuse the existing MPI send and receive routines with correspondingly increased message size. The message size doubles for tangent scalar mode and increases by factor $(1 + n_v)$ for tangent vector mode.

When using the built in MPI reductions of `MPI_Allreduce`, for some reductions additional logic is required. The reductions assume that each entry of the sent data stream contains an

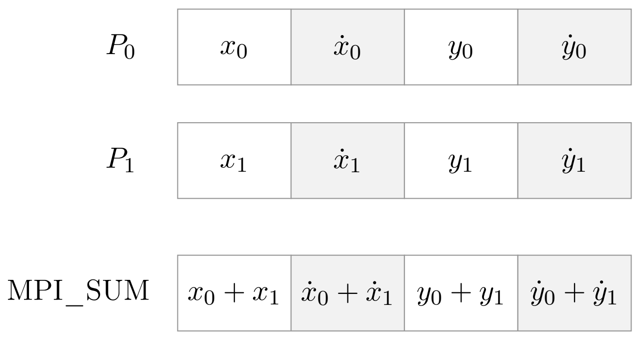


Figure 3.22: Reduction of tangent data using `MPI_Sum`. The MPI reduction treats the two tangent types as four floating point values and sums them up individually.

element for which associativity and commutativity with other elements holds. For a vector of tangent data types, the primal values and tangents are stored interleaved, however MPI interprets the vector as a (bigger) vector of primals.

Inspection of the OpenFOAM source code reveals, that only two of the built-in reductions are used, namely `MPI_SUM` for summation and `MPI_MIN` for finding the minimum value across a vector. For `MPI_SUM` no special actions have to be taken, because the tangent of a sum of two tangent types is the sum of its tangents. The built-in sum reduction will thus calculate the correct primals and tangents by default. Note, that this only holds, because the `MPI_SUM` reduction sums up vectors element-wise without changing their dimension, and thus primals and tangents are summed up without mixing. This reduction is illustrated in Figure 3.22.

For all remaining reduce operations (of which only `MPI_MIN` is currently used in the OpenFOAM code base), we enforce the use of a manual reduction implementation, which separates the communication of data and the reduction operation. As this manual reduction calls functions that are covered by the AD tool, instead of the MPI intrinsic reduce operations, the correct tangents will be calculated.

3.5.5 Parallel Symbolic Differentiation of Linear Solvers with AMPI

The symbolic differentiation of (sparse) linear solvers becomes more complex when parallel communication is involved. For distributed CFD calculations, commonly a ghost cell approach is used, where values of remote domains are cached locally and updated only on demand to reduce communication complexity. In a black-box differentiation setting, the update of the boundaries will be passed through, and handled by, AMPI. However, if such information is passed during a passive section, e.g. during the symbolically differentiated linear solver calls, the flow of adjoints is interrupted. The corresponding adjoint information has to be supplied in a callback function, that is called during the reverse interpretation sweep, as previously discussed in Section 3.4.

In order to correctly adjoin the boundary communication, it is important to first understand how processor boundaries are treated. The geometric domain decomposition utilized in OpenFOAM leads to both decomposed matrix and vector entries. Each processor holds a subset of the solution vector and a subset of coefficients of the discretization matrices. The entries of the solution vector correspond to values defined on cells/faces located in the decomposed domain.

The global coefficient matrix is decomposed row wise to the individual processors. The matrix

```
const label nCells = diag().size();
const label nFaces = upper().size();
// diagonal (cell) coefficients
for (label cell=0; cell<nCells; cell++){
  ApsiPtr[cell] = diagPtr[cell]*psiPtr[cell];
}
// off diagonal (face) coefficients
for (label face=0; face<nFaces; face++){
  ApsiPtr[uPtr[face]] += lowerPtr[face]*psiPtr[lPtr[face]];
  ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}
```

Listing 3.13: Calculation of the local matrix vector product between the LDU coefficients and vector `psi`.

coefficients stored in the local LDU matrix description correspond to a block diagonal sub-matrix in the global discretization matrix and can be multiplied with the contents of the solution vector without inter-processor communication. Matrix entries which lie outside of the diagonal block require multiplication with vector entries that are not located on the same processor. The corresponding vector entries need to be communicated to the local processor, in order to be added to the matrix vector product. Those matrix entries are stored separately from the LDU description and are called *matrix interfaces*. The corresponding entries of the solution vector are called *interface values*.

OpenFOAM separates the matrix vector product into two parts. In the first part the product between all local matrix coefficients and local vector entries is calculated. This part involves no MPI communication and therefore there is no need to supply additional AMPI information.

The second part involves the multiplication of the matrix interface coefficients with the interface values. For each row of the coefficient matrix, the values of the interface coefficients need to be brought in from remote processes to the local process, calculating the part of the matrix vector product corresponding to the row.

To hide communication latency, the implementation separates the parallel matrix vector multiplication into three stages:

Preparation of matrix interfaces: Send matrix interface coefficients $\boldsymbol{B}$ to the corresponding processors. For non-blocking communication, this allows to start with next step before interfaces are received.

Multiplication of local coefficients: Calculate local matrix vector product according to Listing 3.13. Indirect memory access to both the result and the entries of the multiplicant vector, due to the LDU addressing. (Caching friendly if the mesh is numbered efficiently.)

Multiplication of interface coefficients: At a later stage add missing product terms to the result according to Listing 3.14. Entries of the interfaces are already matched correctly, therefore indirect memory lookup only occurs for the result vector.

```
const labelUList& faceCells = this->interface().faceCells();

forAll(faceCells, elemI){
  result[faceCells[elemI]] += coeffs[elemI]*vals[elemI];
}
```

Listing 3.14: Calculation of the matrix vector product between processor interface coefficients `coeffs` with entries of the remote vector `psi`, stored in `vals`. Entries of `coeffs` and `vals` are already correctly aligned, therefore no indirect access on the right hand side. Implementation in `lduInterfaceFieldTemplates.C`

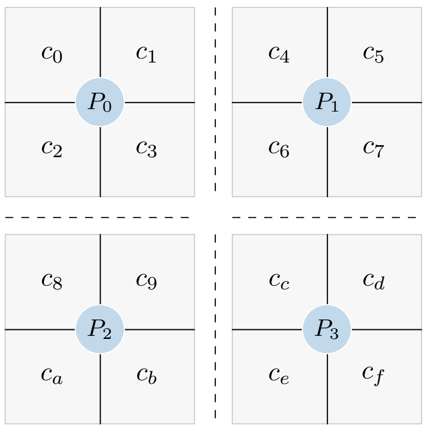


Figure 3.23: Structured 4×4 mesh, decomposed onto four processors P_0–P_3. Cells are numbered such that distance between local cells is minimal. To compress notation, cells are numbered to a hexadecimal base.

An example for the mesh decomposition, the LDU decomposition and the global and local coefficient matrices is given in Figure 3.24. illustrated is a structured 4×4 grid, depicted in Figure 3.23, decomposed onto four processors. The global cells are numbered such that the maximum (index) distance between two cell indices on the individual processors is minimal. To obtain a more compact notation in the global coefficient matrix, we number the 16 cells in a hexadecimal base from c_0 to c_f. There are processor boundaries between processor pairs (P_0, P_1), (P_0, P_2), (P_1, P_3), and (P_2, P_3). The vectors $\boldsymbol{l}, \boldsymbol{d}, \boldsymbol{u}$ in the figure list the processor local entries. The index vectors $\boldsymbol{L}$ and $\boldsymbol{U}$ give the addressing for $\boldsymbol{l}$ and $\boldsymbol{u}$.

For each processor boundary, two entries of the solution vector $\mathbf{x}$ have to be copied from the remote process. For example, $\mathbf{v}^{01} = [x_1, x_3]$ is copied from P^0 to P^1 and $\mathbf{v}^{10} = [x_4, x_6]$ in the opposite direction.

For convenience, the numbering in the figure corresponds to global cell numbers. In the implementation each processor numbers its local cells from zero. Coefficients outside of the diagonal blocks (that is coefficients of faces on a processor boundary), require additional communication. Entries of the distributed vector must be brought to the processor, computing the corresponding rows of the matrix vector product. For example, u_{14} and u_{36} on the processor boundary $(P0, P1)$ require the computation of the product between coefficients associated with processor P_0 and the parts of the solution vector located on P_1, i.e. x_4 and x_6.

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
d_{00} & u_{01} & u_{02} & & & & & & & & & & & & & \\
l_{10} & d_{11} & & u_{13} & u_{14} & & & & & & & & & & & \\
l_{20} & & d_{22} & u_{23} & & & & & u_{28} & & & & & & & \\
& l_{31} & l_{32} & d_{33} & & & u_{36} & & & u_{39} & & & & & & \\
\hline
& l_{41} & & & d_{44} & u_{45} & u_{46} & & & & & & & & & \\
& & & & l_{54} & d_{55} & & u_{57} & & & & & & & & \\
& & & l_{63} & l_{64} & & d_{66} & u_{67} & & & & & u_{cc} & & & \\
& & & & & l_{75} & l_{76} & d_{77} & & & & & & u_{dd} & & \\
\hline
& & l_{83} & & & & & & d_{88} & u_{89} & u_{8a} & & & & & \\
& & & l_{94} & & & & & l_{98} & d_{99} & & u_{9b} & u_{9c} & & & \\
& & & & & & & & l_{a8} & & d_{aa} & u_{ab} & & & & \\
& & & & & & & & & l_{b9} & l_{ba} & d_{bb} & & & u_{be} & \\
\hline
& & & & & & l_{c6} & & & l_{c9} & & & d_{cc} & u_{cd} & u_{ce} & \\
& & & & & & & l_{d7} & & & & & l_{dc} & d_{dd} & & u_{df} \\
& & & & & & & & & & & l_{eb} & l_{ec} & & d_{ee} & u_{ef} \\
& & & & & & & & & & & & & l_{fd} & l_{fe} & d_{ff}
\end{array}\right]
\left[\begin{array}{c}
x_0 \\ x_1 \\ x_2 \\ x_3 \\ \hline x_4 \\ x_5 \\ x_6 \\ x_7 \\ \hline x_8 \\ x_9 \\ x_a \\ x_b \\ \hline x_c \\ x_d \\ x_e \\ x_f
\end{array}\right]$$

$\boldsymbol{l}^0 = [l_{10}, l_{20}, l_{31}, l_{32}]$	$\boldsymbol{l}^1 = [l_{54}, l_{64}, l_{75}, l_{76}]$	$\boldsymbol{l}^2 = [l_{98}, l_{02}, l_{13}, l_{23}]$	$\boldsymbol{l}^3 = [l_{dc}, l_{ec}, l_{fd}, l_{fe}]$
$\boldsymbol{d}^0 = [d_{00}, d_{11}, d_{22}, d_{33}]$	$\boldsymbol{d}^1 = [d_{44}, d_{55}, d_{66}, d_{77}]$	$\boldsymbol{d}^2 = [d_{88}, d_{99}, d_{aa}, d_{bb}]$	$\boldsymbol{d}^3 = [d_{cc}, d_{dd}, d_{ee}, d_{ff}]$
$\boldsymbol{u}^0 = [u_{01}, u_{02}, u_{13}, u_{23}]$	$\boldsymbol{u}^1 = [u_{45}, u_{46}, u_{57}, u_{67}]$	$\boldsymbol{u}^2 = [u_{89}, u_{8a}, u_{9b}, u_{ab}]$	$\boldsymbol{u}^3 = [u_{cd}, u_{ce}, u_{df}, u_{ef}]$
$\boldsymbol{L}^0 = [0, 0, 1, 2]$	$\boldsymbol{L}^1 = [0, 0, 1, 2]$	$\boldsymbol{L}^2 = [0, 0, 1, 2]$	$\boldsymbol{L}^3 = [0, 0, 1, 2]$
$\boldsymbol{U}^0 = [1, 2, 3, 3]$	$\boldsymbol{U}^1 = [1, 2, 3, 3]$	$\boldsymbol{U}^2 = [1, 2, 3, 3]$	$\boldsymbol{U}^3 = [1, 2, 3, 3]$
$\boldsymbol{I}_1^0 = [u_{14}, u_{36}]$	$\boldsymbol{I}_0^1 = []$	$\boldsymbol{I}_0^2 = []$	$\boldsymbol{I}_1^3 = []$
$\boldsymbol{B}_1^0 = []$	$\boldsymbol{B}_0^1 = [l_{41}, l_{63}]$	$\boldsymbol{B}_0^2 = [l_{83}, l_{94}]$	$\boldsymbol{B}_1^3 = [l_{c6}, l_{d7}]$
$\boldsymbol{I}_2^0 = [u_{28}, u_{39}]$	$\boldsymbol{I}_3^1 = [u_{cc}, u_{dd}]$	$\boldsymbol{I}_3^2 = [u_{9c}, u_{be}]$	$\boldsymbol{I}_2^3 = []$
$\boldsymbol{B}_2^0 = []$	$\boldsymbol{B}_3^1 = []$	$\boldsymbol{B}_3^2 = []$	$\boldsymbol{B}_2^3 = [l_{c9}, l_{d7}]$
$\mathbf{x}^0 = [x_0, x_1, x_2, x_3]$	$\mathbf{x}^1 = [x_4, x_5, x_6, x_7]$	$\mathbf{x}^2 = [x_8, x_9, x_a, x_b]$	$\mathbf{x}^3 = [x_c, x_d, x_e, x_f]$

Figure 3.24: Block diagonal matrix, discretizing the mesh from Figure 3.23. Colored coefficients on the main blocks are local to the processors and can be multiplied with the corresponding entries of the vector $\mathbf{x}$ without further communication. Entries in off-diagonal blocks need to be communicated to corresponding processors, to correctly evaluate the matrix vector product.

The SDLS implementation presented in Section 3.4 calculates the adjoints of the local LDU coefficients and vector entries only. In order to correctly capture the adjoints of the parallel matrix vector product, we need to also symbolically calculate the adjoints of the matrix interface coefficients and the interface values. The calculation of the adjoints is outlined in Algorithm 2. Lines 1–20 calculate the adjoints of the local coefficients as outlined in Section 3.4. For symmetric matrices, the adjoints of the off-diagonal coefficients need to also be incremented by the adjoints of the omitted opposite part (Lines 10 and 18).

Lines 21–28 list the calculation of the outer product $-\bar{\mathbf{b}} \cdot \mathbf{x}^T$ for the interface coefficients. In order to calculate the adjoints, the interface values of the remote processors need to be brought into the local scope (Line 23). Using the remote value of $\mathbf{x}$ and the local value of $\bar{\mathbf{b}}$, the individual values of the outer product can be formed. Lines 21–27 directly correspond to Lines 37–55 of the implementation shown in Appendix B.3.

Algorithm 2: Calculation of the entries of $\bar{A}, \bar{\mathbf{b}}$ including processor boundary coefficients.

```
Input: primal solution $\mathbf{x}$, incoming adjoints $\bar{\mathbf{x}}$
Data: matrix coefficients: $(\boldsymbol{l}, \boldsymbol{d}, \boldsymbol{u})$, ldu Addressing $(\boldsymbol{L}, \boldsymbol{U})$, boundary coefficients $\boldsymbol{B}$
Output: adjoints $\bar{\mathbf{b}}$, $\left(\bar{\boldsymbol{l}}, \bar{\boldsymbol{d}}, \bar{\boldsymbol{u}}\right)$, $\bar{\boldsymbol{B}}$
$\bar{\mathbf{b}} \leftarrow \bar{\mathbf{b}} +$ solve$(A^T, \bar{\mathbf{x}})$;
forall diagonal entries $d_i$ at index $i$ in $\boldsymbol{d}$ do
    $\bar{d}_i \leftarrow \bar{d}_i - \bar{b}_i \cdot x_i$;
end
forall lower entries $l_i$ at index $i$ in $\boldsymbol{l}$ do
    $j \leftarrow U_i$;
    $k \leftarrow L_i$;
    $\bar{l}_i \leftarrow \bar{l}_i - \bar{b}_j \cdot x_k$;
    if A symmetric with no upper part then
        $\bar{l}_i \leftarrow \bar{l}_i - \bar{b}_k \cdot x_j$;
    end
end
forall upper entries $u_i$ at index $i$ in $\boldsymbol{u}$ do
    $j \leftarrow L_i$;
    $k \leftarrow U_i$;
    $\bar{u}_i \leftarrow \bar{u}_i - \bar{b}_j \cdot x_k$;
    if A symmetric with no lower part then
        $\bar{u}_i \leftarrow \bar{u}_i - \bar{b}_k \cdot x_j$;
    end
end
forall processor boundary fields $\boldsymbol{p}_j$ with index $j$ do
    update boundary cells of $\mathbf{x}$ and $\bar{\mathbf{b}}$ on patch $\mathbf{p}_j$;
    $\mathbf{x}^\star \leftarrow$ boundary cell values of $\mathbf{x}$ from neighboring processor;
    $\bar{\mathbf{b}}^\star \leftarrow$ boundary cell values of $\bar{\mathbf{b}}$ from this processor;
    forall faces $f_i$ with index $i$ on $\boldsymbol{p}_j$ do
        $\bar{\mathbf{B}}_{i_j} \leftarrow \bar{\mathbf{B}}_{i_j} - \bar{\mathbf{b}}^\star \cdot \mathbf{x}^\star$;
    end
end
```

3.6 Higher Order Differentiation

Obtaining derivatives of order higher than one might be desired for a multitude of reasons. Second order derivatives can e.g. be used to implement a Newton optimization scheme (see Section 2.9.2). Further an additional optimization step, operating on the optimized parameters, might be desired. This motivates the use of higher derivatives. As already outlined in Section 2.7.2, higher derivatives can be obtained by repeatedly applying the first order model of AD.

The AD tool `dco/c++` allows the nesting of first order data types to obtain derivatives of arbitrary order. Nesting a tangent data type inside an adjoint type yields the tangent over adjoint model $x^{(2)}_{(1)}$.

```
typedef dco::ga1s<dco::gt1s<double>::type>::type doubleScalar;
```

Analogously a tangent data type nested inside another tangent data type yields the tangent over tangent model $x^{(1,2)}$.

```
typedef dco::gt1s<dco::gt1s<double>::type>::type doubleScalar;
```

The nesting of data types to obtain second order derivatives, as well as the access routines of `dco/c++` required to extract the individual derivative components, are shown in Figure 3.25.

If implemented naively, the calculation of a Hessian $H \in \mathbb{R}^{n\times n}$ requires n evaluations of the tangent over adjoint model of AD and consequently n full evaluations of the flow equations. For cases like topology optimization, the influence of a parameter to the final objective propagates iteratively through the whole flow domain in a non-linear fashion. This propagation leads to a dense Hessian. Thus, coloring techniques [GMP05] which can be used to reduce the number of evaluations of the AD model required for sparse matrices are not applicable in this case. This makes the computation, and also the storage, of Hessians expensive, such that approximative Quasi-Newton methods like BFGS are a better fit to speed up optimization convergence.

However, for lower dimensional parameter spaces (see e.g. parametric optimization, Section 4.6), problems which are known to exhibit sparsity, or applications where higher order derivatives are needed, higher order adjoints are the superior choice over FD. An example driver for the evaluation of the full Hessian using scalar tangent over tangent mode is given in Listing 3.15.

Improving on the scalar tangent over tangent model, a whole block of the Hessian can be extracted at once using the tangent vector mode (see Section 2.8). For a vector size d, this mode allows to extract a sub block $S \in R^{d\times d}$ of the full Hessian H with one evaluation of the augmented primal function. Listing 3.16 shows the seeding procedure to retrieve the Hessian of an arbitrary function $f : R^n \to R$ under the assumption that the vector size d is identical to n. Because the memory consumption of two nested vector types corresponds to $\mathcal{O}(d^2)$ this is not feasible for even modestly sized problems. Hence in Listing 3.17 we choose $d \ll n$ and extract the smaller Hessian sub blocks S one by one. The calculation of the Hessian in blocks also allows to exploit the symmetry of the Hessian.

Using the tangent over adjoint mode of AD, the computational complexity of calculating the whole Hessian is lowered from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, assuming that m is one or a constant with $m \ll n$. The tape has to be re-recorded for each tangent direction, therefore each tape is only interpreted once. Each evaluation of the tangent over adjoint model yields one row/column of the Hessian. The seeding of the adjoint and tangent directions for a general function `f` is shown in Listing 3.18.

Both tangent over adjoint and tangent over tangent models are implemented in discrete adjoint

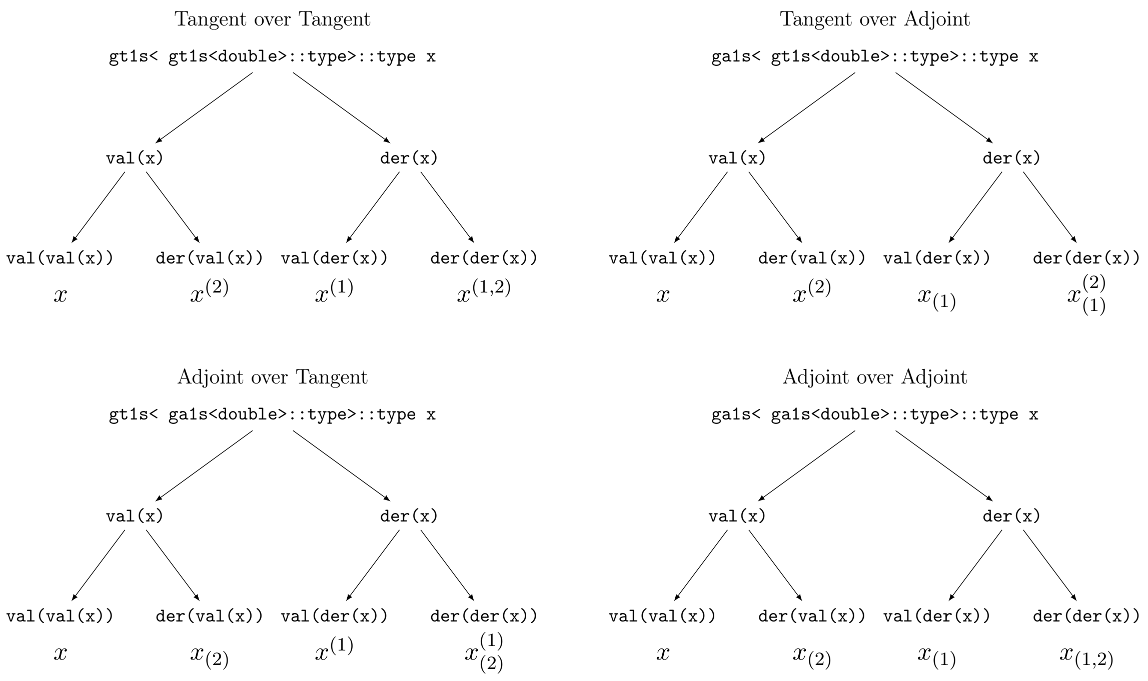


Figure 3.25: Interface calls allowing to access specific components of the higher order models. `dco::derivative()` and `dco::value()` abbreviated as `der()` and `val()`.

```
typedef dco::gt1s<dco::gt1s<double>::type>::type ADtype;
extern ADtype f(ADtype* x,int n);

void t2s_t1s(int n, ADtype* x, double** H){
  // seed all directions, evaluate f n*n times
  for(int i=0; i<n; i++){
    dco::value(dco::derivative(x[i])) = 1.0;
    for(int j=i; j<n; j++){
      dco::derivative(dco::value(x[j])) = 1.0;
      ADtype y = f(x,n);
      H[i][j] = H[j][i] = dco::derivative(dco::derivative(y));
      dco::derivative(dco::value(x[j])) = 0.0;
    }
    dco::value(dco::derivative(x[i])) = 0.0;
  }
}
```

Listing 3.15: Dense seeding for obtaining a full Hessian of function $f : \mathbb{R}^n \to \mathbb{R}$ in scalar tangent over tangent mode.

```cpp
typedef dco::gt1v<dco::gt1v<double,d>::type,d>::type ADtype;
extern ADtype f(ADtype* x,int n);

void t2v_t1v(int n, ADtype* x, double** H){
  // seed all directions, evaluate f once
  for(int i=0; i<n; i++){
    dco::value(dco::derivative(x[i])[i]) = 1.0;
    for(int j=0; j<n; j++)
      dco::derivative(dco::value(x[j]))[j] = 1.0;
  }
  ADtype y = f(x,n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      H[i][j] = dco::derivative(dco::derivative(y)[j])[i];
}
```

Listing 3.16: Seeding of vector tangent over vector tangent mode, under the assumption that the vector size d of gt1v::type equals problem size n.

```cpp
typedef dco::gt1v<dco::gt1v<double,d>::type,d>::type ADtype;
extern ADtype f(ADtype* x,int n);

void t2v_t1v(ADtype* x, int n, int d, double** H){
  for(int i=0;i<n;i+=d){
    for(int j=0;j<=i;j+=d){ // use symmetry
      for(int k=i; k<std::min(i+d,n); k++){
        dco::value(dco::derivative(x[k])[k%d]) = 1.0;
        for(int l=j; l<std::min(j+d,n); l++){
          dco::derivative(dco::value(x[l]))[l%d] = 1.0;
        }
      }
      ADtype y = f(x,n);
      for(int k=i; k<std::min(i+d,n); k++){
        dco::value(dco::derivative(x[k])[k%d]) = 0.0;
        for(int l=j; l<std::min(j+d,n); l++){
          H[k][l] = H[l][k] = dco::derivative(dco::derivative(y)[k%d])[l%d];
          dco::derivative(dco::value(x[l]))[l%d] = 0.0;
        }
      }
    }
  }
}
```

Listing 3.17: Seeding of vector tangent over vector tangent mode for arbitrary matrix size n and tangent vector size d.

```
typedef dco::ga1s<dco::gt1s<double>::type>::type ADtype;
extern ADtype f(ADtype* x,int n);

void t2s_a1s(int n, ADtype* x, double** H){
  // seed all directions, evaluate f n times
  for(int i=0; i<n; i++){
    dco::derivative(dco::value(x[i])) = 1.0;
    ADtype y = f(x,n);
    dco::value(dco::derivative(x[i])) = 1.0;

    ADmode::global_tape->interpret_adjoint();
    for(int j=0; j<n; j++){
      H[i][j] = dco::derivative(dco::derivative(y));
    }
    dco::derivative(dco::value(x[i])) = 0.0;
  }
}
```

Listing 3.18: Seeding of scalar tangent over scalar adjoint mode.

OpenFOAM. They did not require any changes in the code base, compared to the first order models (except the handling of checkpoints, if checkpointing is required). This leads us to believe that the computation of derivatives of order three and higher can also readily be implemented, by introducing the relevant nested data types. The required configuration options to enable second order derivatives in discrete adjoint OpenFOAM are listed in Appendix A.2.

In our observations, using second order AD to compute Hessians to speed up the convergence of optimization methods proved to be ineffective, due to the run time overhead introduced by the additional evaluations of the augmented primal. The potentially lower number of optimization steps required do not outweigh the increase in run time per optimization step. Furthermore, the convergence path of the topology optimizer to a local minimum proved to be quite noisy, leading to poor convergence of second order optimization methods, as they tend to choose small gradient step sizes.

The run time factors for a higher order solver of the `simpleFoam` solver are listed in Table 3.7. Listed is only the run time for a single evaluation of the augmented primal. To obtain a full Hessian, the models must be evaluated repeatedly.

Higher order sensitivities can be beneficial for parametric optimizations with limited number of parameters, as the run time overhead directly scales with the number of parameters for the

Table 3.7: Run times for a single derivative evaluation in first and second order tangent mode. Run time factor for tangent vector mode (vector size 16) is for a single gradient entry.

Solver	Run time (s)	Factor
`simpleFoam`	7.74	1
`t1sSimpleFoam`	28.73	3.71
`t1vSimpleFoam`	451.47	3.65
`t2st1sSimpleFoam`	100.29	12.96

tangent over adjoint model. In the parametric optimization procedure, introduced in Section 4.6, approximately constructed Hessians are used. In the future those approximations could be readily replaced by exact derivatives.

3.7 Profiling of Primal and Adjoint CFD Solvers

3.7.1 Compilation of Discrete Adjoint OpenFOAM

The compilation of (discrete adjoint) OpenFOAM requires a small subset of `C++11` and `C++14` features (mostly for template resolution). Therefore, a somewhat recent compiler is required to compile the discrete adjoint OpenFOAM framework. The compilation has been tested with gcc, clang, and the Intel icc compiler and different MPI implementations (OpenMPI, Intel MPI).

In Table 3.8 the compilation time for the OpenFOAM core package and a subset of solvers for adjoint mode (a1s), tangent mode (t1s) and passive mode are listed. For the release binaries, optimization flags (`-O3`) are set and debug symbols are disabled.

Between passive mode and a1s mode, the compile time increases by roughly 30%, regardless of the used compiler. For adjoint and passive mode, clang compiles the fastest, for tangent mode the compile times between all compilers are roughly on par.

For all configurations (including passive mode), a template instantiation depth of at least 41 is required. In absence of a documented compiler statistic reporting the instantiation depth, this number was obtained by gradually increasing the allowed template depth via the `-ftemplate-depth` compiler flag, until no errors occur during compilation. This finding underlines the complexity and reliance on templating of the code, making modes of AD other than operator overloading very challenging.

3.7.2 Identifying Hotspots

In order to understand the impact of AD on the run time and memory behavior of a given problem, it is important to accurately quantify where most of the run time and memory is spent during program execution. For run time profiling, several different approaches were tried. Namely profiling with `gprof`, the `callgrind` tool of `valgrind` and `gperftools`. All of these methods work by polling the program state at fixed (high frequency) intervals, extrapolating from these intervals how much time is spent in which subroutines. The function names and other program internals are obtained from the debug symbols inside the executable.

Table 3.8: Compilation times of discrete adjoint OpenFOAM (optimized build with `-O3`) on 4 cores with 8 threads for passive, adjoint (A1S) and tangent (T1S) mode with different supported compilers.

Compiler	A1S (min)	T1S (min)	Passive (min)
`g++` 4.9	42.05	33.60	31.05
`g++` 5.4	40.55	32.00	29.60
`g++` 6.3	40.50	32.40	30.20
`clang` 3.8	37.50	32.55	26.20

For assessing the memory consumption related to the adjoint mode, we use function level instrumentation. Function instrumentation allows to inject additional code, which is executed at entry and exit of each function (also for inlined functions). This can be used to precisely track the tape size, as well as current execution time at the entry and exit of each function call.

Three different approaches to inject the additional function calls into the code were investigated:

Using constructors and destructors. An auxiliary object is constructed at the entry of each function. The constructor and destructor (which is called once the function has returned and the object runs out of scope) of the object can be used to call the custom functions, which perform the run time and memory tracking. The auxiliary function is inserted into the code by performing a code transformation with the `clang-rewriter` tool[1]. This tool allows to identify function declarations in the AST, subsequently inject statements into the AST and transform them back to code. One advantage of this method is that the name and signature of the executed functions can be passed to the profiler directly as a string. The disadvantage is that the code rewriting process is complex and time consuming. It tends to miss certain functions, e.g. functions which are generated by C preprocessor macros.

Using the `-finstrument-functions` feature of gcc and clang. The `-finstrument-functions` compile flag defines the functions `func_enter` and `func_exit`. These functions are automatically called at the entry and exit of each function call. They can be overloaded to perform the desired actions. The advantage is that the function hooks are directly injected by the compiler, and thus no function calls are missed. One disadvantage is that the function name and signature of the instrumented functions are not available readily, but have to be reconstructed from the debug symbols in the executable. The lookup from debug symbols is rather expensive, though the overhead for repeated lookups can be kept low with a hash map implementation.

Using the `-fxray-instrument` feature of clang-5.0. This feature, recently added to llvm clang, allows to instrument functions much like `-finstrument-functions`, but gives greater control over the granularity of instrumentation through file and function lists. Furthermore, it respects an instrumentation threshold that allows to exclude functions with very high call counts but minimal impact (e.g. `operator[]`).

Here we focus on the second approach, as it produced the most reliable results. The third option would be preferred, due to the better granularity control, but the implementation in llvm seems incomplete at this point in time and documentation is lacking.

Instrumenting all functions produces a lot of data, which on a very fine granularity level is not of much use and considerably slows down program execution, as the instrument functions are not inlined by the compiler. Therefore, we deliberately disabled the instrumentation for regions of the code that mostly concern the handling, calculation, and storage of data on a very low level (e.g. `operator[]` on vectors). The influence of these omitted functions is not lost, but accumulated into functions higher up in the call tree. Instrumentation was omitted for the `src/OSspecific/` and `src/OpenFOAM/` folders in the source tree. Instrumentation of functions outside of the scope of OpenFOAM was disabled by blacklisting code in the system folders `/usr`, `/lib` and `/etc`.

In instrumentation mode, `dco/c++` creates a trace of the program execution, which lists all encountered functions in chronological order, as well as the connectivity of the functions. From this information, the full call graph of the program can be reconstructed. For every (instrumented) function entered and exited, the following information is stored:

[1] https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html

Number of children: Number of functions called by the function.

Memory size: Number of tape entries created by the function excluding children.

Cumulative memory size: Number of tape entries created by the function including children.

Input count: Number of adjoint inputs.

Output count: Number of adjoint outputs.

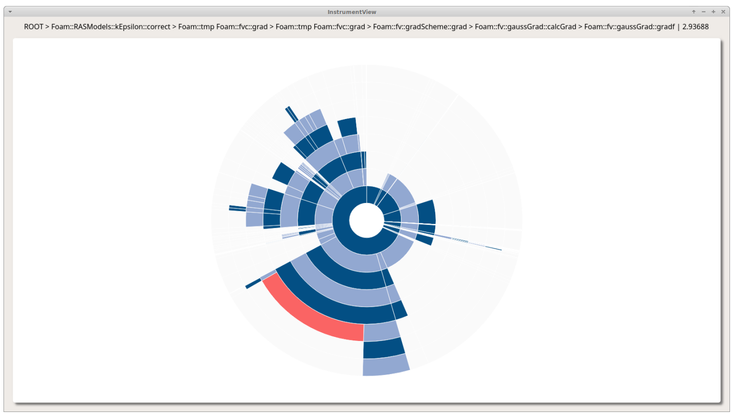
Function name: Name of the function, stripped from function and template arguments.

To obtain a run time profile of `adjointSimpleFoam`, we run the instrumentation on the laminar problem of the angled duct introduced in Section 3.1.2. Due to the high run time overhead of the instrumentation mode, only one iteration of the `adjointSimpleFoam` solver is traced.

In Figure 3.27 the memory sizes for all (direct) child functions of the main function are shown. The blue bars show the tape memory used for a single step of the `adjointSimpleFoam` solver in black-box mode. The red bars show the same functions, but with symbolically differentiated linear solvers. Consequently the red bars are considerably lower (observe the logarithmic scale of the x axis) than the blue ones for the linear solver calls and identical for the rest. Consequently, the main complexity (at least in terms of memory consumption) shifts away from the linear solvers to different places in the code.

In Figure 3.28 we show the same information for the Pitz-Daily example, with k-ϵ turbulence model. We see the same general behavior as with the laminar case, but with an additional memory spike for the correction of the turbulence model. The `kEpsilon::correct()` routine embeds the two linear solvers, required to solve the turbulence equations. Therefore, the memory consumption differs between black-box and symbolically differentiated linear solvers.

The full information of the instrumentation trace is best explored interactively, allowing to limit the high information density to regions of interest. A screenshot of a tool developed to visualize the call history is shown in Figure 3.26. The figure shows a breakdown of the function calls, including sub calls of up to level eight, for one iteration on the Pitz-Daily case. The hierarchical function calls are arranged in a sunburst diagram. The area of the slices correspond to run time or tape memory size.



ROOT > Foam::RASModels::kEpsilon::correct > Foam::tmp Foam::fvc::grad > Foam::tmp Foam::fvc::grad > Foam::fv::gradScheme::grad > Foam::fv::gaussGrad::calcGrad > Foam::fv::gaussGrad::gradf | 2.93688

Figure 3.26: Screenshot of interactive instrumentation visualization tool.

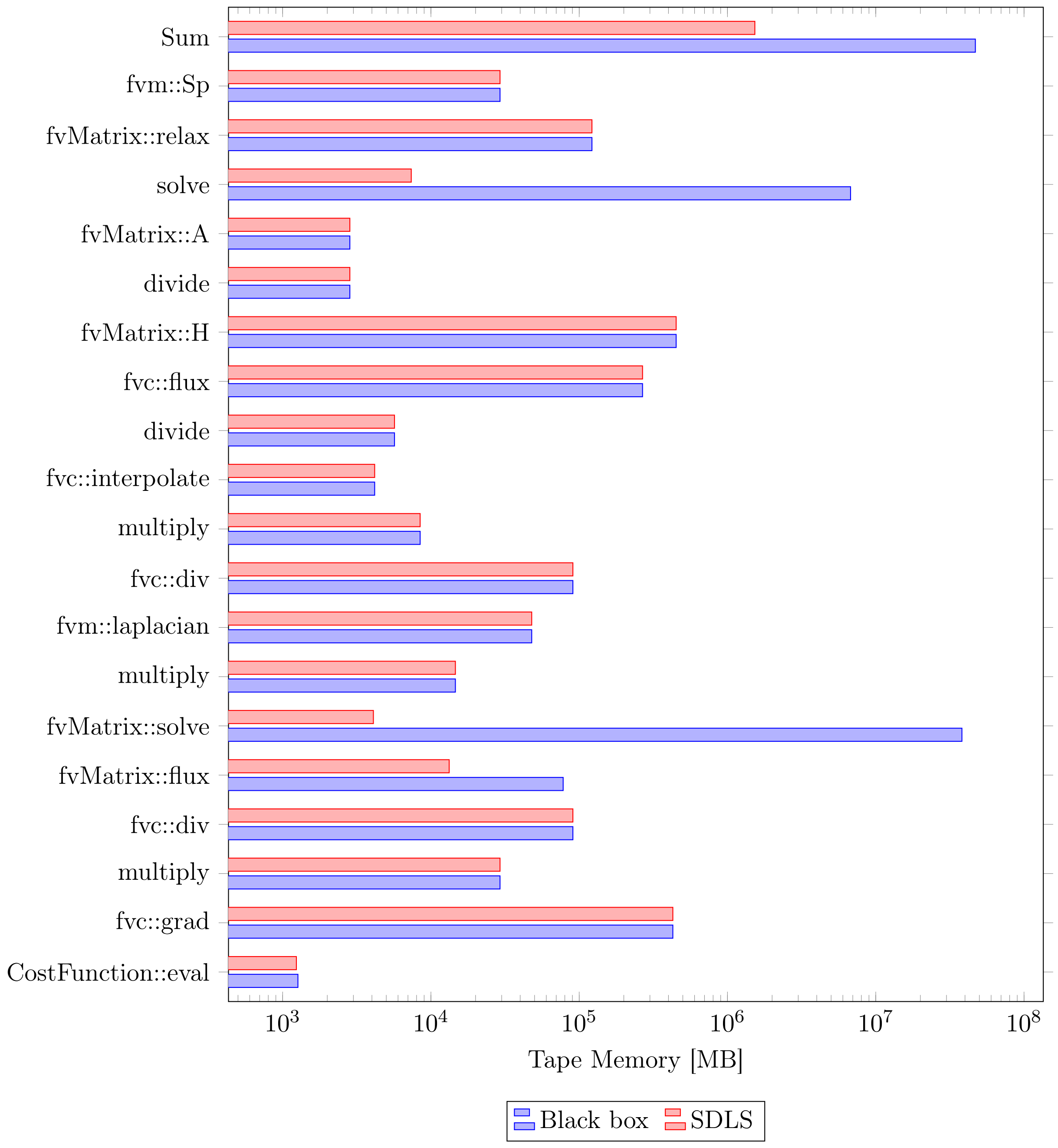


Figure 3.27: Tape size consumed by function calls in `adjointSimpleFoam` for the angled duct case. Black-box differentiation in blue and symbolic linear solvers in red.

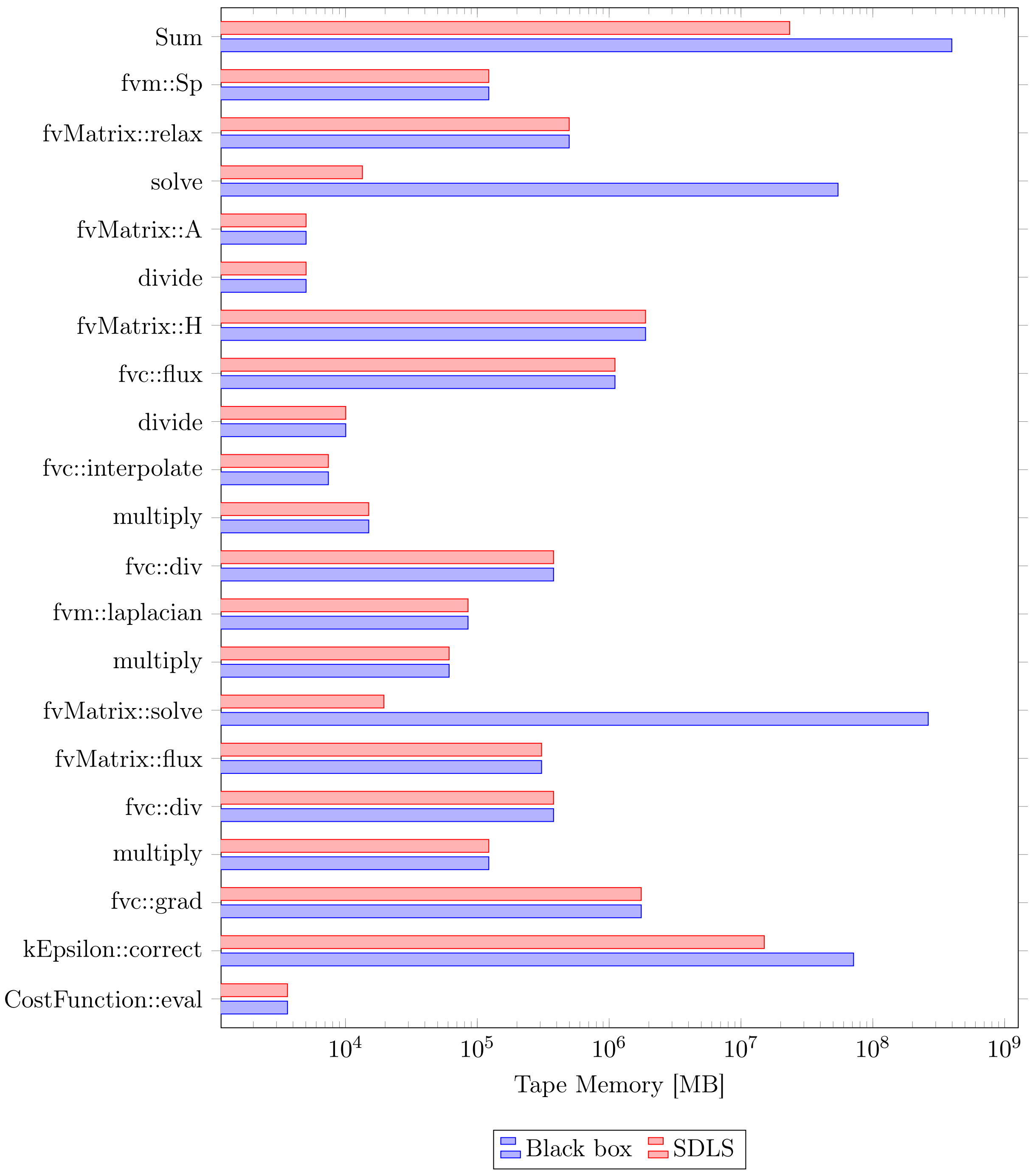


Figure 3.28: Tape size consumed by function calls in `adjointSimpleFoam` for the turbulent Pitz-Daily case. Black-box differentiation in blue and symbolic linear solvers in red.

3.7.3 Application of Profiling Results

Inspecting the detailed breakdown of function cost, one can begin to identify routines which can be further optimized by exploiting application insight. As an exemplary case study, we will inspect the function `lduMatrix::H()`, which ranks highly in the list of functions obtained by the instrumentation. This function calculates the negative product of the off diagonal entries L, U of a sparse matrix $A = L + D + U$ in `lduFormat` with a vector $\boldsymbol{\psi}$:

$$H\boldsymbol{\psi} = -\left(L + U\right)\boldsymbol{\psi} = \left(D - A\right)\boldsymbol{\psi}\,.$$

The product is implemented component wise in a loop over all faces in `src/OpenFOAM/matrices/lduMatrix/lduMatrix/lduMatrixUpdateMatrixTemplates.C` as: where `Hpsi` holds the result of the product, `lower` and `upper` hold the matrix coefficients $\boldsymbol{l}$, and $\boldsymbol{u}$ of the lower and upper triangular matrices L and U, and `lAddr` and `uAddr` store the row and column indices $\boldsymbol{L}$ and $\boldsymbol{U}$ of the matrix coefficients.

One use case for the `H()` operator is the assembly of the pressure correction equation from the calculated velocities (e.g. in `simpleFoam`).

In this application, `lower` and `upper` are scalar fields and `psi` is a vector field (i.e. the individual entries `psi[lAddr[face]]` are in R^3). The adjoints can be calculated symbolically by the following code, where `a1_lower`, `a1_upper`, `a1_psi` and `a1_Hpsi` are the adjoint vectors corresponding to `lower`, `upper`, `psi` and `Hpsi`. This function is a good candidate for introducing symbolic adjoints, due to its limited scope and easy to derive derivative. The adjoint code was implemented by applying AD to the computational kernel from Listing 3.19 by hand, according to the rules introduced in Section 2.7.2. For more complex codes, a source code transformation tool could be used.

The scalar vector multiplication in Lines 3–4 in Listing 3.19 induce scalar products (calculated by the OpenFOAM ampersand operator) in the adjoint code (Listing 3.20) in Lines 4 and 8. This is illustrated by the following example with scalar λ, and vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$:

$$\begin{aligned}
&\mathbf{y} = \lambda\mathbf{x}\\
\Rightarrow\ &\bar{\lambda} = \left(\frac{\partial\mathbf{y}}{\partial\lambda}\right)^T\cdot\bar{\mathbf{y}} = \mathbf{x}^T\cdot\bar{\mathbf{y}} = \sum_{i=0}^{n-1} x_i\cdot\bar{y}_i\\
\Rightarrow\ &\bar{\mathbf{x}} = \left(\frac{\partial\mathbf{y}}{\partial\mathbf{x}}\right)^T\cdot\bar{\mathbf{y}} = \lambda\bar{\mathbf{y}}\,.
\end{aligned}$$

Listing 3.20 shows the hand adjoined loop over the faces, calculating the adjoints of the matrix entries $\bar{L}$ and $\bar{U}$ and the vector $\boldsymbol{\psi}$. Due to the split of primal and adjoint calculation, additional

```
1  // Input: lower, upper, psi
2  for (label face=0; face<nFaces; face++)
3  {
4    Hpsi[uAddr[face]] -= lower[face]*psi[lAddr[face]];
5    Hpsi[lAddr[face]] -= upper[face]*psi[uAddr[face]];
6  }
7  // Output Hpsi
```

Listing 3.19: Implementation of the `H()` operator.

```
for (Foam::label face=0; face<nFaces; face++){
  // adjoin Hpsi[uAddr[face]] -= lower[face]*psi[lAddr[face]];
  a1_lower[face] -= psi[lowerAddr[face]] & a1_Hpsi[upperAddr[face]];
  a1_psi[lowerAddr[face]] -= lower[face] * a1_Hpsi[upperAddr[face]];

  // adjoin Hpsi[lAddr[face]] -= upper[face]*psi[uAddr[face]];
  a1_upper[face] -= psi[upperAddr[face]] & a1_Hpsi[lowerAddr[face]];
  a1_psi[upperAddr[face]] -= upper[face] * a1_Hpsi[lowerAddr[face]];
}
```

Listing 3.20: Loop over faces to calculate adjoints of `H()`.

boilerplate code is needed to correctly insert the symbolic adjoint calculation into the tape. This code, in slightly simplified form, is given in Listings 3.21 and 3.22, showing the creation of the gap in the tape and the symbolic calculation of adjoints during the interpretation. The process using adjoint callback functions is very similar to the implementation of SDLS.

The code calculates the product `Hpsi`, and thus this vector is registered as an output of the function `H()`. Its adjoints become an input to the calculation of the symbolic adjoint.

Due to the product rule, the primal values of `lower`, `upper`, and `psi` are needed in the reverse section. Copies of those vectors are saved in a checkpoint. To access elements of the adjoint vectors at the correct positions, the LDU addressing needs to be available in the symbolic adjoint function as well. The addressing of the matrix indices is assumed to be constant throughout the program execution. Thus, only a pointer to it is stored, instead of a full copy. If the mesh topology, and thus the connectivity of the faces, changes during program execution, full copies have to be saved.

The vectors `lower`, `upper`, and `psi` are the inputs to `H()` and are registered as inputs with the adjoint helper object `D`. Those adjoints have to be incremented with the calculated symbolic adjoints at the end of the adjoint callback function.

The size of the checkpoints, input, and output vectors all correspond to the number of cells in the mesh. In contrast, the number of tape entries created by the loop, which calculates the matrix vector product, over the faces obviously depends on the number of faces in the mesh (two assignments are recorded for each face). The symbolic treatment avoids creating those tape entries. The improvement achieved by the symbolic implementation is therefore expected to be higher the denser the matrix becomes, as the density is determined by the ratio between faces and cells. The savings should thus be best for complex polyhedral 3D meshes, where each cell is connected to multiple other cells.

In Table 3.9 we show the memory consumption and run time for the 2D reference cases from Section 3.1.2, as well as for the OpenFOAM motorbike case, which has a more complex 3D mesh. The savings observed in practice by the symbolic implementation of `H()` are pretty minor ($< 5\%$), due to the rather high amount of data which needs to be checkpointed. However, no cases with decreased performance were observed. Thus, the optimization is still worthwhile and serves as a template for further optimizations.

```
Foam::Field<Foam::vector>> Foam::lduMatrix::H(const Field<Foam::vector>& psi)
{
  Field<vector> Hpsi(lduAddr().size(), Zero);
  if (lowerPtr_ || upperPtr_){ // only needed if matrix not diagonal
    const label nFaces = upper().size();

    ADmode::global_tape->switch_to_passive();
    D = ADmode::global_tape->create_callback_object<ADmode::
        external_adjoint_object_t>();
    // register adjoint inputs
    forAll(lower(), i)
      D->register_input(lowerPtr[i]);

    forAll(upper(), i)
      D->register_input(upperPtr[i]);

    forAll(psi, i){
      D->register_input(psiPtr[i][0]);
      D->register_input(psiPtr[i][1]);
      D->register_input(psiPtr[i][2]);
    }

    D->write_data(lower());
    D->write_data(upper());
    D->write_data(psi);
    D->write_data(&(lduAddr().lowerAddr()));
    D->write_data(&(lduAddr().upperAddr()));

    for (label face=0; face<nFaces; face++){
      Hpsi[uAddr[face]] -= lower[face]*psiPtr[lAddr[face]];
      Hpsi[lAddr[face]] -= upper[face]*psiPtr[uAddr[face]];
    }

    forAll(Hpsi, i){
      Hpsi[i][0] = D->register_output(dco::passive_value(Hpsi[i][0]));
      Hpsi[i][1] = D->register_output(dco::passive_value(Hpsi[i][1]));
      Hpsi[i][2] = D->register_output(dco::passive_value(Hpsi[i][2]));
    }
    ADmode::global_tape->insert_callback<ADmode::external_adjoint_object_t>(gapH
        ,D);
    ADmode::global_tape->switch_to_active();
  }
  return Hpsi;
}
```

Listing 3.21: Primal code for `H()`, augmented to enable symbolic differentiation in the reverse interpretation.

```cpp
inline void gapH(typename Foam::ADmode::external_adjoint_object_t *D){
  const Foam::scalarField& lower = D->read_data<Foam::scalarField>();
  const Foam::scalarField& upper = D->read_data<Foam::scalarField>();
  const Foam::vectorField& psi = D->read_data<Foam::vectorField>();
  const Foam::labelUList& lowerAddr = *(D->read_data<Foam::labelUList*>());
  const Foam::labelUList& upperAddr = *(D->read_data<Foam::labelUList*>());

  Foam::scalarField a1_lower(lower.size(),Foam::Zero);
  Foam::scalarField a1_upper(upper.size(),Foam::Zero);
  Foam::vectorField a1_psi(psi.size(),Foam::Zero);
  Foam::vectorField a1_Hpsi(psi.size(),Foam::Zero);

  forAll(a1_Hpsi,i)
    for(int j=0;j<3;j++)
      a1_Hpsi[i][j] = D->get_output_adjoint();

  Foam::label nFaces = upper.size();
  for (Foam::label face=0; face<nFaces; face++){
    a1_lower[face] -= psi[lowerAddr[face]] & a1_Hpsi[upperAddr[face]];
    a1_psi[lowerAddr[face]] -= lower[face] * a1_Hpsi[upperAddr[face]];

    a1_upper[face] -= psi[upperAddr[face]] & a1_Hpsi[lowerAddr[face]];
    a1_psi[upperAddr[face]] -= upper[face] * a1_Hpsi[lowerAddr[face]];
  }

  forAll(lower,i)
    D->increment_input_adjoint(dco::passive_value(a1_lower[i]));

  forAll(upper,i)
    D->increment_input_adjoint(dco::passive_value(a1_upper[i]));

  forAll(psi,i)
    for(int j=0;j<3;j++)
      D->increment_input_adjoint(dco::passive_value(a1_psi[i][j]));
}
```

Listing 3.22: Symbolic adjoint code for `H()`.

Table 3.9: Comparison between regular `H()` and symbolic adjoint implementation.

Test case	Variant	Adjoint vector size	Total tape size (MB)	Run time (s)
Angled duct	regular `H()`	177467477	6093.35	25.75
	symbolic `H()`	171887480	5966.35	25.68
Pitz-Daily	regular `H()`	121031438	3951.65	10.56
	symbolic `H()`	118963913	3897.06	10.50
Motorbike	regular `H()`	709475801	25049.55	51.53
	symbolic `H()`	686295043	24420.04	51.50

3.8 Overcoming AD Memory Limits

3.8.1 File Tape

The AD tool `dco/c++` supports multiple options to store the tape in memory (RAM or disk):

Blob Tape: Use an adjoint stack of fixed length. Memory is reserved when the tape is created. Most performant option, as no bounds need to be checked.

Chunk Tape: Use an adjoint stack which consists of multiple chunks, that are allocated on demand. Most flexible option, slightly less performant.

File Tape: Use chunks, but buffer full chunks to files on the file system. File IO is handled by the kernel and chunks may be fully buffered in memory. Significantly slower than both blob and chunk tape, but allows running big calculations when checkpointing and offloading with AMPI is not feasible.

The file tape can be utilized when memory demands are high and checkpointing is hard to implement. One such case would be, if for an iterative simulation one time step does not fit into the tape, requiring a more granular checkpoint level. To get reasonable performance from the file tape, fast memory is needed. The access patterns are sequential writes during the recording phase and sequential reads from the adjoint stack and random access read and writes on the adjoint vector during the interpretation phase.

To predict the performance of the file tape, multiple tests were performed on different architectures. Studied were the performance of a SSD RAID ($6 * 1024$ GB in RAID 0) and an Intel NVME installation on the RWTH compute cluster. For RAID 0, the capacity of a volume is the sum of the capacities of the involved disks. The latency and throughput is improved by striping, i.e. data of consecutive writes is distributed over multiple disks, reducing the IO bandwidth needed on the individual disks. For random access, it also improves latency as multiple read/write operations can be performed at the same time.

Table 3.10: SSD benchmark reading/writing of 10 GB of data in 10 MB blocks with GNU `dd`. Average is taken over ten samples and rounded to next multiple of 5 MB.

Mode	`dd` flag	SSD IO (MB/s)	NVME IO (MB/s)
write	no flags	950	1570
	`direct`	1765	1675
	`dsync`	830	705
	`sync`	710	740
	`nocache`	910	1250
read	no flags	350	1580
	`direct`	-	-
	`dsync`	350	755
	`sync`	340	1635
	`nocache`	500	1580

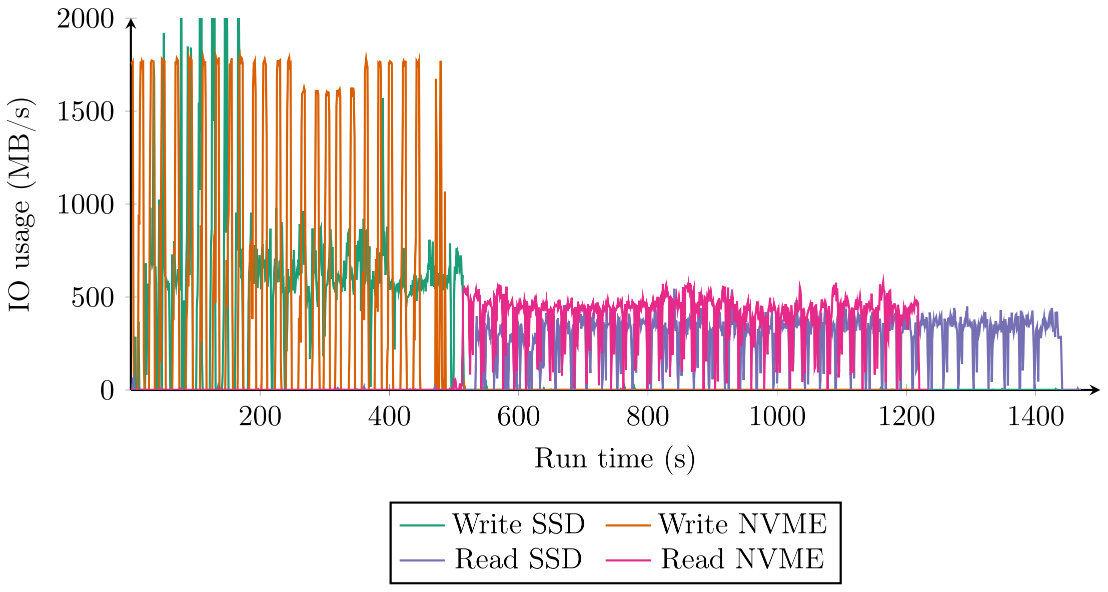


Figure 3.29: Read and write IO usage of the SSD/NVME devices for 40 iteration steps. Approx. 300 GB of data is written to disk during the augmented forward run, and read back during the adjoint propagation phase.

The sustained read and write performance for the SSD and NVME systems, benchmarked by the GNU `dd` command, is shown in Table 3.10. Different modes of disk access are specified by the `-iflag/-oflag`. Data is read/written in consecutive chunks of block size 10 MB. For small blocks, the read and write performance is significantly reduced, however as `dco/c++` reads and writes the adjoint stack in chunks (corresponding to the chunks of a regular tape), the full read and write bandwidth should be used by the `dco/c++` file tape implementation. For the SSD system, the OS caches are flushed after each benchmark run. We do not have root access to the NVME systems, the caches can thus not be manually flushed. For the NVME system, the RAM is artificially filled as much as possible by another process, to avoid skewing of the results by system level read/write buffers. The numbers obtained by these benchmarks correspond to the maximum read/write rates observed with `dco/c++`, with the exception of SSD write speed, where the full bandwidth indicated by `dd` in direct mode could not be achieved.

To evaluate the file tape performance, we study a case run with `adjointSimpleShapeFoam` on a S-bend geometry for 40 iteration steps. The case consumes 307 GB of tape memory, which is written to the hard disk in chunks of one GB, (i.e. one file is created on the disk for every chunk). The adjoint vector is kept fully in RAM, due to the non consecutive memory access pattern involved in the reverse interpretation of the tape. The high latency (compared to RAM) of disk based storage makes non consecutive (random) memory access expensive. For this case, the adjoint vector occupies 70 GB of RAM. For bigger cases, the size of the adjoint vector will quickly become a bottleneck. A technique to overcome this bottleneck will be presented in the next section.

The IO bandwidth (as measured by `iotop`) during augmented forward run and adjoint reverse propagation is shown in Figure 3.29. As SDLS (see Section 3.4) is enabled, new entries for the adjoint stack are only generated outside of the linear solver iterations. During the augmented forward run the NVME system consistently reaches a write speed of 1.8 GB/s. This is fast enough, such that the pauses in the stream of write operations due to the symbolic differentiation of linear solvers become visible in the IO bandwidth. The SSD system reaches a level of 650 MB/s, after an initial peak of higher bursts. We suspect these peaks to be a combination of OS buffering and the SSD write caches, which both become saturated after roughly 100 GB of data has been written. For the SSD system, no idle periods in the IO bandwidth are visible, as the Linux kernel buffers the write operations in memory. Apparently the write performance is not high enough to completely drain those buffers during the idle periods. However, the write performance for the SSD systems is high enough not to majorly impact the forward run time, compared to the NVME system.

The read performance of both the NVME and SSD system is considerably lower than the write performance, maxing out at 300 GB/s for the SSD based system and 550 GB/s for the NVME system. In contrast to the write operations, the read operations are blocking. Therefore, if a chunk of memory is not already contained in the OS cache, the calculation will stall until the chunk is completely fetched from disk. The read performance can potentially be improved by prefetching some data, reducing idle time in the IO subsystem. The lower read performance of the SSD system clearly effects the run time of the adjoint propagation, as it takes considerably longer on the SSD system than on the NVME system.

3.8.2 Adjoint Vector Compression

As already mentioned in the previous section, the adjoint vector, which in `dco/c++` is stored in RAM to retain acceptable random access performance, limits the usefulness of offloading the adjoint stack to secondary storage.

In order to calculate the adjoints for each entry of the adjoint vector, the entries of the adjoint vector connected to it by outgoing edges must be accessible. For iterative methods, where many changes in the states are only local to the current iteration and do not directly influence any values in the next iterations, the required adjoint vector size can be bounded by the longest edge in the tape. Once the tape interpretation has moved past a certain element in the adjoint vector (that is all partial derivatives of the element have been incremented along its outgoing edges), its location in the adjoint vector can be reassigned to another element, which comes later in the interpretation procedure and has not yet been incremented. The number of elements required to be present in the compressed adjoint vector at the same time is bound by the maximum distance between two elements, connected by an edge, in the uncompressed adjoint vector. The concept of the compressed adjoint vector was introduced in [NL18]. This thesis will focus on the implementation in the context of complex iterative algorithms, the performance cost associated with the adjoint vector compression, and a novel implementation to mostly recover the added cost.

We will first illustrate the adjoint vector compression with an example, before showing it more rigorously. As a simple iterative algorithm we consider the Babylonian root finding method. This fixed point iteration approximates the square root of a positive real number $\mathbb{R}^+ \to \mathbb{R}^+ : x = \sqrt{a}$ by the following iteration procedure:

$$x = \frac{1}{2}\left(x + \frac{a}{x}\right) .$$

This iteration procedure can be straightforwardly derived from Newtons method (see Section 2.9.2), applied to find the root of $f(x) = x^2 - a$, as shown in Equation (3.6).

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{3.6}$$

It was developed independent of Newtons method several centuries earlier by Babylonian and Greek mathematicians, hence the name.

From an initial guess $x_0 = a = 2$ we find the square root $x_3 \approx \sqrt{2}$ with an error of 0.00015% after only 3 iterations. To introduce some intermediate variables, which are local to the iteration, we decompose the fixed point iteration into the following equivalent SAC:

```
ADtype x = a; // initial guess
for(int i = 0; i<3; i++){
  ADtype v1 = a/x;
  ADtype v2 = v1 + x;
  x = 0.5*v2;
}
```

Listing 3.23: SAC of Babylonian root finding algorithm for three iterations.

Using the adjoint method of AD we calculate the approximation of the derivative

$$\left.\frac{\partial\sqrt{x}}{\partial x}\right|_{x=2} = \frac{1}{2\sqrt{2}}$$

by adjoining the three iterations of the fix point iteration up to an error of 0.00177%.

Figure 3.31 shows the tape structure generated by the three steps of the algorithm. The parameter a is incremented from each iteration step, the adjoint of a must therefore be available in the adjoint vector during the interpretation of all iterations. However, the individual iterations do not depend on any other iteration steps, except of the state of the directly preceding iteration. Outgoing edges of an entry of the adjoint vector only influence entries which are located before it in the adjoint vector (that is the partial derivatives were created earlier in the primal evaluation phase). Entries below the current interpretation location are never read from or incremented again. Those adjoints can be safely discarded, assuming they are not desired for some other adjoint calculation, and the space in the adjoint vector may be reused for another entry. We will now formalize this observation.

Theorem 6.
Let l denote the maximum distance between two entries $e_i, e_j, i > j$ of the adjoint vector that are connected by an edge $(e_i, e_j) \in E$. Then the number of elements, that need to be retained in the adjoint vector, is bounded by sharply $l + 1$.

Proof. By induction we will show that a rolling window over the adjoint vector of length $l + 1$ is enough to perform all increments. Let $\mathbf{S} = [s_0, \ldots, s_{n-1}] \in \mathbb{R}^n$ be the full adjoint vector and $\mathbf{S}^i = [s_{\max(0,n-i-l)}, \ldots, s_{n-i-1}]$ be a sub vector including at most $l + 1$ consecutive elements of $\mathbf{S}$ starting at index $\max(0, n - i - l)$. This can be pictured as a rolling window moving backwards over the full adjoint vector.

Base Case: Incrementation of adjoints connected to adjoint entry s^{n-i-1} for $i = 0$:
All adjoint vector entries s_j with $(s_{n-1}, s_j) \in E$ have to be incremented. As the distance between s_{n-1} and s_j is at most l, the adjoint vector $\mathbf{S}^0 = [s_{n-l}, \ldots, s_{n-1}]$ contains all required entries s_j. Note how the element from which all edges relevant for this step originate is the last entry of the rolling window.

Inductive step: incrementation of adjoints connected to adjoint entry s^{n-i-1} for $i = k + 1$:
The last element of the previous inductive step is not needed any more, because all outgoing edges s_j with $(s_k, s_j) \in E$ have been incremented and there can not be any backward pointing edges $(s^i, s^j) \notin E$ for all $j > i$. We can thus transform $\mathbf{S}^k$ into $\mathbf{S}^{k+1}$ by removing the last element and inserting one more element at the front, namely $s_{n-l-(k+1)}$. By the definition of l the vector $\mathbf{S}^{k+1}$ again includes all elements s_j with $(s_{n-1-(k+1)}, s_j) \in E$, required to increment all entries connected to the last entry of the adjoint vector $\mathbf{S}^{k+1}$.

The induction terminates when all entries of the adjoint vector have been fully incremented (for $i = n - 1$). □

The naive adjoint vector compression breaks down, if there are edges in the tape that span multiple iteration steps. This is commonly the case if each state depends on a set of parameters, that is defined at the beginning of the program. For illustration, see the dashed arrows in Figure 3.30. To overcome this, the set of parameters is made available at each point of the tape interpretation. Assuming that the first p entries of the adjoint vector $\mathbf{S}$ are parameters, the size of the compressed adjoint vector can again be bound by $\tilde{\mathbf{S}}^i \in \mathbb{R}^{l_s+1+p}$, with the following new definitions for $\tilde{\mathbf{S}}^i$ and l_s:

$$l_s = \max_{(s_i,s_j)\in E,\, j\geq p} i - j$$
$$\tilde{\mathbf{S}}^i = [s_0, \ldots, s_{p-1}, s_{n-l_s-i+p}, \ldots, s_{n-i-1+p}] \ .$$

We call the set of parameters fixed at the start of the adjoint vector *perpetuated* parameters [NL18]. Then l_s is the length of the longest edge in the tape, not connected to a perpetuated parameter.

In practical implementation, the transformation of vector $\mathbf{S}^i$ to $\mathbf{S}^{i+1}$ would either need copying of data, or a more complex linked data structure that allows to remove the first element and append a new element at constant cost. To avoid this added complexity, instead the addressing into the vector is implemented relying on the cyclic properties of the modulo operation.

This allows to reuse the same allocated memory for a sequence of vectors $\hat{\mathbf{S}}_i$ without the need to copy any values. This is best illustrated with an example. The cyclic shifting of the adjoint vector is illustrated in Table 3.11.

Table 3.11: Adjoint vector of length eight, with longest edge three. Rolling window compression (left) and modulo shifting (right). Current position in interpretation underlined.

$\mathbf{S}^0$	$\mathbf{S}^1$	$\mathbf{S}^2$	$\mathbf{S}^3$	$\mathbf{S}^4$	$\mathbf{S}^5$	$\mathbf{S}^6$	$\mathbf{S}^7$
s_4	s_3	s_2	s_1	s_0	-	-	-
s_5	s_4	s_3	s_2	s_1	s_0	-	-
s_6	s_5	s_4	s_3	s_2	s_1	s_0	-
$\underline{s}_7$	$\underline{s}_6$	$\underline{s}_5$	$\underline{s}_4$	$\underline{s}_3$	$\underline{s}_2$	$\underline{s}_1$	$\underline{s}_0$

$\hat{\mathbf{S}}^0$	$\hat{\mathbf{S}}^1$	$\hat{\mathbf{S}}^2$	$\hat{\mathbf{S}}^3$	$\hat{\mathbf{S}}^4$	$\hat{\mathbf{S}}^5$	$\hat{\mathbf{S}}^6$	$\hat{\mathbf{S}}^7$
s_4	s_4	s_4	$\underline{s}_4$	s_0	s_0	s_0	$\underline{s}_0$
s_5	s_5	$\underline{s}_5$	s_1	s_1	s_1	$\underline{s}_1$	-
s_6	$\underline{s}_6$	s_2	s_2	s_2	$\underline{s}_2$	-	-
$\underline{s}_7$	s_3	s_3	s_3	$\underline{s}_3$	-	-	-

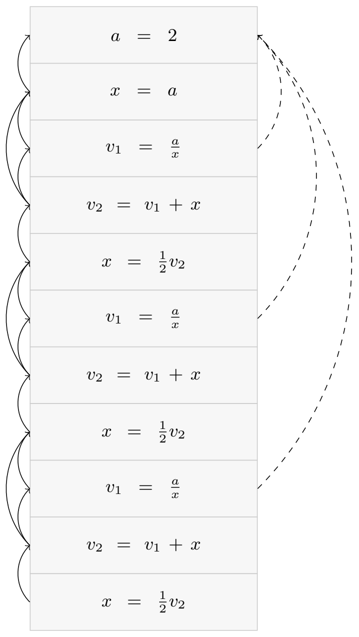


Figure 3.30: Uncompressed tape representation of the Babylon root example.

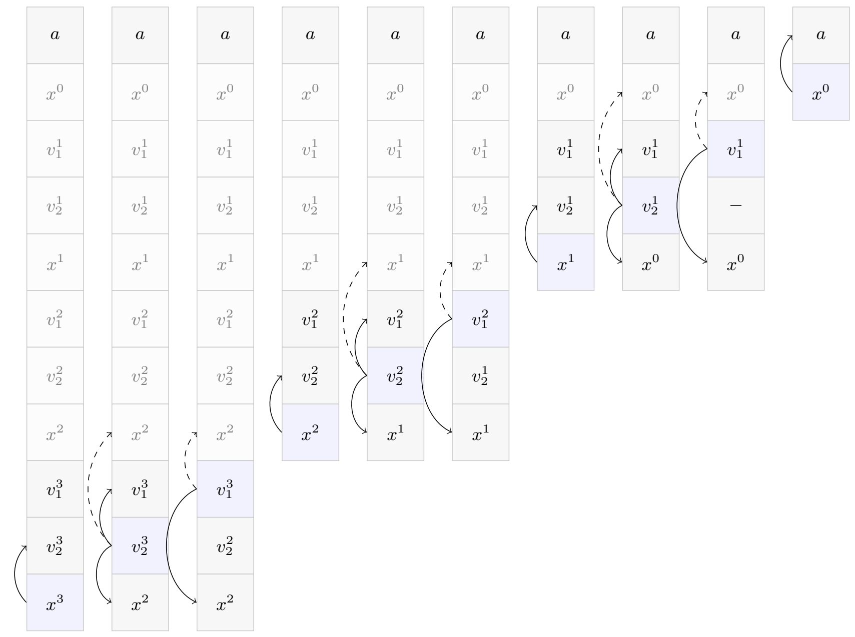


Figure 3.31: Adjoint vector compressed to the longest edge between states, excluding edges from states to parameters. Dashed lines indicate references to the virtual uncompressed adjoint vector.

The interpretation phase, using the adjoint vector compression technique, for the Babylonian example is illustrated in Figure 3.31. There is only one perpetuated parameter a. The longest edge, not connected to parameter a, spans two nodes. The required size for the compressed adjoint vector is thus $n_p + l + 1 = 1 + 2 + 1 = 4$.

To further illustrate the principle of this approach, an implementation of a basic AD tool, applied to the Babylonian root problem, is included in Appendix D. The tool implements both the tape interpretation for a regular tape and a modulo compressed tape. For the exact transformation of the full adjoint vector to the individual entries of the modulo compressed adjoint vector, refer to this code. The implementation in `dco/c++` is functionally identical, however not as transparent due to the added complexity of using a template engine.

The main benefit of the adjoint vector compression is that the part of the tape that needs to be stored in RAM can be made independent of the number of iterations. This allows to differentiate through a high number of iterations by offloading the growing part of the tape to HDD/SSD storage while keeping the latency sensitive adjoint vector in RAM completely.

Implementing Adjoint Vector Compression in OpenFOAM

The only change required in the discrete adjoint OpenFOAM framework, in order to activate the compression of the adjoint vector, is to replace the adjoint datatype `dco::ga1s<double>::type` with the `dco::ga1s_mod<double>::type` data type. The compression of the adjoint vector is not enabled by default, as it adds a run time penalty to the tape interpretation phase. The added modulo operations add additional operations and prohibit the CPU from utilizing speculative execution.

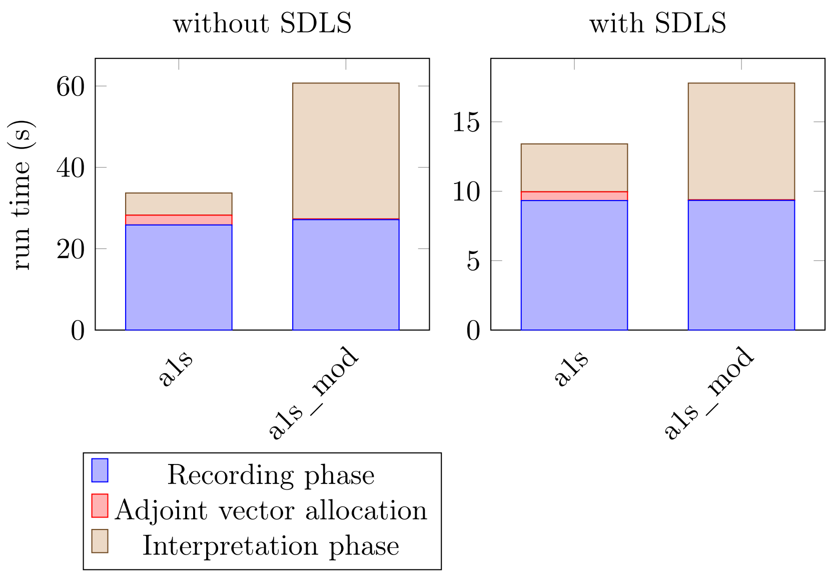


Figure 3.32: Run time of 16 iterations of the Pitz-Daily case using the uncompressed (a1s) and the modulo compressed (a1s_mod) adjoint vector implementation.

As a heuristic to determine the set of parameters which need to be perpetuated, `dco/c++` treats all variables that are explicitly registered with the tape as parameters. After changing the adjoint data type, the remaining code of `adjointSimpleFoam` can be left unchanged, as the first action after creating the tape is already to register all parameters $\boldsymbol{\alpha}$, making them perpetuated by default.

Before we examine the memory savings achieved by the adjoint vector in detail, we will first look at the run time. The run times for 16 iterations of the `adjointSimpleFoam` solver on the Pitz-Daily case, with and without adjoint vector compression, is shown in Figure 3.32. While the run time of the augmented primal evaluation stays nearly identical, the time required for the interpretation of the tape grows considerably. This illustrates the additional computational complexity introduced by the modulo operation, performed with each increment of adjoint variables. The time required for the allocation of the adjoint vector decreases, already hinting at a reduction in RAM usage.

The extent of the run time penalty, introduced by the modulo operator, and consequently an optimization to reduce the overhead of the reverse adjoint propagation, will be presented in the following section.

3.8.3 Bitwise Modulo Optimization

The introduction of the compressed adjoint vector increases the run time of the reverse adjoint propagation phase. Discrete adjoint OpenFOAM seems to be particularly susceptible to this increase in run time, other synthetic benchmarks in the `dco/c++` benchmark suite exhibit a less significant run time increase. The root cause for the run time increase is the execution of the modulo operator in every adjoint incrementation. The data flow reversal, at least in parts of the calculation, is not memory bound, such that this additional numerical operation has a significant impact on the total run time. A naive implementation of the modulo operator might look like

Table 3.12: Calculation of (13 mod 8) = 5 using the bitwise operation optimization for divisors of power two.

	b_3	b_2	b_1	b_0
a	1	1	0	1
b	1	0	0	0
$b-1$	0	1	1	1
$a \mathbin{\&} (b-1)$	0	1	0	1

the following [Knu97]:

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b\,.$$

The actual implementation by the compiler might be optimized, but will still generally require an integer or floating point division.

The run time of the adjoint propagation phase can be reduced by expanding the adjoint vector size to the next power of two, allowing to replace the modulo operation by a more efficient bitwise operation. On all modern platforms bitwise operators are executed very efficiently, especially compared to the floating point division operation involved in the calculation of the modulo remainder. In the worst case, the adjoint vector grows by a factor of two, when expanding the size to the next power of two. Due to the small adjoint vector size, compared to the stack memory size, this should be negligible in most cases. We will now show that the equivalence between modulo operation and bitwise AND operation holds for divisors which are powers of two.

Theorem 7.
Let $i \in \mathbb{N}$, $b = 2^i$ and $a \in \mathbb{N}^+$. Then the following equivalence holds:

$$a \bmod b = a \mathbin{\&} (b-1)\,,$$

where & is the bitwise AND operator.

Proof. The bitmask $b-1$ masks all bits lower than the single significant digit corresponding to b. The bitwise AND operation then strips away all digits of a left of and including the digit in b, removing all integer multiples of b from a. This yields the desired remainder $a - \left\lfloor \frac{a}{b} \right\rfloor b$:

$$b - 1 = \sum_{j=0}^{i-1} 2^i$$

$$a \mathbin{\&} (b-1) = a - \sum_{j=i}^{\lceil \log_2(a) \rceil} 2^i = a - \left\lfloor \frac{a}{b} \right\rfloor b\,.$$

□

The bitwise AND calculation, replacing a modulo operation, is illustrated in Table 3.12 on the example $\left(13 \bmod 2^3\right) = 5$.

To quantify the impact of the operations, we will first analyze the impact of the modulo and bitwise AND operator on a synthetic benchmark and its disassembly. The loops in Listing 3.24

```
// modulo
for(int i = 0; i < 1024; i++){
  a = a % b;
}
// bitwise AND
for(int i = 0; i < 1024; i++){
  a = a & (b-1);
}
```

Listing 3.24: Synthetic benchmark for modulo and bitwise AND operation.

are benchmarked using the google benchmark framework[2] with random positive integers for a and $b = 2^i$.

As can be seen in the disassembly in Listing 3.25, on systems with a `x86` or `IA64` based processor the compiler schedules a call to the `idivl` [Int16] instruction, which computes the quotient, as well as the remainder of two signed integer numbers. The computation can be slightly sped up (approx. 5%) by switching a and b to signed data types. Measuring the execution time of individual instructions on modern architectures is not straightforward, due to instruction fusing and out of order execution. The following benchmarks should thus only be interpreted qualitatively.

The `for`-loops around the operations are necessary to capture a measurable time delta for the bitwise AND operation at all. The benchmark framework polls the loops several hundred thousand times to get a reliable average of the run time. The benchmark tool reports an average run time of 6328 ns for the first loop and a run time of 100 ns for the second loop, with a standard deviation of below one percent. The run time of the second loop can be further reduced to 89 ns by eliminating the subtraction of one (as the adjoint vector size is known when entering the data flow reversal the subtraction can be performed once instead of at each calculation of the modulus). Absent of any memory access (a and b are held in registers) the bitwise AND operation is faster than the modulo operation by a factor of above 60. Assuming the full clock rate of 3.3 GHz of the machine can be utilized for the benchmark, this nets an average execution time of 0.25 CPU cycles per bitwise AND instruction and 20 cycles per regular modulo instruction, indicating that the compiler is able to efficiently vectorize the instructions.

Switching from the synthetic benchmarks to the application in CFD codes, we expect the savings by the bitwise AND operation to be much smaller, as not all operands are stored in processor registers. Therefore, memory latency is introduced which will affect both implementations equally.

In Figure 3.33 the earlier picture showing 16 iterations of the `adjointSimpleFoam` solver is updated with the newly obtained results from the bitwise AND operator. The modulo operations added a significant run time overhead to the regular adjoint solver. Almost the whole overhead can be recovered by switching to the bitwise AND operation instead. As seen previously the allocation of the adjoint vector decreases in run time significantly for both modulo implementations, due to the much smaller adjoint vector.

[2] `https://github.com/google/benchmark`

```
void foo(int a, int b){
  int c = a % b;
  int d = a & (b-1);
}
void foo(int a, int b){
     0: 55               push   %rbp
     1: 48 89 e5         mov    %rsp,%rbp
     4: 89 7d ec         mov    %edi,-0x14(%rbp)
     7: 89 75 e8         mov    %esi,-0x18(%rbp)
  int c = a % b;
     a: 8b 45 ec         mov    -0x14(%rbp),%eax
     d: 99               cltd
     e: f7 7d e8         idivl  -0x18(%rbp)          // signed integer division
    11: 89 55 f8         mov    %edx,-0x8(%rbp)
  int d = a & (b-1);
    14: 8b 45 e8         mov    -0x18(%rbp),%eax
    17: 83 e8 01         sub    $0x1,%eax            // subtract 1
    1a: 23 45 ec         and    -0x14(%rbp),%eax     // bitwise AND
    1d: 89 45 fc         mov    %eax,-0x4(%rbp)
}
```

Listing 3.25: Disassembly of Lines 1-4, as generated by `g++ -c -O0 -g`. Optimization will remove some of the move instructions, however the arithmetic operations remain unchanged.

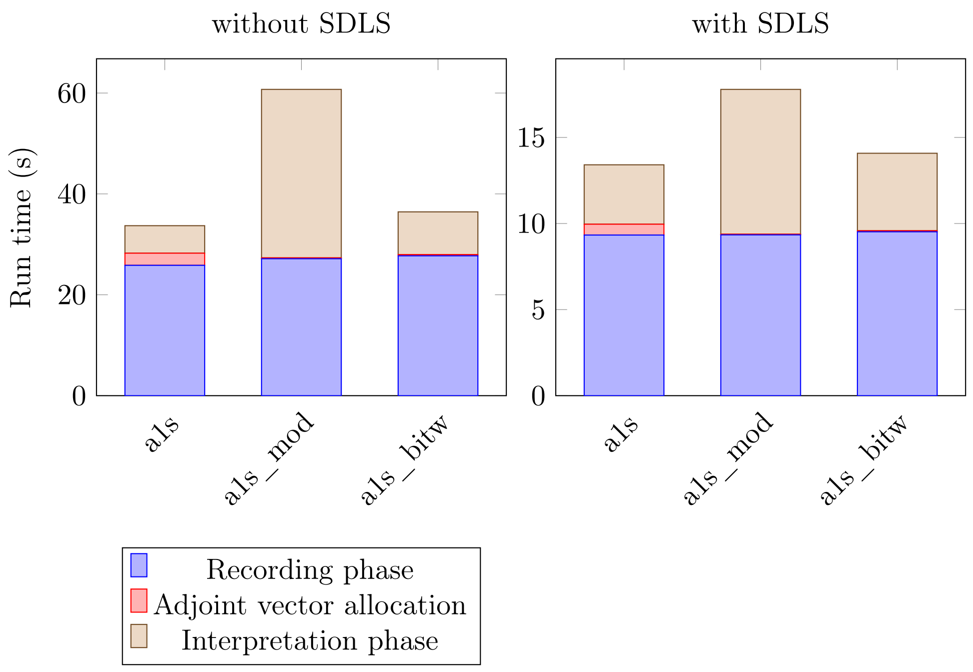


Figure 3.33: Run time of 16 iterations of the Pitz-Daily case using the modulo (a1s_mod) and bitwise AND (a1s_bitw) adjoint vector compression.

The four Figures 3.34–3.37 show the same case, with different number of iterations between one and 64. The maximum size of 64 iterations is chosen such that the tape fills the 128 GB of RAM of the test machine almost completely when not utilizing SDLS. For 64 iterations, the tape consumes around 100 GB of RAM, the uncompressed adjoint vector around 19 GB. Utilizing the modulo operation, the size of the adjoint vector compresses down to 365 MB. When increasing the adjoint vector length to the next power of two, the size of the adjoint vector grows to 512 MB. The figures clearly show, that the compressed adjoint vector size stays constant for increasing iteration counts. The run time for the interpretation increases slightly when utilizing the bitwise AND optimization, but grows with the same rate as the uncompressed version. All effects of the adjoint vector compression, both in terms of run time and compression, are less pronounced for the SDLS enabled runs, as the ratio of instructions treated by `dco/c++` compared to the overall operations is lower in those cases.

Summarizing, the modified version of the adjoint vector compression allows to regain most of the performance, that was lost when the regular modulo adjoint vector compression was introduced. The adjoint vector compression makes large calculations feasible, which would have been limited by the memory size of the adjoint vector before.

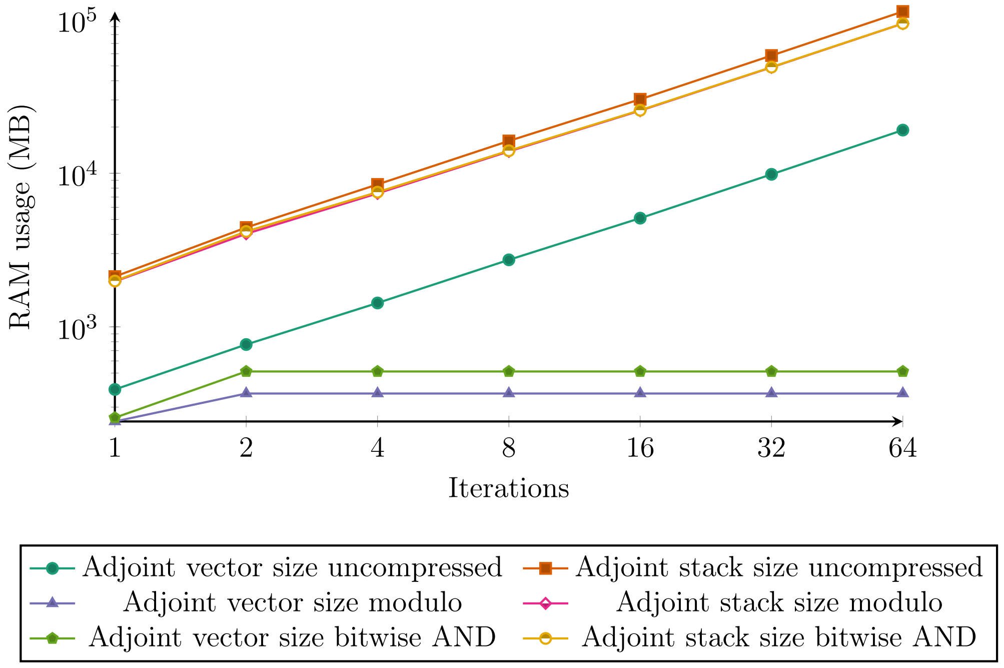


Figure 3.34: RAM usage without SDLS for uncompressed and compressed adjoint vector using the modulo and bitwise AND operators.

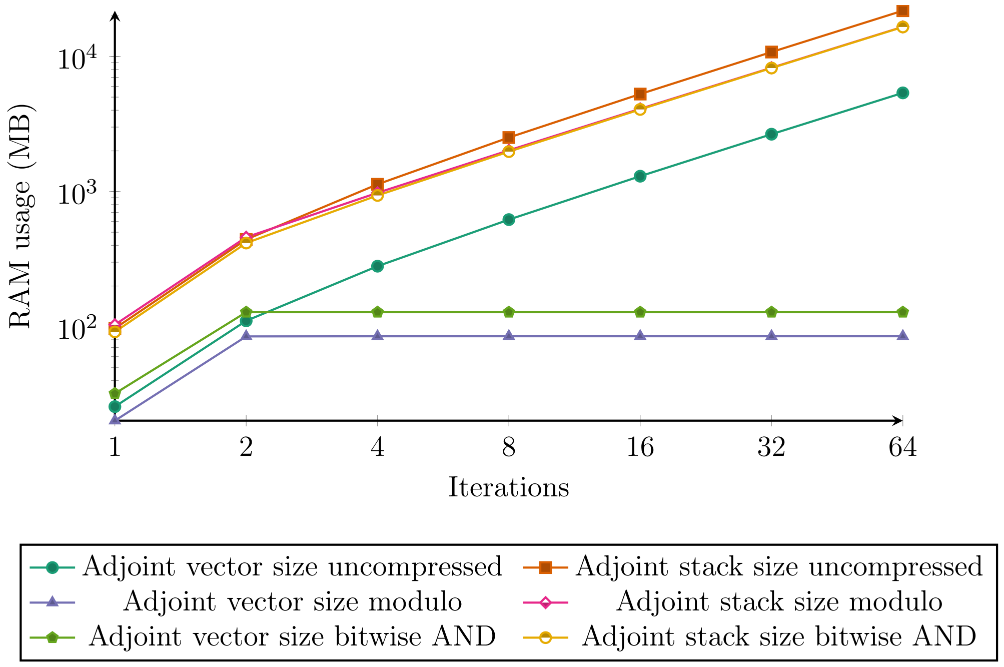


Figure 3.35: RAM usage with SDLS for uncompressed and compressed adjoint vector using the modulo and bitwise AND operators.

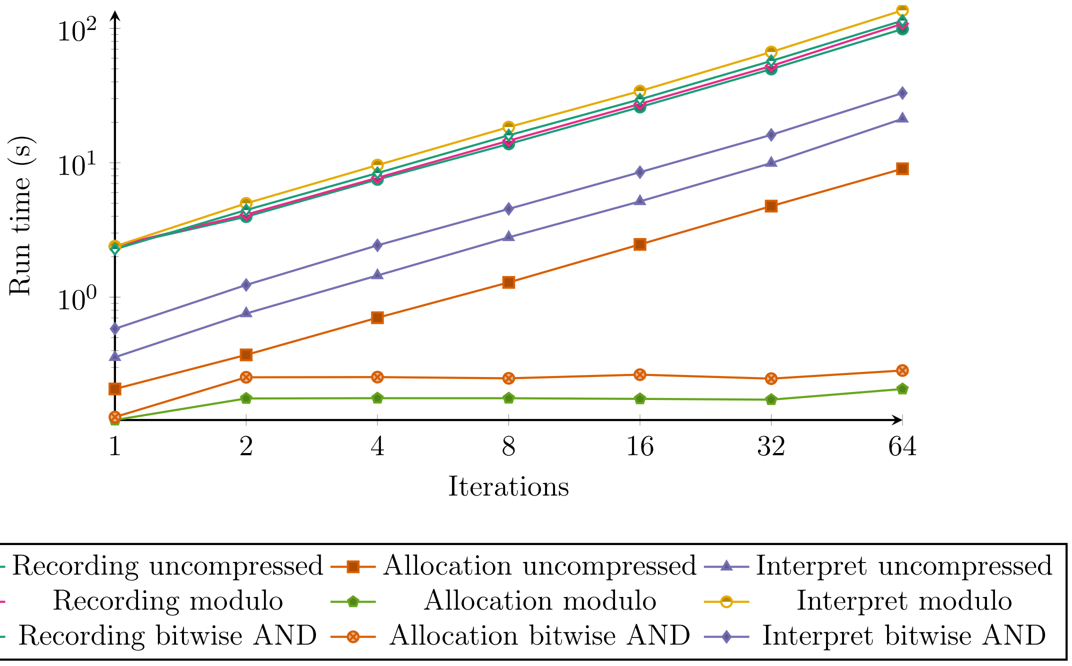


Figure 3.36: Run time without SDLS for uncompressed and compressed adjoint vector using the modulo and bitwise AND operators.

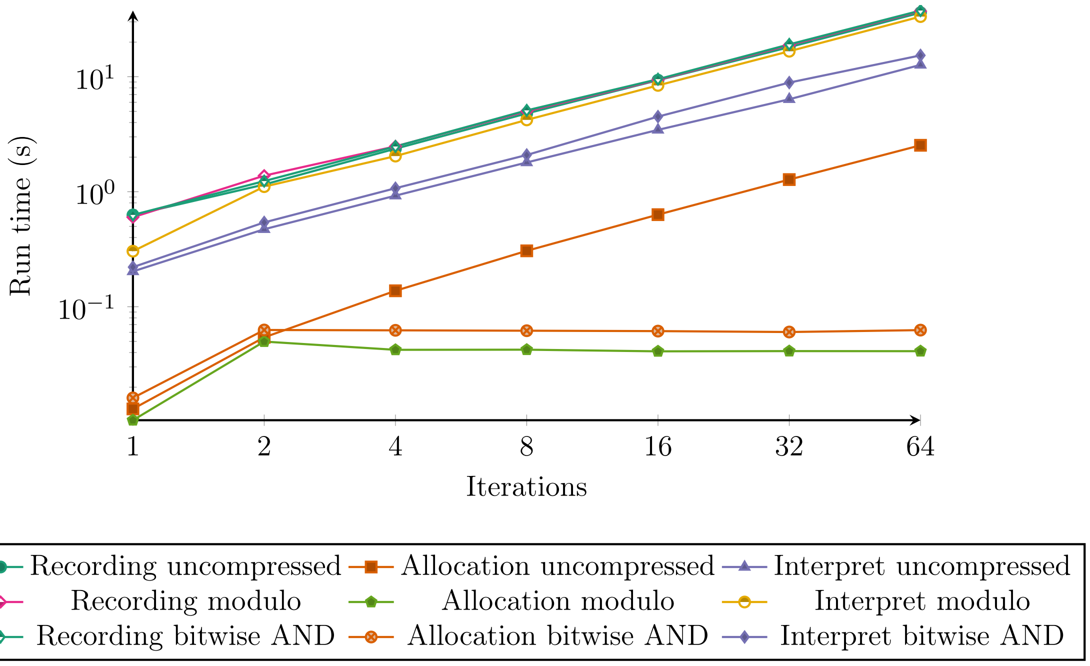


Figure 3.37: Run time with SDLS for uncompressed and compressed adjoint vector using the modulo and bitwise AND operators.

4 Implementing Efficient Algorithms for Steady Flows

After introducing AD to the calculation of full iteration histories, we will now discuss approaches that will improve efficiency of the computations for steady state flow cases.

4.1 Reverse Accumulation

4.1.1 Introduction

Reverse Accumulation allows to iteratively accumulate the adjoints of a problem which contractively converges to a steady solution (fix point) by adjoining the last iteration step repeatedly. The algorithm is extensively documented in the literature [Chr94; Chr92; Gil92], here we focus on the application of the algorithm to typical CFD optimization cases. To this end, we assume the parameters to be the topology optimization parameters $\boldsymbol{\alpha} \in \mathbb{R}^{n_C}$.

For the purpose of describing the reverse accumulation algorithm, we again separate our problem into a pre-processing step $\mathbf{x}^0 = P(\boldsymbol{\alpha})$, (iterative) processing step

$$\mathbf{x}^k = \mathcal{F}(\mathbf{x}^0, \boldsymbol{\alpha}) = f^n\left(\mathbf{x}^{k-1}, \boldsymbol{\alpha}\right) \circ f^{k-1}\left(\mathbf{x}^{k-2}, \boldsymbol{\alpha}\right) \circ \ldots \circ f^1\left(\mathcal{P}(\boldsymbol{\alpha}), \boldsymbol{\alpha}\right) ,$$

and post-processing step $y = \mathcal{J}(\mathbf{x}^k, \boldsymbol{\alpha}) = \mathcal{J}(\mathcal{F}(\mathcal{P}(\boldsymbol{\alpha})), \boldsymbol{\alpha})$. The independent (state) variables $\mathbf{x}$ are initialized in the pre-processing step, potentially depending on the parameters $\boldsymbol{\alpha}$. For laminar steady flow, the independents are the velocity, pressure, and face flux vectors $\mathbf{x} = (\mathbf{U}, \mathbf{p}, \boldsymbol{\phi})$. From the output of the pre-processing step, the state is iteratively converged to the fixed point $\mathbf{x}^*$, bringing the residual of the Navier-Stokes equations towards zero. The iteration function f can, potentially for every iteration step, switch between different execution branches, e.g. due to upwinding in the discretization schemes, therefore a different function f^i is assumed for every iteration step. After the converged state is reached, the cost function $\mathcal{J}$ is calculated from the last state $\mathbf{x}^*$, and potentially also from $\boldsymbol{\alpha}$.

As already stated previously in Section 3.2.2, applying AD to the whole iteration process of k iterations w.r.t. the parameters $\boldsymbol{\alpha}$ yields

$$\frac{\mathrm{d}f^k}{\mathrm{d}\boldsymbol{\alpha}} = \frac{\partial f^k}{\partial \mathbf{x}^{k-1}} \frac{\mathrm{d}\mathbf{x}^{k-1}}{\mathrm{d}\boldsymbol{\alpha}} + \frac{\partial f^k}{\partial \boldsymbol{\alpha}} , \tag{4.1}$$

with the recursion formula

$$\frac{\mathrm{d}\mathbf{x}^k}{\mathrm{d}\boldsymbol{\alpha}} = \begin{cases} \frac{\partial f^k}{\partial \mathbf{x}^{k-1}} \frac{\mathrm{d}\mathbf{x}^{k-1}}{\mathrm{d}\boldsymbol{\alpha}} + \frac{\partial f^k}{\partial \boldsymbol{\alpha}} & k > 1 \\ \frac{\partial f^1}{\partial x^0} \frac{\partial x^0}{\partial \boldsymbol{\alpha}} & k = 1 \, . \end{cases}$$

Reverse Accumulation uses the knowledge, that for a problem which has reached a converged state $\mathbf{x}^k = \mathbf{x}^*$ the Jacobian

$$\boldsymbol{\nabla} f(\mathbf{x}^k) = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}}$$

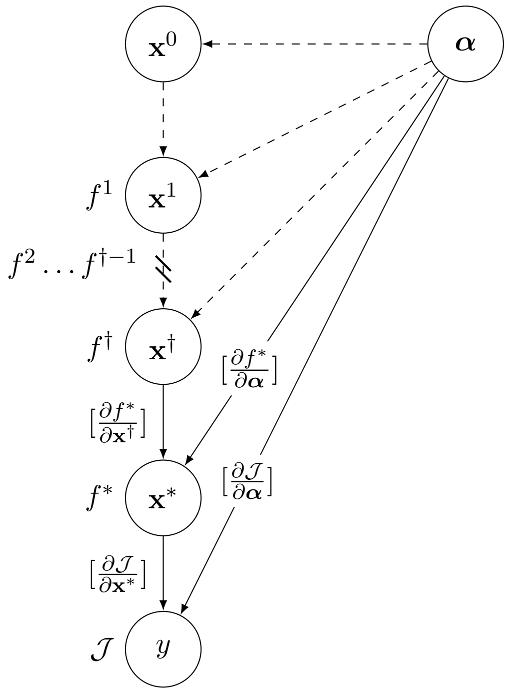

$\mathbf{x}^0$
$\boldsymbol{\alpha}$
f^1
$\mathbf{x}^1$
$f^2 \dots f^{\dagger-1}$
$f^\dagger$
$\mathbf{x}^\dagger$
$[\frac{\partial f^*}{\partial \boldsymbol{\alpha}}]$
$[\frac{\partial f^*}{\partial \mathbf{x}^\dagger}]$
f^*
$\mathbf{x}^*$
$[\frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}]$
$[\frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}]$
$\mathcal{J}$
y

Figure 4.1: DAG of the iterations used for reverse accumulation. $\mathbf{x}^\dagger$ denotes the penultimate state from which on taping is enabled. The final state $\mathbf{x}^*$ is calculated from $\mathbf{x}^\dagger$ by the fixed iteration step f^*. Compare to the black-box differentiation in Figure 3.5.

of the state $\mathbf{x}^k = f(\mathbf{x}^{k-1})$ differentiated w.r.t. to the previous state $\mathbf{x}^{k-1}$ has also converged to a fixed state $\boldsymbol{\nabla} f^*$. Furthermore, the location of the fix point $\mathbf{x}^*$ is independent of the starting point $\mathbf{x}^0$, and therefore

$$\left\| \frac{\mathrm{d}\mathbf{x}^*}{\mathrm{d}\mathbf{x}^0} \right\| \approx 0\,,$$

and consequently

$$\left\| \frac{\mathrm{d}y}{\mathrm{d}\mathbf{x}^0} \right\| \approx 0\,.$$

Let $\mathbf{x}^*$ be the last iterate of a fixed point iteration, $\mathbf{x}\dagger$ the penultimate state, and f^* the last iteration producing the state $\mathbf{x}^*$ from $\mathbf{x}^\dagger$. Instead of evaluating the full chain (4.1) the alternative chain

$$\frac{\mathrm{d}f^*}{\mathrm{d}\boldsymbol{\alpha}} \approx \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^k \cdot \frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^{k-1} \cdot \frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \dots + \frac{\partial f^*}{\partial \mathbf{x}^\dagger} \cdot \frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \frac{\partial f^*}{\partial \boldsymbol{\alpha}} \tag{4.2}$$

can be calculated. This is the earlier recursion formula explicitly unrolled for fixed Jacobians $\partial f^* / \partial \mathbf{x}^\dagger$ and $\partial f^* / \partial \boldsymbol{\alpha}$ instead of $\partial f^i / \partial \mathbf{x}^{i-1}$ and $\partial f^i / \partial \boldsymbol{\alpha}$. The chain can be evaluated for an arbitrary number of iterations k. Convergence predictions are given in [Chr94]. In our application we choose k high enough to ensure convergence of the adjoint fixed point iteration.

The main advantage of the reverse accumulation approach is that only one iteration needs to be captured in the tape. Highlighting the passive and active sections of the procedure, the generation of the output y from $\mathbf{x}^0$ is illustrated in Figure 4.1. Passive computations are connected by dashed lines, computations which need to be captured inside the tape are drawn solid.

The alternative chain (4.2) can be conveniently evaluated by reverse mode AD. Interpreting the tape for the iteration step $f^* : \mathbb{R}^{n_\mathbf{x} \times n_C} \to \mathbb{R}^{n_\mathbf{x}}$, which generates $\mathbf{x}^*$ from $\mathbf{x}^\dagger$, yields incremental projections of the form

$$\begin{aligned} \bar{\mathbf{x}} &= \bar{\mathbf{x}} + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^T \cdot \mathbf{s} \\ \bar{\boldsymbol{\alpha}} &= \bar{\boldsymbol{\alpha}} + \left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right)^T \cdot \mathbf{s}\,, \end{aligned}$$

where $\mathbf{s} \in \mathbb{R}^{n_\mathbf{x}}$ is an arbitrary adjoint seed vector.

By choosing the first seed vector as $\mathbf{s}^0 = (\partial \mathcal{J}/\partial \mathbf{x}^*)^T$, we construct the following iteration which evaluates (4.2) by repeatedly using the result $\bar{\mathbf{x}}$ of the previous iteration as the seed vector for the next iteration:

$$\begin{aligned} \mathbf{s}^1 = \bar{\mathbf{x}}^1 &= \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^T \cdot \mathbf{s}^0 = \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right) \\ \mathbf{s}^2 = \bar{\mathbf{x}}^2 &= \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^T \cdot \mathbf{s}^1 = \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^2 \\ &\dots \\ \mathbf{s}^k = \bar{\mathbf{x}}^k &= \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^T \cdot \mathbf{s}^{k-1} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^k . \end{aligned} \tag{4.3}$$

While the adjoint $\bar{\mathbf{x}}$ is reset to zero after each iteration, the adjoint $\bar{\boldsymbol{\alpha}}$ is allowed to accumulate and yields the desired adjoint approximation for $(\partial \mathcal{J}/\partial \mathbf{x}^*)(\mathrm{d} f^*/\mathrm{d}\boldsymbol{\alpha})$:

$$\begin{aligned} \bar{\boldsymbol{\alpha}}^1 &= \frac{\partial \mathcal{J}}{\partial \bar{\boldsymbol{\alpha}}} + \left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right)^T \mathrm{s}_0 = \frac{\partial \mathcal{J}}{\partial \bar{\boldsymbol{\alpha}}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\frac{\partial f^*}{\partial \boldsymbol{\alpha}} \\ \bar{\boldsymbol{\alpha}}^2 &= \bar{\boldsymbol{\alpha}}^1 + \left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right)^T \mathrm{s}^1 = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \frac{\partial f^*}{\partial \mathbf{x}^\dagger}\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right) \\ \bar{\boldsymbol{\alpha}}^3 &= \bar{\boldsymbol{\alpha}}^2 + \left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right)^T \mathrm{s}^2 = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \frac{\partial f^*}{\partial \mathbf{x}^\dagger}\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^2\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right) \\ &\dots \\ \bar{\boldsymbol{\alpha}}^k &= \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \frac{\partial f^*}{\partial \mathbf{x}^\dagger}\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^2\frac{\partial f^*}{\partial \boldsymbol{\alpha}} + \ldots + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^{k-1}\frac{\partial f^*}{\partial \boldsymbol{\alpha}}\right) \\ &= \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \mathbf{x}^*}\left(1 + \frac{\partial f^*}{\partial \mathbf{x}^\dagger} + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^2 + \ldots + \left(\frac{\partial f^*}{\partial \mathbf{x}^\dagger}\right)^{k-1}\right)\frac{\partial f^*}{\partial \boldsymbol{\alpha}} . \end{aligned} \tag{4.4}$$

With f^* being a contractive function, the norm $\left\|\left(\partial f^*/\partial \mathbf{x}^\dagger\right)^k\right\|$ will tend to zero for $k \to \infty$, making $\bar{\mathbf{x}}^k$ an indicator for the convergence of $\bar{\boldsymbol{\alpha}}$.

4.1.2 Implementation in a CFD Setting

In the following the procedure of accumulating the adjoints is detailed with focus on the usage of an AD tool. The seeding and calculation of the adjoints $\bar{\mathcal{J}}$ of the post-processing step is left unchanged from the standard black-box or checkpointed approaches, yielding the adjoints $\bar{\mathbf{x}}^* = (\partial\mathcal{J}/\partial\mathbf{x}^*)^T \cdot 1$ and $\bar{\boldsymbol{\alpha}} = (\partial\mathcal{J}/\partial\boldsymbol{\alpha})^T \cdot 1$ of the cost function $\mathcal{J}$ w.r.t. the final state $\mathbf{x}^*$ and the parameters $\boldsymbol{\alpha}$.

The tape of the post-processing step can be discarded, once the adjoints $\bar{\mathbf{x}}^*$ have been calculated. The tape of the final iteration step is interpreted from $\mathbf{x}^*$ back to $\mathbf{x}^\dagger$, without discarding the tape. This calculates the adjoints $\bar{\mathbf{x}}^\dagger$, but also begins to accumulate the desired adjoints $\bar{\boldsymbol{\alpha}}$ by incrementing them with the adjoint chain $\bar{\boldsymbol{\alpha}} = \bar{\boldsymbol{\alpha}} + (\partial\mathcal{J}/\partial\mathbf{x}^*)(\partial f^*/\partial\boldsymbol{\alpha})$. Next, the adjoints $\bar{\mathbf{x}}^\dagger$ are extracted from the tape and written to a temporary storage field. This step is slightly complicated by the fact, that usually iterative solvers overwrite the existing state with the newly calculated one, i.e. $\mathbf{x}^{i+1} = f(\mathbf{x}^i, \boldsymbol{\alpha})$ is actually implemented aliased as $\mathbf{x} := f(\mathbf{x}, \boldsymbol{\alpha})$. While the adjoints $\bar{\mathbf{x}}^\dagger$ exist in the tape, they can not be addressed by the variables $\mathbf{x}$ (i.e. `dco::derivative(x)` in `dco/c++` notation), as this would instead yield $\bar{\mathbf{x}}^*$ (memory aliasing). Therefore, the location of $\bar{\mathbf{x}}^k$ in the tape must be saved after first registering the state $\mathbf{x}$ in the tape. This procedure is similar to the extraction of adjoints of the state using checkpointing. For the OpenFOAM implementation, we can therefore reuse parts of the adjoint checkpointing interface for the reverse accumulation iteration.

After extracting $\bar{\mathbf{x}}^\dagger$, the adjoints of all tape entries between $\mathbf{x}^\dagger$ and $\mathbf{x}^*$ in the adjoint vector are reset to zero, leaving only the already partially accumulated adjoints $\bar{\boldsymbol{\alpha}}$ as non-zeroes in the adjoint vector.

Next, the stored adjoints of the state $\bar{\mathbf{x}}^n$ are written back into the adjoint state $\bar{\mathbf{x}}^{n+1}$. The tape is then re-interpreted from $\mathbf{x}^*$ to $\mathbf{x}^\dagger$, yielding a new adjoint state $\bar{\mathbf{x}}^\dagger$ and incrementing the adjoints of the parameters $\bar{\boldsymbol{\alpha}}$.

Summarizing the above procedure the following steps needs to be implemented by a CFD solver utilizing reverse accumulation, enabling to evaluate the adjoint chains (4.3) and (4.4):

- Calculate $n-1$ iterations in *passive mode* up to $\mathbf{x}^*$ (choose n such that solution has sufficiently converged).
- Register all required parameters $\boldsymbol{\alpha}$ in tape.
- Store current position of tape $\rightarrow \tau^\dagger$.
- Register the state variables $\mathbf{x}^\dagger$ in tape.
- Calculate the nth iteration in augmented forward mode, yielding $\mathbf{x}^*$.
- Store the current position of the tape $\rightarrow \tau^*$.
- Evaluate the cost function $\mathcal{J}$ and seed adjoint $\bar{\mathcal{J}} = 1$.
- To evaluate $\frac{\partial\mathcal{J}}{\partial\boldsymbol{\alpha}}$, we proceed as follows:
  - Interpret tape from final position up to τ^*.
  - Get adjoints of final state and store them $\bar{\mathbf{x}}_s = \bar{\mathbf{x}}^*$.
    1. Set adjoints of the state $\bar{\mathbf{x}}^* = \bar{\mathbf{x}}_s$.
    2. Interpret tape from position τ^* to $\tau^\dagger$.
    3. Extract adjoints of initial state and store them $\bar{\mathbf{x}}_s = \bar{\mathbf{x}}^\dagger$.

4. Reset the values of all adjoints from τ^* to $\tau_\dagger$ to zero.

– Repeat steps (1-4) until $\|\bar{\mathbf{x}}_s\| < \epsilon$.

- $\frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}$ is now available in $\bar{\boldsymbol{\alpha}}$.

Utilizing reverse accumulation, the calculation can be sped up considerably, as no checkpointing of an iteration evolution is required. The tape re-evaluation is quite efficient compared to augmented forward steps, further improving runtime. However, it can be difficult to find an iteration step which is a suitable candidate for reverse iteration, if the residuals are noisy. An example for the convergence of reverse accumulation compared to the black-box approach is given in the following section.

The source code of an OpenFOAM solver, implementing reverse accumulation, using the aforementioned checkpointing interface, is given in Appendix C.1.

4.2 Piggyback Adjoint Iteration

The concept of piggyback adjoints in the context of AD was first introduced in [GF03] and further detailed in [GW08]. In this approach, the adjoints are propagated alongside the primal values, somewhat mirroring the behavior of the continuous adjoint, where the adjoint equations are solved alongside the primal equations. In a one-shot optimization setting [Bos+14], the design parameters are immediately optimized using the partially converged derivative information.

The updates of primal, adjoint, and design state are therefore evaluated in one coupled iteration. With the functions $f^i : \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^p \to \mathbb{R}^n$ which calculate one iteration step of the primal, and $g : \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^{n_\mathbf{x}} \times \mathbb{R}^p$ which calculates the adjoints of the states, as well as the gradient required for the update step, the iteration procedure can be outlined as follows:

```
for $k = 0, \ldots, n-1$ :
    $\mathbf{x}^{k+1} = f^k(\mathbf{x}^k, \boldsymbol{\alpha}^k)$
    $\bar{\mathbf{x}}^{k+1} = g^k_{\mathbf{x}}(\mathbf{x}^k, \bar{\mathbf{x}}^k, \boldsymbol{\alpha}^k)$
    $\boldsymbol{\alpha}^{k+1} = \boldsymbol{\alpha}_k - P^{-1} g^k_{\boldsymbol{\alpha}}(\mathbf{x}^k, \bar{\mathbf{x}}^k, \boldsymbol{\alpha}^k)\,,$
```

where P is a suitable preconditioner, to ensure the convergence and stability of the design optimization.

When applying AD, the function g is not given explicitly, but is evaluated by adjoining the iteration f^k, as well as the calculation of the objective $y = \mathcal{J}(\mathbf{x}^{k+1})$ after each iteration step.

The piggyback approach is closely related to reverse accumulation. It differs in that not a fixed (last) iteration step of a fixed point iteration is repeatedly adjoined. Instead always the most recent iteration f^k of the augmented primal is used to obtain the next iterate for the adjoints $\bar{\mathbf{x}}^{k+1}$. New iteration steps are repeatedly calculated and adjoined until the change in both $\mathbf{x}$ and $\bar{\mathbf{x}}$ fall under a prescribed threshold. Only after this threshold is reached, an update of the design parameters $\boldsymbol{\alpha}$ is performed. The change in $\boldsymbol{\alpha}$ will in turn increase the residuals of both $\mathbf{x}$ and $\bar{\mathbf{x}}$, starting another round of inner piggyback iterations.

Even without applying an optimization, one advantage of the piggyback approach is that no fixed point has to be identified in advance. The method lends itself well to one-shot optimization, as the optimization of the parameters $\boldsymbol{\alpha}$ can be started with a not completely converged gradient.

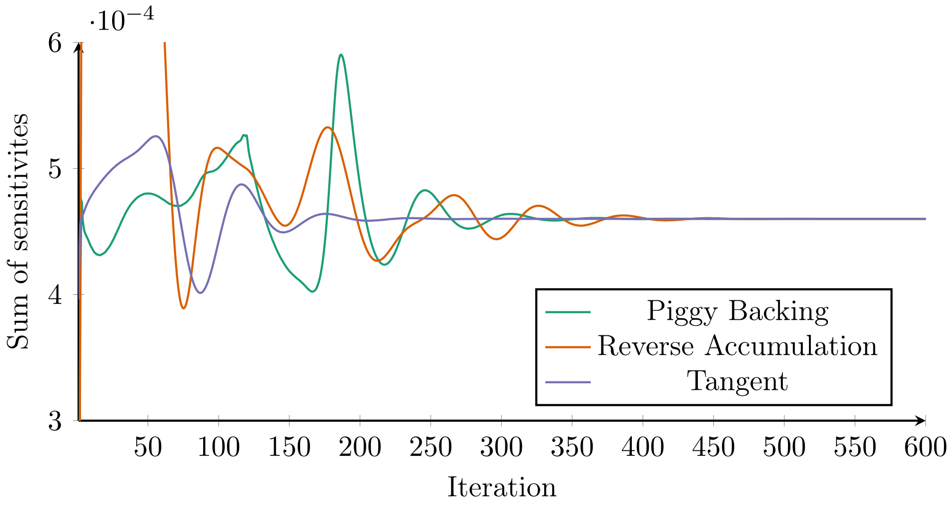


Figure 4.2: Sensitivities obtained by tangent and adjoint modes. The adjoint is piggybacked from the initial state and reverse accumulated by repeatedly adjoining primal iteration step 800.

The piggyback method has been implemented in discrete adjoint OpenFOAM for the SIMPLE algorithm, derived from the `simpleFoam` solver. As with reverse accumulation solvers, the checkpointing interface can be used to help with the handling of the adjoints. Figure 4.2 shows a comparison of the convergence of the sum of sensitivities for black-box tangents (which are identical to the black-box adjoints), reverse accumulation starting after 800 iterations, and piggybacking (without design updates, that is P^{-1} is a zero matrix). The derivatives are calculated on the 2D Pitz-Daily case, with SDLS enabled for piggyback and reverse accumulation solvers. The change in the derivatives from iteration to iteration is detailed in Figure 4.3. Reverse accumulation converges down to machine precision. The convergence of the black-box differentiation and piggyback bottoms out after roughly 600 iterations, however the derivatives have converged sufficiently to not observe any significant changes in the adjoints. The obtained derivative residuals roughly match the chosen linear solver tolerance, a stricter tolerance level will further improve the residuals.

Further applications of the piggyback approach, including optimization using steepest descent algorithm are given in the case study presented in Section 5.1.

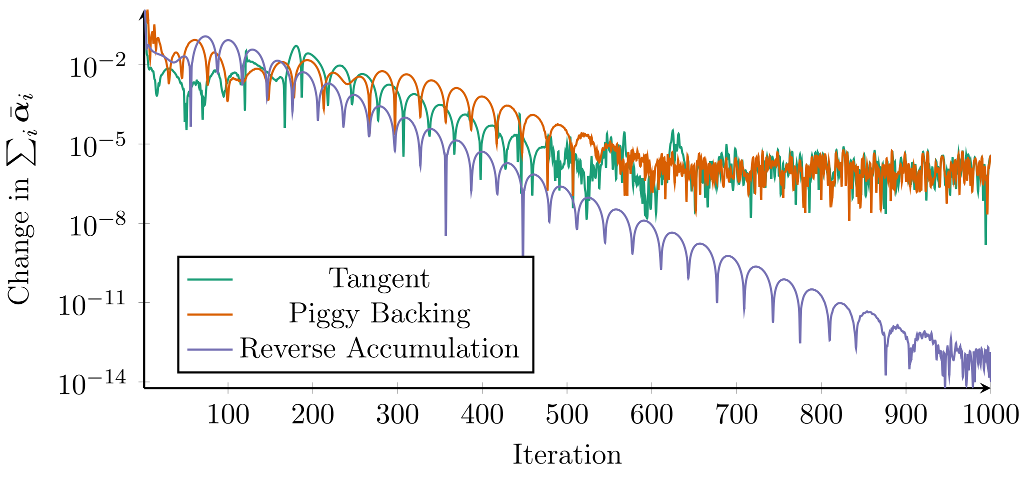


Figure 4.3: Change in the sum of sensitivities between subsequent iterations. Initial residual scaled to one.

4.3 Verification

While the results of the sensitivity calculations obtained so far look plausible, they certainly still need to be verified. We will achieve this using the following steps:

- Check the correct implementation of AD on the algorithms, by comparing results obtained by the adjoint method with results obtained by tangent mode and FD.
- Verify that the differentiated algorithms match the physics of the continuous adjoint, by comparing to results obtained with the continuous adjoint method.
  The `adjointShapeOptimizationFoam` solver, supplied with OpenFOAM, is used for calculating the continuous adjoints.

4.3.1 Mesh Independence of Adjoint Sensitivity

For a steady state problem, the flow fields, obtained by a FVM discretization, should, for increasing mesh resolution, converge towards a final state, as truncation errors decay. As the fields are evaluated at different positions for each discretization, we evaluate the volume integrals of the fields instead to judge convergence.

The angled duct case is evaluated for mesh refinement levels 5 (325 cells) to 80 (83 200 cells). The primal calculation is performed with passive `simpleFoam` iterations, starting from a pre-initialized potential solution. The primal iterations are run until both velocity and pressure fields have reached a prescribed convergence tolerance. Afterwards one augmented forward primal step is executed and then repeatedly adjoined using reverse accumulation. To speed up the calculation, the mesh is decomposed into 8 processor domains. Figure 4.4 shows the number of iterations required to achieve the prescribed solver tolerances. The number of primal evaluations scales roughly linearly with the number of cells in the domain. The number of adjoint reverse accumulations required rises slower, and after a certain point seems to grow logarithmically.

The change in the velocity, pressure, and sensitivity fields, evaluated as volume integrals, is detailed in Figure 4.5. Shown is the change in volume integral, compared to the previous mesh

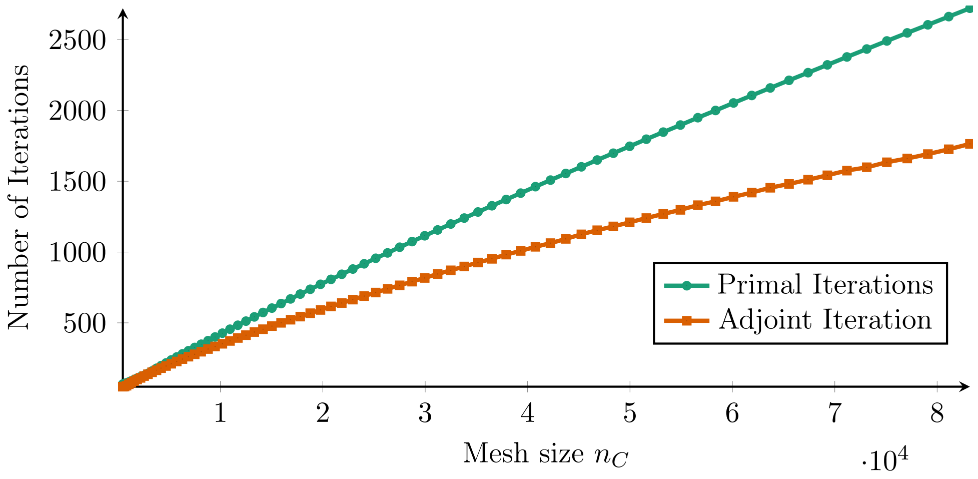


Figure 4.4: Number of primal SIMPLE iterations and number of adjoint reverse accumulation iterations needed to bring case to convergence.

resolution. The curves are normalized and shown in a double logarithmic scale. The change in all fields decreases, as the FVM approximations converge towards the solution of the continuous problem. The pressure convergence lags behind the velocity, which is common with SIMPLE algorithms and is amplified by the lower relaxation factor chosen for the pressure correction equation. The formulation of the cost function is dominated by the influence of the pressure, therefore it is plausible that the convergence rate of the sensitivity seems strongly linked to the convergence rate of the primal pressure.

Summarizing, the adjoint sensitivity field converges towards a fixed point, as the mesh resolution increases. It does so with a convergence speed comparable to the primal iteration. The number of iterations required to achieve adjoint convergence are on par, or lower, than the primal.

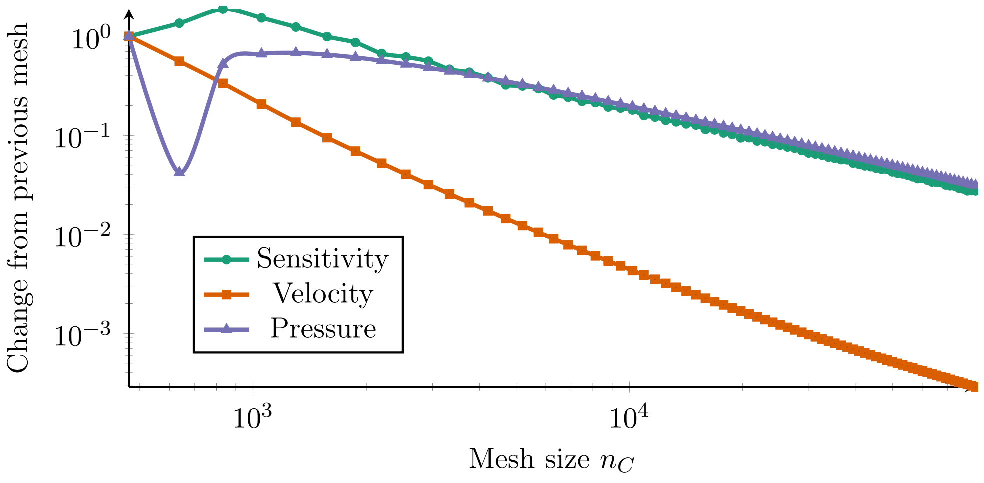


Figure 4.5: Change in velocity, pressure, and adjoint sensitivity field, compared to previous mesh resolution. First value is normalized to one.

4.3.2 Validation Against Tangent and FD Models

To validate the implementation of AD, we first use the laminar angled duct testcase. Units are introduced to the dimensionless geometry by choosing $L = 1\,\mathrm{m}$, with the origin at the lower left corner of the geometry.

In Figure 4.6 we show the values for the derivative $\mathrm{d}\mathcal{J}/\mathrm{d}\alpha$ at different discrete locations along two evaluation lines crossing the domain. The first line L_1 connects the points $(0, 0.55, 0.05)$ and $(5, 0.55, 0.05)$, the second line L_2 connects $(4.55, 0, 0.05)$ to $(4.55, 5, 0.05)$ (all coordinates in meters). The coordinates of the endpoints are chosen, such that the line passes the cell midpoints of the cells it penetrates. The location of the lines L_1 and L_2, as well as the full sensitivity field are shown in Figure 4.7.

Figure 4.6 compares the results obtained by 400 iterations of `adjointSimpleCheckpointingFoam` to the results obtained by a tangent and FD implementation. For the adjoint results, the symbolical differentiation of the linear solvers is disabled, to ensure comparability to the tangents with up to machine precision. The tangent implementation uses the vector mode of `dco/c++`, allowing to evaluate all 16 reference points with one solver run. A tangent vector size of 16 was utilized, using the 16th entry to calculate the sum of all sensitivities.

For FD, a one sided perturbation with $h = 10^{-3}$ was used. The tangent and FD evaluation points are offset by some margin, so that they do not overlap in the plot. The adjoint sensitivities are plotted without interpolating between the cells (to allow us to precisely match the tangent and FD data points), giving a piecewise constant sensitivity evolution along the lines L_1 and L_2.

The lines intersect the inflow and outflow boundary. At the inflow and outflow boundary, we observe a discontinuous behavior in the adjoint field. This is matched by the tangents and FD. It is thus not an error in the differentiation, but rather an issue of the primal implementation. The artifacts presumably arise from the implementation of the boundary conditions, which is continuous for the primal but not necessarily for the adjoint. The artifacts are limited to the cells adjacent to the inflow, outflow, and their neighboring cells. Further up- and downstream no obvious influence of those discontinuities can be observed. The wall boundary conditions do not exhibit such behavior.

In Table 4.1 we summarize the adjoints for five exemplary points. The tangents match the adjoints up to some orders of magnitude of the machine precision. Slight deviations are induced due to the different order of application of arithmetic operations between adjoint and tangent mode, and the use of the `-ffast-math` compiler flag which can lead to additional truncation errors[1].

Table 4.1: Adjoints, tangents, and FD for five exemplary cells. Differences between adjoints and tangents are marked bold.

Cell index	Adjoint	Tangent	FD	Error of FD
970	13351.8234523**369**	13351.8234523**345**	13351.6289	0.0015 %
1013	14929.8557699**385**	14929.8557699**349**	14929.9356	0.0005 %
2785	8605.97203473**893**	8605.97203473**516**	8605.8119	0.0019 %
2817	603.880863106**184**	603.880863106**741**	603.6694	0.0350 %
4099	-81.383599564**168**	-81.383599564**306**	-81.4823	0.1213 %

[1]https://gcc.gnu.org/wiki/FloatingPointMath

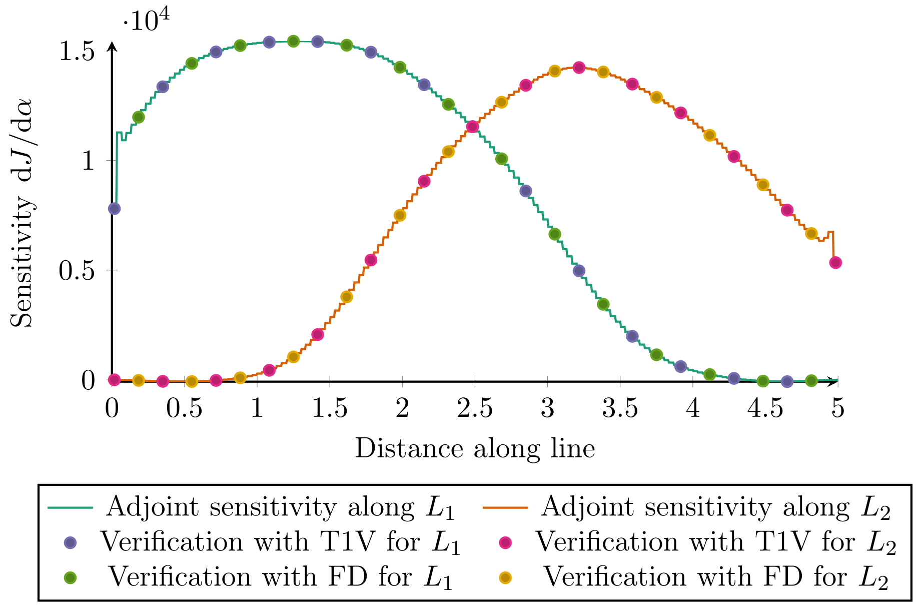


Figure 4.6: Sensitivities on two lines through the domain, computed with adjoint mode without interpolation between cells. The results are verified with 15 points computed by T1V and FD mode respectively.

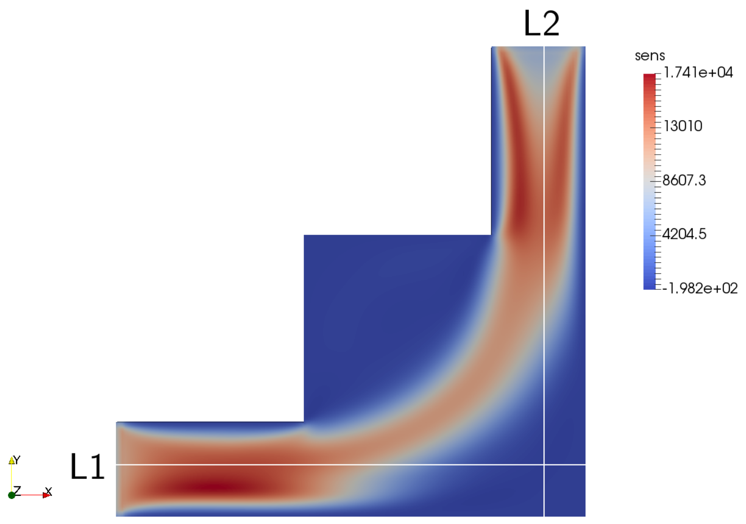


Figure 4.7: Sensitivity field of angled duct case. Evaluation lines L_1 and L_2 are shown in white.

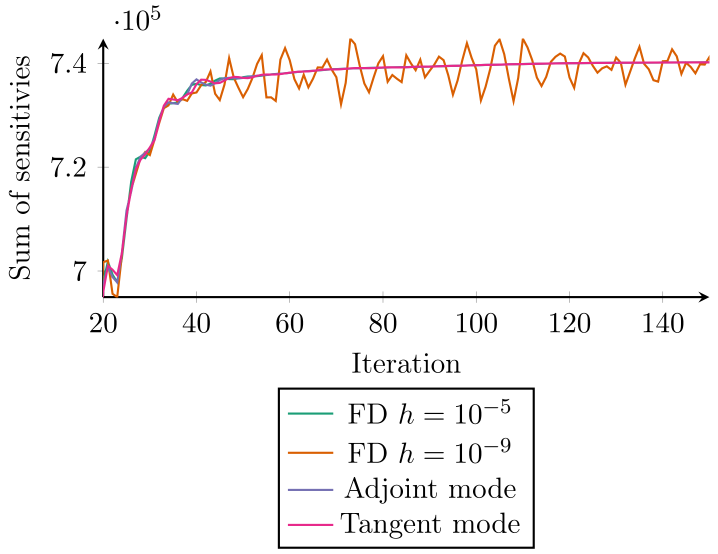


Figure 4.8: Sum of sensitivities computed with FD, tangent mode, and adjoint mode. FD with $h = 10^{-9}$ unreliable, all other curves match.

Figure 4.8 shows the evolution of the sum of sensitivities

$$S^j = \sum_i \frac{\mathrm{d}\mathcal{J}(\mathbf{x}^j)}{\mathrm{d}\alpha_i}$$

over 150 iterations. This sum can be conveniently calculated by FD and tangent mode with a run-time factor of $\mathcal{O}(1) \cdot \mathit{cost}\,(\mathcal{J}(\boldsymbol{\alpha}))$, by seeding all inputs at the same time:

$$\sum_i \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\alpha_i} = \sum_i \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}} \mathbf{e}_i = \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}} \sum_i \mathbf{e}_i = \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}} \cdot [1, \ldots, 1]^T \ .$$

In adjoint mode, all sensitivities of a specific iteration step, and thus also the sum, are calculated in $\mathcal{O}(1) \cdot \mathit{cost}\,(\mathcal{J}(\boldsymbol{\alpha}))$ as well. However, to obtain the sum after every iteration step, the tape has to be completely evaluated from the current position back to the inputs after each iteration step. This raises the complexity to the number of iterations. The sensitivities after each iteration are only needed for this verification task, in practice one would only evaluate the tape once, after the primal iteration has finished.

The tangent and adjoint sums match up to machine precision for each iteration step. FD for $h = 10^{-5}$ also matches very closely. FD for $h = 10^{-9}$ exhibits significant noise, however the sensitivities obtained (or at least the sum) remain stable and oscillate around the correct value.

4.3.3 Validation Against Continuous Adjoint Solver

Next, we validate against the continuous adjoint solver included in OpenFOAM. This solver is based on the principles described in [Oth08], the implementation is further documented in [OVW07]. It implements topology optimization for ducted flows and optimizes for power loss using a steepest descent approach with fixed step width. For an introduction to the primal

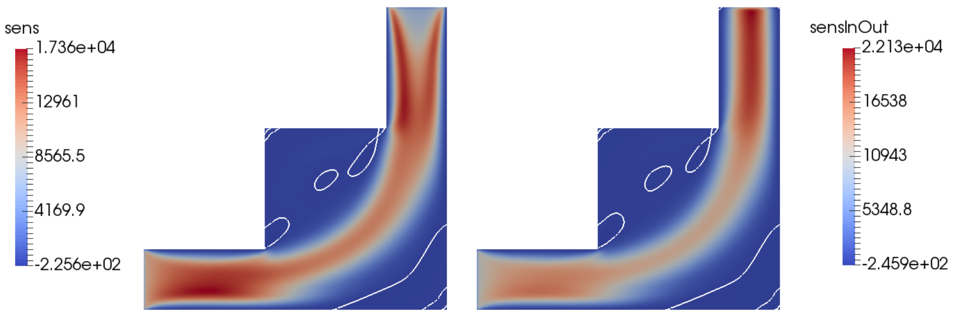


Figure 4.9: Sensitivity fields obtained with discrete piggyback solver. Power loss integrated over the inlet on the left, power loss integrated over inlet and outlet on the right. Contours of zero sensitivity in white. While the field differs at the outflow, regions of negative sensitivity are almost identical.

and continuous adjoint equations, refer to Section 2.5. The cost function is hard coded into the boundary conditions `adjointOutletVelocity` and `adjointOutletPressure`. The primal and adjoint equations are iterated concurrently, the resulting sensitivities are immediately used to update the design field $\boldsymbol{\alpha}$ (this is the continuous equivalent to the discrete piggyback approach). The solver calculates the adjoint velocity and pressure $\bar{\mathbf{U}}$ and $\bar{\mathbf{p}}$, the sensitivities of the individual design parameters α_i are then calculated as the scalar product of primal and adjoint velocity field:

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha_i} = A_i(\mathbf{U}_i \cdot \bar{\mathbf{U}}_i)\,.$$

For the first part of the validation study, we set the step width of the steepest descent optimizer to zero. Thus, the sensitivity field for a fixed (zero) field $\boldsymbol{\alpha}$ is calculated.

The discrete solver is set up to match the cost function and boundary conditions of the stock adjoint solver. This requires to constrain the computation of the power loss to the inlet. If the outlet is included, the sensitivity field changes substantially, while still indicating the same regions of optimization. This effect is shown in Figure 4.9.

The sensitivities are again evaluated along the lines L_1 and L_2. Figure 4.10 shows the sensitivities for the continuous and discrete solvers after convergence (2000 iterations). One can see, that the sensitivities line up very well. The irregularities introduced by the discrete solver at the inflow and outflow boundaries do not introduce any effects in the inner flow domain, which would lead to obvious differences to the continuous solution.

Figure 4.11 shows a condensed y-axis of the previous plot, focusing on the most important region of negative sensitivity. While we see some differences in magnitude between the continuous and discrete adjoint solution here, the sign of the sensitivities is consistent between the adjoint and discrete calculations. A topology optimization approach would thus penalize the same cells, albeit with a slightly different magnitude. For a graphical representation of these regions, we show the sensitivity field for the continuous and discrete solver in Figure 4.12. White iso lines indicate the zero crossings of the sensitivity, bounding regions of negative sensitivity. The results are qualitatively identical to the results obtained with the discrete solver, albeit showing a slightly

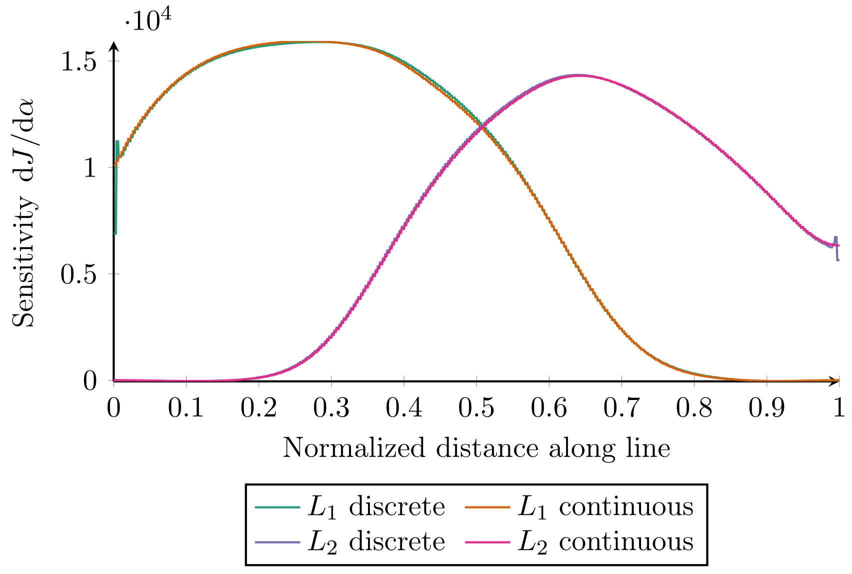


Figure 4.10: Sensitivities along L_1 and L_2 obtained using the continuous and discrete adjoint approach.

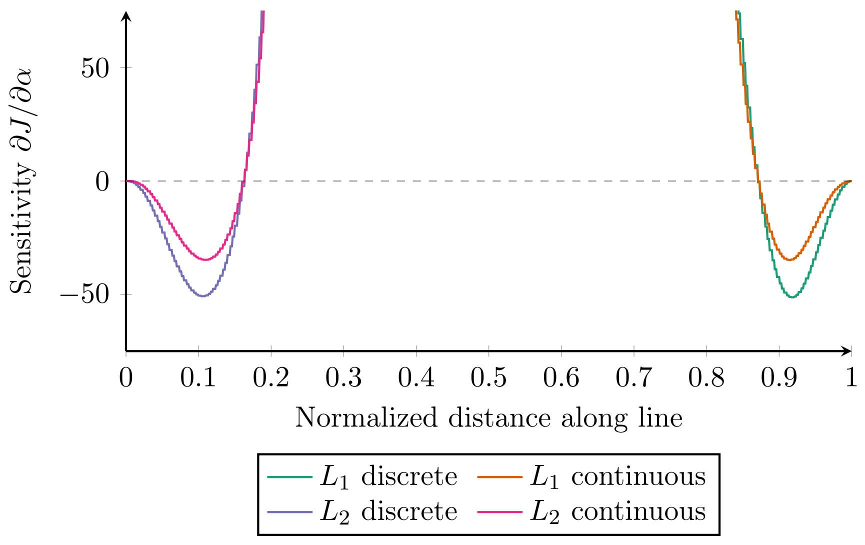


Figure 4.11: Magnification of the negative section of Figure 4.10.

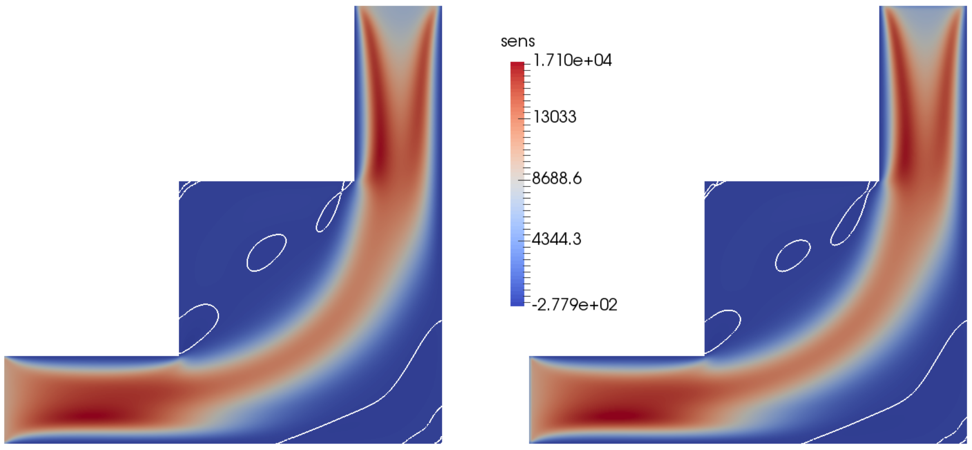


Figure 4.12: Sensitivity field obtained with the continuous (left) and discrete approach (right). Iso-lines of zero sensitivity in white mark the regions of interest for topology optimization.

larger zone of negative sensitivity near the upper right corner of the flow domain.

In Figures 4.13 and 4.14 we compare the convergence speed of the continuous primal and adjoint equations to the convergence of the piggyback approach. As stated earlier, the piggyback approach iterates the adjoint along the primals in a similar fashion to the implicitly coupled continuous adjoint, making this comparison an obvious choice. The former figure shows the convergence of the sum of sensitivities for a mesh of 11 700 cells, the latter for a finer mesh of 187 200 cells. While the continuous and discrete solver converge roughly along the same trajectory, the discrete solver does so with significantly less oscillations, thus arriving earlier at a point where the adjoints can be considered reliable. This potentially leads to a better approximation of the gradient in a one shot optimization setting. Lowering of the under-relaxation factors of the primal and adjoint continuous equations might lead to a reduction of the oscillations, at the expense of slower convergence.

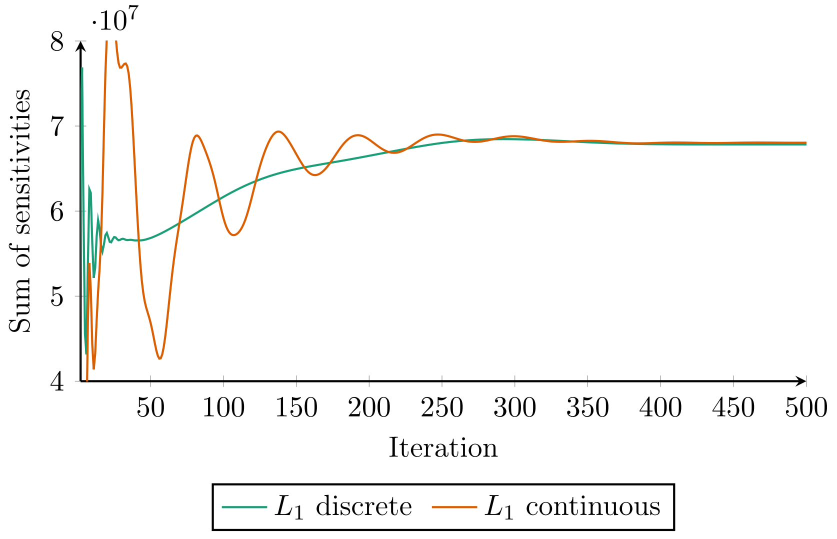


Figure 4.13: Convergence of the sums of sensitivities of piggybacking and continuous adjoint for medium resolution case.

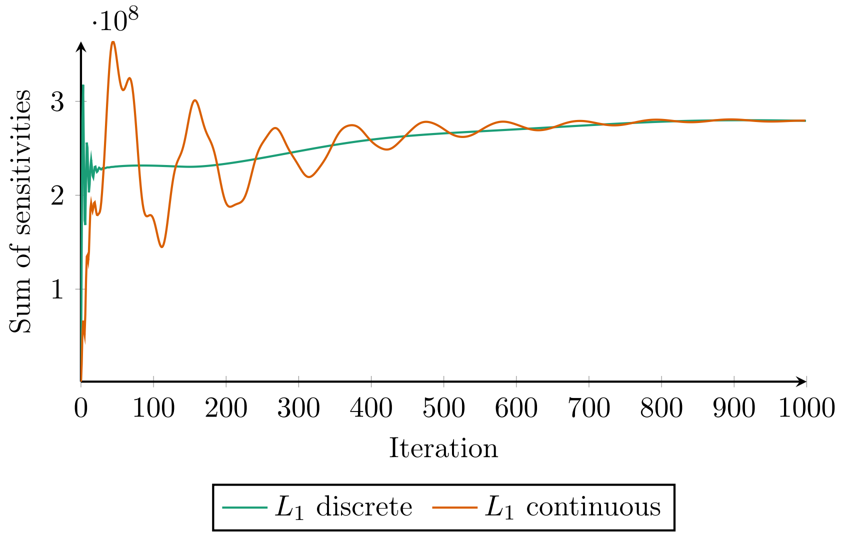


Figure 4.14: Convergence of the sums of sensitivities of piggybacking and continuous adjoint for fine resolution case.

Next, we consider the Pitz-Daily case. For this example, we can not expect a perfect match with the continuous results, as the continuous solver available in the public domain uses a frozen turbulence assumption, which assumes that the influence of the turbulence onto the sensitivities is negligible and can therefore be omitted. The derivation and implementation of continuous equations for turbulence models is complex and sometimes not even possible in a closed form [Car+10; Nem+11]. The frozen turbulence assumption can be approximately replicated with the discrete method, by pausing the tape recording during the calculation of turbulence. The turbulent quantities are thus treated as if they were passive.

The primal flow and the turbulent kinetic energy within the Pitz-Daily geometry is shown in Figure 4.15. The sensitivity results of the simulation after 8000 steps (from a zero initialized solution) are shown in Figure 4.16.

The sensitivities computed with the continuous adjoint solver are shown on top. Below that are the results obtained with the discrete method. The second image is computed by the piggyback approach with differentiated k-ϵ turbulence model. The third image is computed with the same solver, but with the taping of the turbulence model switched off. In this configuration the discrete model thus replicates the frozen turbulence assumption of the continuous solver. The three images look essentially alike, especially the region of negative sensitivity indicated by the white contour lines is very similar. The biggest conceivable difference is located at the location of the step. At this position a solution singularity in the sensitivity can be observed. The maximum sensitivity, obtained at the singularity point by the continuous solver, is roughly 30% higher than the discrete sensitivity obtained with both frozen and fully differentiated turbulence. Further downstream the sensitivities match very well.

For further insight, we again plot the sensitivities over lines through the computational domain. We observe five vertical lines, cutting through the domain at different distances from the inlet. The first line L_1 is located 0.001 m right of the step, capturing the influence of the singularity point. The second to fifth line (L_2 to L_5) are located at $x = \{0.05\,\mathrm{m}, 0.1\,\mathrm{m}, 0.15\,\mathrm{m}, 0.2\,\mathrm{m}\}$ behind the step respectively. For the first line, both discrete solutions are very similar, indicating that the effect of the turbulence on the sensitivities is low. However, the discrete adjoints do not match the continuous solution too well. Nevertheless, the sign of the sensitivities is consistent between the continuous and discrete results. Away from the singularity (which is located at 0.5 along the normalized line distance) the results match better. For lines $L_2 - L_5$, downstream of the singularity, the frozen turbulence and fully differentiated discrete results start to significantly differ. From Figure 4.15 we can see that this coincides with the regions of high turbulent kinetic energy k. The result obtained by the discrete frozen turbulence assumption is in turn now very similar to the continuous result, showing a maximum difference of 7% and an average difference of below 2% along the line. Furthermore, the (interpolated) lines of the discrete solution match all kinks and features of the continuous lines.

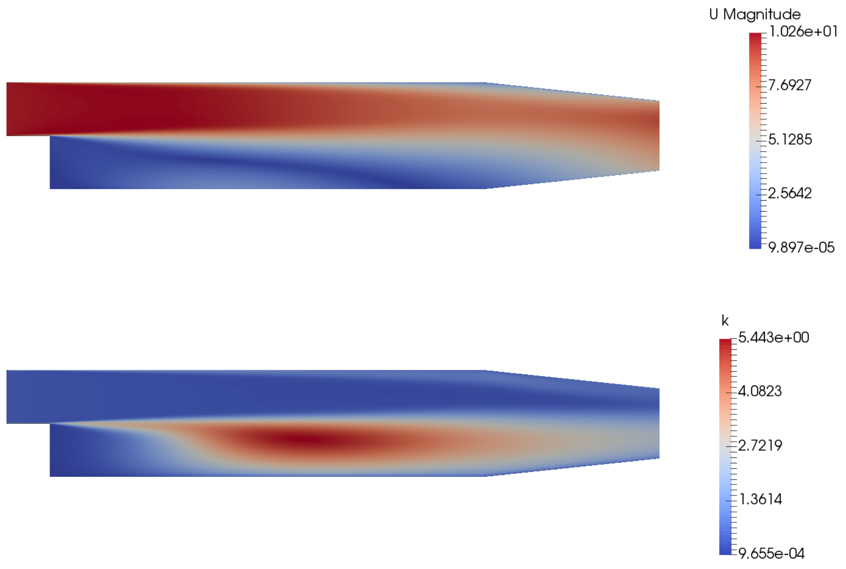


Figure 4.15: Velocity magnitude (absolute length of the velocity vector, top) and turbulent kinetic energy (bottom) of the Pitz-Daily case.

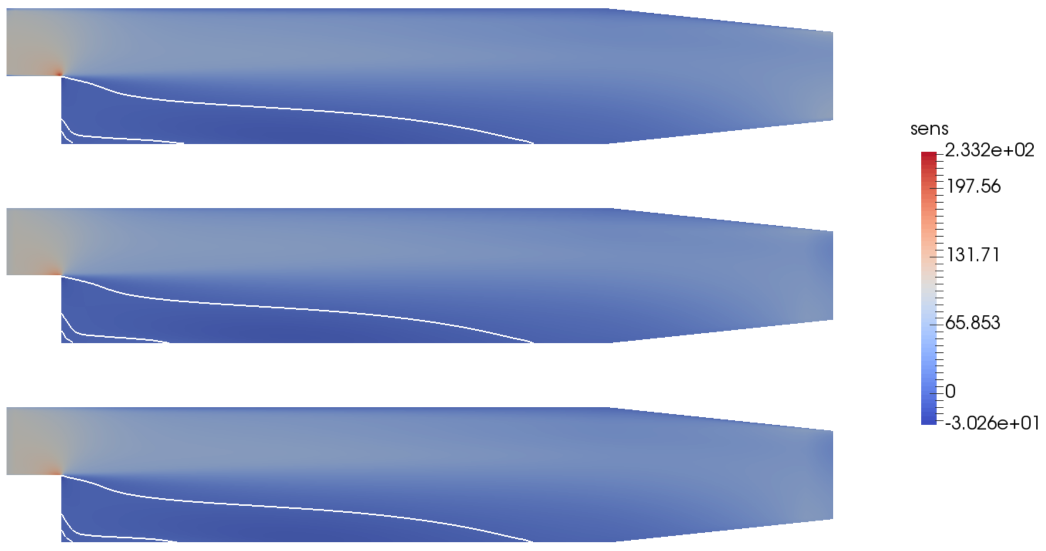


Figure 4.16: Sensitivity fields obtained with the continuous (top), discrete (middle), and discrete approach with frozen turbulence (bottom). Iso-lines of zero sensitivity in white mark the regions of interest for potential topology optimization.

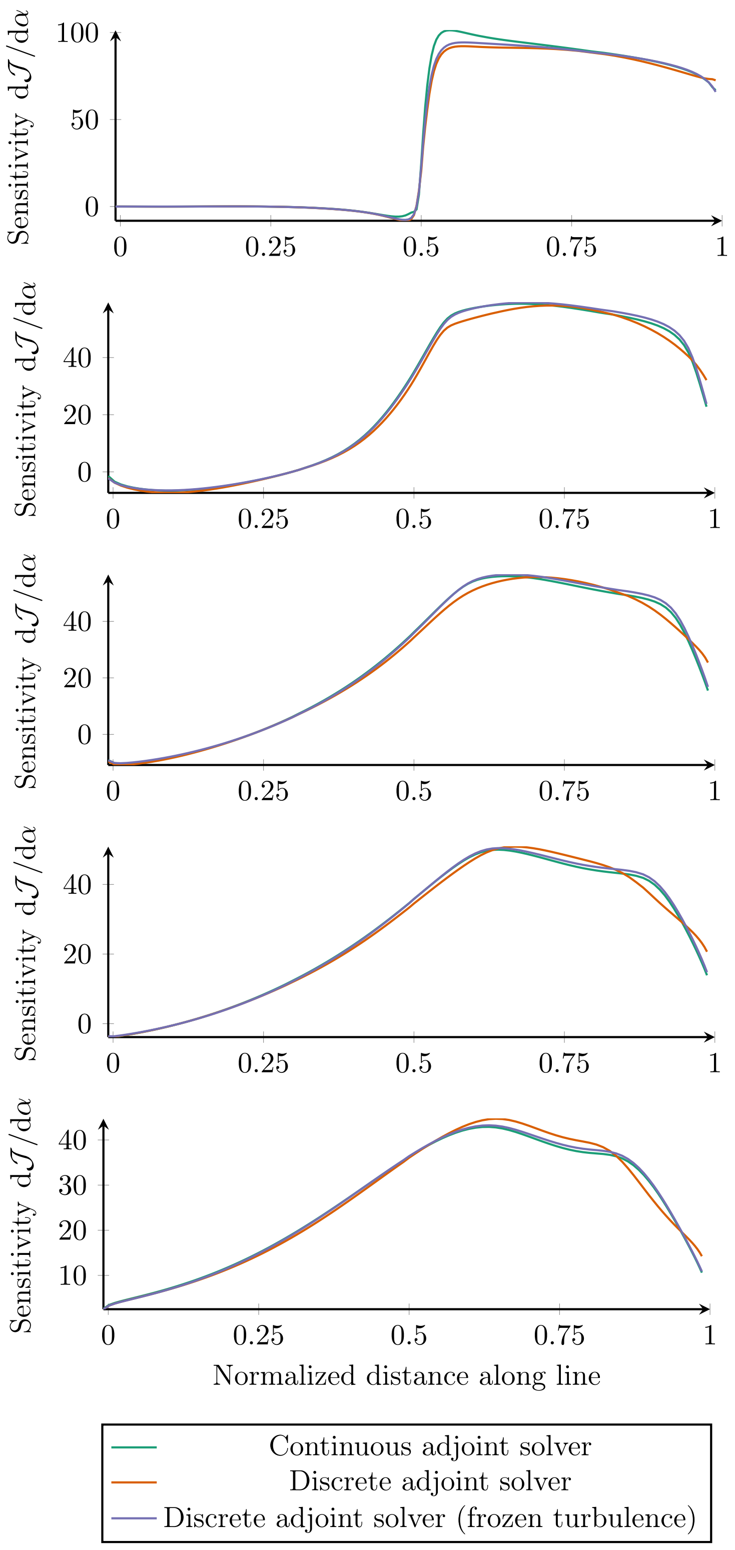


Figure 4.17: Sensitivities along (from top to bottom) lines L_1 to L_5. The continuous adjoints match the discrete adjoints obtained with frozen turbulence.

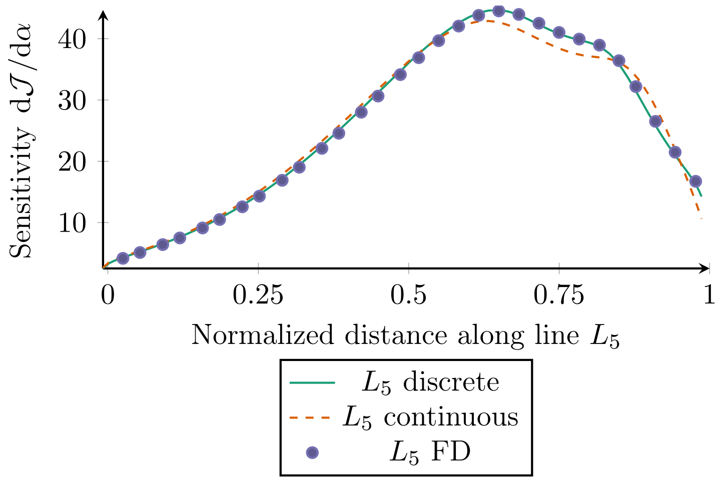


Figure 4.18: Sensitivities along line L_5, obtained by the discrete adjoint model and matched by FD. The continuous model with frozen turbulence (drawn dashed) clearly differs.

As we could not check the correctness of the fully differentiated turbulence models with the continuous adjoint model, we again compare those values against FD and the tangent model. In Figure 4.18 we exemplary verify the fully differentiated turbulence model with FD for 13 points on L_5. The values obtained by FD match the fully discrete adjoint and clearly differ from the frozen turbulence model.

4.3.4 Convergence of FD

As derived in Section 2.7.1, FD approximates the derivative up to a truncation error of order $\mathcal{O}(h)$ for one sided differences and $\mathcal{O}(h^2)$ for two-sided differences. Thus, the difference between FD and the derivatives calculated by AD should shrink according to the relations

$$\begin{aligned}\frac{\mathcal{J}(\boldsymbol{\alpha}+h\mathbf{e}_i)-\mathcal{J}(\boldsymbol{\alpha})}{h}-\frac{\mathrm{d}\mathcal{J}(\boldsymbol{\alpha})}{\mathrm{d}\alpha_i}&=r_{i,1}=\mathcal{O}(h)\\ \frac{\mathcal{J}(\boldsymbol{\alpha}+h\mathbf{e}_i)-\mathcal{J}(\boldsymbol{\alpha}-h\mathbf{e}_i)}{2h}-\frac{\mathrm{d}\mathcal{J}(\boldsymbol{\alpha})}{\mathrm{d}\alpha_i}&=r_{i,2}=\mathcal{O}(h^2)\,,\end{aligned}$$

for some index i. We will now investigate if these relations hold for our implementation.

Figure 4.19 shows the difference between the FD approximation and the derivative calculated by tangent mode for the cell at location $\mathbf{P}=(0.02,-0.02,0.0005)$ of the Pitz-Daily case. Plotted are the absolute values of the difference r_1 and r_2 of FD and tangent, scaled by the tangent, for a range of values h.

The step width of h is chosen in the range $[10^{-5},10^{3}]$. For relatively big values of h, the forward and central finite differences adhere strictly to the reduction factors $\mathcal{O}(h)$ and $\mathcal{O}(h^2)$. For h in the range $[10^{-2},10^{1}]$, the two sided differences, while still providing a better approximation than one sided, behave a bit erratically. We suspect that this is due to the different treatment of the α term for positive and negative signs. While positive values of α contribute to the system matrix and increase the diagonal dominance of the matrix, negative values are put onto the right hand side of the momentum equations. For $h<10^{-5}$, the quality of the approximation begins to

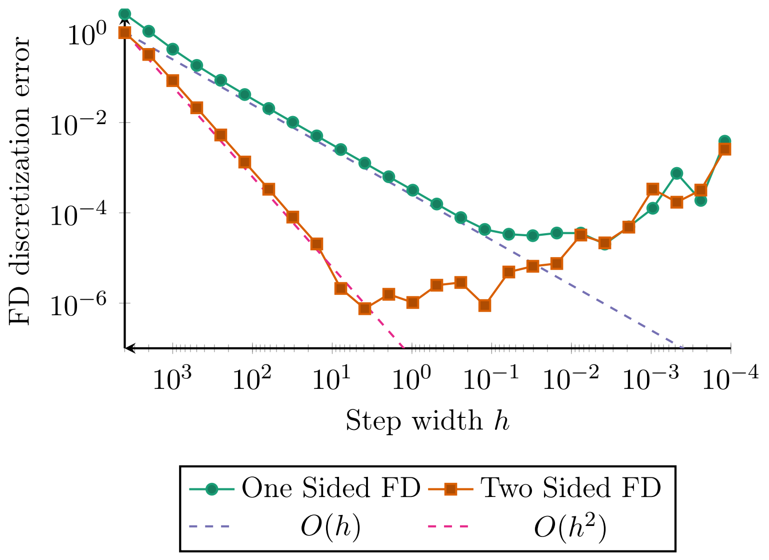


Figure 4.19: Difference between FD and (tangent) AD for decreasing h.

deteriorate. For even lower values, the FD values become noisy and eventually unusable. This highlights the difficulties involved with FD of finding a suitable step size h. For stiff problems, the values of h should be scaled by the central coefficient a_C of the FVM discretization, that is the corresponding diagonal entry of the discretization matrix.

4.4 Shape Adjoints

Shape optimization, in contrast to topology optimization, aims to improve the geometry of the domain directly, by manipulating the position of the nodes on the outer shell. This has the advantage, that the physical wall boundary conditions are retained, allowing the wall stresses to be evaluated more accurately. Other boundary conditions like thermodynamic heat flows can be implemented much easier (or no change is required at all), if a physical wall is present. Usually, the points on the surface are moved in the direction of the normals of the corresponding faces. When the points on the outer surface move, the interior points have to be adapted as well, such that the mesh quality does not degrade, leading to unreliable results or lack of convergence. Two of the simplest approaches to mesh movement are to solve a system of spring/dampener equations or to apply a Laplacian smoothing. The downside of shape optimization is that the mesh movement limits the range of optimization, as the design will retain some resemblance to the original design, making it likely that a local minimum, instead of the global minimum, is hit. It is also challenging to preserve design features during the mesh morphing process, as the morphing tends to smooth out sharp features.

To facilitate the calculation of shape sensitivities in OpenFOAM, we register the positions of the individual points in the mesh as soon as they are created. The mesh is set up in `meshes/polyMesh/polyMesh.C`, and the points are created in the constructors

```
Foam::polyMesh::polyMesh(const IOobject& io){[...]}
```

and

```
Foam::polyMesh::polyMesh
(
  const IOobject& io,
  const Xfer<pointField>& points,
  const Xfer<faceList>& faces,
  const Xfer<labelList>& owner,
  const Xfer<labelList>& neighbour,
  const bool syncPar
){[...]}
```

respectively.

After the mesh and boundaries have been created the points on the boundary can be registered in the tape, making them parameters for the following calculations.

```
  forAll(boundary_,patchI){
    forAll(boundary_[patchI],faceJ){ // loop over all boundary faces
      const labelList ll = boundary_[patchI][faceJ];
      forAll(ll,j){ // register points of boundary faces
        ADmode::global_tape->register_variable( points_[ll[j]][0] );
        ADmode::global_tape->register_variable( points_[ll[j]][1] );
        ADmode::global_tape->register_variable( points_[ll[j]][2] );
      }
    }
  }
```

One could also register all points of the whole mesh. However, as we are only interested to move the points on the surface, and the inner points are adapted by a mesh smoothing technique, we only register the points on the surface to save some tape memory. The set of parameters is thus $\boldsymbol{\gamma} = \mathbf{Q} \in \mathbb{R}^{3n_{Q_\Gamma}}$.

After the adjoint propagation is completed, the adjoints of the individual points of the mesh can be extracted. However, one is usually not interested in the raw adjoints but wants to constrain the movement of points to the normal direction of the associated faces. Those adjoints can not be directly read from the tape, but can be calculated in a post-processing step from the adjoints of the individual points $\bar{\mathbf{q}}$. For each boundary face, the adjoints of the points contained in the patch are interpolated to the face interior by averaging the vectors, giving the face centered adjoint sensitivity vector $\bar{\mathbf{q}}_F$. Once the face center vector is calculated, it can be constrained to the face normal direction by taking the scalar product of $\bar{\mathbf{q}}_F$ with the face normal $\mathbf{n}$. We define this scalar product as the shape sensitivity $\boldsymbol{\beta} \in \mathbb{R}^{n_{F_\Gamma}}$:

$$\beta_i = \frac{\bar{\mathbf{q}}_{F_i} \cdot \mathbf{n}_{F_i}}{A_{F_i}} \quad 0 \leq i < n_{F_\Gamma} \, ,$$

where A_F is the area of the boundary face, required to obtain a mesh independent sensitivity.

For better compatibility with post processing tools, the resulting face defined quantity is copied to the face adjacent cell. The interpolation procedure is outlined in Listing 4.1.

4.4.1 Checkpointing of Shape Adjoints

Conceptually, the application of checkpointing remains unchanged from the case of topology optimization. Compared to topology optimization, the pre-processor stage is much more complex.

```
// loop over all boundary patches
forAll(mesh.boundary(),bi){
  // loop over all faces in boundary bi
  forAll(mesh.boundary()[bi],i){
    // list of point indices for face i on bi
    const labelList face_points = mesh.boundary()[bi].patch()[i];
    // cell in domain corresponding to boundary face
    const label face_cell = mesh.boundary()[bi].faceCells()[i];
    const Foam::vector face_normal = mesh.boundary()[bi].nf()()[i];
    Foam::vector sensVec(0,0,0);
    forAll(face_points,fp){
      const point& pt = mesh.points()[face_points[fp]];
      sensVec[0] += dco::derivative(pt[0]);
      sensVec[1] += dco::derivative(pt[1]);
      sensVec[2] += dco::derivative(pt[2]);
    }
    sensVec /= face_points.size();
    // scalar product of sensitivity vector with face normal
    sens[face_cell] = (sensVec & face_normal) / mesh.boundary()[bi].magSf()[i];
  }
}
```

Listing 4.1: Interpolation of sensitivity vectors, defined at points, to the corresponding faces. The shape sensitivity is computed by computing the scalar product of the resulting averaged sensitivity vector with the face normal vector.

In this stage the parameters, that is the location of the individual points of the mesh (contained in the primitive mesh), are used at various locations in the code to construct the CFD mesh representation. This mesh construction phase is only executed once and can not be restored from a checkpoint easily, therefore it is immediately included in the tape. Following the pre-processing phase, the tape is switched off and the usual checkpointed iteration phase begins. After all iteration steps have been adjoined, the remaining tape of the pre-processor is adjoined, yielding the adjoints of the parameters.

A naive implementation yields results, which are not consistent with black-box adjoints, indicating that some dependencies are missed. Those missing dependencies have been identified as the non-orthogonal correction vectors (compare Section 2.1.4) by manually comparing the tapes of black-box and checkpointed adjoint. The reason the dependencies are missed is the presence of on demand functions in OpenFOAM. Several data fields in the mesh object are stored in dynamic memory, and are only constructed once they are first requested by their access routine.

The following access functions in the `fvMesh` class create their fields on demand:

- `C()`: Constructs the cell center vector,
- `Cf()`: Constructs the face center vector,
- `V()`: Constructs the cell volume vector,
- `Sf()`: Constructs the face area vectors,
- `magSf()`: Constructs the magnitude of face area vectors,
- `deltaCoeffs()`: Constructs delta coefficients,

```
void init_mesh(Foam::fvMesh& mesh){
  mesh.Sf();                   mesh.magSf();
  mesh.C();                    mesh.Cf();
  mesh.V();                    mesh.deltaCoeffs();
  mesh.nonOrthDeltaCoeffs(); mesh.nonOrthCorrectionVectors();
}

int main(int argc, char *argv[])
{
  #include "setRootCase.H"
  #include "createTime.H"

  dco::a1s::global_tape = dco::a1s::tape_t::create();
  #include "createMesh.H"
  #include "createFields.H"

  init_mesh(mesh);
  dco::a1s::global_tape->switch_to_passive();
  [...] // checkpointed SIMPLE algorithm
}
```

Listing 4.2: Forcing the early on demand construction of the `fvMesh` fields by calling their access routines.

- `nonOrthDeltaCoeffs()`: Constructs the non orthogonal delta coefficients,
- `nonOrthCorrectionVectors()`: Constructs the non orthogonal correction vectors.

Most of these functions are first accessed during the pre-processor phase, and thus the construction of the fields is captured by the tape. However, the non-orthogonal correction vectors are first constructed when discretizing the gradient operator in the momentum equations, using the corrected surface-normal gradient scheme. The first occurrence of this discretization is in the first SIMPLE iteration, at which point the tape has already been switched off, to calculate the passive iterations needed for the checkpointing iteration. When the `nonOrthCorrectionVectors()` access function is subsequently called while the tape is active, only a reference to the field created earlier is returned. Therefore the dependence of the correction vectors on the parameters is lost.

To fix this problem, we explicitly call all on demand generator functions of the `fvMesh` instance, after the pre-processing is finished but before the tape is switched off. This might be redundant for some functions, if the field has already been initialized. However, as in that case only a reference is returned, which is subsequently ignored, the run time and memory cost of those additional calls is negligible. The actual constructors generating the data are private to the `fvMesh` class, and would require modifications inside the OpenFOAM code base in order to be accessible from our solvers. Therefore we simply trigger dummy calls to the accessor routines, which have the side effect of creating the required data fields. The changes required in order to obtain a consistent checkpointed shape adjoint are presented in Listing 4.2.

The same fixes apply when using the checkpointing interface to implement reverse accumulation or piggybacking. Figure 4.20 shows sensitivity results over iteration count for a single point on an airfoil surface (compare to Section 4.4). Sensitivities are obtained by tangent mode, adjoint mode with checkpointing (due to the cost involved only evaluated every 20 iteration steps) and

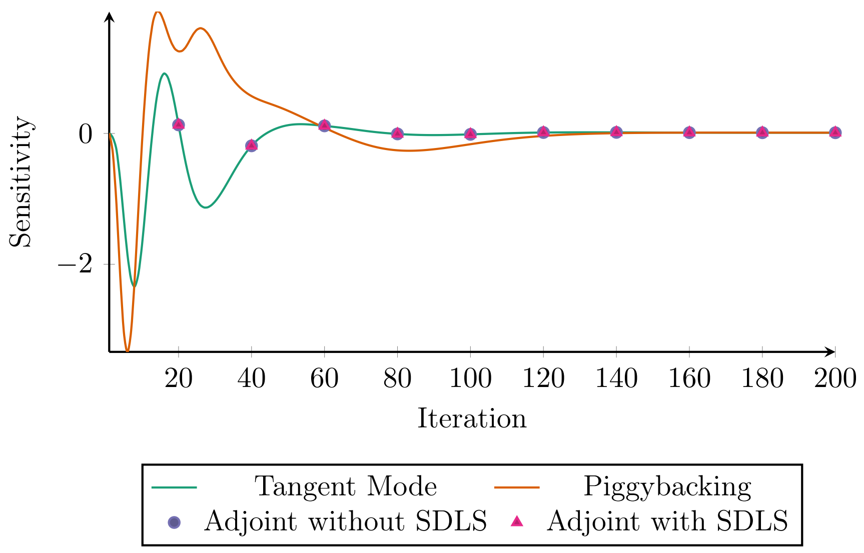


Figure 4.20: Iteration history of shape sensitivity obtained by tangent mode, adjoint mode, and static piggybacking.

static piggybacking (no design update). All converge to the same solution, with tangent and checkpointed results without SDLS being identical within machine precision. The adjoint solution with SDLS exhibits a maximum difference from the tangent solution of 0.6% and the final value matches up to 0.002%.

4.4.2 Verification of Shape Adjoints

Numerical Verification using Higher Order AD

Using the second order differentiation model, the AD implementation of the adjoints can be verified against tangents inside the same solver. As a by-product to the second order derivatives generated by the tangent over adjoint model, the model includes both the first order tangent and adjoint models.

$$\mathbf{x}_{(1)} = \nabla f^T \cdot y_{(1)}$$
$$y^{(2)} = \nabla f \cdot \mathbf{x}^{(2)}$$

Ignoring the second order derivative components, and using the correct seeding, the tangents and adjoints can be calculated within the same solver. To get representative adjoints, the full SIMPLE iteration history has to be adjoined back to the inputs after each iteration step. Using the scalar tangent mode, only one tangent (i.e. the x, y or z component of a single point) can be verified at a given time. Only a subset of points can thus be verified in a reasonable amount of time. We applied this approach and did not observe any discrepancies beyond the usual floating point precision differences. After the correctness of the AD model implementation has been checked, we focus on showing that the results are also consistent to results obtained by the continuous adjoint method.

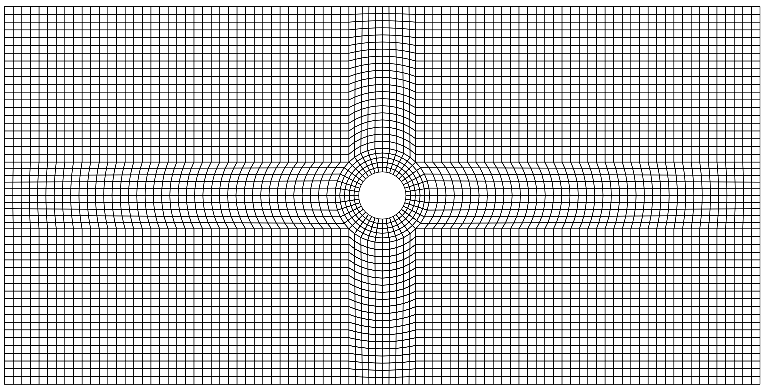


Figure 4.21: Mesh of a cylinder in laminar flow. Inlet on the left, no-slip wall on the cylinder surface, zero pressure outlet on the right, top, and bottom boundaries. The mesh used for the calculation has double the resolution.

Verification Against Continuous Adjoints

To verify the shape adjoints, laminar steady flows around a cylinder in a channel were studied. This is a well studied problem, which exhibits different kinds of behaviors for different flow conditions. For Reynolds numbers of one and lower, the flow closely resembles a potential flow and stays attached to the cylinder. For Reynolds numbers of around 10, a recirculation area begins to form behind the cylinder, however the flow remains steady. For Reynolds numbers of 100 and above, the flow becomes transient. The cylinder begins to shed vortices with a characteristic frequency (characterized by the dimensionless Strouhal number St), due to the oscillating pressure field in the wake of the cylinder. This flow phenomenon is commonly known as the *von Kármán vortex street* [Von54]. While still laminar at first, for growing Reynolds numbers the transient flow becomes increasingly turbulent.

To obtain steady solutions, which exhibit different flow characteristics, two cases with Reynolds numbers $\mathrm{Re} = 2$ and $\mathrm{Re} = 20$ were studied. A structured block mesh around a cylinder of unit diameter was chosen. The structure of the mesh is presented in Figure 4.21, however the actual mesh used for the calculations has double the spatial resolution (and thus four times as many cells). The flow enters the domain through an inlet on the left, with a prescribed flat velocity profile. It exits the domain mainly through the outlet on the right, however also the lower and upper boundaries are configured as outlets. The domain is possibly not big enough to eliminate all influence of the outflow boundaries onto the flow near the cylinder. This would be required to reliably determine the frequency of vortex shedding. However, as we are interested in steady cases and want to compare the sensitivities obtained by different approaches of differentiation, a minor influence of the boundary conditions on the flow is deemed irrelevant.

The laminar flow around the cylinder for both cases is illustrated in Figure 4.22. The former case exhibits flow around the cylinder with the streamlines of the flow near the cylinder following the cylinder surface tangentially. The latter flow is still laminar and steady, however the streamlines detach somewhere after the point of maximum cylinder width and two recirculation areas of opposing rotation form in the wake of the cylinder. Further downstream the streamlines converge together again.

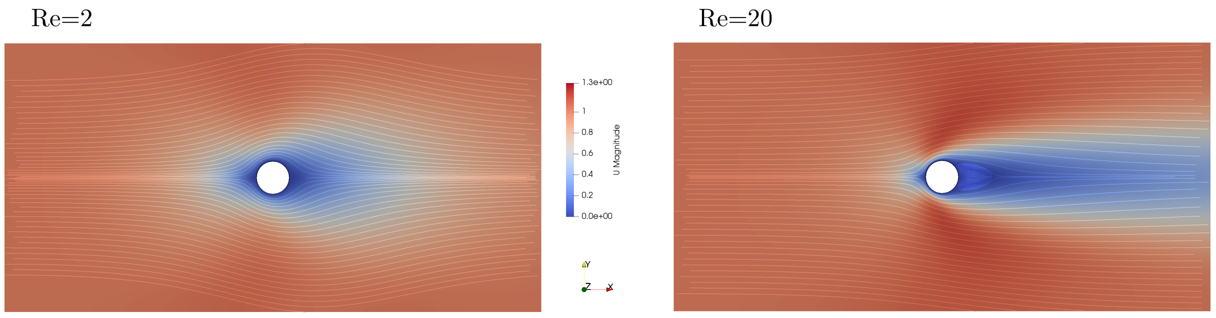


Figure 4.22: Laminar flow around cylinder for *Re* = *2* (left) and *Re* = *20* (right). For the lower viscosity, a recalculation area begins to form in the wake of the cylinder.

Sensitivities are calculated with respect to the power loss between the inlet and outlet, as introduced in Section 3.1.3. Similar results can be obtained by choosing the drag on the surface of the cylinder as cost function. The discrete results are obtained by applying the piggyback method to the simulation, without performing design updates, for 500 iterations. Continuous adjoint results are obtained by running a modified version of the stock OpenFOAM solver `adjointShapeOptimizationFoam`, expanded to calculate the wall sensitivities according to Equation (2.11) after each iteration step.

The results are presented in Figures 4.23 and 4.24. The former shows the sensitivities in a Cartesian plot, the latter in a polar plot. For both plots the abscissa, ranging from $-\pi$ to π, is the position along the surface of the cylinder, starting from the stagnation point at the outmost left position of the cylinder (due to the unit radius it is also the angle between the x-axis and the position on the cylinder surface in rad). The ordinate gives the sensitivity of the cost function to translation of the cylinder surface points in surface normal direction. A negative value indicates a movement in negative normal direction, squishing the cylinder together and reducing the volume of the cylinder. A positive value expands the cylinder in direction of the normal and thus increases the volume.

The results of the discrete adjoint $\boldsymbol{\beta}_D$ have been scaled by a uniform factor λ to best match $\boldsymbol{\beta}_C$:

$$\min_{\lambda} \|\boldsymbol{\beta}_C - \lambda\boldsymbol{\beta}_D\|_2 \,.$$

Due to the linearity of the adjoint momentum equation, the result of the continuous adjoint calculation is linearly dependent on the adjoint inlet velocity, which can be arbitrarily chosen. Therefore a linear factor between the discrete and continuous solution does not indicate a problem, and would be eliminated by the step size control of an optimization scheme.

The results show a very good match between the adjoints produced by the discrete adjoint and the continuous adjoint, especially considering that they are produced in a considerably different way. While the match between the discrete and continuous adjoint is best judged from the Cartesian plot, the influence on the shape can be better recognized in the polar plot. For the low Reynolds number case, the results indicate that an optimization would steer toward an ellipsoidal shape, in order to lower the surface area of the obstacle presented to the flow. If we assume the origin to be the center of the cylinder, the magnitude of the mesh movements is both symmetric around the x-axis and y-axis. This matches the symmetry of the geometry, mesh, and flow field.

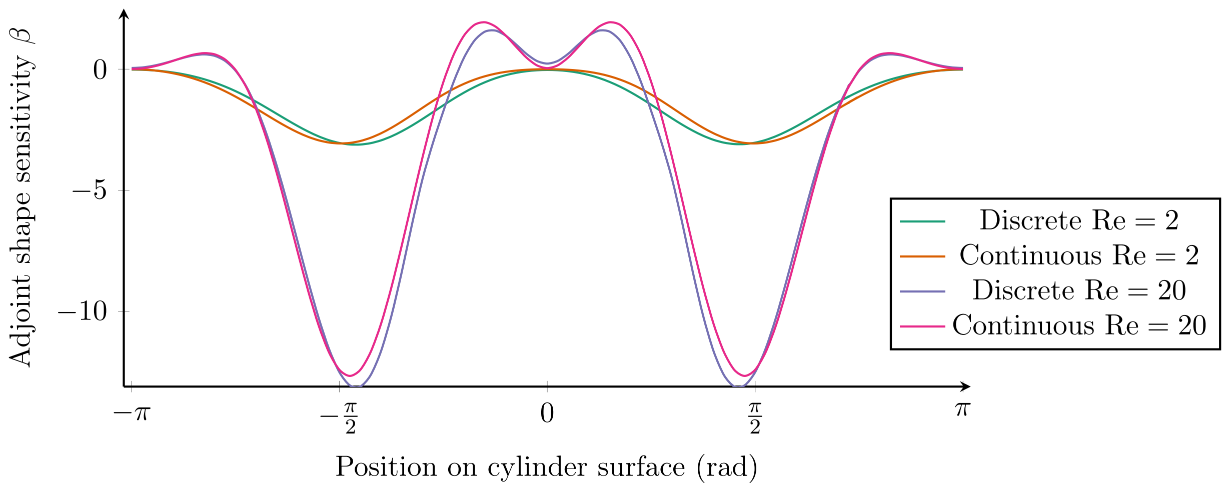


Figure 4.23: Sensitivities over location (in radians) on the cylinder surface in Cartesian coordinate system. Origin is the stagnation point on the front of the cylinder.

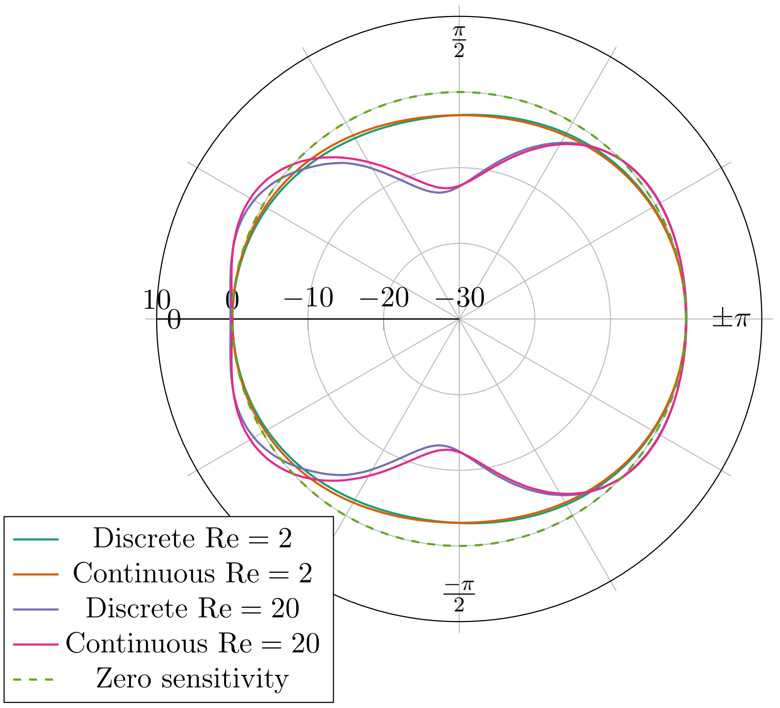


Figure 4.24: Sensitivities over location (in radians) on the cylinder surface in polar coordinate system.

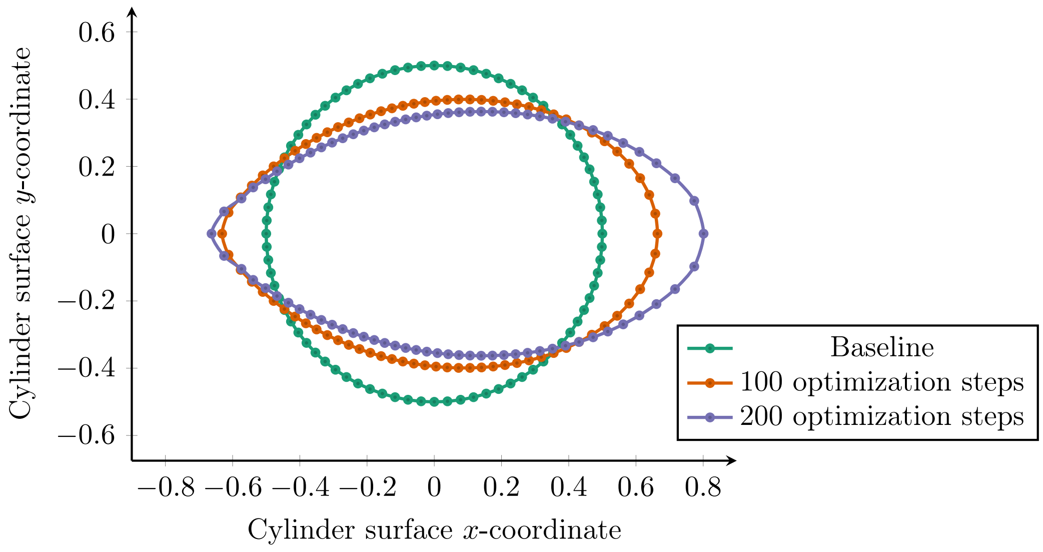


Figure 4.25: Shape optimization with fixed volume constraint of the unit cylinder.

For the higher Reynolds number case, the flow field is not symmetric around the y-axis anymore and neither are the sensitivities. The sensitivities indicate a shape more formed like a hourglass (note, that the linearized gradient only guarantees an improvement of the cost function for an infinitely small perturbation, an actual optimized geometry might look different), likely to prevent the separation of the flow from the cylinder and to combat the formation of the recirculation area. For this solution, there are regions of positive sensitivity, indicating regions where a redirection of the flow is more important than to minimize the surface area of the obstacle.

4.4.3 Shape Optimization

After having established the consistency of the discrete shape adjoints, we will now apply the gradients obtained to optimize the cylinder geometry. A mesh morphing strategy, developed in [Mol18], using adaptations of existing OpenFOAM mesh morphers, will be used. The sensitivities are supplied to the morpher to determine the amount of movement of the surface nodes. The remaining nodes are moved using a Laplacian smoothing technique [Sor+04], distributing the surface movement into the domain, decreasing the movement with increasing boundary distance. The connectivity of the mesh remains unchanged during the optimization. The sensitivity results obtained earlier suggest, consistent to intuition, to reduce the volume of the cylinder in order to obtain a lower drag on the cylinder body. Obviously the optimal solution would be to have no obstacle to the flow at all. In order to obtain a more meaningful optimization target, we will constrain the volume of the (unit) cylinder to its initial volume πh. To enforce the constraint, we choose a simple penalization approach with

$$\mathcal{J} = \mathcal{J}_D\left(\mathbf{x}, \mathbf{q}\right) + \lambda (V_0 - V_1)^2 \,,$$

where V_0 is the sum of all cell volumes in the initial configuration, V_1 the sum of cell volumes in the deformed state and λ a suitable scalar penalization parameter. Cell volumes are positive by definition, no absolute value is thus needed in the summation. Due to the fixed boundaries, the

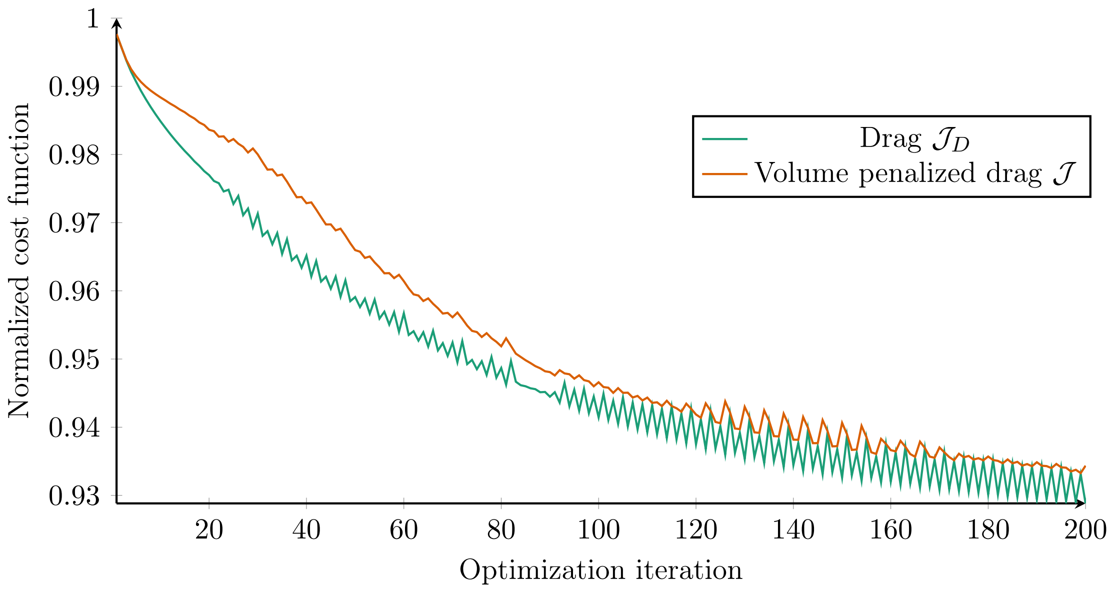


Figure 4.26: Normalized drag and penalized drag on the cylinder surface over 200 iteration steps.

change of sum of cell volumes is directly proportional (with inverse sign) to the change in cylinder volume. We start with a rather low λ and repeatedly increase it during the optimization process to stronger enforce the constraint. The application of the mesh morpher is consistent to a steepest descent method, as it moves the surface points by the gradient, scaled by a constant factor.

The optimization loop repeatedly calls the (piggyback) solver to obtain a gradient and the mesh morpher to translate the sensitivities into an updated mesh. Using the established discrete adjoint framework, it is feasible to combine both steps into one application, potentially executing a mesh update (with small movements to ensure convergence stability) after each piggybacking step. A similar combination of different utilities has been carried out to combine the `blockMesh` and `simpleFoam` utilities, yielding a solver which allows to directly optimize for parameters of the blockMesh mesh description. This approach is detailed in Section 4.6.

The movement of the cylinder surface from the baseline to an optimized configuration is shown in Figure 4.25. After 100 iterations of the shape optimization procedure, the drag is reduced by 5.8%, the volume constraint is violated by 0.85%. After 200 optimizer iterations, the drag is reduced by 6.6%, while the volume of the cylinder is much nearer to the target volume (0.12% violation of the volume constraint).

The gradual reduction in drag is shown in Figure 4.26. Due to the penalization, the constraint violation never exceeds 3%. As the gradient $\partial J/\partial \boldsymbol{\beta}$ nears zero, and the penalization parameter λ is increased, the gap between $\mathcal{J}$ and $\mathcal{J}_D$ closes.

One would suspect, that a globally optimal solution would narrow the cylinder even more. However, with a fixed mesh topology, for which only the point positions are morphed, the mesh quality will degrade for big displacements, leading to distorted meshes and eventually divergence of the solvers. Note, how the center of gravity of the deformed cylinder is not fixed and moves slightly in flow direction. If the cylinder is supposed to stay fixed, the squared movement of the

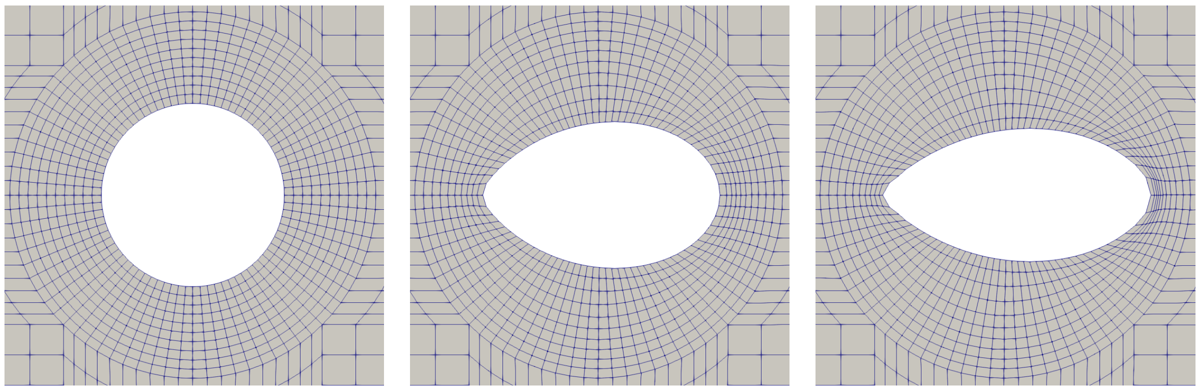

(a) Baseline mesh (b) After 100 optimization steps (c) After 200 optimization steps

Figure 4.27: Baseline mesh and morphed meshes. Mesh after 200 iterations is barely regular with highly distorted cells. The geometry should be remeshed at this point.

center of gravity could be introduced as an additional penalization term.

The original and morphed meshes are depicted in Figure 4.27. The mesh morpher keeps the mesh quality acceptable, such that e.g. no negative volumes occur, however after 200 iterations the quality has significantly degraded and the cylinder should be remeshed.

4.5 Discrete Adjoint Residual Approach

In this Section we will show the steps required to efficiently implement the discrete adjoint residual formulation, introduced in Section 2.10, in OpenFOAM. The big dimension but sparse nature of the involved Jacobians motivates using coloring techniques.

First, we efficiently determine the non-zero pattern of the residual Jacobian. Second, we color the resulting Jacobian, by applying information obtained either directly from the mesh or from an intermediate graph representation. Third, we compute the Jacobian entries using AD or FD and use the resulting linear system to compute the desired sensitivities. Last, we present applications of this methodology to our reference cases.

4.5.1 Calculation of Residuals in OpenFOAM

In the following derivations, we focus on the case of steady laminar flows, and the parameter set required for topology optimization. The flow is thus characterized by the velocity and pressure fields. To incorporate turbulence or other physical quantities, the states, and therefore also the sparse Jacobian of the residuals, can be expanded. The face flux field $\boldsymbol{\phi}$ depends on the velocities, but can not readily expressed by it with an explicit formula, because it is iteratively corrected (see Section 2.3). Therefore, to obtain accurate adjoints, the face flux is introduced as an independent variable to the residual Jacobian [RU13]. The state $\mathbf{x}$ is assembled from both cell centered quantities $(\mathbf{u}, p)$ and face centered quantities (face flux ϕ), yielding the resulting state vector $\mathbf{x} = (\mathbf{U}, \mathbf{p}, \boldsymbol{\phi}) \in \mathbb{R}^{4n_C+n_F}$. Consequently the residual $\mathbf{R} = (\mathbf{R}_U, \mathbf{R}_p, \mathbf{R}_\phi) \in \mathbb{R}^{4n_C+n_F}$ is also split between cell centered $(\mathbf{R}_U, \mathbf{R}_p)$ and face centered $(\mathbf{R}_\phi)$ entries.

The residual vector $\mathbf{R}_U$ of the momentum equation can be calculated, either using the built in residual function of OpenFOAM,

```
fvVectorMatrix UEqn(
  fvm::div(phi, U)
  + turbulence->divDevReff(U)
  + fvm::Sp(alpha, U)
  ==
  - fvc::grad(p)
);
volVectorField URes = UEqn.residual();
```

or by explicitly calculating the residual of the linear equation system:

```
volVectorField URes = (UEqn & U) + fvc::grad(p);
```

Similarly the residual of the mass conservation equation $\mathbf{R}_p$ can be calculated, either using the `fvMatrix` residual function,

```
fvScalarMatrix pEqn(
  fvm::laplacian(rAtU(), p) == fvc::div(phiHbyA)
);
volScalarField pRes = pEqn.residual();
```

or explicitly as:

```
volScalarField pRes = (fvm::laplacian(rAU, p) & p) - fvc::div(phiHbyA);
```

The update of the face flux is not calculated by solving a linear equation, but by an explicit update formula. For a converged case, the difference between two subsequent iterations of ϕ can be interpreted as the residual $\mathbf{R}_\phi$:

$$\mathbf{R}_\phi = \boldsymbol{\phi}^i - \boldsymbol{\phi}^{i-1}$$

With the update formula of the face flux in OpenFOAM,

```
volScalarField rAU(1.0/UEqn.A());
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
surfaceScalarField phiHbyA("phiHbyA", fvc::flux(HbyA));
phi = phiHbyA - fvc::flux(rAU*fvc::grad(p));
```

the calculation of the residual can be implemented in OpenFOAM as:

```
surfaceScalarField phiRes = (phiHbyA - fvc::flux(rAU*fvc::grad(p))) - phi;
```

4.5.2 Prediction of Jacobian Sparsity Pattern

In order to efficiently compress the Jacobian of the residuals, as described in Section 2.11, the sparsity pattern, that is the individual positions of the non-zero entries, of the Jacobian has to be known. A naive way to determine the sparsity pattern is to calculate the Jacobian entries densely, and then check which entries are non-zero. The resulting sparsity pattern can then be used to recompute the Jacobian more efficiently. This obviously is not very efficient and requires that the Jacobian sparsity pattern is reused multiple times to yield any improvement.

For general purpose computer programs, the sparsity pattern can be obtained more efficiently by, instead of calculating the actual entries of the Jacobian, only determining the boolean dependence of the outputs on the inputs. For a function $\mathbf{y} = f(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$, an output y_i depends on an input x_j if the partial derivative $\partial y_i / \partial x_j = J_{ij} \neq 0$, and thus this dependence implies a non-zero in the Jacobian. The determination of boolean dependence can be implemented as the (forward or reverse) propagation of dependency sets. This process is outlined below.

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be implemented by elemental functions φ with intermediate variables $\mathbf{v}$. We assign to each variable a set $D_{v_i} \subset \mathbb{N}$. For the i-th input, we initialize $D_{x_i} = \{i\}$. For a unary function $v_k = \varphi_k(v_i)$, the dependency is propagated unchanged from v_i to v_k:

$$D_{v_k = \varphi_k(v_i)} = D_{v_i} \,.$$

For binary functions $v_k = \varphi_k(v_i, v_j)$, the dependency set is determined by the union of the dependency sets of its inputs:

$$D_{v_k = \varphi_k(v_i, v_j)} = D_{v_i} \cup D_{v_j} \,.$$

Instead of implementing the dependency directly as sets, in `dco/c++` a similar approach is chosen where the sets are modeled as bitsets of fixed length. This increases the memory footprint of the program and may necessitate to split the dependency calculation into multiple parts (similar to a driver for tangent vector mode), but reduces the run time for the individual union operations on the dependencies from $\mathcal{O}(\log(n))$ to $\mathcal{O}(1)$, due to the direct memory lookup of the bitset.

Second order dependencies (needed for the assembly of Hessians) can be obtained by similar, but computationally more expensive methods [Var11].

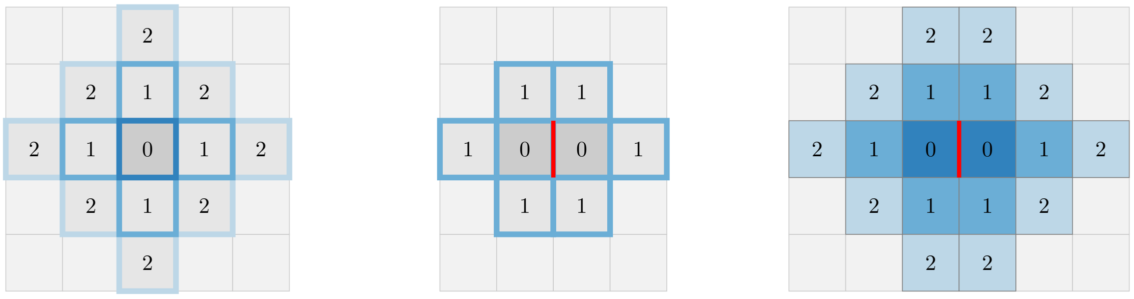
For OpenFOAM meshes, the determination of the full Jacobian sparsity pattern, using the `dco/c++` pattern data type, consumes a significant amount of time. However, if the dimensions of the finite volume stencils are known, the sparsity pattern can be exactly constructed from the mesh connectivity information. A more efficient way to determine the full sparsity pattern is thus to use the pattern type, to obtain the stencil size of an arbitrary cell inside the domain. The full sparsity pattern can then be constructed from this reference stencil and the mesh connectivity information.

The stencil sizes required for the calculation of the desired Jacobian are shown in Table 4.2. As can be seen, the stencil is rather compact, with the biggest stencils required for the calculation of $\mathbf{R}_p$ and $\mathbf{R}_\phi$. The definitions for cell and face centered stencils are illustrated in Figure 4.28.

The resulting block structure of the Jacobian of the state residuals is shown in Figure 4.29. An exemplary sparsity pattern, exhibiting these blocks, for the angled duct case with 325 cells, is presented in Figure 4.30.

Table 4.2: Stencil size for the individual Jacobian residual blocks. Superscript $*$ indicates a cell centered cell stencil, $\dagger$ a cell centered face stencil, $\ddagger$ a face centered cell stencil, and $\diamond$ a face centered face stencil.

	$\mathbf{U}$	$\mathbf{p}$	$\boldsymbol{\phi}$	$\boldsymbol{\alpha}$
$\mathbf{R}_U$	2^*	1^*	$1^\dagger$	0^*
$\mathbf{R}_p$	3^*	1^*	$2^\dagger$	1^*
$\mathbf{R}_\phi$	$3^\ddagger$	$2^\ddagger$	$1^\diamond$	$1^\ddagger$



(a) Cell centered stencils ($*, \dagger$) **(b)** Face centered face stencil ($\diamond$) **(c)** Face centered cell stencil ($\ddagger$)

Figure 4.28: Face stencil around central cell (a), face stencil around central face (b), cell stencil around central face (c).

$$
\begin{array}{c|cccc}
 & n_C & n_C & n_C & n_F \\
\hline
n_C & \frac{\partial R_{U_x}}{\partial U_x} & \frac{\partial R_{U_x}}{\partial U_y} & \frac{\partial R_{U_x}}{\partial p} & \frac{\partial R_{U_x}}{\partial \phi} \\
n_C & \frac{\partial R_{U_y}}{\partial U_x} & \frac{\partial R_{U_y}}{\partial U_y} & \frac{\partial R_{U_y}}{\partial p} & \frac{\partial R_{U_y}}{\partial \phi} \\
n_C & \frac{\partial R_p}{\partial U_x} & \frac{\partial R_p}{\partial U_y} & \frac{\partial R_p}{\partial p} & \frac{\partial R_p}{\partial \phi} \\
n_F & \frac{\partial R_\phi}{\partial U_x} & \frac{\partial R_\phi}{\partial U_y} & \frac{\partial R_\phi}{\partial p} & \frac{\partial R_\phi}{\partial \phi}
\end{array}
$$

Figure 4.29: Sub blocks of the residual Jacobian.

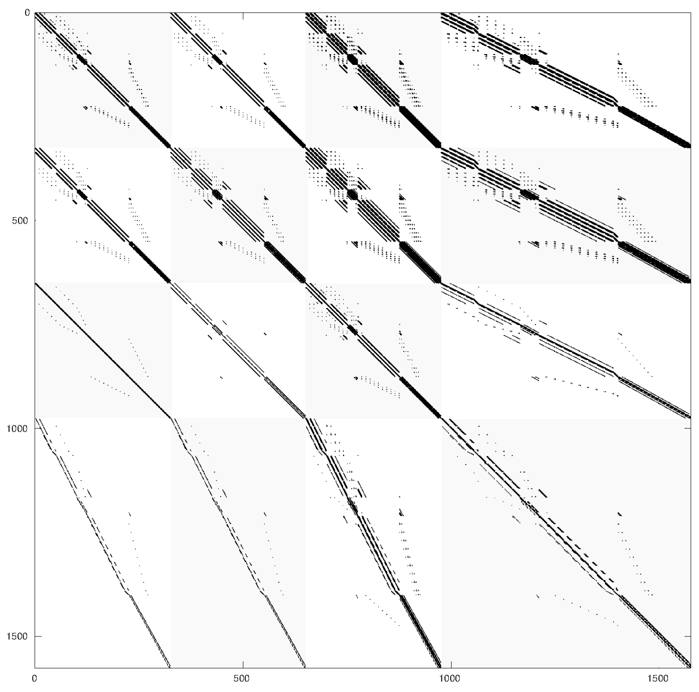


Figure 4.30: Jacobian of residual of the reference case with $n_C = 325$, $n_F = 600$, resulting in a $\mathbb{R}^{1575\times1575}$ sparse matrix with $n_{nz} = 50\,350$ non-zero entries. Sub-blocks of the Jacobian are indicated with different shades of gray background.

4.5.3 Calculation of Discrete Adjoint Residual

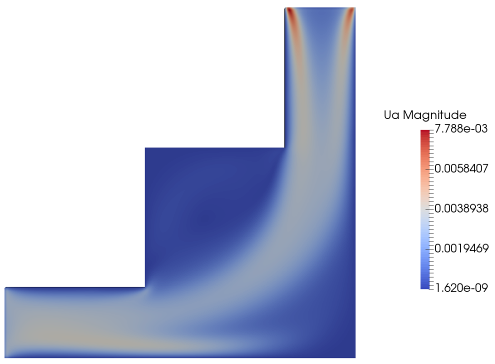
The discrete adjoint residual approach was implemented using the adjoint mode of AD, allowing to determine the Jacobian of the residuals. For validation and performance comparison, it was also implemented in tangent mode and with FD. Colorings of the Jacobian are obtained by using graph coloring algorithms, implemented in ColPack [Geb+13]. When coloring and compressing the Jacobian with ColPack (using smallest last heuristic as ordering) we observe slightly fewer colors when compressing columns than rows, giving the tangent mode and FD a slight advantage. In order to evaluate the matrix vector product $(\partial \mathbf{R}/\partial \boldsymbol{\alpha})^T \cdot \mathbf{\Lambda_x}$ from Equation (2.19) in tangent mode and FD, the matrix $(\partial \mathbf{R}/\partial \boldsymbol{\alpha})^T$ needs to be explicitly calculated. To efficiently calculate this matrix, a separate coloring for the parameters $\boldsymbol{\alpha}$ is performed, yielding significantly less colors than needed to compress the full residual Jacobian, due to the stencil size of the parameter being limited to one. The calculation of the residual Jacobian is fastest in (one sided) FD mode, followed closely by the adjoint mode, which performs well due to the high amount of re-interpretation (one seed and tape interpretation for each color but no additional tape recording). Tangent mode exhibits a constant run time overhead compared to FD. Utilizing tangent vector mode, the performance should be competitive to FD. For simplicity of the driver, this was not pursued further. The adjoint mode can be utilized in a vector mode as well, allowing reverse propagation of different seeds at the same time. This can potentially be used to further speed up the calculation of the Jacobian, at the cost of higher memory usage.

Preparing for the solution of the linear system $(\partial \mathbf{R}/\partial \mathbf{x})^T \cdot \mathbf{\Lambda_x} = (\partial \mathcal{J}/\partial \mathbf{x})^T$, the coefficients of the Jacobian are stored in an Eigen [GJ+10] sparse matrix. Storage in a native OpenFOAM format would be preferable, particularly to preserve parallelism during the solution, however OpenFOAM lacks convenient general purpose linear equation solvers for fields which are not connected to a specific geometry. The solution of the linear equation system is calculated with either the Eigen SparseLU or Eigen BiCGStab (with incomplete LU preconditioner) solvers. In our implementation, the overall run time is heavily dominated by the solution of the linear system, making the overhead of the AD tool less significant. The memory requirement to reverse the residual evaluation by AD is considerably lower than to tape a full SIMPLE iteration step. The memory required for the tape is of the same order of magnitude as the space needed to store the full sparse Jacobian matrix. For reference, to reverse one full iteration of the finer case, introduced below, about 610 MB of tape space (with SDLS enabled) are needed, while the tape size required to capture the calculation of the FVM residual is 408 MB. The total memory consumption including the storage and solution of the sparse matrix is 1 250 MB.

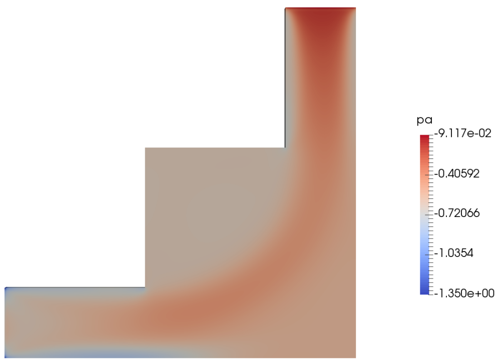
Therefore, despite requiring multiple evaluations of the tape to obtain the full Jacobian, the adjoint method is competitive with FD, in both run time and memory consumption.

To illustrate the results of the discrete adjoint residual solver, we once again turn to the angled duct example. Because this case is a laminar 2D case, the state vector consists of $\mathbf{x} = (\mathbf{u}_x, \mathbf{u}_y, \mathbf{p}, \boldsymbol{\phi})$ and consequently the size of the residual Jacobian is $(3n_C + n_f) \times (3n_C + n_F)$.

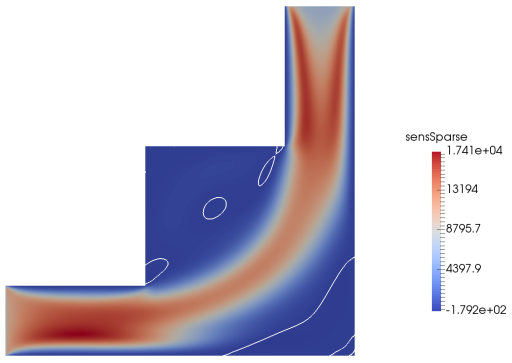
We investigate two different mesh resolutions, a coarse mesh with $n_C = 2\,925$ cells and $n_F = 5\,700$ internal faces and a fine mesh with $n_C = 46\,800$ cells and $n_F = 93\,000$ internal faces. A ColPack bipartite graph representation is build from the sparsity pattern, then partial row/column distance two coloring is applied to obtain a suitable Jacobian coloring. Coloring the columns of the coarse Jacobian using ColPack yields 90 colors, coloring the rows 102. The fine Jacobian yields slightly more colors, namely 95 for coloring the columns and 106 for coloring the rows. As the general connectivity of the mesh is not changed by the mesh refinement, the lower



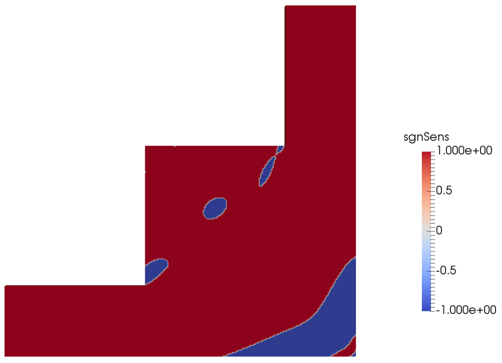
(a) Adjoint velocity



(b) Adjoint pressure



(c) Sensitivity $\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}}$



(d) Sign of sensitivity $\sigma(\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\alpha}})$

Figure 4.31: Solutions for the fine angled duct case obtained by the discrete adjoint residual.

limit for the number of colors likely does not increase with finer meshes; However, the coloring heuristic performs slightly worse for the finer mesh.

Figure 4.31 shows the sensitivity results of the angled duct case, as obtained by the discrete adjoint residual method. The first row of figures shows the adjoint velocity and pressure, which can be extracted from the solution $\Lambda_{\mathbf{x}} = \left(\bar{\mathbf{U}}, \bar{p}, \bar{\boldsymbol{\phi}}\right)$ and have the same physical meaning as the adjoint velocities and pressures defined for the continuous adjoints. The second row shows the final sensitivities $\mathrm{d}\mathcal{J}/\mathrm{d}\boldsymbol{\alpha}$, as well as the sign of the sensitivity for easier cross reference.

The sensitivity results are identical for tangent and adjoint mode (up to machine precision) and align with the FD results very well. The results also match the results obtained by both the discrete black-box differentiation and the continuous adjoint presented in Section 4.3.3.

Note that the adjoint velocities $\bar{\mathbf{u}}$ obtained by this method are vector quantities. They can be used to evaluate Equation (2.11) to obtain shape derivatives, equivalent to the continuous adjoint approach, circumventing the need to differentiate through the generation of the mesh from the individual points.

Table 4.3 lists the run times of the following phases for adjoint, tangent, and FD mode:

Table 4.3: Run times of the different stages; Memory consumption for adjoint, tangent, and FD solver for the coarse and fine level angular duct. Passive run time is calculated over 200 iteration steps.

Coarse Case	A1S	T1S	FD
Colors	102	90 / 23	90 / 23
Passive	6.63 s	4.93 s	2.26 s
Pattern	0.58 s	0.60 s	0.63 s
Diff	0.74 s	1.16 s	0.34 s
Solve	2.81 s	3.15 s	2.48 s
Total run time	10.90 s	10.44 s	6.17 s
Max memory	176.39 MB	155.54 MB	131.35 MB

Fine Case	A1S	T1S	FD
Colors	106	95 / 25	95 / 25
Passive	122.88 s	100.00 s	40.52 s
Pattern	9.94 s	10.24 s	11.47 s
Diff	14.99 s	20.66 s	5.68 s
Solve	147.40 s	166.42 s	152.85 s
Total run time	303.11 s	307.00 s	217.00 s
Max memory	1254.34 MB	871.55 MB	805.51 MB

Passive evaluation: Iteration of the case from initial condition to a converged state.

Assembly of sparsity pattern: Assembly of the expected sparsity pattern of the Jacobian from mesh connectivity.

Coloring: Conversion of sparsity pattern to ColPack graph format and partial coloring of the bipartite graph.

Differentiation: Calculation of the Jacobians $J_{\mathbf{x}}$ and $J_{\boldsymbol{\alpha}}$ ($J_{\boldsymbol{\alpha}}$ only required for tangent mode and FD).

Solution: Solution of the linear system and calculation of (2.18).

All stages are performed in one solver. In practice it can be useful to separate the stages, as the sparsity pattern and coloring remain constant for a specific mesh and are independent of e.g. boundary conditions. They can thus be reused for different configurations of the same case. The iteration procedure can be started from a partially or fully converged state (which can be created by a fully passive version of OpenFOAM) instead of the initial state, lowering the time for passive evaluation.

4.5.4 Directly Obtaining Colors from the Mesh Representation

In order to reduce the computation time of the Jacobian, a coloring approach, as presented in Section 2.11, is used to compress rows or columns of the Jacobian. Previously we used the external software package ColPack to obtain a suitable coloring. This necessitated the calculation of the non-zero pattern, as well as the construction of a graph structure from the Jacobian non-zero pattern. We will now introduce a method to directly obtain a feasible coloring from the FVM mesh representation.

Mesh connectivity graph

To analyze coloring problems, arising from the discretization of CFD problems, one would like to utilize already well known results from graph theory. Therefore, it is desirable to introduce a graph representation of the mesh connectivity, as some properties defined on graphs can be reused on the mesh description.

Definition 19 (Mesh connectivity graph).
Let $M = (\boldsymbol{L}, \boldsymbol{U})$ be a mesh addressing given in LDU *format, with $\boldsymbol{L}, \boldsymbol{U} \in \mathbb{N}^{n_F}$. We define the corresponding mesh connectivity graph $G_M = (V_M, E_M)$, consisting of:*

- *One node for each cell in the mesh: $V = \{v_i \mid i = 0, \dots, n_C - 1\}$;*
- *Each edge in the graph corresponds to an interior face connecting two cells, that is a pair $(l_i, u_i) \in (\boldsymbol{L}, \boldsymbol{U})$: $E = \{(c_{L_i}, c_{U_i}) \mid i = 0, \dots, n_F - 1\}$.*

A cell is directly adjacent to another cell if it shares a face in the mesh. This corresponds to an edge in the mesh connectivity graph. The distance between two cells (c_i, c_j) can be conveniently defined as the length of the shortest path in the mesh connectivity graph connecting both cells.

Definition 20.
Let v_i and v_j be cells of a finite volume mesh, that is they are vertices in G_M. We define the distance $d = \mathcal{D}(v_i, v_j) \in \mathbb{N}$ between cells v_i and v_j as the length of the shortest path in G_M connecting the nodes corresponding to both cells.

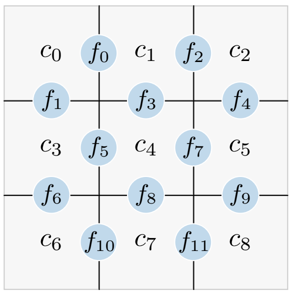
Figure 4.32 gives an example 3×3 mesh with 9 cells and 12 internal faces, its corresponding mesh connectivity graph, and its internal LDU representation.

Application of Mesh Connectivity to CFD problems

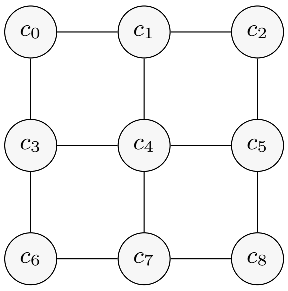
Definition 21 (Cell Stencil).
In CFD the stencil of discretization is defined by the cell neighborhood around a cell which influences the value of the solution at this specific cell in the next iteration step. The size of this stencil is defined as the maximum distance between the two cells in the cell adjacency graph.

An example of two non overlapping stencils of size two on a structured mesh is shown in Figure 4.33. The spreading of information over consecutive iteration steps, as well as the distance $2d$ neighborhood needed for the later proof, is illustrated in Figure 4.34.

We will show that a distance $2d$ coloring on the cell adjacency graph corresponds to a partial distance two coloring of the bipartite graph of the Jacobian. Thus it can be used to compress the Jacobian matrix of the residuals. The same can easily be shown for face stencils.


c_0 f_0 c_1 f_2 c_2
f_1 f_3 f_4
c_3 f_5 c_4 f_7 c_5
f_6 f_8 f_9
c_6 f_{10} c_7 f_{11} c_8

(a) Mesh with cell and face numbers.



(b) Mesh connectivity graph

$$\boldsymbol{L} = (0,0,1,1,2,3,3,4,4,5,6,7)^T$$
$$\boldsymbol{U} = (1,3,2,4,5,4,6,5,7,8,7,8)^T$$

Figure 4.32: Mesh connectivity graph, derived from ldu Addressing corresponding to 3×3 mesh.

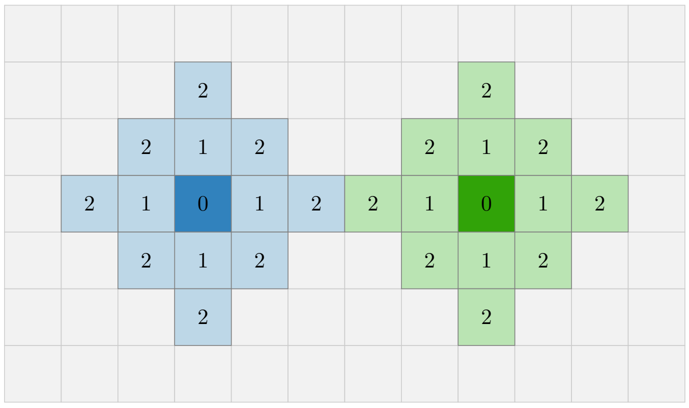


Figure 4.33: Non overlapping stencils of size two around two cells located at distance four.

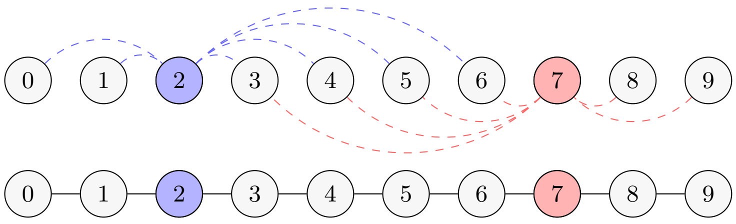


Figure 4.34: Mesh connectivity graph of 1D mesh. Node 2 and 7 are connected with dashed edges to all nodes reachable by distance $2d = 4$.

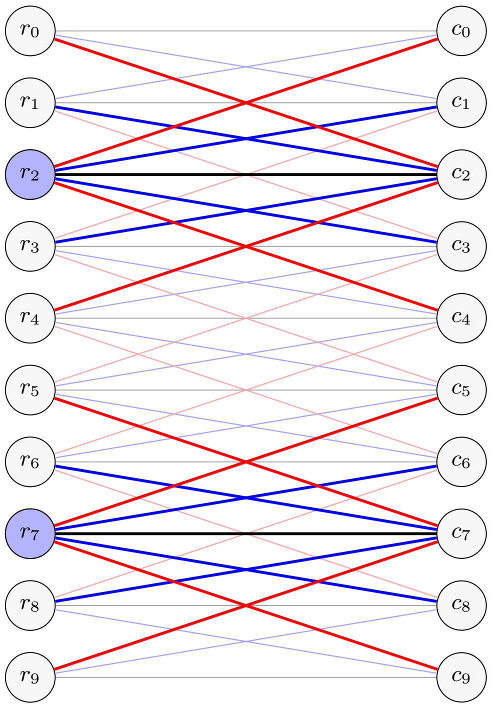


Figure 4.35: Bipartite graph for 1D structured mesh with $n_C = 10$ and finite volume stencil of size two. Nodes r_2 and r_7 (or c_2 and c_7 for column compression) can share the same color, as no path of length two exists between them. Blue edges indicate matrix entries created by stencils of size one, red edges additional matrix entries created by stencil of size two.

Theorem 8.
A distance $2d$ coloring, where d is the maximum stencil size used in the finite volume discretization, on the mesh connectivity graph $G_M = (V_M, E_M)$, can be used to compress the Jacobian of the residual of an equation discretized by this stencil.

Proof. We will transform the coloring problem on G_M into an equivalent problem on the bipartite adjacency graph, which can be colored according to Lemma 1. Let two cells v_i, v_j be at distance d or less in the mesh connectivity graph, that is, starting from cell i, cell j can be reached by crossing at most d faces. Then the undirected edges $(r_i, c_j) \in E_B$ and $(r_j, c_i) \in E_B$ are both part of the bipartite graph $G_B = (V_B, E_B)$ of the Jacobian induced by the finite volume stencils (as v_j is inside the stencil of v_i and vice versa).

Let two cells $v_{i'}, v_{j'}$ be at distance of $2d$ or less then there exists a path in the bipartite graph of length two $r_{i'} \to c_k \to r_{j'}$ with $\mathcal{D}(v_{i'}, v_k) \leq d$ and $\mathcal{D}(v_k, v_{j'}) \leq d$. Thus, cells $v_{i'}$ and $v_{j'}$ may not share a common color for a direct recoverable row compression (Lemma 1). Due to symmetry, the same argument holds for column compression and the path $c_{i'} \to r_k \to c_{j'}$.

Now let the two cells $v_{i'}, v_{j'}$ be at distance of at least $2d+1$. There exists no path $r_{i'} \to c_k \to r_{j'}$ in the bipartite graph, as for each k either $\mathcal{D}(c_k, c_{i'}) > d$ or $\mathcal{D}(c_k, c_{j'}) > d$, or both. Thus, cells $v_{i'}$ and $v_{j'}$ can share a color without breaking the distance two condition on the bipartite graph.

Therefore, a mesh connectivity graph, colored such that no nodes at distance $2d$ or lower are colored with the same color, corresponds to a bipartite graph with a valid (partial) distance two coloring. Such a coloring can be used to compress the Jacobian calculation (Lemma 1). □

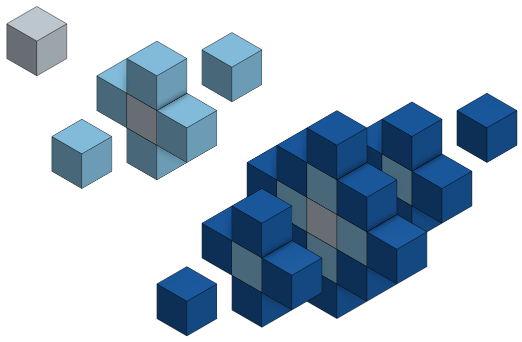


Figure 4.36: Distance-0, distance-1, and distance-2 stencils in three dimensions. Stencils exploded along depth axis.

With the mesh connectivity graph, a valid coloring can be obtained directly on the mesh description (see Section 4.5.2). The OpenFOAM LDU mesh description implicitly models the mesh connectivity graph. The following four ordering heuristics, which are ordered from least to most computationally expensive, have been implemented directly in an OpenFOAM solver.

Natural Ordering: Color cells in order of their cell numbering.

Random Ordering: Color cells according to a random permutation of their cell numbering.

Approximate Largest First: Compute the size of the distance-1 neighborhood for each cell and sort big to small. Color in this order. The distance-1 neighborhood is already available in the OpenFOAM mesh representation, the evaluation of the neighborhood size is thus cheap.

True Largest First: Compute the size of the full distance-4 neighborhood for each cell, and sort big to small. Color in this order.

First we investigate the coloring performance for a structured hexahedral $n \times n \times n$ mesh of the unit cube. A distance two stencil (blue), as well as all cells colored with color zero (red) for the $10 \times 10 \times 10$ unit cube, are shown in Figure 4.39. A graphical representation of structured 3D stencils up to distance-2 are given in Figure 4.36.

All above heuristics exhibit a run time behavior linear in the number of cells $n_C = n^3$ of the mesh. This is to be expected, as each cell is individually colored, and the cost per cell is constant as is argued below. For every cell, a breadth-first search [Moo59] (BFS) is performed, however the BFS is stopped after a fixed number of steps (the desired coloring distance). Thus the run time of a single BFS is independent of n_C. Instead it depends on the connectivity of the mesh and the type of the used cells. The linear run time behavior can be seen in Figure 4.37, where the resolution of the unit cube is scaled from $N = 10^3$ to $N = 100^3$. The corresponding number of colors required to color the Jacobian are shown in Figure 4.38. The number of colors required remains largely constant as n_C rises.

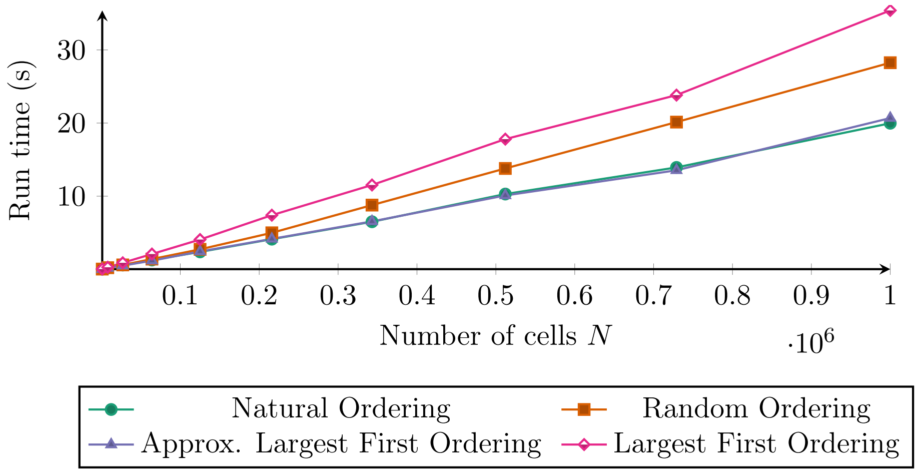


Figure 4.37: Run time behavior of the presented heuristics. Run time scales linear with n_C.

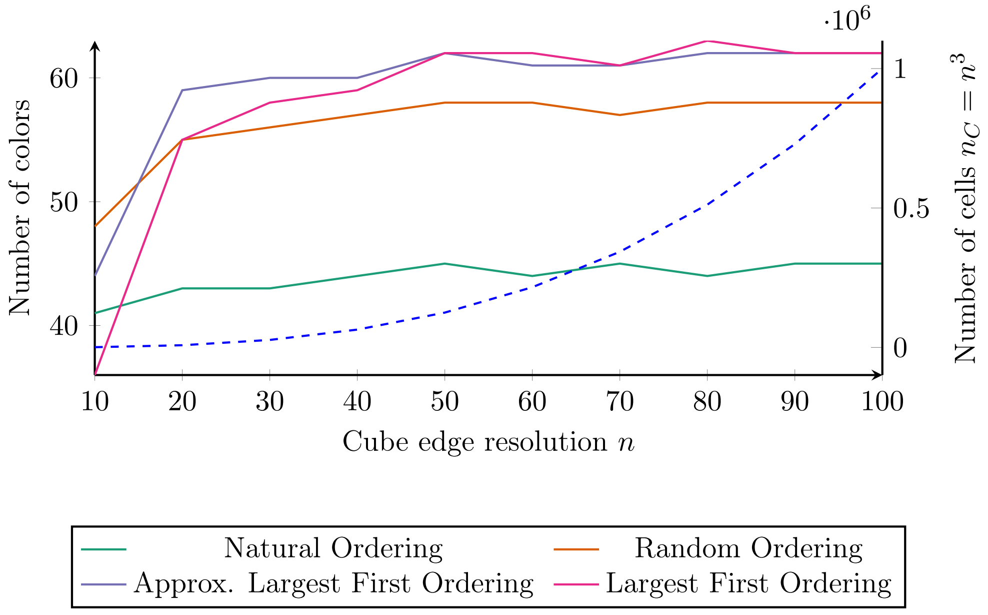


Figure 4.38: Number of colors required to color the unit cube for the presented heuristics. Number of cells n_C dashed on second axis.

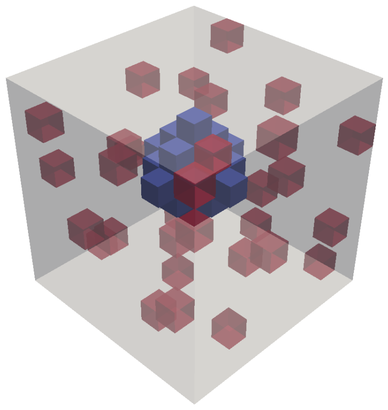


Figure 4.39: Unit cubed meshed with $10 \times 10 \times 10$ cells, colored with 36 colors. Only cells of the first color (red), as well as one stencil (blue) is shown.

For unstructured meshes, the true largest first heuristic performs very well. For structured meshes, it performs worse, which is explained by the uniform size of the cell neighborhoods, yielding many nodes with the same degree. In this case the heuristic basically falls back to a natural ordering (natural ordering is used as tie-breaker), except near boundary patches, where the number of cell neighbors is lower.

In the implementation it has been observed, that using a flat (vector based) set representation instead of the red-black tree [Cor+09] based implementation in the `C++` standard library [Pla+00] is considerably more efficient when performing the BFS on the cell neighborhood. An implementation of the BFS search around a given cell in the mesh representation is given in Listing 4.3. To improve performance and reduce memory, the recursive BFS algorithm is explicitly unrolled to a loop based implementation,

The size of the cell neighborhood is determined by the number of neighboring cells. For structured hexahedral meshes, the size of the neighborhood is determined by

$$n_N = \begin{cases} 2d + 1 & \text{1D} \\ d^2 + (d+1)^2 & \text{2D} \\ \frac{1}{3}(4d^3 - 6d^2 + 8d - 3) & \text{3D}\,, \end{cases}$$

where d is the stencil size. The full distance-4 neighborhood consequently contains 41 cells for 2D and 129 cells for 3D structured hexahedral meshes.

For unstructured meshes, the size of the cell neighborhood is obviously much more varied. Here the largest first heuristic performs well. While this heuristic takes considerably longer to evaluate, it produces an ordering that yields significantly fewer colors. This is illustrated by the *motorbike* and *VW Polo* cases in Table 4.4. The *motorbike* case consists of 352 863 cells and has a maximum cell neighborhood of size 455, due to the polyhedral nature of the mesh. The *VW Polo* case is hex dominated, with tetrahedrons connecting the interior elements to the boundary surfaces. This mesh contains 7.75 million cells, the maximum size of the distance-4 cell neighborhood is 595.

The run time of the coloring heuristics is dominated by the BFS during the coloring stage. Only for the true largest first coloring heuristic the creation of the ordering takes a significant

Table 4.4: Number of required colors and run time in seconds for the different ordering heuristics.

	Cube 10^3		Cube 100^3		Pitz-Daily	
Heuristic	Colors	Run time	Colors	Run time	Colors	Run time
Natural Ordering	41	0.03	45	19	19	0.07
Random Ordering	48	0.02	58	27.4	25	0.08
Approximate Largest First	44	0.03	62	18.4	25	0.08
True Largest First	36	0.04	62	32.3	25	0.13

	Pitz-Daily 3D		Motorbike		VW Polo	
Heuristic	Colors	Run time	Colors	Run time	Colors	Run time
Natural Ordering	48	2.04	99	13.55	110	307.23
Random Ordering	56	3.18	90	15.74	101	384.95
Approximate Largest First	60	2.15	96	14.09	115	315.02
True Largest First	48	3.84	79	25.89	87	622.12

amount of time, as it also evaluates the cell neighborhood using BFS. The overall run time for the true largest first scheme can potentially be improved, at the cost of much higher RAM usage, by saving the BFS results during the ordering phase and reusing them during coloring.

The true largest first distance-4 coloring on the mesh produces results which are competitive to colorings obtained by ColPack [Geb+13] on the adjacency graph of the Jacobian. For the VW Polo case, which is the biggest case we considered, ColPack produced 85 colors, compared to the 87 colors obtained by the coloring implemented directly in OpenFOAM. A visualization of the cell colors of the near wall cells, mapped onto the surface of the car body is shown in Figure 4.40.

```cpp
set<label> calc_cell_neighborhood_BFS(label root, label k, fvMesh& mesh){
  set<label> neigh;
  neigh.insert(root);
  set newCells = neigh; // copy root set
  for(int i = 0; i < k; i++){
    set<label> tmpCells = newCells; // iterate over all cells of old level
    newCells.clear(); // empty set for next level
    for(label it : tmpCells){
      labelList& newNeigh = mesh.cellCells(it);
      forAll(newNeigh,j){
        // returns pair<iterator,bool>, true if element not previously in set
        auto ret = neigh.insert(newNeigh[j]);
        // only add new elements for further exploration in next level
        if(i < k-1 && ret.second)
          newCells.insert(newNeigh[j]);
      }
    }
  }
  return neigh;
}
```

Listing 4.3: BFS algorithm to identify the distance `k` neighborhood around cell index `root`.

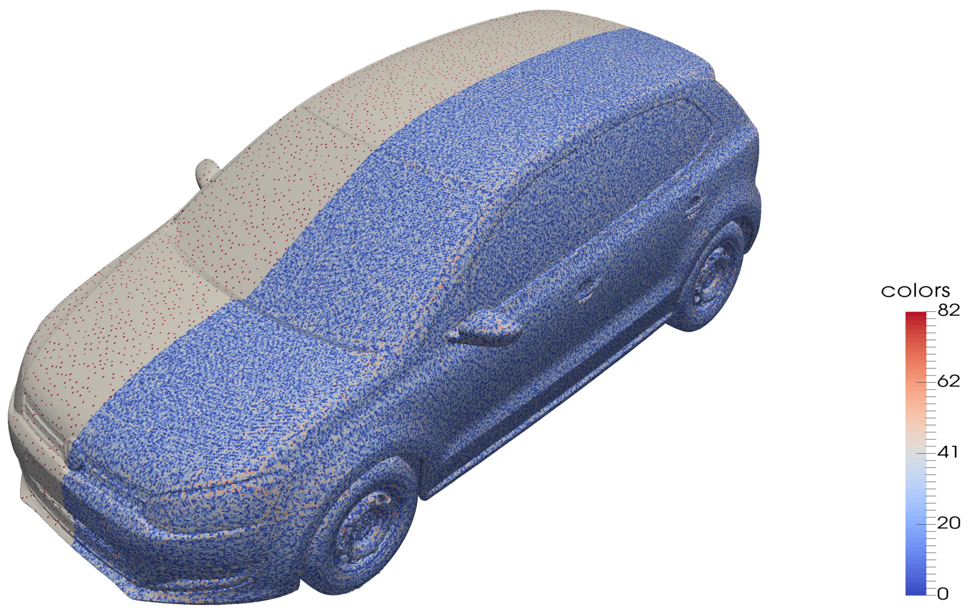


Figure 4.40: Cell colors of distance-4 coloring of near wall cells. Plot is split at the $y = 0$ plane. On the $y > 0$ half of the plot only cells of color zero are shown in red.

4.6 Parametric Optimization

4.6.1 Introduction

In addition to high dimensional optimizations, carried out by topology and shape optimization, the discrete AD model can also be applied to a parametric optimization setting. Parametric optimizations use a set of parameters, which model the shape of the geometry in some way. For example, a pipe could be modeled by a spline, defining the centerline of the pipe, and a set of radii, which model the cross section of the pipe.

Compared to the state vector $\mathbf{x} \in \mathbb{R}^{n_\mathbf{x}}$, the dimension of the parameter vector $\boldsymbol{\gamma} \in \mathbb{R}^{n_\gamma}$ is rather low, with $1 \leq n_\gamma \ll n$. Therefore a calculation using either tangent mode or adjoint mode is feasible. The parameters are an input to the mesher, which transforms the parameters to the full mesh representation.

To circumvent the need to differentiate the mesher and solver at the same time, a hybrid approach is possible. In this setting, the sensitivities of the generated points $\mathbf{Q}$ (output of mesher $\mathcal{M}$) w.r.t. the design parameters $\mathrm{d}\mathbf{Q}/\mathrm{d}\boldsymbol{\gamma}$ are generated using tangent (vector) mode at cost $\mathcal{O}(m) \cdot \mathit{cost}(\mathcal{M}(\boldsymbol{\gamma}))$. The sensitivity of the (scalar) cost function, with respect to all points $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{Q}$, is generated in adjoint mode at cost $O(1) \cdot \mathit{cost}(\mathcal{J}(\mathcal{F}(\boldsymbol{\gamma})))$. The final sensitivities can then be calculated by the following product:

$$\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\boldsymbol{\gamma}} = \frac{\partial \mathcal{J}}{\partial \mathbf{Q}} \cdot \frac{\partial \mathcal{M}}{\partial \boldsymbol{\gamma}}\,.$$

An advantage of parametric designs is that, assuming a reasonable compact parametrization is chosen, the final design can be straightforwardly modeled in CAD tools by changing the parameters of the initial design to the final optimized values. The reduced parameter set of parametric

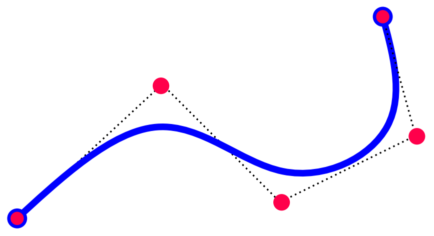


Figure 4.41: Illustration of a B-spline, defined by five control points (red). Spline in solid blue, corresponding control polygon dashed. For the optimization, only the inner three control points are treated as parameters.

optimization also allows the application of advanced optimization schemes, as parameters can more easily be constrained and Hessian approximations can be assembled more effectively. The remeshing after every optimization step ensures that the mesh quality stays consistent, which for a shape optimization, moving all surface points, is much more challenging and might require excessive amounts of smoothing of the surface features.

4.6.2 Parametric BlockMesh Optimizer

As a proof of concept, we differentiate through a combined solver of the mesher `blockMesh` and flow solver `simpleFoam`, using adjoint AD.

The *blockMesh* mesh description syntax is a plain text description of blocks, faces, and edges. From this description, a mesh is generated by the `blockMesh` utility, using hexahedral elements. The resulting meshes resemble structured meshes (OpenFOAM meshes are always stored unstructured). Edges can be curved, where the curves are parametrized by splines and their corresponding control points. Different spline interpretations are available including Bézier curves, B-splines and Catmull–Rom splines. For this proof of concept, we only considered B-spline curves [De 78], as they best retain tangential relations in the geometry. The other spline types can be differentiated analogously. B-splines are defined by a control polygon, that is a piecewise linear path through the control points. The spline connects the start to the endpoint, but does not pass through the intermediate control points. Such a spline is illustrated in Figure 4.41. The function defined by the control points, as well as its derivative are continuous, unless points are multiply defined.

All spline control points are registered as parameters in the tape, as soon as the block edges are created from the control points. The mesh utility afterwards evaluates the spline at the required intervals and constructs the mesh primitives (points, faces, cells, etc.). After the mesher has finished, it returns a `polyMesh` object which holds the mesh information. Instead of reading the mesh information from the mesh files as usual, the finite volume discretization is instead constructed from this `polyMesh` object, leaving the derivative information between mesher and solver intact.

The mesh generation with `blockMesh` is inherently limited to serial execution. To retain the parallelism of the adjoint flow solver, the points generated by the serial mesher need to be distributed to the parallel nodes. The dependencies of the points on the parameters can

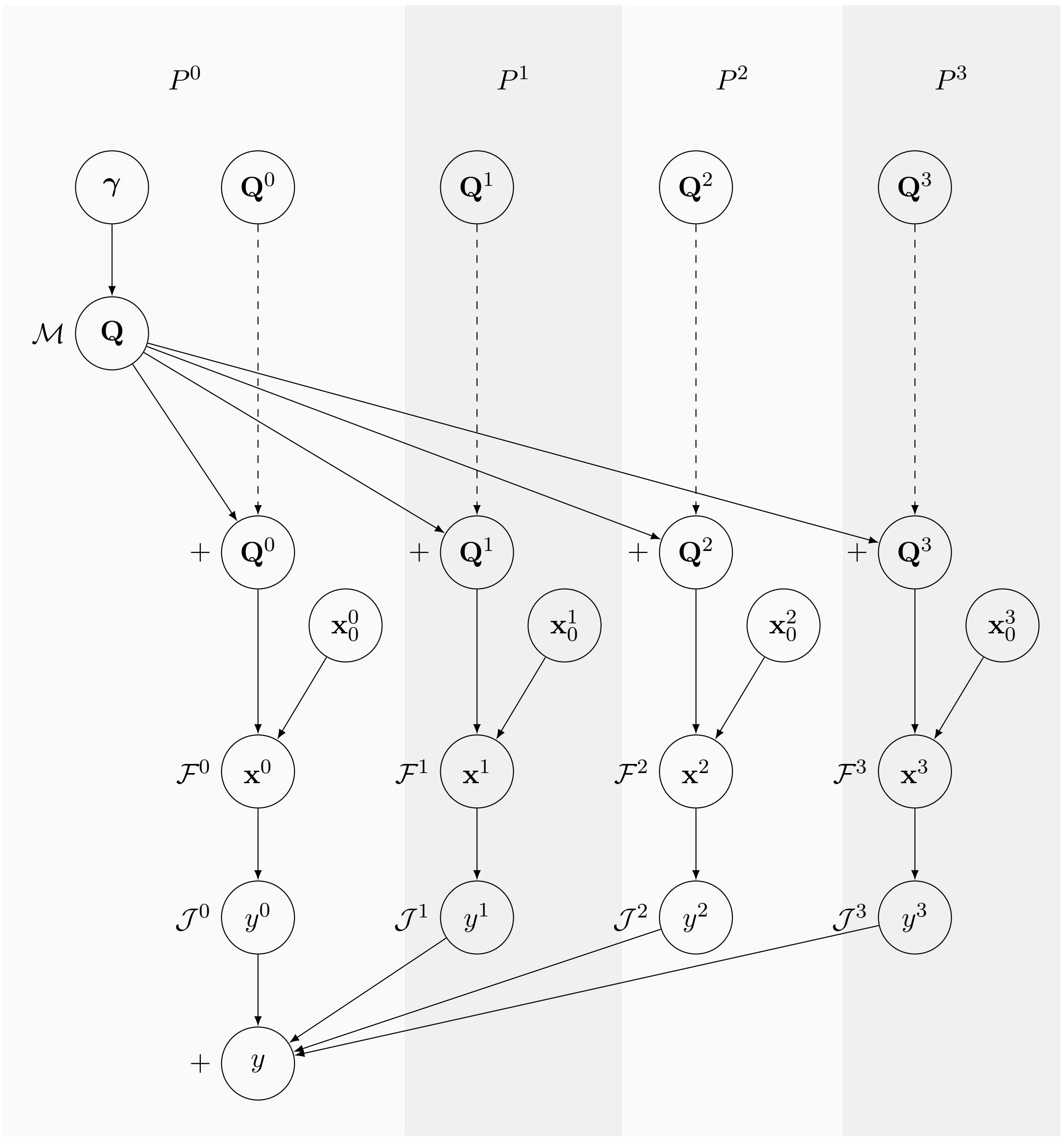


Figure 4.42: Distribution of the global point fields $\mathbf{Q}$, created by the mesher $\mathcal{M}$, to local point fields, required for parallel solver execution $\mathcal{F}$. Adjoints are seeded into $\bar{y}$ on processor P_0 and then propagated back to the parameters $\boldsymbol{\gamma}$.

be retained by allowing AMPI to capture the communication. To facilitate this, we load the distributed finite volume mesh from an existing domain decomposition, but update the processor local point fields from the global field generated by the mesher (which triggers a recalculation of the primitive mesh). The global mesh point vector is thus split into several local parts and distributed by (A)MPI to the corresponding processors. This introduces a communication overhead, as well as the need to hold the whole point mesh on one processor (also required by the mesher anyway).

The derivatives $\mathrm{d}\mathcal{J}/\mathrm{d}\mathbf{Q}$ of the serial or parallel solver $\mathcal{S}$ can be calculated by any mean feasible to adjoin the iteration history. This step is essentially identical to the calculation of the full shape adjoints. For the proof of concept, we choose the piggyback approach, repeatedly adjoining steps of the SIMPLE algorithm until the adjoint of the state $\bar{\mathbf{x}}$ is sufficiently small. The product with $\mathrm{d}\mathcal{M}/\mathrm{d}\gamma$ can be calculated with a single interpretation of the remaining tape of the mesher.

4.6.3 Application to Pitz-Daily Geometry

As an application, we again use the Pitz-Daily backward facing step case, used for topology optimization in previous sections. The step is replaced by a ramp, such that the width of the nozzle gradually expands. Depending on the steepness of the ramp, the flow might still detach or remain attached to the lower wall. Except for the change of mesh topology, the boundary conditions are kept identical to the case previously discussed. As a first step, to find a sensible initial shape, the angle of the ramp is varied, while keeping the walls straight. As can be seen in Figure 4.43, the optimal region is quite broad. Depending on the starting position a local optimization will run into different local minima, located at different angles but with only slightly differing values of the cost function. The parameter $\gamma_0 = S_{0,x}$ is chosen from the middle of the optimal region as 108°. The global optimum is located in this region, both according to a brute force evaluation of the cost functions for different angles, evaluated at 65 different positions of S_{0_x} between 50 mm and 180 mm, and according to a gradient based optimization starting at $S_{0_x} = 50\,\mathrm{mm}$, employing random perturbations after convergence to escape local minima (basin hopping [WD97]). As the variation of S_{0_x} changes the length ratio between the different blocks, the cells in x-direction are dynamically redistributed between blocks B_1 and B_2 (see Figure 4.44), retaining mesh quality while keeping the global cell count constant.

Next, the shape of the wall is modeled by a B-spline with three interior control points. In the initial configuration the control points are placed, such that the wall is tangential to the connecting walls. The control points are allowed to move in x and y-direction, giving the optimizer seven DOF for the spline control points. The gradient is assembled from the derivatives of the cost function, here again power loss, with respect to the positions of the control points, as well as the x-coordinate of the ramp endpoint, to allow corrections of the ramp slope if required. The vector of parameters is thus $\boldsymbol{\gamma} = \left[S_{0_x}, S_{1_x}, S_{1_y}, S_{2_x}, S_{2_y}, S_{3_x}, S_{3_y}\right] \in \mathbb{R}^7$, with the corresponding gradient of same dimension $\nabla_{\boldsymbol{\gamma}}\mathcal{J} \in \mathbb{R}^7$. The control points of the splines are bounded by $[y_{\min}, 0]^3$ to avoid mesh breaking overshoots of the solution (which except for badly behaved FD approximations of the gradient was not strictly needed).

The optimization was carried out using the SLSQP solver [Kra94], implemented in the SciPy [JOP+01] optimization framework, as it both implements parameter bounds and linear constraints and proved more reliable than other solvers. For the unconstrained (but bounded) case, the SLSQP method resembles the (Pseudo-) Newton method. Using a gradient obtained by adjoint mode and a Hessian internally approximated by SciPy, the optimization required 77 passive and 26 active evaluations of the flow field to converge to the prescribed tolerance bounds.

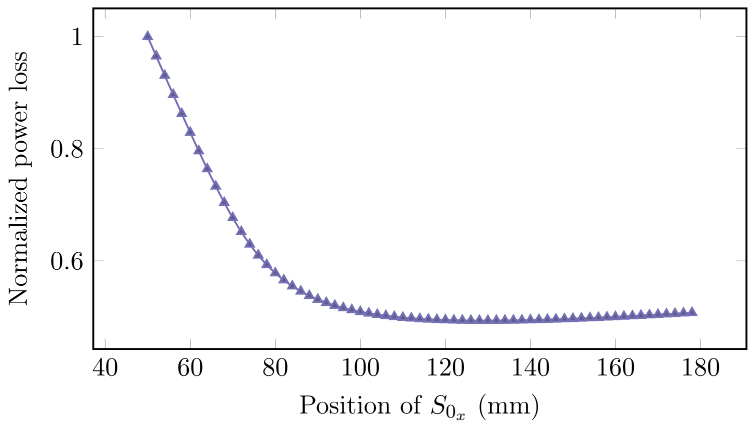


Figure 4.43: Influence of the ramp angle on the power loss. The optimum is located in a broad shallow valley; To find the global minimum, depending on the starting position, a global optimization might be needed.

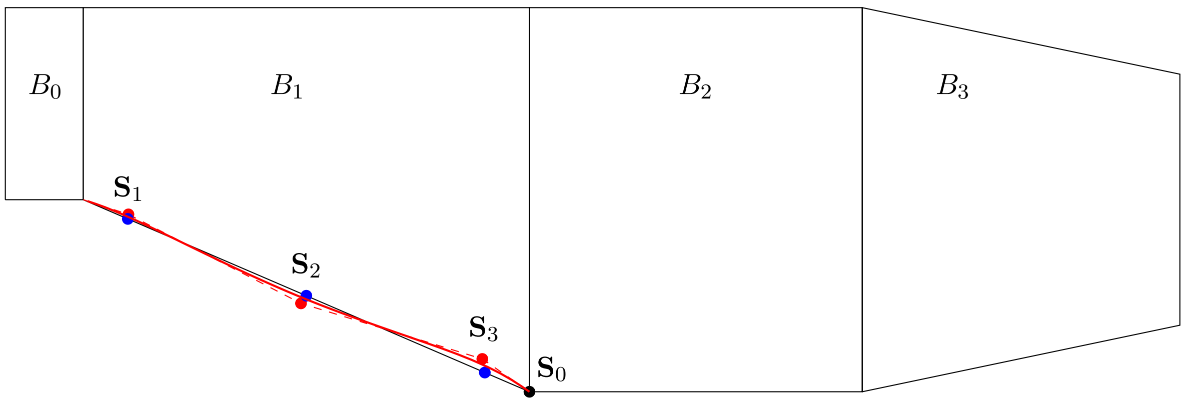


Figure 4.44: Parametrized backward facing step model. Ramp is parametrized by B-Spline with three control points and seven DOF. Design baseline in blue, optimized geometry in red. For better visibility, the y-axis is scaled by a factor of two.

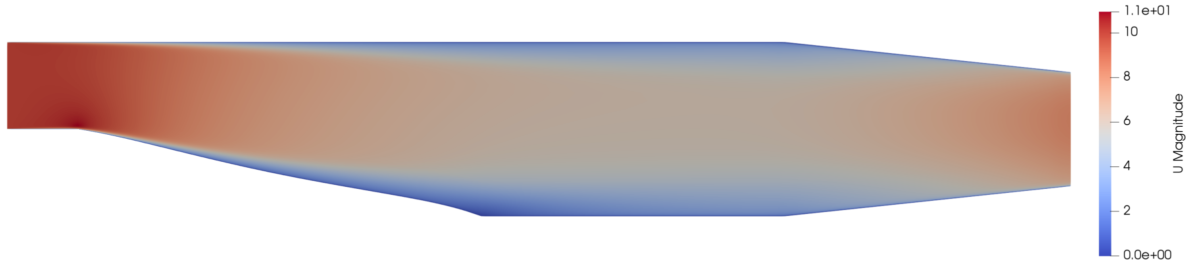


Figure 4.45: Flow inside the optimized Pitz-Daily geometry.

The initial and final geometry are shown in Figure 4.44. Changing from the initial design to the final geometry only changes the geometry slightly, but still improves the power loss by about 6.5%. The flow through the optimized geometry is shown in Figure 4.45. The solution closely resembles solutions obtained by topology optimization.

While the optimization produces reasonable results, it tends to converge into local minima. This is evident from the fact that the optimization does not deviate much from the starting values of the splines in x-direction, regardless on how they are chosen. It is thus advisable to try several initial configurations or to employ a global optimization strategy, e.g. basin hopping.

5 Extended Case Studies

In this chapter two different case studies are presented, highlighting the different approaches to optimization, and the flexibility the discrete adjoint framework allows. The former case is a topology optimization of a finely resolved 3D geometry, showcasing the parallel scalability of the adjoint solvers. The latter case is a shape sensitivity analysis of a 2D airfoil at a high Reynolds number, showcasing the influence of turbulence and the stability of the discrete adjoint approach. The chapter is closed with a brief overview over other applications considered with the discrete adjoint OpenFOAM framework.

5.1 Topology Optimization of 3D Pitz-Daily Case

In this section a three dimensional version of the Pitz-Daily case is topology optimized. Particular emphasis is placed on the scaling behavior of the computation on multiple MPI nodes.

5.1.1 Case Configuration

The test case is derived from the 2D Pitz-Daily case, introduced in Section 3.1.2, by extruding it by 0.25 m in the z-direction. This case is also used by [AU16] to access the scaling of the standard OpenFOAM version on the *Hazel Hen*[1] cluster of HLRS Stuttgart. In its coarsest configuration, the test case is meshed with a 122 250 cell *blockMesh* (the 12 225 cells of the 2D case multiplied by 10 cells in the z-direction). It can then be refined by uniformly increasing the number of cells along each dimension of the geometry. Due to the nature of the blocks for this particular test case, the refinement has to be carried out with an integer factor. The number of cells for refinement levels 1 to 5 are listed in Table 5.1.

Table 5.1: Number of cells for different refinement levels of the Pitz-Daily 3D test case

Refinement Level	Cells
1	122 250
2	978 000
3	3 300 750
4	7 824 000
5	15 281 250

The Reynolds number is set to $Re = 25\,000$ and the k-ϵ turbulence model is used to obtain a non-transient solution. The cost function is implemented as the (total) power loss between inlet and outlet, as defined in Section 3.1.3.

[1]`https://www.hlrs.de/de/systems/cray-xc40-hazel-hen/`

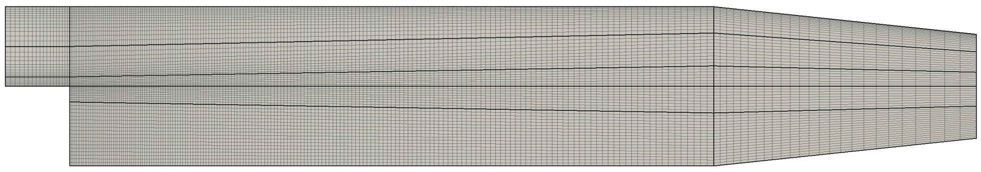


Figure 5.1: Geometry of the Pitz-Daily 3D test case with blocks (bold black lines) and cells (gray lines) for refinement level one. Inflow on the upper left, outflow on the right. The geometry is extruded 0.25 m in z-direction.

5.1.2 First Simulations

To assess the convergence speed of the primal and derivatives, we first run a tangent simulation on the base configuration. The initial flow field is set to a solution of the potential flow equation (`potentialFoam`). We then run a tangent solver for 400 iterations. By seeding the tangents of all parameters $\dot{\alpha}_i$ to one, we obtain the sum of all partial derivatives

$$\dot{\mathcal{J}} = \nabla\mathcal{J}(\boldsymbol{\alpha}) \cdot \dot{\boldsymbol{\alpha}} = \sum_{i=0}^{n-1} \frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\alpha_i} ,$$

with one evaluation of the tangent augmented simulation code. The convergence history of the sensitivity $\dot{\mathcal{J}}$ is depicted in Figure 5.2. We see no major changes in both the primals and the sum of tangents after 1 500 iterations. The sensitivities do not lag behind the primals significantly. However, the momentum equation lags behind the pressure correction by a few hundred iterations, hinting that the momentum equations might be too strongly relaxed.

After having determined the approximate convergence rate with the tangent simulation, we now switch to the adjoint mode. The flow is initialized to a mostly converged state, obtained by 2 500 passive iterations of the `simpleFoam` solver. We run the adjoint simulation with a uniform starting field $\boldsymbol{\alpha}$, initialized to zero. First we run for 400 iterations, to check if the adjoints match the values predicted by the tangent simulations. Figure 5.2 shows that the sensitivities obtained by piggybacking match the value obtained by tangent mode. Further the sensitivity obtained with reverse accumulation by repeatedly adjoining the 400th iteration step match the results obtained by piggy-backing and tangent mode.

5.1.3 Run Time and Optimization Results

For the optimization, we utilize the piggyback algorithm (introduced in Section 4.2), which is run until both the primals and adjoints are sufficiently converged.

Table 5.2 lists the run time factors and memory usage of the piggyback simulation. Shown is the average run time of a single iteration step of the `simpleFoam` and `piggySimpleFoam` solvers. By introducing the `dco/c++` data type into `simpleFoam`, without executing an augmentation of the forward section, the run time increases by a factor of over two and the memory consumption by a factor of 1.6.

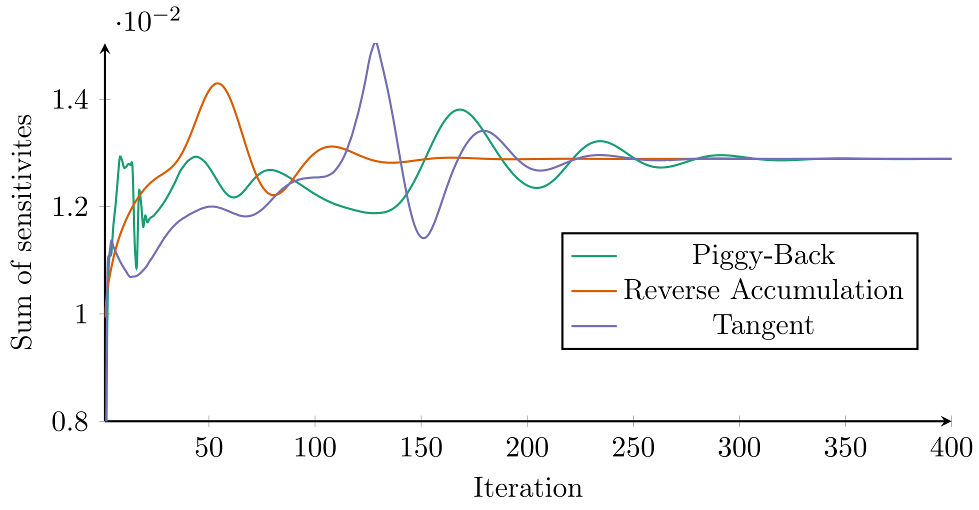


Figure 5.2: Sensitivities obtained by tangent and adjoint modes, with piggybacking from the initial state, and reverse accumulation by repeatedly adjoining time step 400.

Table 5.2: Global run time and memory, including factors, for the Pitz-Daily 3D case.

	Run time (s)	Factor	Memory (MB)	Factor
`simpleFoam` passive	26.92	1.00	239.14	1.00
`simpleFoam` active	75.08	2.79	381.93	1.60
`piggySimpleFoam` w. SDLS	218.64	8.12	5979.15	25.00
`piggySimpleFoam` w.o. SDLS	463.48	17.22	38474.52	160.89

Table 5.3: Run time and run time factors (compared to `simpleFoam`) of the individual solver components. The run times and run time factors for the individual linear equation system solvers are given for the augmented forward and reverse interpretation (only for SDLS) execution.

	`simpleFoam`	`simpleFoam` active		Piggyback w. SDLS		Piggyback w.o. SDLS	
	Run time (s)	Run time (s)	Factor	Run time (s)	Factor	Run time (s)	Factor
Total	26.91	75.81	2.82	213.00	7.92	463.48	17.22
Augm. forward	25.80	73.80	2.86	134.69	5.22	296.26	11.48
U	6.39	15.43	2.41	16.78	2.62	62.69	9.80
p	7.74	19.36	2.50	20.98	2.71	108.65	14.04
k	2.14	5.10	2.38	5.14	2.40	21.18	9.88
epsilon	1.68	4.01	2.38	4.65	2.76	16.09	9.57
Seed	—	—	—	3.47	—	15.42	—
Interpretation	—	—	—	66.39	—	143.32	—
U reverse	—	—	—	16.86	—	—	—
p reverse	—	—	—	14.96	—	—	—
k reverse	—	—	—	5.01	—	—	—
epsilon reverse	—	—	—	4.52	—	—	—

Enabling the augmented forward and reverse interpretation for the `piggySimpleFoam` solver increases the run time factor to a factor of approximately eight, and the memory factor to 25. This is still considerably better than with black-box differentiated linear solvers, which consumes double the run time and over six times more memory.

The run times are broken down to the individual solver phases in Table 5.3. The solver run is broken down into the *forward phase*, *seeding phase* and *interpretation phase*. The forward phase is further broken down into the individual linear solver calls (velocity, pressure and turbulence (k-ϵ)). The seeding phase is dominated by the first allocation and initialization of the adjoint vector, and therefore profits from SDLS, due to reduction in size of the adjoint vector. For the SDLS case, the interpret phase also breaks down the individual linear solver calls, executed from the adjoint callback objects. As with the primal, the solution time of the adjoint equation systems are dominated by the velocity and pressure systems, the turbulence equations are comparatively cheap to solve.

When both the primal and adjoint have converged, we update the parameters $\boldsymbol{\alpha}$ corresponding to the steepest descent algorithm. The values for $\boldsymbol{\alpha}$ are clamped below zero to eliminate non-physical momentum sources and capped at a maximum value to obtain a solution which is upwards bounded.

Figure 5.5 shows the convergence history of the cost function $\mathcal{J}(\boldsymbol{\alpha})$ over 2000 steepest-descent iterations. The average total pressure drop between inlet and outlet drops from $18.74\,\mathrm{m}^2\,\mathrm{s}^{-2}$ in the baseline version to $11.49\,\mathrm{m}^2\,\mathrm{s}^{-2}$ in the optimized version.

The top and bottom part of Figure 5.3 show the velocity field and total pressure contours for the baseline and optimized design respectively, while Figure 5.4 shows the distribution of the penalization parameter α for the optimized case. The gap in the penalization field in the area after the step is an artifact visible at different mesh refinement levels and presumably tries to shift some flow from the centerline of the duct more towards the near- and far-field of the geometry.

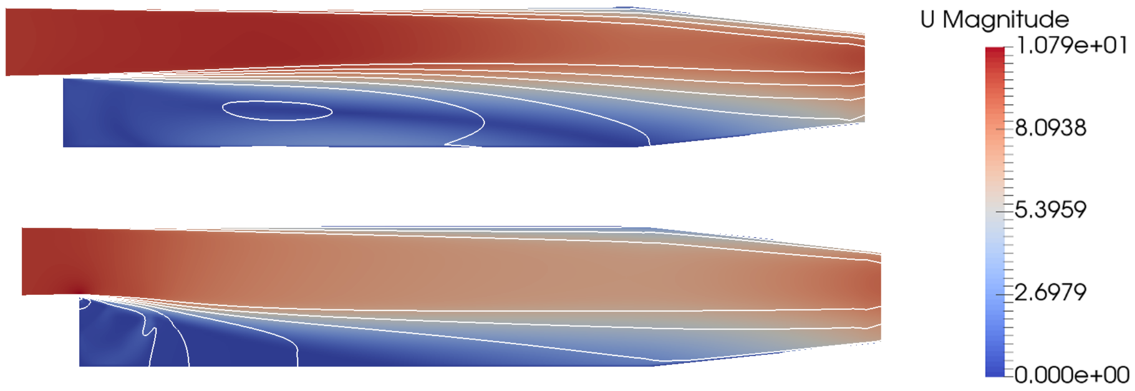


Figure 5.3: Velocity plot on the z-midplane of the initial (top) and optimized (bottom) configuration. White contour lines show levels of total pressure.

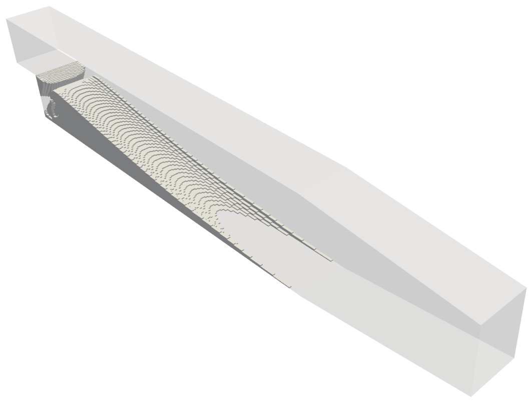


Figure 5.4: Geometry with penalized regions where $\alpha > 0$.

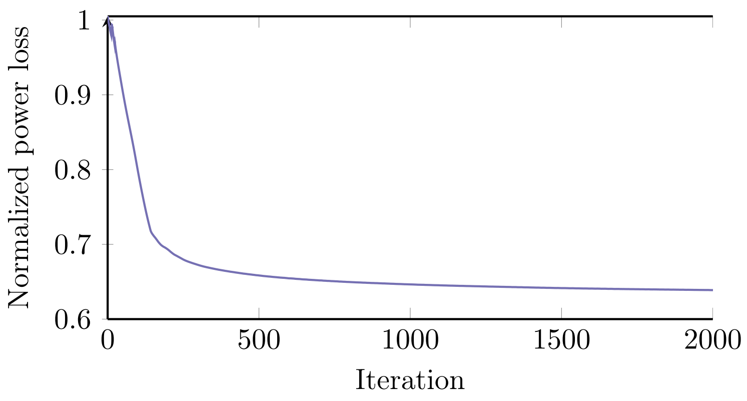


Figure 5.5: Convergence of the cost function $\mathcal{J}$ for the optimization of the 3D Pitz Daily test case, improving the predicted power loss by 38%.

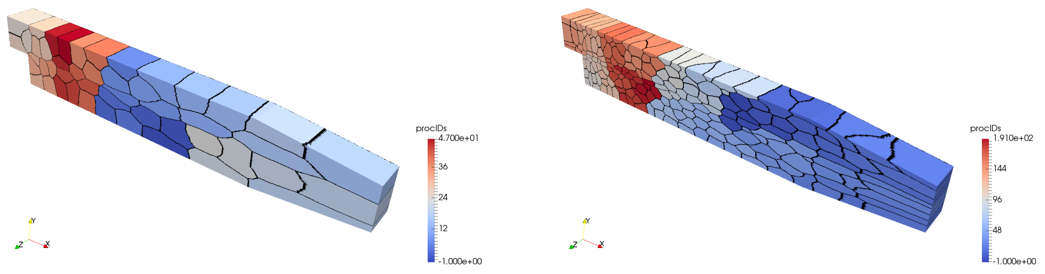


Figure 5.6: Mesh decomposition for 48 and 192 processors using scotch decomposition. Processor boundaries are shown in black, remaining mesh colored by processor id.

Table 5.4: Relevant Hardware of the RWTH Aachen Compute Cluster.

Section	CLX-MPI	CLX-SMP
LSF Node Type	c24m128	c144m1024
#Nodes	600	6
#Sockets per Node	2	8
CPU Codename	Intel Broadwell EP	Intel Broadwell EX
CPU Model	E5-2650v4	E7-8860v4
Clock Speed (GHz)	2.2	2.2
#Cores per Chip	12	18
#Cores per Node	24	144
Memory per Node (GB)	128	1024
Memory per Core (GB)	5.33	7.11

5.1.4 Scaling Behavior on HPC Cluster

Having established the feasibility of this test case for topology optimization we now use this case for benchmarking the scaling behavior of the implementation. The case is decomposed onto multiple processors using the `ptScotch` decomposition algorithm [CP08]. The decomposition for 48 and 192 processors is depicted in Figure 5.6.

In Figure 5.8 we show the memory consumption and run time results for refinement level three. We again observe a major reduction of tape memory when utilizing the symbolically differentiated linear solvers (3088 GB down to 253 GB). For not well conditioned problems, which require more solver iterations, the memory improvements are even higher, as the memory consumption without symbolically differentiated solvers is directly dependent on the number of solver iterations. Also one rogue iteration, that needs more linear iterations than average to complete, may kill the whole simulation due to lacking RAM space.

With SDLS we also generally see a reduction in run time, due to improved linear solver efficiency (calculation in passive mode), less memory allocation, less adjoint propagation, and less adjoint communication, which outweighs the need to solve additional linear equation systems during the adjoint propagation. In our case, the adjoint propagation phase is slightly slower due to the additional equation systems which need to be solved. This is offset by the more efficient augmented primal section, leading to an overall reduction in run time of roughly 40%.

Both the run time and the memory consumption are dominated by the solution of the pressure equation, for which the geometric-algebraic multi-grid solver (GAMG) is used. The remaining equations are solved with a Gauss-Seidel solver variant, which is often used in near transient cases due to its stability.

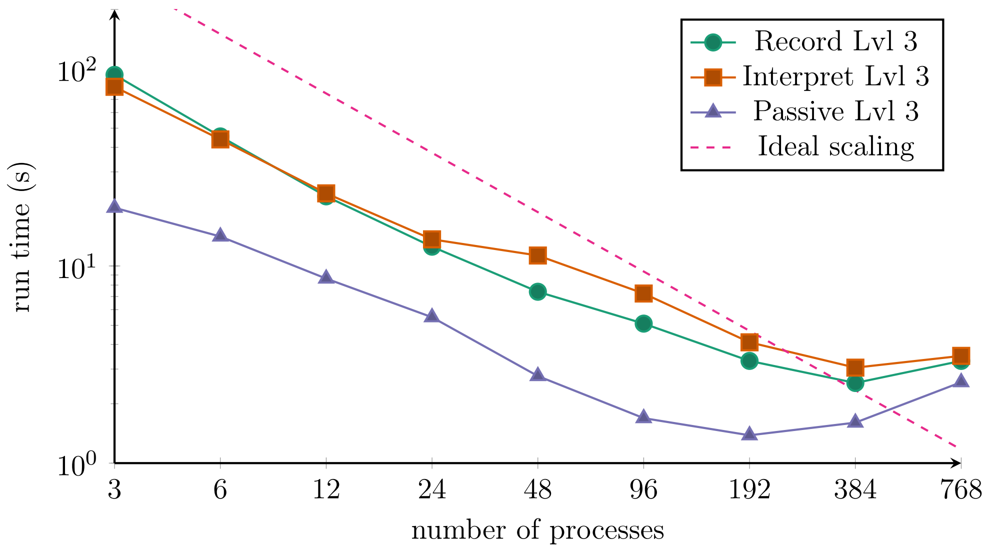
In Figure 5.7 we show the scaling behavior of our implementation on the RWTH University compute cluster. Benchmarked are refinement levels three and four of the test case. Table 5.4 lists the two systems types most relevant to the application of AD. They both provide a relatively high amount of RAM per core with 5.33 GB and 7.11 GB respectively. With currently 600 installed nodes, the former system is the most common node of the RWTH cluster. It supplies 24 cores per node and 128 GB of RAM, making it a good general purpose choice. The latter system provides 1024 GB of RAM per node, making it the preferred choice if a very high amount of RAM is needed locally. However, many of the 144 cores need to remain idle to utilize the full memory for single threads, which is punished by the job queueing system.

The simulation was run on the former machines supplying 128 GB each. As the finer simulation (level 4) consumes about 650 GB RAM (see Figure 5.8), for cases decomposed onto few processors ($n = \{12, 24, 48, 96\}$), we cannot use all physical cores of the nodes, as not enough RAM is available locally. For those cases, we only place 3, 6 and 12 threads respectively per node, leaving the remaining cores unutilized. The total available memory bandwidth is thus shared by less threads for these cases, at the expense of more communication between distant nodes (connected by InfiniBand).

For $n = \{192, 384, 768\}$, we can fully saturate the nodes with 24 threads each. For the benchmark, we time 20 piggyback iterations and calculate the average time needed for both the augmented primal and adjoint propagation phases. In the figure we see scaling of both phases. For reference, also the average run time for passive calculation with `simpleFoam` is shown. For the most part, the scaling behavior of the discrete adjoint and passive version are comparable. The passive calculation stops scaling earlier than the discrete adjoint, because fewer operations need to be performed during each iteration for the passive solver. The run time of the passive calculation is thus dominated by the communication overhead earlier.

For the coarser case of refinement level three, scaling stops after 192 threads for the passive computation and after 384 threads for the discrete adjoint. At this point no further scaling is to be expected, as the number of cells per thread has already fallen below 10 000. For the finer case, scaling stops for the passive computation after 192 threads, but continues onto 768 threads for the discrete adjoint. At this stage each individual thread only holds around 12 000 cells. We thus expect the scaling of the discrete adjoint to stop beyond that point for this case as well, as the numeric work load for each process becomes too small.

Scaling of (primal) OpenFOAM has been shown to extend into thousands of processors [Dur+15]. For even higher processor numbers, some design decisions limit the scalability [Cul11; AU16].



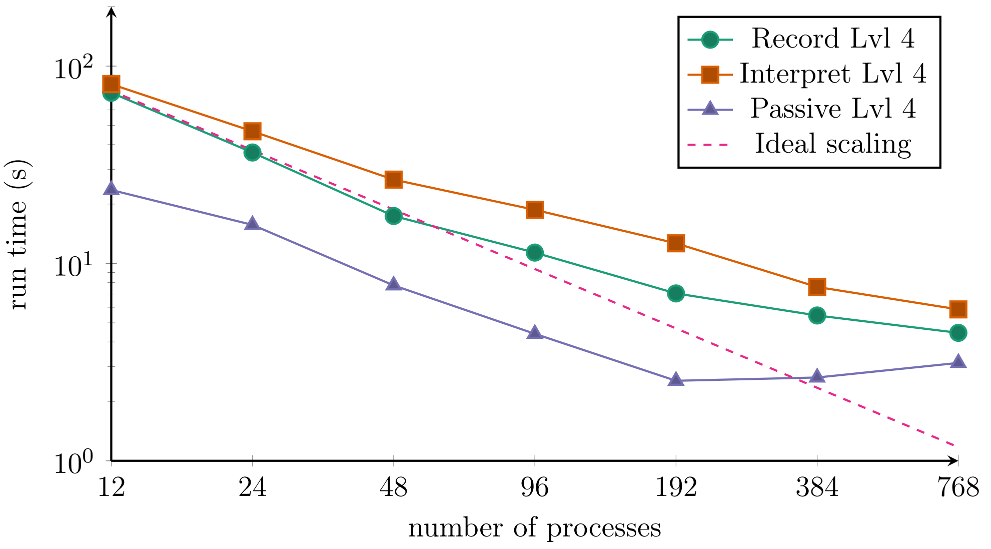


Figure 5.7: Run time scaling of the discrete adjoint on the RWTH cluster. The average run time out of twenty piggyback steps is shown for the recording and interpretation phases. For reference, also the scaling behavior of the passive `simpleFoam` solver and the theoretical ideal scaling is shown.

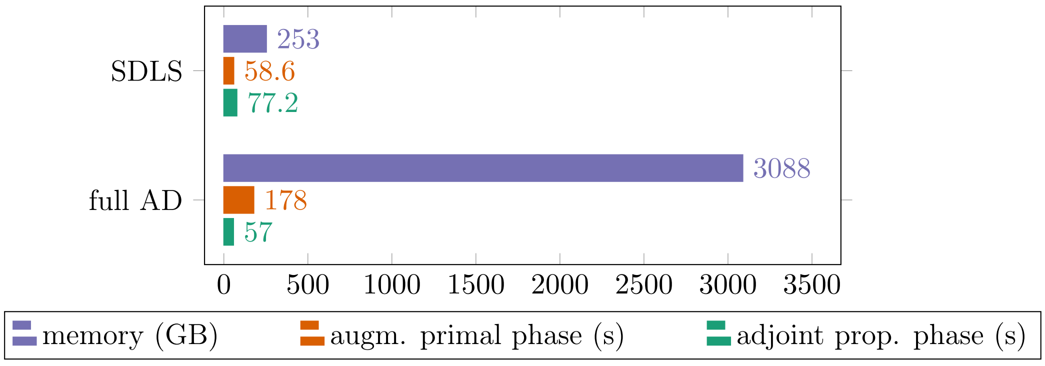


Figure 5.8: Memory consumption (over all processes) and run time of the Pitz-Daily example for 768 processors with (top) and without (bottom) symbolically differentiated linear solvers.

5.2 Shape Sensitivities of NACA Airfoil

In this section we apply the procedures introduced in Section 4.4 to generate the surface sensitivities of a NACA airfoil. The flow is calculated using the Spalart-Allmaras turbulence model, the sensitivities of the airfoil are evaluated w.r.t. viscous lift and drag.

5.2.1 Modelling of NACA Airfoils

The concept of NACA airfoils was introduced by the National Advisory Committee for Aeronautics (NACA) in the early 20th century, to efficiently describe different airfoil shapes. The most commonly used parametrization is the 4-digit parametrization, e.g. `NACA 4412`, which parametrizes an asymmetric 2D airfoil. A special case is the generation of symmetric airfoil profiles, where the distance from the camber line is equal for both the upper and lower surface. Those profiles are described by only one parameter t, encoded with two digits, e.g. `NACA 0012`. Due to the symmetric pressure profile at zero angle of attack (that is the angle between spanwise direction of the airfoil and the direction of the freestream flow), symmetric profiles produce no lift.

For a symmetric airfoil, the distance from the camber line is defined as

$$y_t = 5t\left(0.2969x^{\frac{1}{2}} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1036x^4\right),$$

giving the coordinates of the upper and lower surfaces as $x_L = x$, $x_U = x$, $y_L = -y_t(x)$ and $y_U = +y_t(x)$.

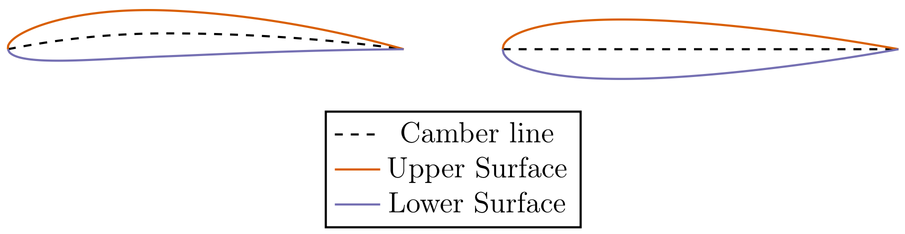


Figure 5.9: Cross section of asymmetric `NACA 4412` airfoil (left) and symmetric `NACA 0012` (right) with zero angle of attack.

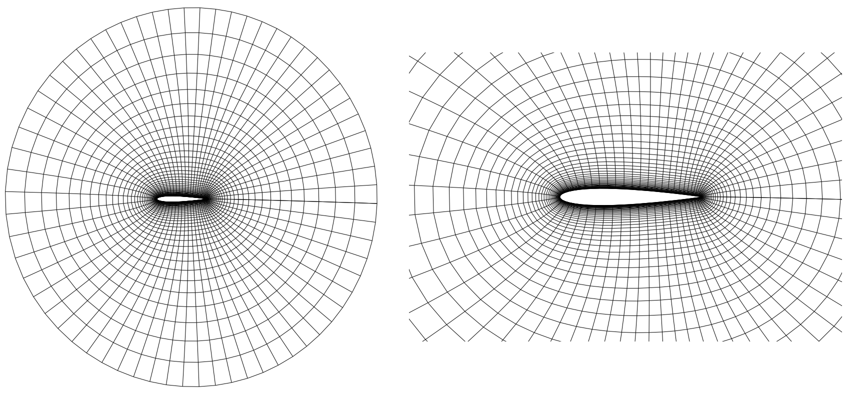


Figure 5.10: Illustration of the O-type meshing of the airfoil. Zoom on the boundary layers on the right. For the actual mesh, the radius of the bounding circle is considerably bigger and the boundary layers are finer.

The wing cross section and the camber line for the asymmetric `NACA 4412` and symmetric `NACA 0012` wings are shown in Figure 5.9. In the following, we will further investigate the `NACA 0012` airfoil. A symmetric airfoil is useful to verify the adjoint implementation, as one expects a symmetric sensitivity field at zero angle of attack.

An O-type mesh [TWM85] is generated for a circular domain around the airfoil, with a radius of thirty times the wing length. The spacing of the boundary layers yields a $y+$ value of approximately 0.1 on average and a maximum $y+$ value of 0.25 at the chosen Reynolds number of $\mathrm{Re} = 2 \cdot 10^6$. The angle of attack is varied by changing the direction of the incoming flow. An illustration of the mesh layout is shown in Figure 5.10. In order to make the mesh features clearly visible, the radius of the circular domain is lowered and the mesh resolution is reduced in the figure. The resulting mesh consists of approximately 80 000 cells, most of which are needed to form suitable boundary layers and to refine the mesh regions in the wake after the trailing edge.

As cost function the lift and drag of the wing are considered. The lift force on a wing is defined as the aerodynamic force of the fluid exerted onto the wing, perpendicular to the freestream flow. Analogously the drag force on a wing is defined as the aerodynamic force parallel to the freestream flow.

For low velocities, the aerodynamic force is dominated by the pressure forces, which act normal to the skin surface. The total pressure force acting on the wing can be calculated by integrating along the airfoil surface Γ:

$$\mathbf{F}_p = \oint_\Gamma p(\mathbf{w})\; \mathbf{n}\, \mathrm{d}\mathbf{w}\,.$$

For higher velocities, the contribution of viscous effects, that is forces caused by the shear stresses between the fluid layers, to the aerodynamic forces become non-negligible. This part of the forces is called friction or shear forces $\mathbf{F}_s$, which act tangentially to the skin surface:

$$\mathbf{F}_s = \oint_\Gamma s(\mathbf{w})\; \mathbf{t}\, \mathrm{d}\mathbf{w}\,,$$

where s is the friction force at each location on the airfoil. The calculation formula for the friction force depends on the chosen turbulence model, but can in general be obtained from the

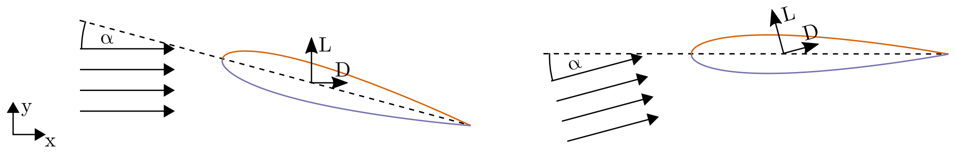


Figure 5.11: Airfoil with $\alpha = 15\,\text{deg}$ angle of attack. For the left airfoil, the freestream is aligned with the x-axis, for the right airfoil the camber line is aligned with the x-axis. As a result the lift and drag vectors are tilted to the coordinate system for the right airfoil.

implementation of the viscous stress tensor. The total (viscous) aerodynamic force $\mathbf{F}_v = \mathbf{F}_p + \mathbf{F}_s$ is then the sum of pressure and friction force.

For an angle of attack α, the lift induced by the aerodynamic forces is defined as the pressure force projected in the direction perpendicular to the freestream facing upwards:

$$\mathbf{L}_v = \mathbf{F}_v \cdot \begin{pmatrix} -\sin(\alpha) \\ \cos(\alpha) \\ 0 \end{pmatrix} ,$$

and the drag as the force parallel and opposed to the freestream:

$$\mathbf{D}_v = \mathbf{F}_v \cdot \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \\ 0 \end{pmatrix} .$$

A graphical representation of the drag and lift vectors is given in Figure 5.11.

5.2.2 Primal and Sensitivity Results

The pressure distribution along the airfoil surface for three different angles of attack is shown in Figure 5.12. The zero surface integral between the pressure on the upper and lower surface of the airfoil, for zero angle of attack, shows that no lift is produced by the airfoil in this configuration.

The sensitivities have been found to strongly depend on a finely resolved boundary layer ($y^+ \leq 1$), even more so than the primal pressure distribution. Also the sensitivity fields for drag differ fundamentally between viscous and non-viscous formulation of the cost function. This is to be expected, because at the chosen Reynolds numbers the viscous drag contributes a considerable amount to the total drag. The lift is less influenced by the viscous effects.

To obtain a simulation state which converges with reverse accumulation, i.e. a state where the adjoint iteration is contractive, the relaxation factors for the primal equations had to be slightly lowered. While the primal converges smoothly for under relaxation factors of 0.9 for the momentum equations (using the consistent SIMPLEC scheme) and 0.7 for the Spalart-Allmaras turbulence equations, convergence of the adjoint could only be obtained after lowering the under relaxation factors for the primal equations to 0.8 and 0.6 respectively. That is, the under-relaxed field (here for the pressure) $\mathbf{p}_u^{i+1}$ is calculated with under relaxiation factor λ as

$$\mathbf{p}_u^{i+1} = \lambda \mathbf{p}^{i+1} + (1-\lambda)\mathbf{p}^i .$$

The sensitivity results for the drag w.r.t. movement of the surface nodes in surface normal direction are shown in Figure 5.14. Negative sensitivities indicate the desire to move the surface

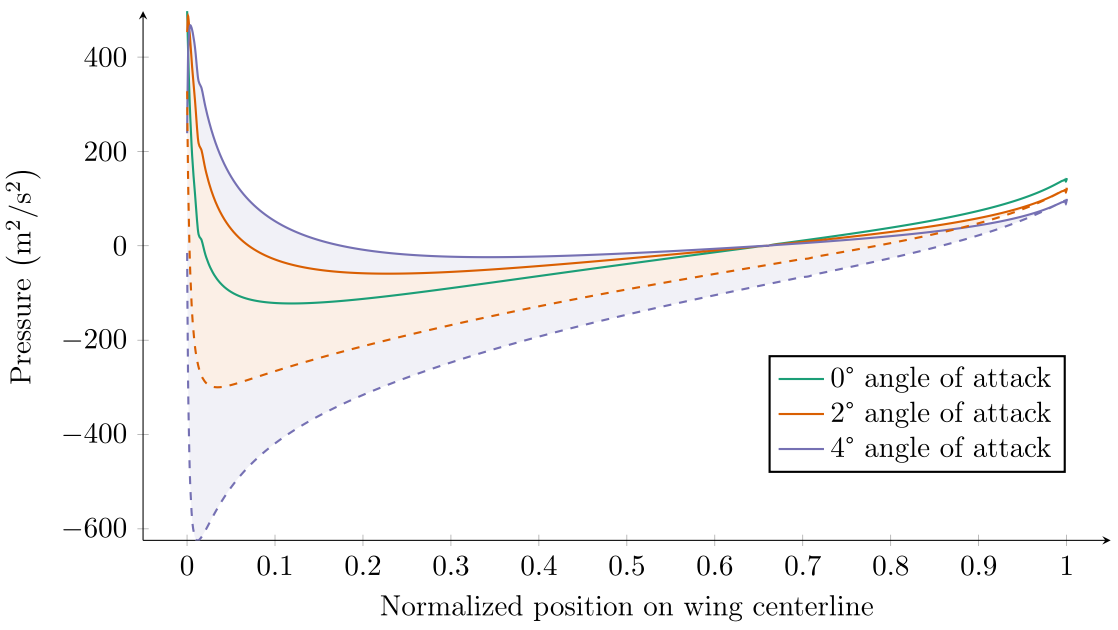


Figure 5.12: Pressure distribution along the surface of the `NACA 0012` wing for zero, two and four degree angle of attack. Pressure on lower surface solid and dashed on upper surface. The area enclosed by the lower and upper pressure curve corresponds to the total lift of the wing (excluding the viscous forces).

nodes against the airfoil surface normal direction, lowering the cross section of the airfoil. As expected, the sensitivities for zero angle of attack are symmetric along the centerline of the airfoil. For two and four degree angle of attack, the sensitivities are asymmetric, with the lower chord of the airfoil exhibiting higher sensitivity values. Overall the sensitivities are visibly correlated to the pressure distribution. The sensitivities are smooth along the airfoil surface, except for a singularity point limited to the trailing edge of the geometry.

Using the Spalart-Allmaras turbulence model, an issue with the calculation of the wall distance y^+, needed for the calculation of the turbulence equations, was identified. If this calculation is left unchanged, it introduces considerable noise into the adjoints, especially near the leading edge. If the calculation of the wall distance is treated with a frozen adjoint assumption, the adjoints become very smooth, while still retaining the same overall shape. Both the uncorrected and corrected sensitivities for a coarse mesh with two degree angle of attack can be seen in Figure 5.13. The fix almost completely removes the instabilities observed around the trailing edge of the airfoil, only leaving a singularity around the trailing edge. To make sure the correction does not completely remove the adjoint sensitivities of the turbulence model from the calculation, the same configuration is run with a frozen turbulence assumption. The results of this calculation are also shown in Figure 5.13. This reveals that the sensitivities of the airfoil obtained with frozen turbulence, while still indicating that the cross section of the airfoil needs to be decreased to improve drag, exhibit a completely different behavior. In particular the frozen adjoint solution (even with the previous fix to wall distance calculation applied) exhibits issues at the leading edge, which do not appear with the fully differentiated Spalart-Allmaras turbulence model. To summarize, calculating the wall distance with a frozen adjoint calculation only changes the overall

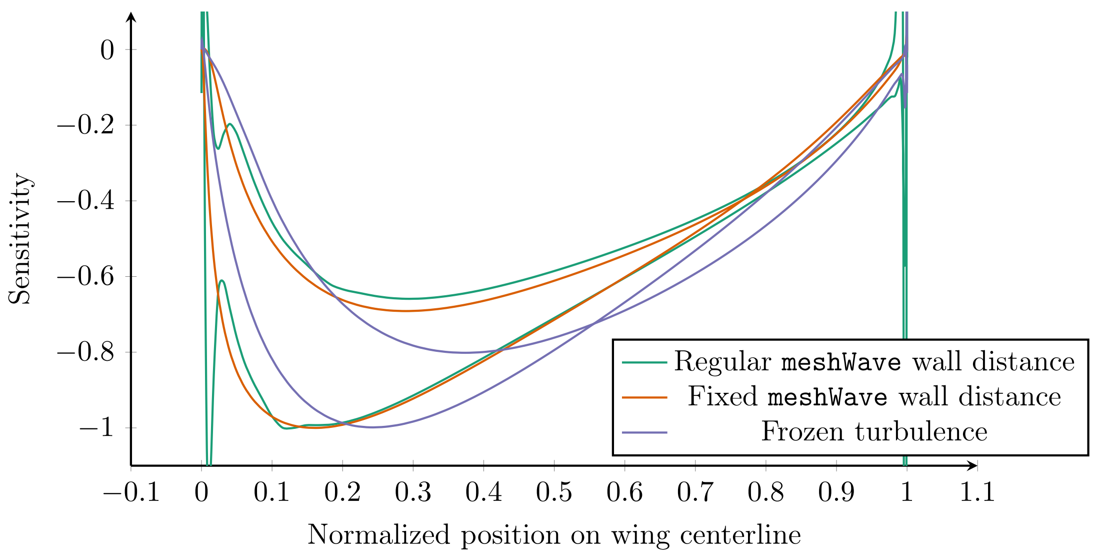


Figure 5.13: Sensitivity of the drag w.r.t. movement of surface nodes of the `NACA 0012` airfoil in surface normal direction. The noisy green curve is calculated without the modified adjoint wall distance, the smooth orange curve with the modification.

perception of the adjoints a little, while considerably improving smoothness of the adjoints. In contrast, introducing a completely frozen turbulence assumption changes the overall appearance of the adjoint sensitivities and exhibits additional issues.

The results for the lift on the same case are presented in Figure 5.15. For zero angle of attack, the results are symmetrical around the x-axis, indicating that the top surface needs to be moved outwards to increase lift, while the lower surface needs to be pushed inwards. This is consistent to the shape of an asymmetric NACA wing. For increasing angle of attack, the sensitivity distribution shifts in positive direction. Again the sensitivities exhibit issues at the trailing edge and are smooth otherwise.

The sensitivities can be subsequently used as an input to a mesh morpher, e.g. the one introduced in Section 4.4, with the aim to obtain an improved geometry.

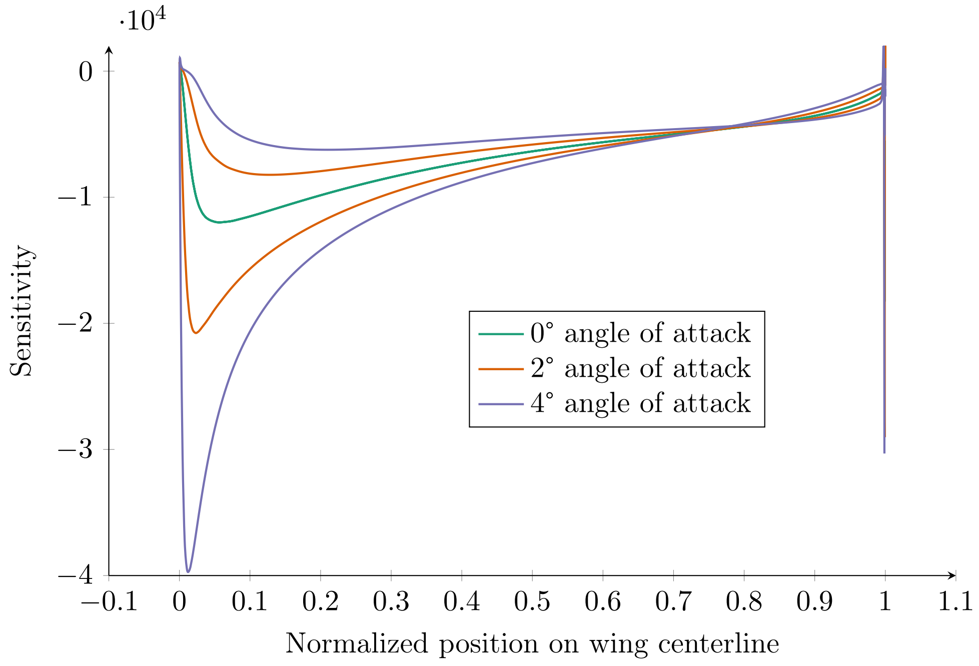


Figure 5.14: Sensitivities of drag w.r.t. movement of surface nodes of the `NACA 0012` airfoil in surface normal direction.

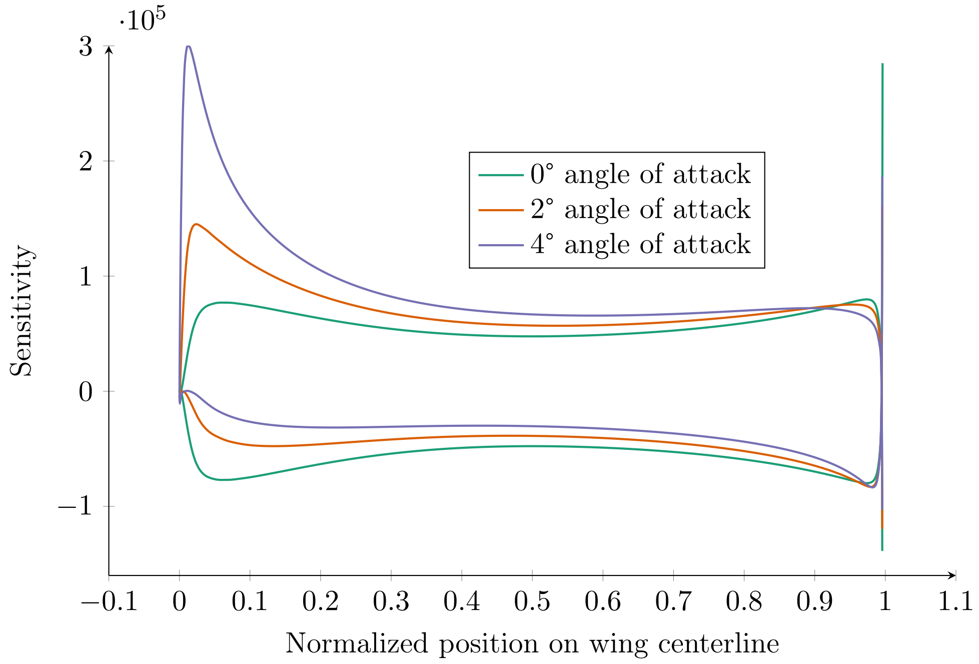


Figure 5.15: Sensitivities of lift w.r.t. movement of surface nodes of the `NACA 0012` airfoil in surface normal direction.

5.3 Further Applications of Discrete Adjoint OpenFOAM

Discrete adjoint OpenFOAM has been applied to a variety of different applications by other researchers, or by students writing their thesis at STCE.

The tangent and adjoint model has been used to differentiate through the OpenCascade CAD environment, also including the whole OpenFOAM mesh generation procedure, using the `snappyHexMesh` mesher. The solution process is differentiated using the `adjointSimpleFoam` solver. This allows to optimize with parameters defined directly in the CAD environment, by using the spline control points of the intermediate NURBS representation as parameters [Gez16].

A parametric optimization study has been performed, coupling the sensitivities obtained by shape adjoints on a wing shaped rudder, with the proprietary parametric optimization framework CAESES by Friendship Systems AG. Studied were the thickness and twist of the wing in a flow with low angle of attack [FT16].

Some of the concepts presented in this thesis have been applied to the Foam-extend project, allowing to run an even wider range of solvers, as well as advanced discretization methods, such as explicitly coupled block solvers [STN]. Furthermore, discrete adjoint OpenFOAM has been used to obtain transient adjoints in the context of the aboutFLOW project [EU16]. Higher order approaches and the accumulation of full Hessians have been explored in [Pee16].

The discrete shape optimization approach, using the Spalart-Allmaras turbulence model, was used to obtain sensitivities and improve the drag of the student competition solar car *Sonnenwagen* [Mol18]. A shape sensitivity, with respect to the drag of the car in a wind tunnel configuration, is shown in Figure 5.16, demonstrating the applicability to complex geometries. Similarly, the sensitivities of a flow around the rear wing of a touring car has been studied in [Pes16].

Discrete adjoint OpenFOAM is available as open-source under the GNU GPLv3[2].

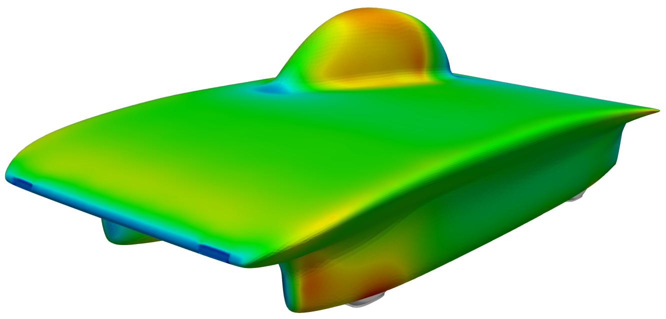


Figure 5.16: Shape sensitivities w.r.t. drag on *Sonnenwagen* solar competition car geometry [Mol18]. Red cells need to be moved outwards, blue cells inwards.

[2]stce.rwth-aachen.de/foam

6 Summary & Outlook

6.1 Summary

Starting from a black-box application of AD, introduced as a proof of applicability, different strategies were explored to lower the runtime and memory impact of the adjoint mode of AD. For steady state cases, the fixed point properties of the solution algorithms have been extensively exploited, using reverse accumulation, the piggyback method, or the discrete adjoint residual approach.

For the residual approach, different coloring techniques have been implemented to speed up the calculation of the Jacobians using either tangent or adjoint mode, exploiting the sparse nature of FVM matrices. For transient cases, or if the convergence of the steady primal case is not sufficient to reliably obtain adjoints using the aforementioned methods, the whole iteration history can be adjoined with low memory overhead, while still retaining acceptable run time behavior, by using binomial checkpointing. The differentiated versions of all solution methods, which rely on embedded linear solvers, profit immensely from the optimizations implemented with SDLS.

Per time step, piggyback run time factors under ten can be achieved, when compared to a passive primal execution. The memory consumption of the adjoint also generally is increased by a factor of around ten. Using reverse accumulation, the run time factor between adjoint and primal is lower, as no augmented primal needs to be calculated for each reverse accumulation iteration.

During the implementation of the adjoint framework, an emphasis was to retain the parallel scalability of the primal and adjoint code. In particular this involved the introduction of AMPI to the discrete adjoint CFD framework. In addition to providing improved run time behavior, the decomposition of cases onto multiple processing nodes spreads the memory demand onto multiple machines. This both improves the memory throughput and also removes the demand for machines with unusually high amounts of available RAM. Using these capabilities, cases on complex meshes with over 7 million cells could be calculated. If, even with all applied optimizations, the RAM demand remains higher than what is available in hardware, the `dco/c++` tape can be offloaded onto secondary storage (preferably low latency, high throughput, such as SSD storage). Due to its random access nature, the adjoint vector is retained in RAM. In order to remove the bottleneck of the adjoint vector outgrowing the amount of available RAM, adjoint vector compression techniques have been discussed and implemented. The improved adjoint vector techniques proved to be very effective when adjoining long iteration histories, without majorly impacting the run time of the augmented primal and reverse propagation. Having the discrete adjoint available for a whole simulation environment proves to be very valuable. While arguably not as efficient as an approach tailored specifically to a specific application, it provides much more flexibility.

Another advantage of a tool based AD implementation is the availability of different models of differentiation on the same code base. A feature added to the primal is immediately available in adjoint, tangent, and higher order versions, by just changing some environment variables.

The discrete adjoint framework has been applied to a variety of CFD cases, ranging from ducted flows to external aerodynamics. The availability of fully differentiated turbulence models makes the discrete adjoint method attractive for applications, where the turbulent quantities are believed to have a major impact on the objective. A variety of cost functions, mostly regarding external aerodynamic flows, have been implemented. New and blended cost functions can be readily implemented, due to the flexibility of the discrete adjoint. Penalization approaches can be used to implement basic constraints. Using the on demand compilation features of OpenFOAM, this can potentially even be done on a case by case basis, implementing the cost function as part of the case configuration. In addition to the high dimensional sensitivities, produced by topology and shape optimization, a parametric optimization approach was implemented, demonstrating the full AD treatment of a meshing tool.

6.2 Outlook

The discrete adjoint approaches presented in this thesis can be applied to a variety of different applications, either within the OpenFOAM framework, or in the general CFD field. The application to large scale transient simulations remains challenging, but also has the potential to create results not obtainable by other methods. Furthermore, for transient applications, the continuous adjoint approach faces some of the same challenges as the discrete adjoint, closing the performance gap. With the advances in the file tape and parallel adjoints made in this thesis, the computation of sensitivities for very large and complex geometries becomes feasible. With the developed profiling methods, further avenues of optimization for the AD implementation should be identified and implemented. In addition to the already employed forward activity analysis, additional optimizations can be applied to the tape, improving performance, especially if the tape is evaluated multiple times.

To expand the capabilities of the discrete adjoint framework, the application to multi-physical optimization is promising. The flexibility of the discrete adjoint makes it applicable to a wide variety of simulation approaches and code bases already in existence, without requiring major code rewrites. A planned development is the incorporation of the discrete adjoint into coupled heat transfer (CHT) problems.

Another advanced topic is the incorporation of robust optimization, to obtain designs which are feasible under a variety of operating conditions. The availability of higher order adjoints facilitates the usage of sophisticated robust optimization schemes.

More complex constrained optimization methods should be explored, in order to generate solutions which can be produced cost effectively (design to manufacture), or which adhere to certain design constraints (e.g. fixed amount of volume). The already implemented parametric optimization capabilities could be used to directly couple the simulated geometries to their CAD representations, allowing for a more construction driven optimization approach.

Besides optimization problems, other applications of the adjoint methods are conceivable, and already pursued by other researchers, using the discrete adjoint OpenFOAM framework. For example, the application of adjoint error estimators for adaptive mesh refinement, or the usage of adjoints for uncertainty quantification.

Appendices

A Developer Documentation

A.1 Location of Discrete Adjoint OpenFOAM Solvers

Discrete adjoint OpenFOAM strives to integrate into the native OpenFOAM framework as seamlessly as possible. The general folder layout remains unchanged from the native OpenFOAM implementation. Let `$FOAM_INST_DIR` point to the OpenFOAM base directory, then the solvers unique to discrete adjoint OpenFOAM are located in `$FOAM_INST_DIR/applications/discreteAdjointOpenFOAM`. Adjoint solvers are located in `$FOAM_INST_DIR/applications/disreteAdjointOpenFOAM/adjoint`, tangent solvers in `$FOAM_INST_DIR/applications/disreteAdjointOpenFOAM/tangent`, passive solvers (e.g. for finite differences) in `$FOAM_INST_DIR/applications/disreteAdjointOpenFOAM/passive`, and support libraries (for checkpointing and the calculation of cost functions) in `$FOAM_INST_DIR/applications/disreteAdjointOpenFOAM/libs`. The `dco/c++` header files are located in `$FOAM_INST_DIR/src/OpenFOAM/dco`.

The most relevant subdirectories for the usage and configuration of discrete adjoint OpenFOAM are listed below:

```
. $FOAM_INST_DIR
├── applications
│   ├── discreteAdjointOpenFOAM
│   │   ├── adjoint
│   │   ├── experimental
│   │   ├── fd
│   │   ├── libs
│   │   │   ├── libCostfunction
│   │   │   └── libCheckpointing
│   │   ├── passive
│   │   ├── tangent
│   │   └── tools
│   └── solvers
├── etc
│   ├── bashrc
│   └── derivativeSettings.sh
├── src
│   └── OpenFOAM
│       └── dco
│           └── dco.hpp
├── tutorials
└── wmake
```

Figure A.1: Folder structure of discrete adjoint OpenFOAM, with root at `$FOAM_INST_DIR`. Folders not immediately relevant are omitted for space reasons.

A.2 Compile Options

Discrete adjoint OpenFOAM uses the `wmake` build system of OpenFOAM. The environment of OpenFOAM is set by sourcing the `etc/bashrc` script file. Discrete adjoint OpenFOAM extends the environment variables of OpenFOAM (identifiable by the prefixes `WM_*` and `FOAM_*`) with a set of variables with the prefix `DOF_*`.

The following variables are currently implemented:

`DOF_AD_OPTION`: Choice of AD option to be used.
One of { `A1S` | `T1S` | `T1V` | `T2A1S` | `T2T1S` | `Passive` }, default: `A1S`.

`DOF_COMPILER`: Choice of compiler, gets passed along to `WM_COMPILE_OPTION`.
One of { `Gcc` | `Clang` | `Icc` }, default: `Gcc`.

`DOF_COMPILE_OPTION`: Specify level of optimization.
One of { `Opt` | `Debug` | `Prof` } default: `Opt`.

`DOF_BUILD_PROCS` number of threads on `localhost` for parallel compilation, default: `8`.

The flags for `dco/c++` are set in the environment variable `DOF_DCO_FLAGS`. The `DOF_DCO_FLAGS` are set by the `bashrc` script, depending on the choice of `DOF_AD_OPTION`. The flags are added to the compilation flags and configure `dco/c++`, such that only the needed features are instantiated. The choice of flags is detailed in the following subsections.

A.2.1 Passive Mode

For passive mode, all non-essential features of `dco/c++` are disabled. Neither adjoint nor tangent data types are available. To enable the use of the same code base for all variants, some `dco/c++` functions such as `dco::passive_value()` are still used. However they should be optimized out by the compiler.

```
export DOF_DCO_FLAGS="-DDCO_NO_DEFAULT"
```

A.2.2 First order adjoint mode (A1S)

For adjoint mode, the generic adjoint type (`ga1s`) is instantiated, allowing the definition of `Foam::scalar` as `dco::ga1s<double>::type`. A chunk tape is used, allowing the tape to gradually grow in chunks. Tape callbacks are needed for the symbolic differentiation of linear solvers. Activity analysis is enabled, reducing the tape size for regions not dependant on the registered inputs. For cases which require tape indices bigger than 2^{31}, a 64 bit datatype has to be requested to enumerate the tape entries with `DDCO_TAPE_USE_LONG_INT`.

```
export DOF_DCO_FLAGS="\
  -DDCO_NO_DEFAULT \
  -DOF_DCO_MODE_A1S \
  -DOF_DCO_A1S_CONT_LINEAR \
  -DDCO_GA1S \
  -DDCO_CHUNK_TAPE \
  -DDCO_TAPE_CALLBACKS \
  -DDCO_TAPE_USE_LONG_INT \
  -DDCO_TAPE_ACTIVITY \
  -DDCO_ALLOW_TAPE_SWITCH_OFF"
```

A.2.3 First Order Tangent Scalar Mode (T1S)

For scalar tangent mode, the generic tangent type (`gt1s`) is enabled, allowing the definition of `Foam::scalar` as `dco::gt1s<double>::type`.

```
export DOF_DCO_FLAGS="\
  -DOF_DCO_MODE_T1S \
  -DDCO_NO_DEFAULT \
  -DDCO_GT1S \
  -DDCO_T1S_ACTIVITY"
```

A.2.4 First Order Tangent Vector Mode (T1V)

For the vector tangent mode, the generic tangent type (`gt1v`) is enabled, allowing the definition of `Foam::scalar` as `dco::gt1v<double,d>::type`. The vector size d is set to 5 and is fixed at compile time for performance reasons. If another vector size is required, it can be changed in `DCO_T1V_SIZE`. All object files have to be recompiled in order to apply the changes to the vector size. The optimal vector size is dependant on cache behavior, the amount of available RAM and the number of needed derivatives. A reference to the i-th tangent of a variable `x` can be accessed by using the `dco::derivative(x)[i]` method.

```
export DOF_DCO_FLAGS="\
  -DOF_DCO_MODE_T1V \
  -DDCO_NO_DEFAULT \
  -DDCO_GT1V \
  -DDCO_T1V_ACTIVITY \
  -DDCO_VECTOR_SIZE=5"
```

A.2.5 Second Order Tangent Over Adjoint Mode (T2A1S)

For the second order tangent over adjoint mode, both the generic adjoint and tangent types are instantiated. Those types can be arbitrarily nested to obtain higher order derivative models. The scalar tangent over adjoint mode is obtained by nesting an tangent type inside an adjoint type, yielding `dco::ga1s<dco::gt1s<double>::type>::type`.

```
export DOF_DCO_FLAGS="\
  -DOF_DCO_MODE_T2A1S \
  -DDCO_NO_DEFAULT \
  -DOF_DCO_A1S_CONT_LINEAR \
  -DDCO_GT1S \
  -DDCO_GA1S \
  -DDCO_TAPE_CALLBACKS \
  -DDCO_TAPE_USE_LONG_INT \
  -DDCO_TAPE_ACTIVITY \
  -DDCO_ALLOW_TAPE_SWITCH_OFF"
```

A.2.6 Second Order Tangent Over Tangent Mode (T2T1S)

The (scalar) second order tangent over tangent mode nests a tangent type inside another tangent type, yielding `dco::gt1s<dco::gt1s<double>::type>::type`. The flags are identical to first order tangent scalar mode.

```
export OF_DCO_FLAGS="\
  -DOF_DCO_MODE_T2T1S \
  -DDCO_NO_DEFAULT \
  -DDCO_GT1S \
  -DDCO_T1S_ACTIVITY"
```

B Full SDLS Code

```
template<>
Foam::solverPerformance Foam::fvMatrix<Foam::scalar>::solveSegregated
(
  const dictionary& solverControls
)
{
  GeometricField<scalar, fvPatchField, volMesh>& psi =
    const_cast<GeometricField<scalar, fvPatchField, volMesh>&>(psi_);

  scalarField saveDiag(diag());
  addBoundaryDiag(diag(), 0);
  scalarField totalSource(source_);
  addBoundarySource(totalSource, false);

  ADmode::global_tape->switch_to_passive();

  auto* D = ADmode::global_tape->create_callback_object<ADmode::
      external_adjoint_object_t>();

  int nu = 0; if(this->hasUpper()) nu = this->upper().size();
  int nd = this->diag().size();
  int nl = 0; if(this->hasLower()) nl = this->lower().size();

  // register rhs as adjoint inputs
  forAll(totalSource, i)
    D->register_input(totalSource[i]);

  // register matrix coefficients as adjoint inputs
  if(this->hasUpper())
    for(int i = 0; i < nu; i++)
      D->register_input(this->upper()[i]);

  if(this->hasLower())
    for(int i = 0; i < nl; i++)
      D->register_input(this->lower()[i]);

  for(int i = 0; i < nd; i++)
    D->register_input(this->diag()[i]);

  // register boundary coefficients if on parallel boundary
  forAll(psi.boundaryField(),i)
    if(psi.boundaryField().types()[i] == "processor")
      forAll(boundaryCoeffs_[i],j)
        D->register_input(boundaryCoeffs_[i][j]);

  solverPerformance solverPerf = lduMatrix::solver::New
  (
    psi.name(),
    *this,
    boundaryCoeffs_,
    internalCoeffs_,
    psi_.boundaryField().scalarInterfaces(),
    solverControls
  )->solve(psi.primitiveFieldRef(), totalSource);

  D->write_data(psi.name());
  D->write_data(Foam::direction(-1)); // dummy direction
  D->write_data(*this);
  D->write_data(psi);

  forAll(psi.primitiveField(),i)
    psi.primitiveFieldRef()[i] = D->register_output(dco::passive_value((psi.
        primitiveFieldRef()[i])));

  ADmode::global_tape->insert_callback<ADmode::external_adjoint_object_t>(
      symbolic::fillSolverGap<Foam::scalar>,D);
  ADmode::global_tape->switch_to_active();

  diag() = saveDiag;
  psi.correctBoundaryConditions();
}
```

Listing B.1: `fvMatrix` solve with creation of solver gap and checkpoints of the necessary data.

```cpp
template<class Type>
void fillSolverGap(typename Foam::ADmode::external_adjoint_object_t *D){
  const Foam::word& fieldName = D->read_data<Foam::word>();
  const Foam::direction& cmpt = D->read_data<Foam::direction>();

  const Foam::fvMatrix<Type>& A = D->read_data<Foam::fvMatrix<Type> >();
  const Foam::volScalarField& x_ref =  D->read_data<Foam::volScalarField>();
  const Foam::scalarField& x = x_ref.primitiveField();

  Foam::scalarField a1_x(x);
  forAll(a1_x, i)
    a1_x[i] = D->get_output_adjoint(); // read incoming adjoints from tape

  Foam::fvMatrix<Type> A_T(A); // will hold transpose of A
  Foam::label nu = 0; if(A.hasUpper()) nu = A.upper().size();
  Foam::label nl = 0; if(A.hasLower()) nl = A.lower().size();
  Foam::label nd = A.diag().size();

  // component will return this for scalar field
  Foam::FieldField<Foam::Field,Foam::scalar> bcmpts = A_T.boundaryCoeffs().
      component(cmpt)();
  Foam::FieldField<Foam::Field,Foam::scalar> icmpts = A_T.internalCoeffs().
      component(cmpt)();

  // transpose matrix if necessary
  bool sym = A.symmetric();
  if(!sym){
    A_T.lower() = A.upper();
    A_T.upper() = A.lower();
    // switch boundary and internal coeffs for transposed
    icmpts = A_T.boundaryCoeffs().component(cmpt)();
    bcmpts = A_T.internalCoeffs().component(cmpt)();
  }

  Foam::word reverseFieldName = fieldName + Foam::word("Reverse");
  Foam::volScalarField a1_b(reverseFieldName,x_ref);
  const dictionary& reverseSolverControls = a1_b.mesh().solverDict([...]);

  Foam::lduMatrix::solver solver = Foam::lduMatrix::solver::New
  (
    fieldNameCmpt + Foam::word("Reverse"),
    A_T,
    bcmpts,
    icmpts,
    a1_b.boundaryField().scalarInterfaces(),
    reverseSolverControls
  );

  // solve for b_1
  Foam::solverPerformance solverPerf = solver->solve(a1_b.primitiveFieldRef(),
      a1_x);
  [...] // continued in next listing
```

Listing B.2: Adjoint callback routine, solution of the adjoint system.

```
  [...] // continuation of previous listing
  // increment input adjoint for b
  for(int i = 0; i < a1_b.size(); i++)
    D->increment_input_adjoint(dco::passive_value(a1_b[i]));

  // Adressing for upper and lower half of matrix
  const Foam::label* uPtr = A_T.lduAddr().upperAddr().begin();
  const Foam::label* lPtr = A_T.lduAddr().lowerAddr().begin();

  // increment input adjoint for A (upper part)
  double tmp = 0;
  if(A.hasUpper()){
    for(int i = 0; i < nu; i++){
      tmp = dco::passive_value(-a1_b[lPtr[i]]*x[uPtr[i]]);
      if(lPtr[i] != uPtr[i] && sym)
        tmp += dco::passive_value(-a1_b[uPtr[i]]*x[lPtr[i]]);
      D->increment_input_adjoint(tmp);
    }
  }
  // increment input adjoint for A (lower part)
  if(A.hasLower()){
    for(int i = 0; i < nl; i++){
      tmp = dco::passive_value(-a1_b[uPtr[i]]*x[lPtr[i]]);
      if(lPtr[i] != uPtr[i] && sym)
        tmp += dco::passive_value(-a1_b[lPtr[i]]*x[uPtr[i]]);
      D->increment_input_adjoint(tmp);
    }
  }
  // increment input adjoint for A (diag part)
  for(int i = 0; i < nd; i++)
    D->increment_input_adjoint(-a1_b[i]*x[i]);

  Foam::volScalarField xSF(x_ref);
  forAll(a1_b.boundaryField(),i){
    if(a1_b.boundaryField().types()[i] == "processor"){
      a1_b.boundaryFieldRef()[i].initEvaluate();
      a1_b.boundaryFieldRef()[i].evaluate();
      xSF.boundaryFieldRef()[i].initEvaluate();
      xSF.boundaryFieldRef()[i].evaluate();

      forAll(A_T.boundaryCoeffs()[i],j){
        Foam::Field<Foam::scalar> x_other =
          xSF.boundaryField()[i].patchNeighbourField()();
        Foam::Field<Foam::scalar> a1_b_this =
          a1_b.boundaryField()[i].patchInternalField()();
        tmp = dco::passive_value(x_other[j]*a1_b_this[j]);
        D->increment_input_adjoint(tmp);
      }
    }
  }
}
```

Listing B.3: Continuation of adjoint callback routine, incrementation of input adjoints.

C Reference Solver and Case

C.1 Reverse Accumulation for Incompressible Steady Flows

```
// [...] GPLv3 License Header

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "simpleControl.H"
#include "fvOptions.H"

#include "CheckInterface.H"
#include "CheckDict.H"
#include "costFunctionLibrary.H"

int main(int argc, char *argv[])
{
  #include "setRootCase.H"
  #include "createTime.H"
  #include "createMesh.H"
  simpleControl simple(mesh);
  #include "createFields.H"
  #include "createFvOptions.H"
  #include "initContinuityErrs.H"

  #include "adjointSettings.H"

  CheckInterface check(runTime);
  CheckDict checkDict(&runTime);
  CheckDatabase checkDB(&runTime,&checkDict);

  turbulence->validate();

  dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
  auto& global_tape = dco::ga1s<double>::global_tape;
  global_tape->switch_to_passive(); // iterate passive until primal convergence

  auto zero_to = global_tape->get_position();
  auto interpret_to = global_tape->get_position();

  bool frozenTurbulence = false;
  scalar adjointEps = 1e-6;

  while (simple.loop()){ // iterate passive until primal convergence
    // --- Pressure-velocity SIMPLE corrector
    #include "UEqn.H"
    #include "pEqn.H"

    laminarTransport.correct();
    turbulence->correct();

    if(simple.criteriaSatisfied())
      break;
  }

  // run and tape one active step
  global_tape->switch_to_active();
  global_tape->register_variable(alpha.begin(),alpha.end());

  zero_to = global_tape->get_position(); // don't zero alphas
  checkDB.registerAdjoints(); // register state
  interpret_to = global_tape->get_position();

  #include "UEqn.H"
  #include "pEqn.H"
  laminarTransport.correct();
  if(frozenTurbulence)
    global_tape->switch_to_passive();
  turbulence->correct();
  global_tape->switch_to_active();

  J = CostFunction(mesh).eval(); // eval cost, seed later
  checkDB.registerAsOutput();
  global_tape->switch_to_passive(); // no more recording needed

  int iter = 0;
  scalar eps = Foam::GREAT;
  while(eps > adjointEps )
  {
    if(iter == 0 && Pstream::master())
      dco::derivative(J) = 1;
    else
      checkDB.restoreAdjoints();

    global_tape->interpret_adjoint_to(interpret_to);
    checkDB.storeAdjoints();

    static scalar firstEps = checkDB.calcNormOfStoredAdjoints();
    eps = checkDB.calcNormOfStoredAdjoints() / firstEps; // normalize eps
    global_tape->zero_adjoints_to(zero_to);
    iter++;
  }
  forAll(alpha,i) // extract sensitivities after reverse acc converged
    sens[i] = dco::derivative(alpha[i]) / mesh.V()[i];
  runTime.write();
  dco::ga1s<double>::tape_t::remove(global_tape);
  return 0;
}

// ************************************************************************* //
```

Listing C.1: Source of `reverseAccSimpleFoam`

C.2 Example Case Configuration

```
dimensions       [0 1 -1 0 0 0 0];
internalField    uniform (0 0 0);

boundaryField{
  inlet{
    type             fixedValue;
    value            uniform (100 0 0);
  }
  walls{
    type             fixedValue;
    value            uniform (0 0 0);
  }
  outlet{
    type             zeroGradient;
  }
  defaultFaces{
    type             empty;
  }
}
```

Listing C.2: Velocity boundary conditions `0/U`

```
dimensions       [0 2 -2 0 0 0 0];
internalField    uniform 0;

boundaryField{
  inlet{
    type             zeroGradient;
  }
  walls{
    type             zeroGradient;
  }
  outlet{
    type             fixedValue;
    value            uniform 0;
  }
  defaultFaces{
    type             empty;
  }
}
```

Listing C.3: Pressure boundary conditions `0/p`

```
transportModel  Newtonian;
nu              nu [0 2 -1 0 0 0 0] 1;
```

Listing C.4: Viscosity and transport model `constant/transportProperties`

```
simulationType laminar;

RAS
{
  RASModel        kEpsilon;
  turbulence      off;
  printCoeffs     on;
}
```

Listing C.5: Turbulence model `constant/turbulenceProperties`

```
checkpointSettings{
  checkpointingMethod  revolve; // equidistant, revolve or none
  nCheckpoints         10;
  nTapeSteps           1;
}

checkpointRequired{
  U;
  p;
  phi;
}
```

Listing C.6: General settings `system/checkpointingDict`

```
/*---------------------------------*- C++ -*-----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  plus                                  |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*-----------------------------------------------------------------------------*/
FoamFile{ [...] }

application       simpleFoam;
startFrom         startTime;
startTime         0;
stopAt            endTime;
endTime           50;
deltaT            1;
writeControl      timeStep;
writeInterval     100;
purgeWrite        0;
writeFormat       ascii;
writePrecision    6;
writeCompression  off;
timeFormat        general;
timePrecision     6;
runTimeModifiable true;
```

Listing C.7: system/controlDict

```
/*---------------------------------*- C++ -*-----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*-----------------------------------------------------------------------------*/
FoamFile{ [...] }

ddtSchemes{ default steadyState; }
gradSchemes{ default Gauss linear; }

divSchemes{
  default                          none;
  div(phi,U)                       bounded Gauss upwind;
  div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes{ default Gauss linear corrected; }
interpolationSchemes{ default linear; }
snGradSchemes{ default corrected; }
wallDist{ method meshWave; }
```

Listing C.8: system/fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  plus                                  |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile{ [...] }
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

SDLS yes; // global flag to enable or disable SDLS

solvers
{
  "(.*)" // catch all equations
  {
    solver          smoothSolver;
    smoother        symGaussSeidel;
    tolerance       1e-05;
    relTol          0;
    SDLS            $SDLS;
  }

  "(p|pReverse)" // specialize for pressure correction equation
  {
    solver          GAMG;
    tolerance       1e-06;
    relTol          0;
    smoother        DIC;
    SDLS            $SDLS;
  }
}

SIMPLE
{
  nNonOrthogonalCorrectors 0;
  consistent               yes;
  costFunctionPatches      (inlet);
  costFunction             "pressureLoss";
  adjointEps               1e-5; // treshold for reverse acc / piggyback
}

relaxationFactors
{
  equations
  {
    U               0.9; // 0.9 is more stable but 0.95 more convergent
    ".*"            0.9; // 0.9 is more stable but 0.95 more convergent
  }
}

// ************************************************************************* //
```

Listing C.9: system/fvSolution

```
/*---------------------------------*- C++ -*-----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.7.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*-----------------------------------------------------------------------------*/
FoamFile{ [...] }
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
convertToMeters 1;

vertices(
  (0 0 0) (2 0 0) (4 0 0) (5 0 0) (0 1 0) (2 1 0) (4 1 0)
  (5 1 0) (2 3 0) (4 3 0) (5 3 0) (4 5 0) (5 5 0)
  (0 0 1) (2 0 1) (4 0 1) (5 0 1) (0 1 1) (2 1 1) (4 1 1)
  (5 1 1) (2 3 1) (4 3 1) (5 3 1) (4 5 1) (5 5 1)
);

m4_define(lvl,1)
blocks(
  hex (0 1 5 4 13 14 18 17)    ( m4_eval(lvl*2)  m4_eval(lvl*1) 1 )
      simpleGrading (1 1 1)
  hex (1 2 6 5 14 15 19 18)    ( m4_eval(lvl*2)  m4_eval(lvl*1) 1 )
      simpleGrading (1 1 1)
  hex (2 3 7 6 15 16 20 19)    ( m4_eval(lvl*1)  m4_eval(lvl*1) 1 )
      simpleGrading (1 1 1)
  hex (5 6 9 8 18 19 22 21)    ( m4_eval(lvl*2)  m4_eval(lvl*2) 1 )
      simpleGrading (1 1 1)
  hex (6 7 10 9 19 20 23 22)   ( m4_eval(lvl*1)  m4_eval(lvl*2) 1 )
      simpleGrading (1 1 1)
  hex (9 10 12 11 22 23 25 24) ( m4_eval(lvl*1)  m4_eval(lvl*2) 1 )
      simpleGrading (1 1 1)
);

patches(
  patch inlet(
    (0 4 17 13)
  )
  wall walls(
    (0 1 14 13) (1  2 15 14) ( 2  3 16 15) (4 5 18 17)
    (3 7 20 16) (7 10 23 20) (10 12 25 23)
    (5 8 21 18) (8  9 22 21) ( 9 11 24 22)
  )
  patch outlet(
    (11 12 25 24)
  )
);
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Listing C.10: `system/blockMeshDict`

D Demonstator AD Tool

```cpp
#include <iostream>
#include <vector>
#include <cmath>

typedef std::vector<double> adjoint_vector;

struct partial_edge{
  partial_edge(double v=0, int i=-1) : partial_val(v),target_idx(i){}
  double partial_val;
  int target_idx;
};

struct adjoint_stack : std::vector<std::vector<partial_edge>>{
  int longest_edge;
  int n_params;
};
adjoint_stack s;

struct adjoint_type{
  double val;
  int tape_idx;
  adjoint_type(const double& x): val(x), tape_idx(-1){}

  void operator=(const adjoint_type& x){
    this->val = x.val;
    std::vector<partial_edge> p(1,partial_edge(1.0,x.tape_idx));
    this->tape_idx = s.size();
    s.emplace_back(1,partial_edge(1.0,x.tape_idx));
  }
};

void register_variable(adjoint_type& adt){
  s.n_params++;
  s.emplace_back(); // push empty entry to tape
  adt.tape_idx = s.size()-1;
}
```

Listing D.1: Data structures for tape and partial edges; Interface.

```cpp
adjoint_type operator*(const adjoint_type& x1,const adjoint_type& x2){
  adjoint_type y(x1.val*x2.val);
  std::vector<partial_edge> partials;
  if(x1.tape_idx >= 0) partials.emplace_back(x2.val,x1.tape_idx);
  if(x2.tape_idx >= 0) partials.emplace_back(x1.val,x2.tape_idx);
  y.tape_idx = s.size();
  s.push_back(partials);
  return y;
```

```
}
adjoint_type operator+(const adjoint_type& x1,const adjoint_type& x2){
  adjoint_type y(x1.val+x2.val);
  std::vector<partial_edge> partials;
  if(x1.tape_idx >= 0) partials.emplace_back(1.0,x1.tape_idx);
  if(x2.tape_idx >= 0) partials.emplace_back(1.0,x2.tape_idx);
  y.tape_idx = s.size();
  s.push_back(partials);
  return y;
}
adjoint_type operator/(const adjoint_type& x1,const adjoint_type& x2){
  adjoint_type y(x1.val/x2.val);
  std::vector<partial_edge> partials;
  if(x1.tape_idx >= 0) partials.emplace_back(1.0/x2.val,x1.tape_idx);
  if(x2.tape_idx >= 0)
    partials.emplace_back(-x1.val/(x2.val*x2.val),x2.tape_idx);
  y.tape_idx = s.size();
  s.push_back(partials);
  return y;
```

Listing D.2: Operators

```

int find_longest_edge(const adjoint_stack& s){
  int max_length = 0;
  for(int i = s.size()-1; i>=0;i--)
    for(const partial_edge& p : s[i])
      if(p.target_idx>=s.n_params)
        max_length = std::max(max_length,i-p.target_idx);
  return max_length;
}

void interpret_tape(const adjoint_stack& s, adjoint_vector& v){
  for(int i = s.size()-1; i>=0;i--)
    for(const partial_edge& p : s[i])
      v[p.target_idx] += p.partial_val * v[i];
}

void interpret_tape_mod(const adjoint_stack& s, adjoint_vector& v){
  const int nC = v.size();
  const int nP = s.n_params;
  const int l = s.longest_edge;
  for(int i = 0; i<(s.size()-nP);i++){
    const int ti = s.size()-1-i; // position in tape
    const int j = nC-1-(i % (l+1)); // position in mod adjoint vector
    std::vector<partial_edge> partial_edges = s[ti];
    for(partial_edge& p : partial_edges){
      if(p.target_idx >= 0){
        if(p.target_idx < nP){
          v[p.target_idx] += p.partial_val * v[j];
        }else{
          const int tgt = ((j-nP)-(ti-p.target_idx)+(l+1))%(l+1)+nP;
          v[tgt] += p.partial_val * v[j];
        }
      }
    }
    v[j] = 0; // reset adjoint after all tape edges adjoined
  }
}
```

Listing D.3: (Compressed) Tape interpretation

```
int main(){
  adjoint_type a = 2;
  register_variable(a);
  adjoint_type x = a;
  for(int i = 0; i<3;i++){
    x = 0.5*(a/x+x);
  }
  std::cout << "f(x) = " << x.val << std::endl;

  { // interpret compressed
    s.longest_edge = find_longest_edge(s);
    adjoint_vector av(s.n_params + s.longest_edge+1);
    av[av.size()-1] = 1.0; // seed
    interpret_tape_mod(s,av);
    std::cout << "df/da = " << av[a.tape_idx] << std::endl;
  }
  { // interpret full
    adjoint_vector av(s.size());
    av[av.size()-1] = 1.0; // seed
    interpret_tape(s,av);
    std::cout << "df/da = " << av[a.tape_idx] << std::endl;
  }
  return 0;
}
// output:
// f(x) = 1.41422
// df/da = 0.353566
// df/da = 0.353566
```

Listing D.4: Driver calculating derivative of approximation to $\sqrt{x}$ using full and compressed adjoint vector representations.

Bibliography

[Aba+03] M. G. Abadi, J. F. Navarro, M. Steinmetz, and V. R. Eke. “Simulations of Galaxy Formation in a Λ Cold Dark Matter Universe. II. The Fine Structure of Simulated Galactic Disks”. In: *The Astrophysical Journal* 597.1 (2003), p. 21.

[AHM16] S. Akbarzadeh, J. Hückelheim, and J. Müller. “Consistent Treatment of Incompletely Converged Iterative Linear Solvers in Reverse-Mode AD”. In: *AD 2016, The 7th International Conference on Algorithmic Differentiation, Programme and Abstracts* (2016), pp. 19–22.

[Air18] Airbus. *A350 XWB Bracket Produced with 3D Printing, Reproduced with Permission.* `https://www.airbus.com/content/dam/products-and-solutions/structure/A350_XWB_Bracket.jpg`. 2018.

[AJ12] S. R. Allmaras and F. T. Johnson. “Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model”. In: *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*. 2012, pp. 1–11.

[Aln+15] M. Alnæs et al. “The FEniCS Project Version 1.5”. In: *Archive of Numerical Software* 3.100 (2015), pp. 9–23.

[Anc+97] F. Ancilotto, G. L. Chiarotti, S. Scandolo, and E. Tosatti. “Dissociation of Methane into Hydrocarbons at Extreme (Planetary) Pressure and Temperature”. In: *Science* 275.5304 (1997), pp. 1288–1290.

[And16] J. D. Anderson Jr. “Some Reflections on the History of Fluid Dynamics”. In: *Handbook of fluid dynamics*. Ed. by R. W. Johnson. CRC Press, 2016.

[ASG16] T. A. Albring, M. Sagebaum, and N. R. Gauger. “Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2”. In: *17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. 2016, p. 3518.

[AU16] G. Axtmann and R. Ulrich. “Scalability of OpenFOAM with Large Eddy Simulations and DNS on High-Performance Systems”. In: *High Performance Computing in Science and Engineering ´16: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2016*. Cham: Springer International Publishing, 2016, pp. 413–424.

[Aur+16] S. Auriemma, M. Banovic, O. Mykhaskiv, H. Legrand, J. Müller, and A. Walther. “Optimisation of a U-Bend using CAD-Based Adjoint Method with Differentiated CAD Kernel”. In: *ECCOMAS Congress*. 2016.

[Bar+00] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon. “Automatic Differentiation of Algorithms”. In: *Journal of Computational and Applied Mathematics* 124.1 (2000). Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations, pp. 171–190.

[Bar+97] J. Bardina, P. Huang, T. Coakley, J. Bardina, P. Huang, and T. Coakley. “Turbulence Modeling Validation”. In: *28th Fluid Dynamics Conference*. 1997, p. 2121.

[BBV06] C. H. Bischof, H. M. Bücker, and A. Vehreschild. “A Macro Language for Derivative Definition in ADiMat”. In: *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006, pp. 181–188.

[Ben89] M. P. Bendsøe. “Optimal Shape Design as a Material Distribution Problem”. In: *Structural Optimization* 1.4 (1989), pp. 193–202.

[Bos+14] T. Bosse, N. R. Gauger, A. Griewank, S. Günther, and V. Schulz. “One-Shot Approaches to Design Optimzation”. In: *Trends in PDE Constrained Optimization*. Springer, 2014, pp. 43–66.

[BP03] T. Borrvall and J. Petersson. “Topology Optimization of Fluids in Stokes Flow”. In: *International Journal for Numerical Methods in Fluids* 41.1 (2003), pp. 77–107.

[Bra77] A. Brandt. “Multi-Level Adaptive Solutions to Boundary-Value Problems”. In: *Mathematics of Computation* 31.138 (1977), pp. 333–390.

[Bro70] C. G. Broyden. “The Convergence of a Class of Double-Rank Minimization Algorithms: 1. General Considerations”. In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90.

[BW97] J. Bonet and R. D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, 1997.

[Car+10] A. Carnarius, F. Thiele, E. Oezkaya, and N. R. Gauger. “Adjoint Approaches for Optimal Flow Control”. In: *5th Flow Control Conference*. 2010, p. 5088.

[Che+09] J. H. Chen et al. “Terascale Direct Numerical Simulations of Turbulent Combustion using S3D”. In: *Computational Science & Discovery* 2.1 (2009), p. 015001.

[Chr18] B. Christianson. “Differentiating through Conjugate Gradient”. In: *Optimization Methods and Software* (2018), pp. 1–7.

[Chr92] B. Christianson. “Automatic Hessians by Reverse Accumulation”. In: *IMA Journal of Numerical Analysis* 12.2 (1992), pp. 135–150.

[Chr94] B. Christianson. “Reverse Accumulation and Attractive Fixed Points”. In: *Optimization Methods and Software* 3.4 (1994), pp. 311–326.

[CKS00] B. Cockburn, G. E. Karniadakis, and C.-W. Shu. “The Development of Discontinuous Galerkin Methods”. In: *Discontinuous Galerkin Methods*. Springer, 2000, pp. 3–50.

[CM69] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices”. In: *Proceedings of the 1969 24th National Conference*. ACM ’69. New York, NY, USA: ACM, 1969, pp. 157–172.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[Cox+69] H. S. M. Coxeter, H. S. M. Coxeter, H. S. M. Coxeter, and H. S. M. Coxeter. *Introduction to Geometry*. Vol. 136. Wiley New York, 1969.

[CP08] C. Chevalier and F. Pellegrini. “PT-Scotch: A Tool for Efficient Parallel Graph Ordering”. In: *Parallel Computing* 34.6-8 (2008), pp. 318–331.

[Cro00] T. J. Crowley. "Causes of Climate Change Over the Past 1000 Years". In: *Science* 289.5477 (2000), pp. 270–277.

[CS93] W. Craig and C. Sulem. "Numerical Simulation of Gravity Waves". In: *Journal of Computational Physics* 108.1 (1993), pp. 73–83.

[Cul11] M. Culpo. "Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters". In: *PRACE White Paper, http://www.praceri.eu* (2011).

[CV97] T. F. Chan and H. A. Van der Vorst. "Approximate and Incomplete Factorizations". In: *Parallel Numerical Algorithms*. Springer, 1997, pp. 167–202.

[De 78] C. De Boor. *A Practical Guide to Splines*. Vol. 27. Springer New York, 1978.

[Deu11] P. Deuflhard. *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms*. Vol. 35. Springer Science & Business Media, 2011.

[Dur+15] A. Duran, M. S. Celebi, S. Piskin, and M. Tuncel. "Scalability of OpenFOAM for Bio-Medical Flow Simulations". In: *The Journal of Supercomputing* 71.3 (2015), pp. 938–951.

[Dur08] F. Durst. *Fluid Mechanics: An Introduction to the Theory of Fluid Flows*. Springer Science & Business Media, 2008.

[Dvo09] D. Dvorak. "NASA Study on Flight Software Complexity". In: *AIAA Infotech in Aerospace Conference*. 2009, p. 1882.

[Eco18] T. D. Economon. "Simulation and Adjoint-based Design for Variable Density Incompressible Flows with Heat Transfer". In: *2018 Multidisciplinary Analysis and Optimization Conference*. 2018, p. 3111.

[EGH00] R. Eymard, T. Gallouët, and R. Herbin. "Finite Volume Methods". In: *Handbook of Numerical Analysis* 7 (2000), pp. 713–1018.

[Emm+11] C. Emmelmann, P. Sander, J. Kranz, and E. Wycisk. "Laser Additive Manufacturing and Bionics: Redefining Lightweight Design". In: *Physics Procedia* 12 (2011), pp. 364–368.

[EU16] European Commission. *AboutFlow, Adjoint-Based Optimisation of Industrial and Unsteady Flows*. `http://cordis.europa.eu/project/rcn/105535_en.html`. 2016.

[Far+13] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. "Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C369–C393.

[Fle70] R. Fletcher. "A New Approach to Variable Metric Algorithms". In: *The Computer Journal* 13.3 (1970), pp. 317–322.

[Fle76] R. Fletcher. "Conjugate Gradient Methods for Indefinite Systems". In: *Numerical Analysis* (1976), pp. 73–89.

[För14] M. Förster. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Springer, 2014.

[FT16] C. Fütterer and M. Towara. “Parametric-Adjoint Optimization using OpenFOAM and CAESES.” In: *Abstract, 4th Annual OpenFOAM User Conference, Cologne, Germany.* 2016. URL: `https://www.esi-group.com/sites/default/files/resource/other/5284/abstract_fuetterer_friendshipsystems_parametric-adjointoptimizationusingopenfoamandcaeses1.pdf`.

[Geb+13] A. Gebremedhin, D. Nguyen, M. Patwary, and A. Pothen. “ColPack: Software for Graph Coloring and Related Problems in Scientific Computing”. In: *ACM Trans. Math. Softw.* 40.1 (Oct. 2013), 1:1–1:31.

[Gez16] S. Gezgin. *Algorithmic Differentiation of a CAD Geometry Kernel and a Mesh Generation Tool.* Master Thesis, RWTH Aachen University, 2016.

[GF03] A. Griewank and C. Faure. “Piggyback Differentiation and Optimization”. In: *Large-Scale PDE-Constrained Optimization* (2003), pp. 148–164.

[GG06] M. Giles and P. Glasserman. “Smoking Adjoints: Fast Monte Carlo Greeks”. In: *Risk* 19.1 (2006), pp. 88–92.

[Gil+03] M. B. Giles, M. C. Duta, J.-D. Müller, and N. A. Pierce. “Algorithm Developments for Discrete Adjoint Methods”. In: *AIAA journal* 41.2 (2003), pp. 198–205.

[Gil08] M. B. Giles. “Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation”. In: *Advances in Automatic Differentiation.* Springer, 2008, pp. 35–44.

[Gil92] J. C. Gilbert. “Automatic Differentiation and Iterative Processes”. In: *Optimization Methods and Software* 1.1 (1992), pp. 13–21.

[GJ+10] G. Guennebaud, B. Jacob, et al. *Eigen v3.* http://eigen.tuxfamily.org. 2010.

[GM03] J. Gausemeier and S. Moehringer. “New Guideline VDI 2206-a Flexible Procedure Model for the Design of Mechatronic Systems”. In: *DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design, Stockholm.* 2003.

[GMP05] A. H. Gebremedhin, F. Manne, and A. Pothen. “What Color is Your Jacobian? Graph Coloring for Computing Derivatives”. In: *SIAM Review* 47.4 (2005), pp. 629–705.

[Gol70] D. Goldfarb. “A Family of Variable-Metric Methods Derived by Variational Means”. In: *Mathematics of Computation* 24.109 (1970), pp. 23–26.

[Goo17] Google LLC. *Tangent: Source-to-Source Debuggable Derivatives.* `https://research.googleblog.com/2017/11/tangent-source-to-source-debuggable.html`. 2017.

[GP00] M. B. Giles and N. A. Pierce. “An Introduction to the Adjoint Approach to Design”. In: *Flow, Turbulence and Combustion* 65.3-4 (2000), pp. 393–415.

[GPL] *GNU General Public License.* Version 3. Free Software Foundation, June 29, 2007. URL: `http://www.gnu.org/licenses/gpl.html`.

[GW00] A. Griewank and A. Walther. “Algorithm 799: Revolve: An implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation”. In: *ACM Transactions on Mathematical Software* 26.1 (Mar. 2000).

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, 2008.

[He+18] P. He, C. A. Mader, J. R. Martins, and K. J. Maki. “An Aerodynamic Design Optimization Framework Using a Discrete Adjoint Approach with OpenFOAM”. In: *Computers & Fluids* 168 (2018), pp. 285–303.

[Hes+08] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. “GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation”. In: *Journal of Chemical Theory and Computation* 4.3 (2008), pp. 435–447.

[HNP05] L. Hascoët, U. Naumann, and V. Pascual. ““To be Recorded” Analysis in Reverse-Mode Automatic Differentiation”. In: *Future Generation Computer Systems* 21.8 (2005), pp. 1401–1417.

[HP13] L. Hascoët and V. Pascual. “The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification”. In: *ACM Transactions on Mathematical Software* 39.3 (2013), 20:1–20:43.

[HS52] M. R. Hestenes and E. Stiefel. *Methods of Conjugate Gradients for Solving Linear Systems.* Vol. 49. 1. NBS, 1952.

[HTK12] A. A. Houck, H. E. Türeci, and J. Koch. “On-Chip Quantum Simulation with Superconducting Circuits”. In: *Nature Physics* 8.4 (2012), p. 292.

[HUB16] A. Hück, J. Utke, and C. Bischof. “Source Transformation of C++ Codes for Compatibility with Operator Overloading”. In: *Procedia Computer Science* 80 (2016). International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, pp. 1485–1496.

[Hug12] T. J. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Courier Corporation, 2012.

[Hun98] J. C. Hunt. “Lewis Fry Richardson and his Contributions to Mathematics, Meteorology, and Models of Conflict”. In: *Annual Review of Fluid Mechanics* 30.1 (1998), pp. 13–36.

[Int16] Intel Corp. “Intel 64 and IA-32 Architectures Software Developer’s Manual”. In: *Volume 2: System Programming Guide, Instruction Set Reference, A-Z* (2016).

[Jam03] A. Jameson. “Aerodynamic Shape Optimization Using the Adjoint Method”. In: *Lectures at the Von Karman Institute, Brussels* (2003).

[Jas+18] H. Jasak et al. *Foam-Extend Repository.* `https://sourceforge.net/projects/foam-extend/`. 2018.

[Jas96] H. Jasak. “Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows.” PhD thesis. Imperial College London (University of London), 1996.

[JOP+01] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open Source Scientific Tools for Python.* 2001. URL: `http://www.scipy.org/`.

[Kav+15] I. Kavvadias, E. Papoutsis-Kiachagias, G. Dimitrakopoulos, and K. Giannakoglou. “The Continuous Adjoint Approach to the k–ω SST Turbulence Model with Applications in Shape Optimization”. In: *Engineering Optimization* 47.11 (2015), pp. 1523–1542.

[KGG18] M. Kütt, M. Göttsche, and A. Glaser. “Information Barrier Experimental: Toward a Trusted and Open-Source Computing Platform for Nuclear Warhead Verification”. In: *Measurement* 114 (2018), pp. 185–190.

[Knu97] D. E. Knuth. *The Art of Computer Programming.* Vol. 3. Pearson Education, 1997.

[Kra94] D. Kraft. “TOMP-Fortran Modules for Optimal Control Calculations”. In: *ACM Transactions on Mathematical Software* 20.3 (1994), pp. 262–281.

[Leh96] M. M. Lehman. “Laws of Software Evolution Revisited”. In: *European Workshop on Software Process Technology.* Springer. 1996, pp. 108–124.

[Lep+17] K. Leppkes, J. Lotz, U. Naumann, and J. du Toit. “Meta Adjoint Programming in C++”. In: AIB-2017-07 (Sept. 2017). URL: `http://aib.informatik.rwth-aachen.de/2017/2017-07.pdf`.

[LLN16] K. Leppkes, J. Lotz, and U. Naumann. *Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features.* Tech. rep. AIB-2016-08. RWTH Aachen University, Sept. 2016. URL: `http://aib.informatik.rwth-aachen.de/2016/2016-08.pdf`.

[Lot16] J. Lotz. *Hybrid Approaches to Adjoint Code Generation with dco/c++.* Dissertation, RWTH Aachen University, 2016.

[LS83] B. E. Launder and D. B. Spalding. “The Numerical Computation of Turbulent Flows”. In: *Numerical Prediction of Flow, Heat Transfer, Turbulence and Combustion.* Elsevier, 1983, pp. 96–116.

[LW10] A. Logg and G. N. Wells. “DOLFIN: Automated Finite Element Computing”. In: *ACM Transactions on Mathematical Software (TOMS)* 37.2 (2010), p. 20.

[LW90] M. Lüscher and U. Wolff. “How to Calculate the Elastic Scattering Matrix in Two-Dimensional Quantum Field Theories by Numerical Simulation”. In: *Nuclear Physics B* 339.1 (1990), pp. 222–252.

[Maf14] L. O. Mafteiu-Scai. “The Bandwidths of a Matrix. A Survey of Algorithms”. In: *Annals of West University of Timisoara-Mathematics* 52.2 (2014), pp. 183–223.

[Mav07] D. J. Mavriplis. “Discrete Adjoint-Based Approach for Optimization Problems on Three-Dimensional Unstructured Meshes”. In: *AIAA Journal* 45.4 (2007), pp. 741–750.

[Men93] F. Menter. “Zonal two Equation k-ω Turbulence Models for Aerodynamic Flows”. In: *23rd fluid dynamics, plasmadynamics, and lasers conference.* 1993, p. 2906.

[Mes94] Message Passing Forum. *MPI: A Message-Passing Interface Standard.* Tech. rep. Knoxville, TN, USA, 1994.

[MHM18] J.-D. Müller, J. Hückelheim, and O. Mykhaskiv. “STAMPS: A Finite-Volume Solver Framework for Adjoint Codes Derived with Source-Transformation AD”. In: *2018 Multidisciplinary Analysis and Optimization Conference.* 2018, p. 2928.

[Mit+08] R. Mittal, H. Dong, M. Bozkurttas, F. Najjar, A. Vargas, and A. von Loebbecke. “A Versatile Sharp Interface Immersed Boundary Method for Incompressible Flows with Complex Boundaries”. In: *Journal of Computational Physics* 227.10 (2008), pp. 4825–4852.

[MMD+16] F. Moukalled, L. Mangani, M. Darwish, et al. “The Finite Volume Method in Computational Fluid Dynamics”. In: (2016).

[Mol18] L. Moltrecht. *Adjoint-Based Aerodynamic Optimization of a Solar Vehicle Concept.* Master Thesis, RWTH Aachen University, 2018.

[Moo59] E. F. Moore. “The Shortest Path Through a Maze”. In: *Proc. Int. Symp. Switching Theory, 1959.* 1959, pp. 285–292.

[Nau+15] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. “Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations”. In: *ACM Transactions on Mathematical Software (TOMS)* 41.4 (2015), p. 26.

[Nau08] U. Naumann. “Call Tree Reversal is NP-Complete”. In: *Advances in Automatic Differentiation.* Springer, 2008, pp. 13–22.

[Nau12] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* SIAM, 2012.

[Nav23] C. Navier. “Mémoire sur les Lois du Mouvement des Fluides”. In: *Mem. Acad. Sci. Inst. Fr* 6.1823 (1823), pp. 389–416.

[Nem+11] A. Nemili, E. Özkaya, N. Gauger, F. Thiele, and A. Carnarius. “Optimal Control of Unsteady Flows Using Discrete Adjoints”. In: *41st AIAA Fluid Dynamics Conference and Exhibit.* 2011, p. 3720.

[Nin+15] F. Ning, W. Cong, J. Qiu, J. Wei, and S. Wang. “Additive Manufacturing of Carbon Fiber Reinforced Thermoplastic Composites Using Fused Deposition Modeling”. In: *Composites Part B: Engineering* 80 (2015), pp. 369–378.

[NJ07] S. K. Nadarajah and A. Jameson. “Optimum Shape Design for Unsteady Flows with Time-Accurate Continuous and Discrete Adjoint Method”. In: *AIAA Journal* 45.7 (2007), pp. 1478–1491.

[NL18] U. Naumann and K. Leppkes. “Low-Memory Algorithmic Adjoint Propagation”. In: *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing.* SIAM. 2018, pp. 1–10.

[NW11] C. J. Nannini and H. Wan. “Designs for Large-Scale Simulation Experiments, with Applications to Defense and Homeland Security”. In: *Design and Analysis of Experiments, Volume 3: Special Designs and Applications* 810 (2011), p. 413.

[OF18] OpenCFD Ltd. *OpenFOAM User Guide - Basic Input/Output File Format.* `https://openfoam.com/documentation/user-guide/basic-file-format.php`.

[Ors70] S. A. Orszag. “Analytical Theories of Turbulence”. In: *Journal of Fluid Mechanics* 41.2 (1970), pp. 363–386.

[Osb+05] L. Osborne, J. Brummond, R. Hart, M. Zarean, and S. Conger. *Clarus, Concept of Operations: A Nationwide Surface Transportation Weather Observing and Forecasting System.* U.S. Derpartment of Transportation, Federal Highway Administration, Oct. 2005.

[Oth08] C. Othmer. “A Continuous Adjoint Formulation for the Computation of Topological and Surface Sensitivities of Ducted Flows”. In: *International Journal for Numerical Methods in Fluids* 58.8 (2008), pp. 861–877.

[Ott+03] K. N. Otto et al. *Product design: techniques in reverse engineering and new product development.* Pearson, 2003.

[OVW07] C. Othmer, E. de Villiers, and H. G. Weller. “Implementation of a Continuous Adjoint for Topology Optimization of Ducted Flows”. In: *18th AIAA Computational Fluid Dynamics Conference, June.* 2007.

[Pat80] S. Patankar. *Numerical Heat Transfer and Fluid Flow.* CRC press, 1980.

[PD83] R. W. Pitz and J. W. Daily. “Combustion in a Turbulent Mixing Layer Formed at a Rearward-Facing Step”. In: *AIAA Journal* 21.11 (1983), pp. 1565–1570.

[Pee16] B. Peeters. *Second-Order Adjoint Methods with `dco/c++` in OpenFOAM.* Bachelor Thesis, RWTH Aachen University, 2016.

[Pes02] C. S. Peskin. “The Immersed Boundary Method”. In: *Acta numerica* 11 (2002), pp. 479–517.

[Pes16] A. Pesch. *Discrete Adjoint Optimization of a Rear Wing.* Master Thesis, RWTH Aachen University, 2016.

[Pla+00] P. J. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library.* Prentice Hall PTR, 2000.

[Pop00] S. B. Pope. *Turbulent Flows.* Cambridge University Press, 2000.

[PS72] S. V. Patankar and D. Spalding. “A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-dimensional Parabolic Flows”. In: *Int. J. of Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806.

[RBM03] O. Reynolds, A. W. Brightmore, and W. H. Moorby. *The Sub-Mechanics of the Universe.* Vol. 3. Cambridge University Press, 1903.

[RC83] C. Rhie and W. L. Chow. “Numerical Study of the Turbulent Flow Past an Airfoil with Trailing Edge Separation”. In: *AIAA Journal* 21.11 (1983), pp. 1525–1532.

[Ros60] H. Rosenbrock. “An Automatic Method for Finding the Greatest or Least value of a Function”. In: *The Computer Journal* 3.3 (1960), pp. 175–184.

[RU13] R. Roth and S. Ulbrich. “A Discrete Adjoint Approach for the Optimization of Unsteady Turbulent Flows”. In: *Flow, Turbulence and Combustion* 90.4 (2013), pp. 763–783.

[Rum12] C. Rumsey. *2D Backward Facing Step.* NASA Langley Research Center. 2012. URL: `https://turbmodels.larc.nasa.gov/backstep_val.html`.

[SA92] P. Spalart and S. Allmaras. “A One-Equation Turbulence Model for Aerodynamic Flows”. In: *30th Aerospace Sciences Meeting and Exhibit.* 1992, p. 439.

[SAG17] M. Sagebaum, T. Albring, and N. R. Gauger. “High-Performance Derivative Computations using CoDiPack”. In: *arXiv preprint arXiv:1709.07229* (Jan. 1, 2017). URL: `https://arxiv.org/abs/1709.07229`.

[Sch14] M. Schanen. *Semantics Driven Adjoints of the Message Passing Interface.* Dissertation, RWTH Aachen University, 2014.

[Sch97] R. R. Schaller. "Moore's Law: Past, Present and Future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59.

[SG18] The OpenFOAM Foundation Ltd. *OpenFOAM Coding Style Guide.* `https://openfoam.org/dev/coding-style-guide/`.

[Sha04] J. Shang. "Three Decades of Accomplishments in Computational Fluid Dynamics". In: *Progress in Aerospace Sciences* 40.3 (2004), pp. 173–197.

[Sha70] D. F. Shanno. "Conditioning of Quasi-Newton Methods for Function Minimization". In: *Mathematics of Computation* 24.111 (1970), pp. 647–656.

[SM13] O. Sigmund and K. Maute. "Topology Optimization Approaches". In: *Structural and Multidisciplinary Optimization* 48.6 (2013), pp. 1031–1055.

[SN12] M. Schanen and U. Naumann. "A Wish List for Efficient Adjoints of One-Sided MPI Communication". In: *Recent Advances in the Message Passing Interface.* Springer, 2012, pp. 248–257.

[Sor+04] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. "Laplacian Surface Editing". In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry processing.* ACM. 2004, pp. 175–184.

[STN] A. Sen, M. Towara, and U. Naumann. "Sensitivity Computation for Ducted Flows using Adjoint of Implicit Pressure-Velocity Coupled Solver Based on Foam". In: *EUROGEN 2015, EUROGEN 2015, 14-16 September 2015, Glasgow, UK.*

[Sto51] G. G. Stokes. *On the Effect of the Internal Friction of Fluids on the Motion of Pendulums.* Vol. 9. Pitt Press Cambridge, 1851.

[SV16] P. Spalart and V. Venkatakrishnan. "On the Role and Challenges of CFD in the Aerospace Industry". In: *The Aeronautical Journal* 120.1223 (2016), pp. 209–232.

[Tho98] S. H. Thomke. "Simulation, Learning and R&D Performance: Evidence from Automotive Development". In: *Research Policy* 27.1 (1998), pp. 55–74.

[TN86] H. Takeuchi and I. Nonaka. "The New New Product Development Game". In: *Harvard business review* 64.1 (1986), pp. 137–146.

[TS11] K. Thulasiraman and M. N. Swamy. *Graphs: Theory and Algorithms.* John Wiley & Sons, 2011.

[TWM85] J. F. Thompson, Z. U. Warsi, and C. W. Mastin. *Numerical Grid Generation: Foundations and Applications.* Vol. 45. North-Holland Amsterdam, 1985.

[Utk+09] J. Utke, L. Hascoët, C. Hill, P. Hovland, and U. Naumann. "Toward Adjoinable MPI". In: *Proceedings of IPDPS 2009.* 2009.

[Van92] H. A. Van der Vorst. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 13.2 (1992), pp. 631–644.

[Var11] E. Varnik. *Exploitation of Structural Sparsity in Algorithmic Differentiation.* Dissertation, RWTH Aachen University, 2011.

[Von54] T. Von Kármán. *Aerodynamics: Selected Topics in the Light of their Historical Development.* Cornell University Press, 1954.

[VR84] J. Van Doormaal and G. Raithby. "Enhancements of the SIMPLE Method for Predicting Incompressible Fluid Flows". In: *Numerical Heat Transfer* 7.2 (1984), pp. 147–163.

[WD97] D. J. Wales and J. P. K. Doye. "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms". In: *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116.

[Wel+98] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. "A Tensorial Approach to Computational Continuum Mechanics using Object-Oriented Techniques". In: *Computers in Physics* 12.6 (1998), pp. 620–631.

[Wen08] J. F. Wendt. *Computational Fluid Dynamics: An Introduction.* Springer Science & Business Media, 2008.

[WG12] A. Walther and A. Griewank. "Getting started with ADOL-C". In: *U. Naumann and O. Schenk, Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science* (2012), pp. 181–202.

[Whi86] S. Whitaker. "Flow in Porous Media I: A Theoretical Derivation of Darcy's Law". In: *Transport in Porous Media* 1.1 (1986), pp. 3–25.

[Wil+98] D. C. Wilcox et al. *Turbulence Modeling for CFD.* Vol. 2. DCW Industries La Canada, CA, 1998.

[Wol69] P. Wolfe. "Convergence Conditions for Ascent Methods". In: *SIAM Review* 11.2 (1969), pp. 226–235.

[Zho+18] B. Y. Zhou, S. R. Koh, N. R. Gauger, M. Meinke, and W. Schröder. "A Discrete Adjoint Framework for Trailing-Edge Noise Minimization via Porous Material". In: *Computers & Fluids* (2018).

[Zim06] W. B. Zimmerman. *Multiphysics Modeling with Finite Element Methods.* Vol. 18. World Scientific Publishing Company, 2006.

[Zym+09] A. Zymaris, D. Papadimitriou, K. Giannakoglou, and C. Othmer. "Continuous Adjoint Approach to the Spalart–Allmaras Turbulence Model for Incompressible Flows". In: *Computers & Fluids* 38.8 (2009), pp. 1528–1538.