Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

Advanced techniques for the semi automatic transition from simulation to design software

von

Max Sagebaum

Datum der wissenschaftlichen Aussprache: 11.06.2018

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

D 386

Danksagung

Zuerst möchte ich meinen Gutachtern Nicolas Gauger, Christoph Garth und Christian Bischof für ihre Zeit und Anmerkungen zu der Arbeit danken. Ohne sie hätte die Doktorprüfung nicht stattfinden können. Meinen Doktorvater Nicolas Gauger möchte ich besonders danken, dass er mich über die letzten sieben Jahre begleitet hat und mir immer sehr freie Hand für die Lösungsansätze und Nebenprojekte gegeben hat.

Ich möchte auch den vielen Menschen danken, die mich auf dem Weg durch die Arbeit begleitet haben. Allen voran Uwe Naumann und seiner Arbeitsgruppe. Hier im besonderen Klaus Leppkes und Johannes Lotz für ihre Unterstützung im Umgang mit dco/c++ und Algorithmischen Differenzieren im generellen. Auch möchte ich Matthias Sonntag danken, für Diskussionen im Bereich der Softwareentwicklung und des Softwaredesigns. Weiterhin möchte ich auch Jan Backhaus vom DLR Köln und Sebastian Man von MTU Aero Engines für ihre Zusammenarbeit danken.

Am meisten möchte ich aber meinen Eltern danken, die mich immer bei allem Unterstützt haben und mir meinen Lebensweg und meine Entwicklung bis zu dieser Arbeit erst ermöglicht haben.

Danke.

Max Sagebaum

Zusammenfassung

Für den Einsatz von ableitungsbasierten Optimierungsverfahren für die Verbesserung von Zielgrößen industrieller Produkte, wie z.B. dem Wirkungsgrad einer Turbine, werden exakte Ableitungen benötigt, um die letzten Prozent an möglicher Effizienzsteigerung zu erhalten. Dafür ist es notwendig, dass die Ableitungen immer konsistent zu den Lösungsverfahren in der eingesetzten Lösersoftware sind. Problematisch ist dabei die fortlaufende Weiterentwicklung an der Lösersoftware, die meist dazu führt, dass approximative Ableitungsverfahren eingesetzt werden oder die Ableitungsberechnung in der Software nicht erneuert wird und somit veraltet und inkonsistent ist.

In dieser Arbeit wird ein universeller Ansatz vorgeschlagen, der zum einen die gesamte Software mittels Algorithmischen Differenzierens (AD) ableitet und zum anderem eine korrekte und konsistente Ableitung nach jeder Änderung sicherstellt. Um die Korrektheit und Konsistenz zu garantieren, wird die Technik des *Variable Tagging* entwickelt. Diese überprüft zur Laufzeit ob alle Abhängigkeiten von den eingesetzten Ableitungsmethoden korrekt berücksichtigt werden. Für die generelle Überprüfung, ob eine Ableitungsmethode korrekt implementiert ist, wird ein Theorem für die Vergleichbarkeit von AD Ableitungen entwickelt. Aus diesem Theorem ergeben sich Techniken, wie eine Implementierung getestet werden kann. Durch Analyse von vorliegenden industriellen Fällen wird die praktische Anwendbarkeit der entwickelten Techniken erfolgreich bestätigt.

Die so erzeugten Ableitungen sind damit bereits konsistent und korrekt, jedoch kann die Effizienz des Verfahrens noch gesteigert werden. Dazu werden neue Ableitungsalgorithmen entwickelt. Eine konsistente Herleitung mit einem Fix-Punkt-Iterator beinhaltet alle dem Stand der Technik entsprechenden Algorithmen und ergibt zwei neue Algorithmen, die alle implementationstechnischen Details mitberücksichtigen und damit die Konsistenz der Ableitungsergebnisse nicht verletzen.

Des Weiteren werden bekannte Techniken für das automatische Auffinden der größ-ten Ressourcenverbraucher vorgestellt und erweitert. Das Datenlayout für die Berechnung der Kenngrößen für die Ressourcenverbraucher, wie z.B. die Anzahl der Eingabe- und Ausgabewerte oder der Speicherverbrauch, wurde so erweitert, dass die Geschwindigkeit der Anwendung nur geringfügig verändert wird. Die Ressourcenverbraucher können mit Techniken wie *Checkpointing* und *Preaccumulation* behandelt werden. Die bisher in der Literatur fehlende Analyse, wie diese Techniken den Zeit- und Speicherverbrauch verändern, wird durchgeführt und Besonderheiten bezüglich verschiedener AD-Werkzeuge werden betrachtet.

Als letzten Schritt werden die benutzten AD Werkzeuge genauer analysiert. Basierend auf den existierenden Implementierungen wird eine generelle Übersicht von allen möglichen Implementationstechniken durch Operatorüberladen gezeigt. Das Hauptaugenmerk liegt auf dem minimalen Speicherverbrauch eines AD-Werkzeuges und stellt implementationstechnische Verbesserungen vor. Es resultiert die erstmals mögliche Vergleichbarkeit von AD-Werkzeugen auf einer theoretischen Ebene. Dies dient als Basis zur Entwicklung eines optimalen Werkzeugs. Das neue Software-Werkzeug *CoDiPack* geht aus diesen Überlegungen hervor und wird in seinem Design und Konzepten vorgestellt.

Die in dieser Arbeit vorgestellten Verbesserungen und Analysen ermöglichen es, eine automatisierte, konsistente und korrekte Ableitung für industrielle Zwecke effizient zur Verfügung zu stellen.

Abstract

If gradient based derivative algorithms are used to improve industrial products by reducing their target functions, the derivatives need to be exact. The last percent of possible improvement, like the efficiency of a turbine, can only be gained if the derivatives are consistent with the solution process that is used in the simulation software. It is problematic that the development of the simulation software is an ongoing process which leads to the use of approximated derivatives. If a derivative computation is implemented manually, it will be inconsistent after some time if it is not updated.

This thesis presents a generalized approach which differentiates the whole simulation software with Algorithmic Differentiation (AD), and guarantees a correct and consistent derivative computation after each change to the software. For this purpose, the *variable tagging* technique is developed. The technique checks at run-time if all dependencies, which are used by the derivative algorithms, are correct. Since it is also necessary to check the correctness of the implementation, a theorem is developed which describes how AD derivatives can be compared. This theorem is used to develop further methods that can detect and correct errors. All methods are designed such that they can be applied in real world applications and are used within industrial configurations.

The process described above yields consistent and correct derivatives but the efficiency can still be improved. This is done by deriving new derivative algorithms. A fixed-point iterator approach, with a consistent derivation, yields all state of the art algorithms and produces two new algorithms. These two new algorithms include all implementation details and therefore they produce consistent derivative results.

For detecting hot spots in the application, the state of the art techniques are presented and extended. The data management is changed such that the performance of the software is affected only marginally when quantities, like the number of input and output variables or the memory consumption, are computed for the detection. The hot spots can be treated with techniques like *checkpointing* or *preaccumulation*. How these techniques change the time and memory consumption is analyzed and it is shown how they need to be used in selected AD tools.

As a last step, the used AD tools are analyzed in more detail. The major implementation strategies for operator overloading AD tools are presented and implementation improvements for existing AD tools are discussed. The discussion focuses on a minimal memory consumption and makes it possible to compare AD tools on a theoretical level. The new AD tool *CoDiPack* is based on these findings and its design and concepts are presented.

The improvements and findings in this thesis make it possible, that an automatic,

consistent and correct derivative is generated in an efficient way for industrial applications.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

F	General function considered for differentiation.
f	Target functional for the minimization.
G	Fixed-point iterator for the solution of a PDE.
g	Not fixed, mostly used for function concatenations.
I	Identity matrix or identity mapping.
M	Function for the computation of the mesh metrics, identification for the master tape.
P	Preconditioner for a quasi-Newton step.
ϕ	Elemental operator.
Φ	Elemental mapping.
R	Residual formulation for the solution of a PDE.
$\mathcal{R}$	Remainder term in taylor approximation.
S	Identification for the secondary tape.
U	Vector space of the intermediate variables for F.
V	Vector space of a program evaluation.
X	Vector space of the input variables for F and the design space for G and f.
Y	Vector space of the output variables for F and G.
η	Counts for quantities, that are recorded on the tape.
k	Size of the intermediate steps in a program function.
l	Size of the intermediate variables U.
m	Size of the output variables Y.
n	Size of the input variables X.
o	Life cycle index, index for contracted formulation, output for communication.
a	Used as a general parameter.
b	Used as a general parameter.
c	Output value of a binary or unary operator.
γ	Specific heat ratio, lighthouse example.
ω	Used in the lighthouse example and for angle definitions.
p	Various uses.
$\mathfrak{p}$	Position in tape.
q	Various uses.
ρ	Pressure.
σ	Contraction rate for the fixed-point iterator G.
t	Various uses.
u	Vector of intermediate values.
v	State vector of a program evaluation.

$\mathfrak{v}$	Speed in the Navier Stokes equation.
w	Variables for the precomputed mesh metrics.
x	Vector of input values for F or design parameters.
y	Vector of output values for F or state of a PDE.
z	Value of the target functional f. Alternative output value if c is not available.
ζ	General object on which time and memory measurements are performed.
θ	Used in the γ-Re_θ model.

1. Introduction and outline

1.1. Motivation

The topic of the thesis is about the transition of simulation software to design software, which is done in such a way, that the transformation is automated. This raises several questions:

- What is simulation software?
- What is design software?
- Why does the former need to be transformed into the latter?
- Why should the process be automated?
- How is the automation done?
- Why is a thesis required about this topic?

An in depth introduction to all these different research fields is not possible, but each question will be motivated such that the general point of the thesis can be made.

What is simulation software? Simulation software can be described by the effort to model the real world in a computer, which is a reproduction of the physical laws. Such things can be the behavior of a bridge under strain in order to know if the bridge will break. It can also be the question how heat is propagated in the oceans such that natural phenomena as El Nino can be understood and forecast, how the melting of ice sheets is affected by the global climate change, how the fluid flow around an airfoil affects the drag and lift or what will be the thrust of a newly developed jet engine? The simulation of phenomena like these, helps to make our lives easier while also preventing the occurrence of catastrophic events. It is therefore necessary to develop computer software that can simulate the real world, but the task is quite involved and requires several steps until a problem can be simulated with a reasonable accuracy.

First, a deep understanding of the physical laws and their formulation as mathematical equations is necessary. The modeling can be done in several different ways and with different abstraction mechanisms. The Boltzmann equations [Har04] can be used as a starting point and can for example be modeled as the Lattice-Boltzmann method [HL97] or as the DNS (direct numerical simulation) method [Spa88]. There are many other methods available but in this introduction partial differential equations are chosen as an example. The Navier-Stokes partial differential equations

[Tem84] can be derived from the Boltzmann equations and describe how a viscous flow behaves. They can be used to simulate such phenomena as mentioned above. A direct implementation of these equations is not possible as they are formulated in a continuum domain, that can not be represented on a computer. The domain of the problem needs to be broken down into discrete units. This discretisation can be done in many ways and one choice is to use small blocks each representing a small volume of the domain. If the blocks are small enough, then we can assume that all fluid properties (like temperature) are constant inside of each block. The combination of millions of blocks will give a discrete representation of the problem domain in finite volumes, and is called the finite volume method [VM07]. Several other choices, how the discretization can be achieved exist e.g. the finite difference method [LeV07], the finite element method [ZTZ13] or the discontinuous Galerkin method [CKS00].

In order to enable the computations to be executed on the available hardware, several different approximations have been invented over the past few decades. A large range of methods is available that differ in their computational complexity. These range from simple panel methods [AJ10] to the Reynolds averaged Navier-Stokes (RANS) equations [Alf09]. The RANS equations are established under the assumption that the motions in the fluid can be decomposed into two components: one that describes the time averaged motion of the fluid, and one that models the small fluctuations [Ran06].

On current computer clusters it is possible to simulate the unsteady RANS equations for full aircrafts [ND17], but the implementation of such numerical schemes is a major effort. In order to be accurate the software needs to be validated against several experiments such that it is clear that the physical laws are correctly simulated [RJL$^+$15]. Each experiment, like a wind-tunnel experiment [SG03], can cost several thousands of euros and accumulate into millions of euro over the development time of a simulation code. Next to the simulation, the code needs to be written such that it performs well on super computers, which requires an entire team of software experts [WGNP07] and can not be accomplished by a single person. Until a simulation code has reached a state where it can be used reliably to investigate new problems of large size, several hundreds of people have to work for several years. It is then an ongoing effort to improve the accuracy, performance and capabilities of the code which easily leads to software, where thousands people-years of work and millions of euro have been investigated. Each line of such a software can have easily an estimated worth of 100 euro.

What is design software? We now have a software that can be used to simulate the flow around an airplane, or the stresses acting upon a bridge. It is now natural to want to use the software to design an airplane that requires less fuel or carry a larger cargo with the same amount of fuel. Usually an engineer tries to maximize or minimize a certain target value. If the drag of an airfoil can be decreased, the efficiency gain will decrease the airplanes overall fuel consumption leading to reduced ticked prices for our next holiday. An optimization can also avoid a bridge from breaking when it is hit by a storm and therefore save lives [LMHF13].

Most designs developed over the last few decades [Irv93] could either be optimized by an engineers intuition, from experience gained by previous designs or by small model tests. Experience however, goes both ways. It is gained through successful designs, but is also gained through design failures and the casualties they may cause. A sharp edge of a top window design lead to unusual amounts of high pressure and caused the cabin of the BOAC Flight 781 to explode, killing everybody on board [Ste88].

Intuition and experience can only lead to designs that are close to a certain optimal solution. Current industrial designs are already quite optimal but in order to gain the last few percents of improvement a manual trial and error approach is not feasible. Recent designs can have several millions of parameters and it is impossible for a human to gain an intuition for each of these parameters. It is therefore necessary to develop a process that will find the optimal solution in an automated fashion.

The mathematical notion for this is the derivative of a function. The derivative describes how a change in the design parameters (e.g. the thickness of a beam in a bridge) will change the target value used for optimization (e.g. the maximum load of the bridge). The derivative, with respect to all design parameters, is called the gradient and points in the direction of the steepest ascent [NW99]. It can be used by an engineer or by an automated algorithm to generate a new design. Gradient based optimization methods like the steepest descent algorithm [NW99] provide an automated way to find an optimal design.

Why simulation software needs to be transformed into design software and why the process should be automated? In order to manage the complexity of a design process with millions of design parameters and gain the last percent of efficiency the simulation software needs to calculate the derivatives required for optimization, in doing so it becomes a design software. Derivatives can be integrated into a simulation software in different ways. The simplest method is the finite difference approximation [Mic91]. One design parameter can be modified by a small amount and the difference in the solutions can be used to estimate the derivatives for this parameter. If this process is repeated for all design parameters the full gradient can be approximated. The finite difference method has two drawbacks. It is not clear by how much a parameter should be modified, such that the truncation and cancellation errors are minimized. For some parameters the modifications can be in the order of 10^{-3} and for others in the order of 10^{-6}. The second drawback is the feasibility for a large set of design parameters. If one simulation takes one day and there are 10,000 design parameters, one approximation of the gradient will take 27 years.

The second option is to provide a manual implementation of the derivative computation. This can either be achieved by looking at physical laws like the Navier-Stokes equations and deriving an equation from them for the sensitivity computation [JMP98], which is then discretised, implemented and validated, or by visiting each code line and implementing the respective derivative. The equation centered approach is called “first optimize and then discretize” while the code centered approach is called “first discretize and then optimize” [BC05]. Both approaches lead to new code that is added to the simulation software and if done properly will provide the fastest way to compute the derivatives with the optimal amount of resources. The new code however, needs to be validated and tested in the same way as the simulation code. The development of the simulation code already requires several people-years and now a second code base has to be developed, tested, maintained as well as updated. This would require the same amount of people as previously, which is usually not feasible. Also each code change in the simulation code needs to be reflected in the derivative code, otherwise the results will be inconsistent representing yet another complication to overcome.

Further inconsistencies can arise due to the development procedure for both codes. Usually a new feature is first introduced in the simulation code and tested there until it is determined fit for productive use. Only afterwards the development of the derivative code for this feature starts. This yields the discrepancy, that the simulation code is capable of computing something that is not available in the corresponding derivative computation. Until the same feature is available in the derivative code, several month or years can pass. During this time the simulation code will have evolved further and a difference in the capabilities of the simulation code and the derivative code will almost always be present. In order to avoid this gap, the transformation for the simulation code into the design code needs to be done automatically.

How to automatize? Both presented options are not feasible for highly complex simulation codes. It would be most appropriate if the differentiation of the software could be automated. As a first step the insecurity about the step size of the finite difference approach can be eliminated. Instead of perturbing the design variables, the directional derivative with respect to the design variables can be computed. For this purpose each design parameter is given a direction in which it is differentiated and the program is viewed as a chain of intrinsic functions like the addition, multiplication, sine or cosine, that are sent to the cpu. As the first intrinsic function is handled by the cpu, the directional derivative needs to be computed. The derivative for each of the intrinsic functions is quite simple and therefore the gradient can be evaluated and multiplied with the derivative direction that is set for the input variables. The directional derivative for the computed intrinsic function is therefore also available and computed alongside the intrinsic function. For the second intrinsic function the process is repeated and so on. After millions of statements, the simulation will finish and for every computation the directional derivative and the original primal value will have been computed. The target values will have a directional derivative associated with them. This gives us the information how sensitive the target value is with respect to the initial direction. The nice thing about this procedure is, that only a direction needs to be chosen and no step size for the perturbation. The resulting derivatives will also be accurate to machine precision. This process is called nowadays the forward mode of Algorithmic Differentiation (AD) [GW08]. It has been rediscovered several times in the past decades [Wer82] and it still is.

The forward mode interprets the problem of the derivative calculation from the viewpoint of the design parameters. For each parameter the question “How will a change in the design parameter change my target quantities?” is answered. A more natural way of formulating the question would be “How will a change in the target quantity change my design parameters?”, as the engineers goal is to improve the target quantities. The answer to this question can be directly computed with the reverse AD mode [GW08].

The forward mode interprets the program as a chain of intrinsic functions and applies the chain rule to these intrinsic functions. The result is a concatenation of millions of simple Jacobi matrices. By applying the directional derivative to this chain of Jacobi matrices, a vector is multiplied from the right. If now a vector is multiplied from the left instead, the propagation of the derivative information is performed from the target quantities to the design parameters. The vector that is multiplied from the left is called the adjoint direction. If the sensitivities for a change in the first target quantity is computed, then the adjoint vector will be set to the first unit vector. The result then describes, how the design parameters will need to be changed in order to gain the desired change in the target quantity. This describes the process for the reverse AD mode. It has also been rediscovered several times over the past decades [Lin76] and is know by the machine learning community as back propagation [Wer94]. With this technique the derivatives to all design parameters can be computed with one evaluation of the simulation code and one reverse interpretation.

The advantage of the reverse AD mode in contrast to the hand implementation is, that it can be fully automated. With this method every change to the simulation software is automatically available in the reverse AD mode, eliminating the drawbacks of the hand coded derivatives. Nevertheless, the application of the reverse AD mode to a large software includes several challenges. The reverse mode changes the information flow in the program. For each input, the adjoint value is now an output and for each output the adjoint value is now an input value. Due to this reversal of the information flow, the computation for the reverse mode can not be computed directly alongside the primal computation. For each function or statement a certain set of data needs to be stored, such that the reverse interpretation can be done. Proper management of the data as well as the correct reversal of the program are the major challenges in the application of the reverse AD mode, which makes the task none trivial.

An alternative to the reverse mode of AD would be the symbolic differentiation of the mathematical formulation for the program. This approach is not considered here since several problems arise. The mathematical equations are not directly implemented and therefore a discrepancy exist. The symbolic differentiation yields new formulations, that need to be implemented or generated. Also the symbolic differentiation may yield large formulations that increase the complexity of the software.

Since, an important aspect for the automation is the consistency of the derivatives with respect to the actual software implementation, the reverse mode of AD is selected for the automation.

Why is a thesis required on this topic? Since the original development of the AD theory started in the late 1970s, there should be sufficient knowhow and tools available such that the efficient transformation from simulation software to design software should already be possible. If a developer wants to start with such a task, then the decision as to which AD tool is appropriate needs to be decided first.

Several tools are available: ADOL-C [WG09], AD model builder [FSA$^+$12], CppAD [BB08], etc., but the user has usually no means to compare the tools on a theoretical level. Nearly every tool has a different advantage or disadvantage. The AD model builder can be used to handle higher order structures in an efficient way. ADOL-C provides various drivers, that enable a user to easily perform different differentiation tasks like the forward mode, the reverse mode, the computations of Hessians etc.. Often the different tools are compared only upon very small problems. These tests e.g. by the Stan Math library [CHB$^+$15] do not exhibit the performance characteristics of large scale software that is evaluated on a cluster node. Here the memory bandwidth limitation is most critical and small tests do not show problems resulting from the management of large memory chunks. Due to the small test size the tools cannot be compared for the memory consumption and there is nearly no literature about how much memory an AD tool will use per operation or statement. The Stan Math library is the only major tool where the memory per operation is shown.

Other decision criteria for the appropriate AD tool would be the properties of the calculation type provided by the tool. There is currently no literature available that explains the properties of the tools and no tool developer states this in their development examples. The most important property as to whether or not the type is safe to use in C-like memory operations is scarcely documented. For example, ADOL-C is a tool where it is strictly forbidden to use such operations, on the other hand dco/C++ [LLN16] can be used in such operations without any problems.

AD tools that apply AD to a code base via operator overloading are also always classified as one big category lacking any distinctions. Juedes [Jue91] provides six different categories for AD tools but operator overloading tools are only classified as operational tools. Bischof et al. [BHN08] provides also an overview of AD tools, but uses only one category for operator overloading tools. That there are different taping, data management and various implementation approaches is nowhere reflected. The resulting properties of the tools often go unexplained in the literature. Especially for libraries like operator overloading AD tools, that change how each operation in the code is performed, it should be well documented which properties the implementation has. For HPC applications, the data storing, the access patterns and the kind of computations are very important.

The final decision criteria could be which improvements have been applied to the AD tool in order to reduce memory consumption or computing time. Only recently papers about the efficient implementation of expression templates have been published by Hogan [Hog14] and Phipps [PP12]. Phipps gives a good demonstration on the implementation of this approach and discusses different improvements that were made to the implementation, a topic seldom covered in literature. Hogan provides also a good insight into the implementation process, but his comparisons to other tools only occur on small scale examples. Small tricks are found in several different AD tools but there are no papers about these implementation optimizations.

The decision for the most appropriate AD tool can also be made by reports of the successful application of AD to large scale HPC codes. For programs written in Fortran some results are available. In [BBL$^+$01, BBL$^+$03] the AD tool ADIFOR is applied on the finite element package SEPRAN. ADIFOR is also applied in [FE02] on the CFD solvers mheuler and in [CGBN94, GNH96] it is applied on the CFD solver TLNS3D. R. Giering et al. present in [GKE$^+$09] the application of the AD tool TAF on the CFD solver FLOWer and the grid generator MUGRIDO. Most of these reports only apply the forward AD mode, which does not exhibit the memory management challenges of the reverse AD mode. If the reverse AD mode is applied in large scale software, then it is usually only done for small parts of the application.

For C/C++ there are no results for large scale software available so far. In [SGN$^+$13] a first approach for the semi automatic transition of a simulation to a design software has been achieved. Further research in that direction is also done by the chair for Scientific Computing from the University of Kaiserslautern in cooperation with the group of Naumann [LNSS13].

Most other approaches on large scale software in C/C++ apply AD only on a small part of the application. F. Zaoui presents in [Zao08] the application of the AD tool ADOL-C to a large code base but the solution process is dominated by the solution of a linear system and only the setup of the matrix and the right hand side is differentiated with AD.

It is therefore quite hard to decide which tools are appropriate for the use in an HPC context, if the tool uses the optimal memory layout and how to apply the AD tool to large scale software. The necessary information is only available after a careful study of the AD tools and by experimenting how these tools are best introduced into the software. In this area Hück et. al [HBU15] first analyzed how common problems in the introduction of an operator overloading AD tool to a software can be detected automatically. In a further work [HUB16], they provide a tool for the automatic correction of the detected problems.

The choice of the AD tool is not the only challenge that needs to be tackled, when the software is differentiated. The validation of the derivative results is another important aspect, but usual techniques only employ the validation with finite differences and the comparison with the forward mode of AD. How problems are covered, when the validation fails, is mostly not described in detail or only mentioned. A detailed analysis on how to find errors in the computation of the AD derivatives or how to validate special techniques is not available. But these techniques are necessary in order to automate the maintenance of the primal and adjoint code.

Further aspects in the differentiation of large code bases involves the handling of code locations that use the most resources, the so called hot spots. An automated analysis for the hot spots is introduced by J. Lotz [Lot16, LNM16]. The work concentrates on the correct measurements of the required resources for AD and collects the number of input variables for each function call in the program. The hot spots can then be handled with the general known techniques of function checkpoints and preaccumulation [Utk05]. These techniques are generally known but details on their implementation for specific taping strategies are not available.

1.2. Content and general approach

The aim of the thesis is to provide a general approach for the transformation of large scale simulation software to design software. The two most important aspects are the efficiency and maintainability of the approach.

The efficiency is defined in terms of execution speed, memory consumption and development time for the application of an AD tool onto the software. The maintainability covers the long term development aspects of large scale software. That is, the derivative code should always be up to date with the simulation software and every change should be automatically transferred to the derivative code.

The approach of this thesis is therefore to treat the simulation software as a black box. This means that the software is differentiated completely with AD and no part of the software is skipped. This approach by itself is not new but in general avoided. The initial workload can be quite high and the result is not guaranteed to satisfy the performance requirements. Nevertheless, it has the advantage that the derivative code is always consistent with the implementation and computes exact derivatives. Through the use of AD it is guaranteed, that the derivative code will always be up to date and the maintenance aspect is mostly covered.

The initial workload and further work on maintenance can be reduced with proper validation techniques for the derivatives. Since there are no general techniques available in the literature, new techniques are developed in this thesis. The developed *tagging* technique is an online validation technique that traces the dependencies of variables in the application. It is a novel technique that is used to validate all kinds of interfaces and dependencies required by AD. A second validation technique is described for the AD tapes. It is based on a theorem derived in this thesis for the comparability of two tapes. In the resulting algorithm one tape is considered a correct master tape and a second tape is considered to be a wrong secondary tape. The algorithm finds all differences in the tapes and provides an indication for wrong implementations, bugs in the AD tools, or other errors in the dependencies.

Since the whole application is differentiated with AD the analysis for the discrete adjoint method needs to be done in this framework. In contrast to the usual mathematical formulation, where an abstract model of the application is considered that does not fit to the implementation, a consistent modeling is considered here. The modeling in the thesis is based on a fixed-point solution process, which is used by nearly all state of the art solvers. For the fixed-point solver, an initial guess for the solution is made. The guessed solution is then updated by the numerical scheme in an iterative process until no further changes occur. The reached fixed-point is then the solution of the problem. By using the fixed-point iterator in the mathematical modeling, a direct link to the software implementation exists. The fixed-point iterator approach will contain all techniques like flux limiting or turbulence modeling which do not need to be considered separately in the differentiation process.

Two modeling theories are applied to generate derivative algorithms. The derivation in the standard optimization framework (Lagrange function with fulfilled KKT conditions [KT51]) yields the *Reverse Accumulation* algorithm. The theory requires the solution to be good enough in order to yield appropriate convergence behavior and therefore meaningful derivative results. But theses conditions are weaker than the conditions that are required by state of the art algorithms for the derivative computation.

The second modeling theory comes from AD itself. Here, on the whole computation algorithm AD is applied, yielding the *Black Box* derivative algorithm that always provides consistent derivative information. For both algorithms it is shown that they are generalizations of the current state of the art algorithms. Thus they provide improved consistency over the currently used algorithms.

After the code is differentiated and the new derivative algorithms are implemented, the analysis of the code for resource hot spots needs to be done. The current theory in this area is extended such that also the number of output variables can be tracked for all regions of the code. In addition, the algorithms and data layouts are changed such that unnecessary overheads are eliminated. Only then the analysis of large scale applications can be performed.

For the treatment of hot spots, checkpointing and preaccumulation techniques are presented. Since they are usually not applied in a framework where the whole application is differentiated and no special code is written for the hot spots, both techniques are adapted to these constraints. The implications of these constraints are analyzed with respect to the available AD implementation techniques. Possible problems are highlighted and their solutions presented. In addition the memory and time changes for the application of checkpointing and preaccumulation are investigated in great detail, consequently the most appropriate technique can be chosen for each hot spot.

A minimal invasive hot spot treatment technique is presented for an example software. It is based on the global structure of the software and applied such that the required memory can be reduced without changing time requirements.

The selection of an appropriate AD tool is discussed after a general presentation of AD implementation strategies. All known AD implementation techniques for operator overloading are presented in a generalized framework and for each technique the minimal required memory is derived. Known improvements to all techniques are presented, too. These improvements are again analyzed for their minimal required memory.

New ideas for further improvements are presented and analyzed for minimal memory as well. The linear subexpression improvement tries to reduce the required memory for code statements that are linear and therefore require less information for the derivative computation. The removal of duplicated arguments is a second new technique that tries to optimize the graph of each statement. The graph is modified so each duplicated argument is only stored once. This removes unnecessary information from the storage.

The resulting memory efficient AD implementation strategies are compared to currently available AD tools. The comparison shows that some tools already provide a memory efficient implementation but in an strategy that is not too optimal. This leads to the design of a new AD tool, CoDiPack (Code Differentiation Package). The main goal of this tool is to provide the memory efficient AD implementations presented in the thesis and to provide a framework for further AD based research.

1.3. Outline

The first chapter of the thesis will introduce the basic theory of AD and provides a first approach for the implementation of an AD tool. This also covers the introduction of basic naming conventions in an AD tool. The following chapter introduces the analysis of the available derivative algorithms for the outer derivative, that is the global structure of the program. Here, the new framework is introduced and specialized such that the state of the art algorithms are recovered. This sets all the available algorithms in context to each other. Chapter 4 will cover the Black-Box differentiation approach. It is exemplary performed in an exemplary fashion upon the TRACE code and then the techniques for the automated validation and maintenance of the differentiated software are discussed. After that, the optimization techniques for whole functions are presented. First the hot spot analysis is shown and then the checkpointing and preaccumulation techniques are analyzed. Chapter 6 handles the selection and improvements of the AD tools. The chapter builds on the implementation approach from Chapter 2 and introduces several different implementation strategies for the reverse AD mode. Improvements for the different strategies are presented that provide a large variety of options. Current AD tools are put in relation to the presented strategies and analyzed if they can be applied to large scale software. The chapter is concluded with the presentation of the new AD tool CoDiPack, a tool specifically designed for large scale software which implements all of the presented implementation strategies. The final chapter presents performance results of the different AD tools, function optimization techniques and derivative algorithms. For the derivative algorithms an analysis of their robustness under various conditions is provided.

2. Algorithmic Differentiation

Several mathematical algorithms require derivatives in order to work properly. Some are partial differential equations, gradient based optimization, Newton's method and parameter fitting. In order to provide these derivatives a common choice is to implement them by hand. For small functions or software packages this is still possible but the derivation and implementation quickly become a tedious task for large scale software.

The theory of Algorithmic Differentiation (AD) tries to provide the basis for the automation of derivative generation in a computer program. For this purpose the program is viewed as a sequence of simple elemental (or intrinsic) functions that include the sine, cosine, multiplication and addition. By applying the chain rule to this sequence, a representation of the derivative for the computer program is created. If the directional derivative is required for this representation, then the computation can be evaluated quite efficiently. Let

$$w = (a + b) * (c - d) \tag{2.1}$$

be the computer program. This program can now be split into the elemental function which yields the intermediate steps:

$$\begin{aligned} t_1 =& a + b \\ t_2 =& c - d \\ w =& t_1 * t_2 \ . \end{aligned} \tag{2.2}$$

For each of these steps the Jacobi matrix can be computed quite easily, for $a + b$ the matrix is $(1, 1)$. In order to compute the directional derivative, the Jacobi matrix is multiplied with direction $(\dot{a}, \dot{b})^T$, where the dot notation is used to describe the corresponding derivative direction of a variable. If this process is applied to the full procedure, the result is

$$\begin{aligned} \dot{t}_1 =& \dot{a} + \dot{b} \\ \dot{t}_2 =& \dot{c} - \dot{d} \\ \dot{w} =& t_2 * \dot{t}_1 + t_1 * \dot{t}_2 \ . \end{aligned} \tag{2.3}$$

By computing the corresponding directional derivative statement in procedure (2.3) alongside the original statement in procedure (2.2), the directional derivative is computed for the whole computer program (2.1). If the same process is done for each operation in a computer program, then the directional derivative can be computed for arbitrarily complex software.

In summary, the directional derivative of a software can be computed by introducing a derivative variable for each variable, splitting the software into elemental functions, computing the Jacobi matrix for every elemental function and multiplying it with the directional derivative of the respective variables. This procedure is called the forward mode of AD. From a mathematical viewpoint it consists of applying the chain rule to the whole computer program and then computing the directional derivative. A rigorous derivation will be done in this chapter.

The second operation mode of AD is called the reverse AD mode. It can be introduced in a very crude way by multiplying an adjoint direction from the left side instead of multiplying a derivative direction from the right. The adjoint for a variable a is denoted with the bar notation $\bar{a}$. For the statement $t_1 = a + b$, the Jacobian is $(1, 1)$ and multiplied with the adjoint direction $\bar{t}_1$ from the left. This yields the two updates $\bar{a} = \bar{t}_1$ and $\bar{b} = \bar{t}_1$. The reverse AD procedure for procedure (2.2) is then

$$\begin{aligned}
\bar{t}_1 =& t_2 * \bar{w} \\
\bar{t}_2 =& t_1 * \bar{w} \\
\bar{c} =& \bar{t}_2 \\
\bar{d} =& -\bar{t}_2 \\
\bar{a} =& \bar{t}_1 \\
\bar{b} =& \bar{t}_1 \ .
\end{aligned} \tag{2.4}$$

Because of the multiplication from the left, the order of the adjoint statements is reversed, which is also the advantage of the reverse AD mode. The full gradient of a real valued function (e.g. a computer program with one output value) can be computed with one reverse AD evaluation. If the program has one million input variables, then the full gradient can be computed in one reverse sweep.

The first two sections will provide a very strict way of deriving the forward and reverse AD mode. How both methods work will be shown in a more sophisticated example. Afterwards, a first naive implementation of the forward and reverse AD mode is provided. This implementation will introduce a user defined type that uses operator overloading to perform the additional AD operations alongside the primal computations. This section will also introduce basic notations and definitions for the AD implementations, that are used throughout the thesis. The naive implementation will be refined in Chapter 6 and possible avenues for improvements are explored there.

Remark 2.1 (Notations)
The notation for AD in this thesis is taken from the book of Griewank et al. [GW00], and is used by Griewank only to introduce the reverse mode. It is used here to introduce the forward mode as well, which yields a consistent notation and derivation for both modes. The only difference is that no negative indices are used for the input variables, this is also the notation that Naumann uses in [Nau12]. The basic calculus definitions are taken from the book of Apostol [Apo69] and are repeated in Appendix A.

Remark 2.2 (AD on algorithms)
The theory of AD is often seen just as an intermediate step for the implementation of some tools that apply AD to a computer program. But, the theory of AD can also be used to generate a differentiated version of mathematical algorithms which could be used to describe the global structure of a numerical process. By applying AD to the mathematical algorithm, it is easy to see how the derivative of this algorithm will look and this insight can be used to program the derivative of such an algorithm in a more efficient way.

2.1. The forward mode

In order to differentiate a computer program or (in short) a code, a mathematical representation for the same computer program is required. The assumption is that the program computes the function $F : X \subset \mathbb{R}^n \to Y \subset \mathbb{R}^m$. The program consist of the input values x from the set X, the output values y in the set Y and all intermediate results u in the set U. The intermediate results are created during the computation of the output values y from the input values x. The theory is based on the assumption that the intermediate values u and the output values y are computed by simple functions for which the derivatives are known. First the evaluation space of the program is defined.

Definition 2.3 (Program space)
X, Y and U are defined by:

- $X \subset \mathbb{R}^n$ is the vector space of the input values.
- $Y \subset \mathbb{R}^m$ is the vector space of the output values.
- $U \subset \mathbb{R}^l$ is the vector space of the intermediate values.

Then
$$V := X \times U \times Y \subset \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R}^m = \mathbb{R}^{n+l+m}$$
is the vector space of the program evaluation. The vector $v \in V$ can also be written in component form $v = (x, u, y)^T$.

The program space views the computer program as a large state where all intermediate variables are always available. The operations in the program create a new state, where only one element is modified. These transitions are defined by elemental operators and mappings.

Definition 2.4 (Elemental operator)
An elemental operator is a function $\phi_i : V \to \mathbb{R}$ with $i \in \{1, ..., l + m\}$, that computes the i-th operation in the program space V. The operator depends only on the first $n + i - 1$ entries of the vector $v \in V$.

Remark 2.5

If the i-th operation in the program is

$$c = sin(a) \quad ,$$

then the corresponding elemental operator is

$$\phi_i(v) = sin(v_j)$$

where $j \in \mathbb{N}$ is the corresponding index for $a \in \mathbb{R}$ in the program space.

Definition 2.6 (Elemental mapping)

An elemental mapping is a function $\Phi_i : V \to V$ with $i \in \{1, ..., l+m\}$, that sets the $n+i$ entry in the program space V. The elemental mappings are defined by

$$\Phi_i(v) = (v_1, \dots, v_{n+i-1}, \phi_i(v), \underbrace{0, ..., 0}_{l+m-i}) \tag{2.5}$$

where the $n+i$-th entry is set by the elemental operator ϕ_i and the first $n+i-1$ entries stay the same.

In order to handle the input and output values the mapping from X to V is defined by

$$\begin{aligned} I_{X,V} : X &\to V \\ x &\mapsto (x, 0, 0)^T \end{aligned}$$

and the projection from V to Y is defined by

$$\begin{aligned} I_{V,Y} : \qquad V &\to Y \\ (x, u, y)^T &\mapsto y \ . \end{aligned}$$

These definitions will be used to describe the relation between the mathematical function F and the evaluation of the computer program.

Definition 2.7 (Program function)

Let $x \in X$ be the input values of the program and $y \in Y$ be the output values. The program consists of $k \in \mathbb{N}$ elemental mappings, which calculate the function $F : X \to Y$.

F is called the program function if and only if the equation

$$F(x) = I_{V,Y} \circ \Phi_k \circ \Phi_{k-1} \circ ... \circ \Phi_2 \circ \Phi_1 \circ I_{X,V}(x) \tag{2.6}$$

holds and $l = k - m$ is the size of the intermediate values.

Remark 2.8 (Output variables)

Through the definition of the program space V and the program function F it is always implicitly assumed, that the last m elemental mappings compute the output y of the function.

The program function represents the link between mathematical theory and the computer program. The computer program is represented in this definition as a single assignment code (SAC) [Nau12]. Each operation performs only one assignment and only one variable is changed per operation. But, how does the derivative of the function $F(x)$ apply itself to the elemental mappings and how do the resulting equations describe an algorithm for the derivative calculation? In order to show this an assumption about the elemental operations ϕ is established.

Assumption 2.9 (Differentiability)
The elemental operations $\phi : V \to \mathbb{R}$ are differentiable up to the order $d \in \mathbb{N}$.

Elemental mappings are a combination of the identity and the elemental operations, and are also d times differentiable. The concatenation of the mappings Φ_i for all $i = 1 \dots k$ is then d times differentiable by the chain rule. This yields the differentiability of the program function F, which is represented by the elemental mappings. The derivative of F exists and the derivative F' can be represented with the derivatives of $I_{X,V}$, $I_{V,Y}$ and Φ_i.

The derivative of $I_{X,V}(x) = \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}$ is:

$$\frac{dI_{X,V}}{dx} = \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} .$$

The derivative of $I_{V,Y}(x, u, y) = y$ is:

$$\frac{dI_{V,Y}}{dv} = \begin{pmatrix} 0 & 0 & I \end{pmatrix} .$$

The derivative of an elemental mapping Φ_i defined in eq. (2.5) is presented for each component separately. The notation $\Phi_{i_j}(v) := (\Phi_i(v))_j$ is used as an abbreviation in the following statements.

- Let $j < i + n$, then $\Phi_{i_j}(v) = v_j$ and it follows

$$\frac{d\Phi_{i_j}}{dv} = (0, ..., 0, \underbrace{1}_{\text{j-th}}, 0, ..., 0) .$$

- Let $j > i + n$, then $\Phi_{i_j}(v) = 0$ and it follows

$$\frac{d\Phi_{i_j}}{dv} = (0, ..., 0) .$$

- Let $j = i + n$, then $\Phi_{i_j}(v) = \phi_i(v)$ and it follows

$$\frac{d\Phi_{i_j}}{dv} = \frac{d\phi_i}{dv} = \left(\frac{\partial \phi_i}{\partial v_1}, ..., \frac{\partial \phi_i}{\partial v_{j-1}}, \underbrace{0}_{\text{j-th}}, 0, ..., 0 \right) .$$

The full Jacobi matrix of Φ_i can now be written as:

$$\frac{d\Phi_i}{dv} = \begin{pmatrix} I & 0 & 0 \\ \frac{d\phi_i}{dv} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} .$$

The first step for the derivation of F is the analysis of two concatenated elemental mappings. Let $\Phi_1 : V \to V$ and $\Phi_2 : V \to V$ be two elemental mappings, which are evaluated at the point $v \in V$ and let $\dot{v} \in \mathbb{R}^{n+l+m}$ be a direction for the directional derivative. The concatenation is defined as $g(v) := (\Phi_2 \circ \Phi_1)(v)$. The chain rule yields

$$\begin{aligned} \dot{v}_{(1)} :=& \frac{d\Phi_1}{dv}(v)\dot{v} \\ \dot{v}_{(2)} :=& \frac{d\Phi_2}{dv}(\Phi_i(v))\dot{v}_{(1)} \\ \frac{dg}{dv}(v)\dot{v} =& \frac{\Phi_2 \circ \Phi_1}{dv}(v)\dot{v} = \frac{d\Phi_2}{dv}(\Phi_i(v)) \cdot \frac{\Phi_1}{dv}(v)\dot{v} \end{aligned}$$

The last term in the derivative of $g(v)$ shows that the factor $\frac{\Phi_1}{dv}(v)\dot{v}$ could be substituted by $\dot{v}_{(1)}$. The result of the substitution is $\frac{d\Phi_2}{\Phi_i(v)}(dv)\dot{v}_{(1)}$ which is equivalent to the definition of $\dot{v}_{(2)}$ and therefore equivalent to the derivative of $\Phi_2(\Phi_i(v))$. The important notion of this example is that it does not matter if the whole Jacobi matrix is computed directly and then multiplied by $\dot{v}$ or if the Jacobi matrices are computed step by step and each time multiplied with the derivative direction.

This yields the basic idea on which Algorithmic Differentiation is based and it is used later to formulate the forward mode of Algorithmic Differentiation. Nevertheless, the program function F needs to be differentiated first. The Jacobian of F is computed by applying the chain rule to each term:

$$\begin{aligned} \frac{dF}{dx}(x) &= \frac{dI_{V,Y} \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \circ I_{X,V}}{dx}(x) \\ &= \frac{dI_{V,Y} \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2 \circ \Phi_1}{dv}(I_{X,V}(x))\frac{dI_{X,V}}{dx}(x) \\ &= \frac{dI_{V,Y} \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2}{dv}(\Phi_1 \circ I_{X,V}(x))\frac{d\Phi_1}{dv}(I_{X,V}(x))\frac{dI_{X,V}}{dx}(x) \\ &\vdots \\ &= \frac{dI_{V,Y}}{dv}\frac{d\Phi_k}{dv}(\Phi_{k-1} \circ \cdots \circ \Phi_1 \circ I_{X,V}(x)) \cdot \ldots \cdot \frac{d\Phi_1}{dv}(I_{X,V}(x))\frac{dI_{X,V}}{dx}(x) . \end{aligned} \tag{2.7}$$

Let $\dot{x} \in \mathbb{R}^n$ be a directional vector. As it is shown in the example with the two elemental mappings, the directional derivative of the program function F can now also be written in an iterative form. The intermediate results after each step will be denoted with $v_{(i)} \in V$ and the intermediate derivative directions as $\dot{v}_{(i)} \in \mathbb{R}^{n+m+l}$. With these abbreviations the iterative form is described as

$$\begin{array}{rclrcl}
v_{(0)} & = & I_{X,V}(x) & \dot{v}_{(0)} & = & \frac{dI_{X,V}}{dx}(x)\dot{x} \\
v_{(1)} & = & \Phi_1(v_{(0)}) & \dot{v}_{(1)} & = & \frac{d\Phi_1}{dv}(v_{(0)})\dot{v}_{(0)} \\
v_{(2)} & = & \Phi_2(v_{(1)}) & \dot{v}_{(2)} & = & \frac{d\Phi_2}{dv}(v_{(1)})\dot{v}_{(1)} \\
 & \vdots & & & \vdots & \\
v_{(i)} & = & \Phi_i(v_{(i-1)}) & \dot{v}_{(i)} & = & \frac{d\Phi_i}{dv}(v_{(i-1)})\dot{v}_{(i-1)} \\
 & \vdots & & & \vdots & \\
v_{(k)} & = & \Phi_l(v_{(k-1)}) & \dot{v}_{(k)} & = & \frac{d\Phi_k}{dv}(v_{(k-1)})\dot{v}_{(k-1)} \\
y & = & I_{V,Y}(v_{(k)}) & \dot{y} & = & \frac{dI_{V,Y}}{dv}\dot{v}_{(k)} \ .
\end{array} \tag{2.8}$$

The only difference between equation (2.7) and iteration (2.8) is that the latter only computes the directional derivative. Otherwise both state the same evaluation process. The iterative form of the process is much clearer to read and nearer to the evaluation in a computer. The iteration also states very clearly that it is not necessary to build the full Jacobi matrix $F'(x)$. It is sufficient to apply the directional derivatives to the Jacobi matrices of the intermediate steps.

The abstraction in Definition 2.7 is also valid on a computer, but the evaluation is done differently. If the whole state of the program will be copied it would be the same as creating the new vector $v_{(i)}$ from the vector $v_{(i-1)}$. In order to avoid this, procedure 2.8 needs to be simplified such that only one state vector v is used for the algorithm. Each elemental operator ϕ_i changes only the component $i+n$ of the vector. This component has never been written before because i is always increasing and the component has never been used before because each elemental operator ϕ_j with $j \leq i$ depends only upon the first $n+j-1$ elements of the state vector. It is therefore safe to just use one state vector v instead of copying the whole state vector each time. The same argument as for the primal computation can be made for the derivatives in iteration (2.8). There is no need to copy the derivative vector $\dot{v}_{(i-1)}$ to $\dot{v}_{(i)}$ only to change one component. The algorithm has to only update the derivative value of the corresponding variable. This final analysis yields the evaluation procedure in Figure 2.1 and the definition of the forward mode of Algorithmic Differentiation.

Theorem 2.10 (AD forward mode)
Let a program be given by $k \in \mathbb{N}$ elemental operations $\phi_i : V \to \mathbb{R}$ with $i = 1, ..., k$. $x \in X$ are the input variables, $y \in Y$ are the output variables and $\dot{x} \in \mathbb{R}^n$ is a direction. Then the algorithm defined in Figure 2.1 computes the directional derivative of the program at the point x in the direction $\dot{x}$.

	Function	Primal statement	Forward AD statement
Input:	$I_{X,V}(x)$	$v_i = x_i, \forall i = 1...n$	$\dot{v}_i = \dot{x}_i, \forall i = 1...n$
	$\Phi_1(v)$	$v_{n+1} = \phi_1(v)$	$\dot{v}_{n+1} = \frac{\partial \phi_1}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_1}{\partial v_n}\dot{v}_n$
	$\Phi_2(v)$	$v_{n+2} = \phi_2(v)$	$\dot{v}_{n+2} = \frac{\partial \phi_2}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_2}{\partial v_{n+1}}\dot{v}_{n+1}$
	$\vdots$		
	$\Phi_i(v)$	$v_{n+i} = \phi_i(v)$	$\dot{v}_{n+i} = \frac{\partial \phi_i}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_i}{\partial v_{n+i-1}}\dot{v}_{n+i-1}$
	$\vdots$		
	$\Phi_k(v)$	$v_{n+k} = \phi_k(v)$	$\dot{v}_{n+k} = \frac{\partial \phi_k}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_k}{\partial v_{n+k-1}}\dot{v}_{n+k-1}$
Output:	$I_{V,Y}(v)$	$y_i = v_{l-m+i}, \forall i = 1...m$	$\dot{y}_i = \dot{v}_{l-m+i}, \forall i = 1...m$

Figure 2.1.: Evaluation procedure for the AD forward mode.

If the program corresponds to the function $F : X \subset \mathbb{R}^n \to \mathbb{R}^m$, that can be represented by equation (2.6)*, then the algorithm in Figure 2.1 computes the equation*

$$\dot{y} = \frac{dF}{dx}(x)\dot{x} \tag{2.9}$$

which is the directional derivative of F in the direction $\dot{x}$.

Proof. The proof is just a recapitulation of the argumentation from the beginning of the section. Equation (2.6) is the starting point. Each component of this equation is differentiated. The Jacobi matrix of the equation is written and transformed into the iterative formulation in equation (2.8). For each elemental mapping Φ_i the first $n+i-1$ components are the identity operation and the $n+i$-th component is computed by the elemental operation ϕ_i. It is sufficient to have one vector v and update each component by the elemental operation ϕ_i. The same is true for the Jacobies of the elemental mappings. The Jacobies perform the identity operation in the first $n+i-1$ components and they update the $n+i$-th component with the derivative $\frac{d\phi_i}{dv}\dot{v}_{(i)}$. It is sufficient to update $\dot{v}_i$ with $\frac{d\phi_i}{v}\dot{v}$. The mappings $I_{X,V}$ and $I_{V,Y}$ perform identity operations and these operations can be handled by copying the values from x to v, $\dot{x}$ to $\dot{v}$, v to y and $\dot{v}$ to $\dot{y}$. □

In Figure 2.2 the forward derivatives for selected functions are shown.

Remark 2.11 ($\dot{x}$ (The dot and the x))
As each variable now has a corresponding derivative value, a meaningful name has to be found for the derivative variable. Since all possible notations of the

Elemental operation	AD forward mode
$c = a + b$	$\dot{c} = \dot{a} + \dot{b}$
$c = a * b$	$\dot{c} = b \cdot \dot{a} + a \cdot \dot{b}$
$c = \frac{1}{a}$	$\dot{c} = -\frac{1}{a^2} \cdot \dot{a}$
$c = \sin(a)$	$\dot{c} = \cos(a) \cdot \dot{a}$
$c = e^a$	$\dot{c} = e^a \cdot \dot{a}$
$c = \ln(a)$	$\dot{c} = \frac{1}{a} \cdot \dot{a}$
$c = a^p$	$\dot{c} = pv^{p-1} \cdot \dot{a}$

Figure 2.2.: AD forward mode for selected elemental operators.

letter d are already used and associated with different meanings, another notation of d would not be helpful to anybody. Therefore it is established in the AD community that the forward derivative of a variable is always denoted with a dot above that specific variable.

2.2. The reverse mode

Let $F : X \subset \mathbb{R}^n \to \mathbb{R}^m$ be the same function as in the derivation of the AD forward mode. This means that F is described by $F(x) = I_{V,Y} \circ \Phi_k \circ ... \circ \Phi_1 \circ I_{X,V}(x)$.

In order to derive the AD reverse mode, the adjoint operator [DS58, Apo69] is derived for the forward AD equation $\dot{y} = \frac{dF}{dx}(x)\dot{x}$ from Theorem 2.10. The vector $\bar{y} \in \mathbb{R}^m$ as well as the standard scalar products $\langle \cdot, \cdot \rangle_Y$ and $\langle \cdot, \cdot \rangle_X$ in $\mathbb{R}^m$ and $\mathbb{R}^n$ respectively are needed for the derivation. The adjoint operator is computed for the forward AD equation

$$\langle \bar{y}, \dot{y} \rangle_Y = \langle \bar{y}, \frac{dF}{dx}\dot{x} \rangle_Y = \langle \frac{dF}{dx}^T \bar{y}, \dot{x} \rangle_X \qquad \text{for all } \bar{y} \in \mathbb{R}^m \ .$$

The shift of $\frac{dF}{dx}$ to the left hand side changes the scalar product from $\mathbb{R}^m$ to the scalar product in $\mathbb{R}^n$. The equation holds for all $\dot{x} \in \mathbb{R}^n$ without any constraints. Therefore the equation

$$\langle \frac{dF}{dx}^T \bar{y}, \dot{x} \rangle_X = \langle \bar{x}, \dot{x} \rangle_X$$

holds for all $\dot{x} \in \mathbb{R}^n$. $\bar{x} \in \mathbb{R}^n$ can then be defined as

$$\bar{x} := \frac{dF}{dx}^T \bar{y} \ . \tag{2.10}$$

Remark 2.12 ($\bar{y}$ (The bar and the y))

The forward mode of AD introduces a new variable for each variable in the program. The same problem arises again for the reverse mode of AD. In solving this issue another symbol has had to be found in order to distinguish the reverse mode variables from the forward mode variables and the original variables. The choice of $\bar{y}$ is arbitrary but well established in the AD community.

The Jacobi matrix of F in (2.10) is analyzed. In (2.7) the Jacobi matrix for the program function is derived and is

$$\frac{dF}{dx} = \frac{dI_{V,Y}}{dv} \cdot \frac{d\Phi_k}{dv} \cdot \ldots \cdot \frac{d\Phi_1}{dv} \cdot \frac{dI_{X,V}}{dx} \ .$$

The transposed matrix is

$$\frac{dF}{dx}^T = \frac{dI_{X,V}}{dx}^T \cdot \frac{d\Phi_1}{dv}^T \cdot \ldots \cdot \frac{d\Phi_k}{dv}^T \cdot \frac{dI_{V,Y}}{dv}^T \ . \tag{2.11}$$

The Jacobi matrices are now applied in the reverse order, therefore the derivatives of the elemental operations are also evaluated in the reverse order. The question is now what happens with the elemental operators Φ_i, the mapping $I_{X,V}$ and the projection $I_{V,Y}$ when they are transposed.

For $I_{X,V}(x) = \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}$ the Jacobi matrix is:

$$\frac{dI_{X,V}}{dx} = \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} \qquad \text{it follows} \qquad \frac{dI_{X,V}}{dx}^T = \begin{pmatrix} I & 0 & 0 \end{pmatrix} \ .$$

For $I_{V,Y}(x, u, y) = y$ the Jacobi matrix is:

$$\frac{dI_{V,Y}}{dv} = \begin{pmatrix} 0 & 0 & I \end{pmatrix} \qquad \text{it follows} \qquad \frac{dI_{V,Y}}{dv}^T = \begin{pmatrix} 0 \\ 0 \\ I \end{pmatrix} \ .$$

The transposed Jacobi matrix changes the original meaning of the operators. $\frac{dI_{V,Y}}{dv}^T$ behaves as an identity projection of Y into V and $\frac{dI_{X,V}}{dx}^T$ behaves as a projection from V onto X.

The Jacobi of the elemental operation Φ_i is derived in the section for the forward mode:

$$\frac{d\Phi_i}{dv} = \begin{pmatrix} I & 0 & 0 \\ \frac{d\phi_i}{dv} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

and the transposed matrix is

$$\frac{d\Phi_i}{dv}^T = \begin{pmatrix} I & \frac{d\phi_i}{dv}^T & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \ .$$

The important notion in the transposed matrix is that the $i+n$ entry of the vector is set to zero. Exactly the same value is set in the forward mode. The product $\bar{v} = \frac{d\Phi_i^T}{dv}\bar{z}$ can be written as the update

$$\bar{v}_j = \bar{z}_j + \frac{\partial \phi_i}{\partial v_j}\bar{z}_{i+n}$$

for $j < i+n$.

Now it is clear what each Jacobi matrix from equation (2.11) means in the adjoint operator. The evaluation procedure can again be rewritten in its iterative form. The difference with respect to the forward iteration is that the the reverse statements can not be evaluated directly next to primal statements. They can only be evaluated after the primal statements. The iteration has the form

$$\begin{aligned}
v_{(0)} &= I_{X,V}(x) \\
v_{(1)} &= \Phi_1(v_{(0)}) \\
v_{(2)} &= \Phi_2(v_{(1)}) \\
&\vdots \\
v_{(i)} &= \Phi_i(v_{(i-1)}) \\
&\vdots \\
v_{(k)} &= \Phi_k(v_{(k-1)}) \\
y &= I_{V,Y}(v_{(k)})
\end{aligned}$$

$$\begin{aligned}
\bar{v}_{(k)} &= \frac{dI_{V,Y}}{dv}^T \bar{y} \\
\bar{v}_{(k-1)} &= \frac{d\Phi_k}{dv}^T (v_{(k-1)})\bar{v}_{(k)} \\
&\vdots \\
\bar{v}_{(i-1)} &= \frac{d\Phi_i}{dv}^T (v_{(i-1)})\bar{v}_{(i)} \\
&\vdots \\
\bar{v}_{(1)} &= \frac{d\Phi_2}{dv}^T (v_{(1)})\bar{v}_{(2)} \\
\bar{v}_{(0)} &= \frac{d\Phi_1}{dv}^T (v_{(0)})\bar{v}_{(1)} \\
\bar{x} &= \frac{dI_{X,V}}{dx}^T \bar{v}_{(0)} \ .
\end{aligned} \tag{2.12}$$

In a computer the evaluation is never done in such a way. The transfer from multiple states $v_{(0)}$, $v_{(1)}$, etc. to just one state v is done with the same reasoning as for the evaluation procedure in Figure 2.1. The argumentation for the vectors $\bar{v}_{(i)}$ is more involved. The information flow is reversed by the transposed gradient of the elemental operations ϕ_i. This has the consequence, that an input variable is now an output variable. Each input variable is used multiple times and therefore, the reverse iteration over the vectors $\bar{v}_{(i)}$, can update a component several times. One step in the iteration (2.12) can be denoted as the product $\bar{v} = \frac{d\Phi_i}{dv}^T(v)\bar{z}$. The step then has the form

$$\bar{v}_j = \bar{z}_j + \frac{\partial \phi_i}{\partial v_j}\bar{z}_{i+n}$$

with $j < i + n$. The value of $\bar{z}_{i+n}$ is updated during previous steps for all elemental mappings with an index that is larger than i. For the elemental mapping with the index i, the value $\bar{z}_{i+n}$ is used in the multiplication with the gradient of ϕ_i. For all elemental mappings, that have a smaller index than i, the value $\bar{z}_{i+n}$ is never used again. If only one vector $\bar{v}$ is used to represent the adjoint state of the program, the update remains correct when all elemental mappings are performed in the correct order from k to 0. Then each component of the state is assembled before being used. The update changes to

$$\bar{v}'_j = \bar{v}_j + \frac{\partial \phi_i}{\partial v_j}\bar{v}_{i+n} \ .$$

where $\bar{v}'_j$ is the updated state of the vector $\bar{v}$. The update can be written with the short notation $\mathrel{+}=$, denoting for $a \mathrel{+}= b$ the update $a' = a + b$. The final equation for the update is

$$\bar{v}_j \mathrel{+}= \frac{\partial \phi_i}{\partial v_j}\bar{v}_{i+n} \ .$$

This final equation concludes all definitions necessary to define the evaluation procedure in Figure 2.3 and to define the reverse mode of AD.

Theorem 2.13 (AD reverse mode)
Let a program be given by $k \in \mathbb{N}$ elemental operations $\phi_i : V \to \mathbb{R}$ with $i = 1 \dots k$. $x \in X$ are the input variables, $y \in Y$ are the output variables and $\bar{y} \in \mathbb{R}^m$ is a direction. Then the algorithm defined in Figure 2.3 computes the discrete adjoint of the directional derivative of the program at the point x in the adjoint direction $\bar{y}$.

If the program corresponds to the function $F : X \subset \mathbb{R}^n \to \mathbb{R}^m$, that can be represented by equation (2.6)*, then the algorithm in Figure 2.3 computes*

$$\bar{x} = \frac{dF}{dx}^T(x)\bar{y} \ .$$

Proof. The proof is just a recapitulation of the argumentation from the beginning of the section. Theorem 2.10 for the AD forward mode is the starting point. The discrete adjoint for this equation is derived and each of the transposed components are analyzed. The Jacobi evaluation in equation (2.11) is transformed into an iterative formulation. For each elemental mapping Φ_i the first $n + i - 1$ components are the identity operation and the $n + i$-th component is computed by the elemental operation ϕ_i. It is sufficient to have just one vector v and update each component by the elemental operation ϕ_i. For the transposed Jacobi of Φ_i an update is performed

	Function	Primal statement	Reverse AD statement
Input:	$I_{X,V}(x)$	$v_i = x_i \quad \forall i = 1...n$	
	$\Phi_1(v)$	$v_{n+1} = \phi_1(v)$	
	$\Phi_2(v)$	$v_{n+2} = \phi_2(v)$	
	$\vdots$		
	$\Phi_i(v)$	$v_{n+i} = \phi_i(v)$	
	$\vdots$		
	$\Phi_k(v)$	$v_{n+k} = \phi_k(v)$	
Output:	$I_{V,Y}(v)$	$y_i = v_{l-m+i} \quad \forall i = 1...m$	
Input:	$I_{V,Y}(v)$		$\bar{v}_{l-m+i} = \bar{y}_i \quad \forall i = 1...m$
	$\Phi_k(v)$		$\bar{v}_j \mathrel{+}= \dfrac{\partial \phi_k}{\partial v_j}\bar{v}_{n+k} \quad \forall j = 1...(n+k-1)$
			$\bar{v}_{n+k} = 0$
	$\vdots$		
	$\Phi_i(v)$		$\bar{v}_j \mathrel{+}= \dfrac{\partial \phi_i}{\partial v_j}\bar{v}_{n+i} \quad \forall j = 1...(n+i-1)$
			$\bar{v}_{n+i} = 0$
	$\vdots$		
	$\Phi_1(v)$		$\bar{v}_j \mathrel{+}= \dfrac{\partial \phi_1}{\partial v_j}\bar{v}_{n+1} \quad \forall j = 1...n$
			$\bar{v}_{n+1} = 0$
Output:	$I_{X,V}(x)$		$\bar{x}_i = \bar{v}_i \quad \forall i = 1...n$

Figure 2.3.: Evaluation procedure for the AD reverse mode.

Elemental operation	AD reverse mode
$c = a + b$	$\bar{a} \mathrel{+}= \bar{c}, \quad \bar{b} \mathrel{+}= \bar{c}, \quad \bar{c} = 0$
$c = a \cdot b$	$\bar{a} \mathrel{+}= b \cdot \bar{c}, \quad \bar{b} \mathrel{+}= a \cdot \bar{c}, \quad \bar{c} = 0$
$c = \frac{1}{a}$	$\bar{a} \mathrel{+}= -\frac{1}{a^2} \cdot \bar{c}, \quad \bar{c} = 0$
$c = \sin(a)$	$\bar{a} \mathrel{+}= \cos(a) \cdot \bar{c}, \quad \bar{c} = 0$
$c = e^a$	$\bar{a} \mathrel{+}= e^a \cdot \bar{c}, \quad \bar{c} = 0$
$c = \ln(a)$	$\bar{a} \mathrel{+}= \frac{1}{a} \cdot \bar{c}, \quad \bar{c} = 0$
$c = a^p$	$\bar{a} \mathrel{+}= pv^{p-1} \cdot \bar{c}, \quad \bar{c} = 0$

Figure 2.4.: AD reverse mode for selected elemental operators.

on the first $n + i - 1$ components where the value of the $n + i$-th component is used for that update. The $n+i$-th component is set to zero afterwards. The set of written variables is strictly decreasing and the variable for the update is never contained in this set. This makes it possible to perform the update on just one vector $\bar{v}$. The mappings $I_{X,V}$ and $I_{V,Y}$ perform the identity operation and these operations can be handled by copying values from x to v, $\bar{v}$ to $\bar{x}$, v to y and $\bar{y}$ to $\bar{v}$. $\square$

Reverse evaluations are shown in Figure 2.4 for a selection of elemental operations.

Remark 2.14 (Gradient evaluation with the forward and the reverse mode)
The benefits of the forward and reverse mode are best explained with small examples. Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be an arbitrary function, with $y = F(x)$. The gradient of F is computed with the forward mode and with the reverse mode. F has n inputs, so the forward mode needs to be evaluated n times in order to get the full gradient. For each evaluation the derivative direction $\dot{x}$ is set to a different unit vector. If $\dot{x}$ is set to $(1, 0, \ldots, 0)$, then the evaluation of $\dot{y} = \frac{dF}{dx}\dot{x}$ computes $\frac{\partial F}{\partial x_1}$. The same can be done for the reverse mode, here the adjoint direction $\bar{y}$ is set m-times to all unit vectors in order to build the full Jacobi matrix.

Assume that $n = 1$, $m = 1,000,000$ and that the evaluation of F takes one second are our initial assumptions. If the forward mode is used for the evaluation of the gradient then $\dot{x}$ is set to 1.0 and with one evaluation the whole gradient is computed. But if the reverse mode is used then $1,000,000$ evaluations of F and the reverse mode are needed. $\bar{y}$ is set to the first unit vector $(1, 0, \ldots, 0)$ then to the second $(0, 1, 0, \ldots, 0)$ and so on. If the additional time for the forward and reverse mode is neglected, then the computation of the gradient takes one second with the forward mode and approximately 12 days with the reverse mode.

Assume that $n = 1,000,000$ and $m = 1$. When $\bar{y}$ is set to 1.0, the reverse mode can compute the gradient of F in one sweep. On the other hand the forward mode needs to set $\dot{x}$ to all unit vectors and evaluate F for each. The forward mode needs 12 days for the evaluation and the reverse mode one second.

It is clear that the gradient computation can be done with both modes of Algorithmic Differentiation. The efficiency depends on the relationship between the number of input and output variables. In design optimization the relation is most of the time $m << n$ and therefore the reverse mode is the more appropriate choice for such cases.

Remark 2.15 (Comparison of evaluation procedures and the AD functions)
The mathematical theory makes a distinction between the elemental mapping Φ and the elemental operation ϕ. In the result of the forward mode only the elemental operation is used. The elemental mapping Φ is a helper function, and is needed to create a consistent theory. The forward mode Theorem 2.10 states the equation

$$\dot{y} = \frac{dF}{dx}(x)\dot{x} \ .$$

A look into Figure 2.1 for the theorem reveals that the update for all $\dot{v}_i$ perform the same operation. The difference is in the function and the variables on which the function is evaluated. Otherwise the same update is defined. The only equation necessary to remember for the forward mode is

$$\dot{y} = \frac{dF}{dx}(x)\dot{x} \ .$$

The same differentiation procedure is used on all levels.

The equation in the reverse mode Theorem 2.13 has nearly the same layout

$$\bar{x} = \frac{dF}{dx}^T(x)\bar{y} \ .$$

The equation in Figure 2.3 is slightly different. For each elemental operation ϕ_i the update $+=$ is performed instead of the assignment $=$. The formulation is

$$\begin{aligned} \bar{v}_j \ &+= \frac{\partial \phi_i}{\partial v_j}\bar{v}_{n+i} \quad \forall j = 1...(n+i-1) \\ \bar{v}_{n+i} \ &= 0 \ . \end{aligned}$$

This can be written in the context of F in the following way

$$\bar{x} \ += \frac{dF}{dx}\bar{y} \tag{2.13}$$

$$\bar{y} = 0 \ . \tag{2.14}$$

If $\bar{x}$ is assumed to be zero and if it is unimportant that $\bar{y}$ is set to zero, then the equation can be simplified to the one of the theorem. Equation (2.13) can therefore be used on all levels.

Remark 2.16 (Input and output dependencies in the reverse mode)
The access mode of each variable changes in the reverse evaluation. x is the

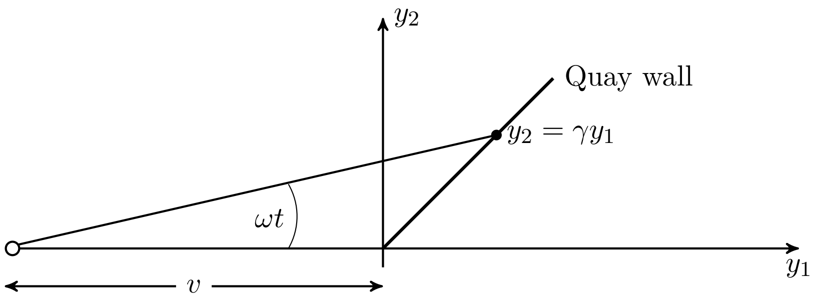


Figure 2.5.: The geometry for the lighthouse example.

input for the function F and y is the output. The reverse mode changes this so that $\bar{y}$ is the input and $\bar{x}$ is the output.

In a computer program it is often the case that the same operations are performed on each cell of a grid. The grid point (i, j) needs the surrounding grid points $(i \pm 1, j \pm 1)$ in order to be updated. In a parallel environment, the cell updates can be computed simultaneously because each grid cell is written once. The reverse mode of that algorithm can not apply the same parallel strategy because the point (i, j) is read and the points $(i \pm 1, j \pm 1)$ are written.

2.3. Examples

The theory behind the forward and reverse mode has now been introduced. To better understand how AD modifies an algorithm, AD is applied on the lighthouse example found in Griewanks book [GW08]. The function $F : \mathbb{R}^4 \to \mathbb{R}^2$ computes the light point of a light house along a quay wall. The definition is

$$\begin{aligned} y_1 =& \frac{v * \tan(\omega * t)}{\gamma - \tan \omega * t} \\ y_2 =& \frac{\gamma * v * \tan(\omega * t)}{\gamma - \tan \omega * t} \end{aligned} \tag{2.15}$$

where v is the distance of the lighthouse from the quay wall, ω the rotation speed of the lighthouse, γ the angle of the quay wall with respect to the lighthouse and t the time. The geometry is pictured in the Figure 2.5. The computational algorithm for (2.15) is shown in Figure 2.6. Common expressions are evaluated first and the code is written such that it directly represents the theory developed in the sections about the forward and reverse mode. Every line represents one elemental operator and can be handled according to the rules of Theorem 2.10 for the forward mode and Theorem 2.13 for the reverse mode.

The forward AD mode of the lighthouse example is shown in Figure 2.7. For each input and output variable an additional variable for the forward mode is added. The “_d” comes from the dot notation of the forward mode and is spoken e.g. “v dot” for the variable v. The same is done for every intermediate variable $t1$ to $t6$. The name

```
void lighthouse(double v, double w, double gamma, double t, double& y1,
    double& y2) {
  double t1 = w * t;
  double t2 = tan(t1);
  double t3 = v * t2;
  double t4 = gamma - t2;
  double t5 = t3 / t4;
  double t6 = t5 * gamma;

  y1 = t5;
  y2 = t6;
}
```

Figure 2.6.: Computational algorithm for the light house example.

of the function is extended by "_d" in order to be consistent in naming conventions. For every statement the forward AD statements is added in front of the original one. The derivation of the forward AD statement comes from the forward AD Theorem 2.10. The steps for every statement are identical. The partial derivatives with respect to every input variable are built and they are multiplied with the direction of the variables. The final step sums all the products. The resulting algorithm computes the primal value of the lighthouse example and the directional derivative. If the function is called at the point $(v, \omega, \gamma, t) = (20.0, 0.1, 2.0, 1.0)$ with the derivative direction $(\dot{v}, \dot{\omega}, \dot{\gamma}, \dot{t}) = (1.0, 0.0, 0.0, 0.0)$, the result for the derivative of the lighthouse example with respect to v is $(0.05282, 0.10563)$.

The reverse AD mode of the lighthouse example is shown in Figure 2.8. Here the variables have been added, too. The naming convention adds "_b" to the variables. This comes from the bar notation of the reverse mode and is spoken e.g. "v bar" for the variable v. It is important to note, that the bar version of the input and output variables change there meaning. v is an input variable and *v_b* is an output variable. For *y1* the opposite is true. *y1* is an output variable and *y1_b* is an input variable. This is according to the flow reversal of the reverse AD mode. See also Remark 2.16. The next difference occurs in the reverse statements, which are added after the primal evaluation in reverse order. The steps for each statement are nearly the same as for the forward mode. First the partial derivatives with respect to every input variable are built and they are multiplied against the adjoint direction of the left hand side variable. The final step updates the adjoint directions of the input variables. The resulting algorithm computes the primal value of the lighthouse example and the adjoint direction. If the function is called at the point $(v, \omega, \gamma, t) = (20.0, 0.1, 2.0, 1.0)$ with the adjoint direction $(\bar{y}_1, \bar{y}_2) = (1.0, 0.0)$, the result for the sensitivities in the lighthouse example with respect to the first output parameter y_1 are $(0.05282, 11.19582, -0.55607, 1.11958)$.

```
void lighthouse_d(double v, double v_d, double w, double w_d,
                  double gamma, double gamma_d, double t, double t_d,
                  double& y1, double& y1_d, double& y2, double& y2_d) {
  double t1_d = w * t_d + t * w_d;
  double t1 = w * t;
  double t2_d = t1_d / (cos(t1) * cos(t1));
  double t2 = tan(t1);
  double t3_d = v * t2_d + t2 * v_d;
  double t3 = v * t2;
  double t4_d = gamma_d - t2_d;
  double t4 = gamma - t2;
  double t5_d = t3_d / t4 - t3 * t4_d / (t4 * t4);
  double t5 = t3 / t4;
  double t6_d = t5 * gamma_d + gamma * t5_d;
  double t6 = t5 * gamma;

  y1_d = t5_d;
  y1  = t5;
  y2_d = t6_d;
  y2  = t6;
}
```

Figure 2.7.: Computational algorithm for the forward AD mode of the light house example. v_d stands for $\dot{v}$.

```
void lighthouse_b(double v, double& v_b, double w, double& w_b,
                  double gamma, double& gamma_b, double t, double& t_b,
                  double& y1, double y1_b, double& y2, double y2_b) {
  double t1 = w * t;
  double t2 = tan(t1);
  double t3 = v * t2;
  double t4 = gamma - t2;
  double t5 = t3 / t4;
  double t6 = t5 * gamma;
  y1  = t5;
  y2  = t6;

  double t1_b = 0.0, t2_b = 0.0, t3_b = 0.0;
  double t4_b = 0.0, t5_b = 0.0, t6_b = 0.0;
     t6_b += y2_b;                                       y2_b = 0.0;
     t5_b += y1_b;                                       y1_b = 0.0;
  gamma_b += t5 * t6_b; t5_b += gamma * t6_b;            t6_b = 0.0;
     t3_b += t5_b / t4; t4_b += -t3 * t5_b / (t4 * t4); t5_b = 0.0;
  gamma_b += t4_b; t2_b += -t4_b;                        t4_b = 0.0;
     t2_b += v * t3_b; v_b += t2 * t3_b;                 t3_b = 0.0;
     t1_b += t2_b / (cos(t1) * cos(t1));                 t2_b = 0.0;
      t_b += w * t1_b; w_b += t * t1_b;                  t1_b = 0.0;
}
```

Figure 2.8.: Computational algorithm for the reverse AD mode of the light house example. v_b stands for $\bar{v}$.

2.4. Generalization of the theory

The theory in Sections 2.1 and 2.2 restricts elemental operations to one output argument. This restriction can be lifted by generalizing the theory.

Definition 2.17 (Function operator)
A function operator is a function $\phi_{i...j} : V \to \mathbb{R}^\kappa$ with $i, j \in \{1, ..., l+m\}$ and $i <= j$. Where $\kappa = j - i + 1$ is the number of output values of this function and the operator computes the i-th to j-th operations in the program space V. The operator depends upon the first $n+i-1$ entries of the vector $v \in V$.

Definition 2.18 (Function mapping)
A function mapping is a function $\Phi_{i...j} : V \to V$ with $i, j \in \{1, ..., l+m\}$ and $i <= j$, that sets the $n+i$-th to $n+j$-th entries in the program space V. The function mapping is defined by

$$\Phi_{i...j}(v) = (v_1, \ldots, v_{n+i-1}, \underbrace{\phi_{i...j}(v)}_{\kappa}, \underbrace{0, \ldots, 0}_{l+m-j})$$

where the $n+i$-th to $n+j$-th entries are set by the function operator $\phi_{i...j}$ and the first $n+i-1$ entries stay the same.

The extension to the elemental mapping allows these operators to modify multiple elements in the vector v. The derivative of an arbitrary $\Phi_{i...j}$ is now

$$\frac{d\Phi_{i...j}}{dv} = \begin{pmatrix} I & 0 & 0 \\ \frac{d\phi_{i...j}}{dv} & 0 & 0 \\ \underbrace{0}_{n+i-1} & \underbrace{0}_{\kappa} & 0 \end{pmatrix}$$

and its transpose is

$$\frac{d\Phi_{i...j}}{dv}^T = \begin{pmatrix} I & \frac{d\phi_{i...j}}{dv}^T & 0 \\ 0 & 0 & 0 \\ \underbrace{0}_{n+i-1} & \underbrace{0}_{\kappa} & 0 \end{pmatrix} .$$

The changes to the derivatives are very small and the derivative of a function mapping can be interpreted as κ elemental mappings with special dependencies. Because of this, the extension of the theory to elemental operators with an arbitrary amount of output values is straight forward and does not need to be developed in full detail. From Chapter 3 on the definitions of elemental mappings and function mappings will be used interchangeably and will be referred to as elemental mappings.

2.5. Contracted formulation

The contracted forms of the elemental operators and function operators will be used to work with functions that have a direct representation. This is the case when implementations in software are discussed. The contracted formulations avoid the overhead to define everything in terms of the program space V and program functions. This is done by removing unnecessary dependencies.

Definition 2.19 (Contracted form of an elemental operator)
Let $\phi_i : V \to \mathbb{R}$ be an elemental operator. Then ϕ_i depends only on $o \in \mathbb{N}$ of the first $n + i - 1$ entries of the vector v, with $o \leq n + i - 1$. The contracted form of ϕ_i is written as $h : \mathbb{R}^o \to \mathbb{R}$ with $q_1, \dots, q_o \in \mathbb{N}$ such that

$$\phi_i(v) = h(v_{q_1}, \dots, v_{q_o})$$

holds for all $v \in V$.

Definition 2.20 (Contracted form of a function operator)
Let $\phi_{i\dots j} : V \to \mathbb{R}$ be a function operator. Then $\phi_{i\dots j}$ depends only on $o \in \mathbb{N}$ of the first $n + i - 1$ entries of the vector v, with $o \leq n + i - 1$. The contracted form of $\phi_{i\dots j}$ is $h : \mathbb{R}^o \to \mathbb{R}^\kappa$ with $\kappa = j - i + 1$ and $q_1, \dots, q_o \in \mathbb{N}$ such that

$$\phi_{i\dots j}(v) = h(v_{q_1}, \dots, v_{q_\kappa})$$

holds for all $v \in V$.

2.6. Example implementation and software notations

The theory for AD is established, but a clear understanding on how AD can actually be applied to a program is missing. From the theoretical derivation and the mathematical representation of AD comes a very natural method of implementation. Every statement is directly associated with a forward or reverse statement. A computer program can be modified in such a way that before each operation the forward AD expression is evaluated. For a reverse mode implementation all reverse statements are added to the end of the function. The examples in Section 2.3 applied this technique to the lighthouse example. The source code of the lighthouse example is transferred to a new source code that computes either the forward AD mode or the reverse AD mode. This approach is called *source transformation*.

A second approach is called *operator overloading* and is mostly applied to languages like C++, which support this technique. A new type is implemented that overloads all mathematical operators *operator +*, *operator **, etc. and mathematical functions *sin*, *exp*, etc. such that the additional forward or reverse AD statements are computed in the background. This type is implemented in such a way, that it can be used like a normal floating point type. The only change required by the user is the exchange of the current floating point type like *double* with the new type that performs AD in the background. This section will give a naive implementation of the forward and reverse AD mode and apply them to the lighthouse example.

This section gives a basic overview of the concepts and terminology that are used in AD implementations for operator overloading. The presented concepts will be referenced throughout the thesis as they require special consideration in some cases.

2.6.1. Forward mode

In Section 2.1 the forward mode is derived and Figure 2.1 shows the evaluation procedure. The procedure states two things clearly. First, for each primal variable a forward variable is introduced and second, the evaluation of the forward derivative can be computed next to the primal statements. The first is implemented directly in the type FReal (forward real). The structure definition is:

```
struct FReal {
  double p; \\ primal
  double d; \\ dot for dot value
}; .
```

The member p represents the original primal variable and d represents the corresponding dot variable.

The derivative statement is seen in the operator implementations. It will be shown for the multiplication and the sinus of a value, all other operators and functions can be implemented in the same way. Let the elemental function $\phi_i : V \to \mathbb{R}$ be defined as $\phi(v) = v_o \cdot v_p$ with $o, p \in \mathbb{N}$ and $o < n + i$ and $p < n + i$. Then $\dot{v}_{i+n}$ is defined as

$$\dot{v}_{i+n} = \sum_{j=1}^{i+n-1} \frac{\partial \phi_i}{\partial v_j} \dot{v}_j = v_o \cdot \dot{v}_p + v_p \cdot \dot{v}_o \ . \tag{2.16}$$

Most partial derivatives are zero. Only the indices o and p give a nonzero value. Let $h_1 : \mathbb{R}^2 \to \mathbb{R}$ be the contracted form of ϕ with $c = h_1(a, b) = a \cdot b$ then the gradient of the function is $h_1'(a, b) = (b, a)$. Therefore the forward AD equation (2.9) can be applied on h_1 and the result is

$$\dot{c} = \frac{\partial h_1}{\partial(a, b)} \begin{pmatrix} \dot{a} \\ \dot{b} \end{pmatrix} = (b, a) \begin{pmatrix} \dot{a} \\ \dot{b} \end{pmatrix} = b \cdot \dot{a} + a \cdot b \tag{2.17}$$

and is identical to equation (2.16) (See also Remark 2.15.). Equation (2.17) is now implemented in the overloading of *operator **:

```
FReal operator * (const FReal& a, const FReal& b) {
  FReal c;
  c.d = b.p * a.d + a.p * b.d;
  c.p = a.p * b.p;
  return c;
}
```

Now the elemental mapping for the sinus is analyzed. Let $h_2 : \mathbb{R} \rightarrow \mathbb{R}$ be the contracted form and defined as $c = h_2(a) = sin(a)$. The forward derivative is then

$$\dot{c} = \frac{\partial h_2}{\partial a}\dot{a} = cos(a)\dot{a} \ .$$

The implementation of the sinus is:

```
FReal sin(const FReal& a) {
  FReal c;
  c.d = cos(a.p) * a.d;
  c.p = sin(a.p);
  return c;
}
```

For each other method the implementation is done in the same way and would yield a simple but complete forward AD tool.

The *FReal* type is used on the lighthouse example from Section 2.3. With the operator overloading approach, it is enough to extend the function with a template parameter. The function can then be used with the regular *double* type or the *FReal* type. The template implementation and how it is used with the *FReal* type is shown in Figure 2.9. In line 15 the derivative value of v is set and the derivative values of y_1 and y_2 contain the derivative of the lighthouse function with respect to v. The solution is then:

```
Position (1.05634, 2.11268)
Derivative with respect to v (0.05282, 0.10563)
```

And is identical to the solution in Section 2.3.

2.6.2. Reverse mode

In Section 2.2 the reverse AD mode is derived and Figure 2.3 shows its evaluation procedure. The difference with respect to the forward procedure is that the reverse statements can not be evaluated directly next to the primal statements. They must be evaluated in reverse order after the last primal statement. The required information for each reverse statement is the state of the primal values. The state changes during the evaluation of a computer program and it is not guaranteed that the required information will still be available when the reverse statement is reached. The problem is seen in the following code:

```
1 void func(const double& a, double& b) {
2   double t1 = a*a;
3   t1 = sin(t1);
4   b = t1 / a;
5 }
```

In line 3 the value of *t1* is overwritten. If the reverse AD statements are evaluated at the end of the function, then the first value from *t1* needs to be restored. Because

```
template<typename Number>
void lighthouse(const Number& v, const Number& gamma, const Number& t,
    const Number& omega, Number& y1, Number& y2) {
  Number direction = tan(omega * t);
  y1 = v * direction / (gamma - direction);
  y2 = gamma * y1;
}

int main(int nargs, char** args) {
  FReal v = 20.0;
  FReal w = 0.1;
  FReal t = 1.0;
  FReal gamma = 2.0;
  FReal y1, y2;

  v.d = 1.0; // derivative with respect to time
  lighthouse(v, gamma, t, w, y1, y2);

  printf("Position (%.5f, %.5f)\n", y1.p, y2.p);
  printf("Derivative with respect to t (%.5f, %.5f)\n", y1.d, y2.d);
}
```

Figure 2.9.: The application of the *FReal* forward mode AD type to the lighthouse example. It computes the derivative with respect to t.

variables are constantly overwritten, the state for each statement needs to be recorded and restored.

This structure is called a tape in the AD community. The tape provides functions that store and retrieve information for the reverse evaluation process. Access to the information is strictly linear and always occurs in a stack like (aka. last in, first out (LiFo)) fashion. Each statement in the program puts some data onto the stack and when the last statement is reached the data is read in reverse order. In this thesis it is always assumed that an appropriate *Tape* implementation is available.

The second difference between the forward and reverse mode is the creation of the adjoint variables (e.g. $\bar{a}$, $\bar{v}$, etc.). They can not be stored in the structure for the reverse type, because the variables will run out of scope and are not accessible in the reverse evaluation procedure. The scope problem can be seen in the following code:

```
1 void(const double& a, double& b) {
2   double t1;
3   {
4     double t2 = a * a;
5     t1 = sin(t2);
6   }
7   b = t1 / a;
8 }
```

After variable *t2* runs out of scope in line 6 it will be deleted and all information that is stored inside the new AD type will be lost. The solution is already shown in the reverse mode of Procedure 2.3. Each statement has a unique index which is used to access the state vectors v and $\bar{v}$. The introduction of the index to the reverse type structure will allow for the identification of each variable and can be used in the reverse process to access a vector containing the adjoint variables. The structure for the reverse type *RReal* (reverse real) is defined as:

```
struct RReal {
 double p; // primal
 int  i; // index
}
```

The implementation for the elemental operations will be shown for the multiplication and the sinus of a value, all other operators and functions can be implemented in the same way. Let the elemental operator $\phi_i : V \to \mathbb{R}$ be defined as $\phi_i(v) = v_o * v_p$ with $o, p \in \mathbb{N}$, $o < i + n$ and $p < i + n$. The adjoint equation for ϕ_i is defined as

$$\begin{aligned} \bar{v}_j \mathrel{+}=& \frac{\partial \phi_i}{\partial v_i} \bar{v}_{i+n} \quad \forall j = 1 \dots i + n - 1 \\ \bar{v}_{i+n} =& 0 \end{aligned}$$

which yields

$$\begin{aligned} \bar{v}_o \mathrel{+}=& v_o \cdot \bar{v}_{i+n} \\ \bar{v}_p \mathrel{+}=& v_p \cdot \bar{v}_{i+n} \\ \bar{v}_{i+n} =& 0 \ . \end{aligned}$$

Let h_1 be the contracted form of $\phi_i(v)$ and be defined as $h_1 : \mathbb{R}^2 \to \mathbb{R}$ with $c = h_1(a, b) = a \cdot b$. Then the gradient of the function is $h_1'(a, b) = (b, a)$ and if the reverse equation (2.13) is applied to h_1, the result is

$$\begin{aligned} \begin{pmatrix} \bar{a} \\ \bar{b} \end{pmatrix} \mathrel{+}=& \frac{\partial h_1}{\partial(a, b)}^T \bar{c} = \begin{pmatrix} b \\ a \end{pmatrix} \bar{c} \\ \bar{c} =& 0 \ . \end{aligned}$$

Therefore the results are the same for both approaches. (See also Remark 2.15.) The information required in the reverse mode are the values of a, b and the indices of $\bar{a}$, $\bar{b}$ and $\bar{c}$. The overloaded operator in Figure 2.10 stores this information on the tape and generates a new index for c. The *MUL* identifier is required by the implementation in order to know which function needs to be called in the reverse evaluation procedure, and is additionally stored on the tape. Figure 2.11 shows how this is handled in the case of a lookup table or in the case of a function pointer. It is important that the information about the operation type be stored last such that in the reverse sweep this information is the first available. The lookup table variant is more convenient if a fixed set of operations exists. The function pointer implementation is more appropriate if lots of functions exist which are not known in advance. The actual implementation of the reverse evaluation of the multiplication

```
RReal operator * (const RReal& a, const RReal& b) {
  tape.push(a.p);
  tape.push(b.p);
  tape.push(a.i);
  tape.push(b.i);

  RReal c;
  c.p = a.p * b.p;
  c.i = ++tape.globalStatementCounter;

  tape.push(c.i);
  tape.push(MUL);

  return c;
}
```

Figure 2.10.: Basic implementation of the overload multiplication operator for the reverse AD mode. It stores the required primal values, the indices for the adjoint variables and the indicator for the operation.

```
//Lookup table:
  ...
  ID id = tape.pop<ID>();
  if(MUL == id) {operator_mul_b(adjoints);}
  ...
//Function pointer:
  ...
  FUNC func = tape.pop<FUNC>();
  func(adjoints);
  ...
```

Figure 2.11.: Example code for calling the reverse evaluation functions either via a lookup table or a function pointer.

is shown in Figure 2.12. As the *MUL* identifier is already popped to identify the operator and call the function, it does not need to be popped inside of the function. The other information is retrieved in the reverse order and the values are stored in the global adjoint vector which is equivalent to $\bar{v}$ in the Algorithm 2.3.

The implementation for the sinus function is done in a similar fashion. If the function $h_2 : \mathbb{R} \to \mathbb{R}$ is defined as $c = h_2(a) = sin(a)$, then the reverse equation is

$$\begin{aligned} \bar{a} \mathrel{+}=& cos(a)\bar{c} \\ \bar{c} =& 0 \ . \end{aligned}$$

The implementation in the Figures 2.13 and 2.14 is analog to the implementation of the multiplication.

For every other method the implementation is analog and yields a simple but complete reverse AD tool.

```
void operator_mul_b(double* adjoint) {
  int c_i = tape.pop<int>();
  int b_i = tape.pop<int>();
  int a_i = tape.pop<int>();
  double b_p = tape.pop<double>();
  double a_p = tape.pop<double>();

  adjoint[a_i] += b_p * adjoint[c_i];
  adjoint[b_i] += a_p * adjoint[c_i];
  adjoint[c_i] = 0.0;
}
```

Figure 2.12.: Basic implementation for the reverse evaluation of the multiplication for the reverse AD mode. All the data is retrieved from the tape and the reverse equation is then evaluated. The operation indicator is already accessed in the calling function (see Figure 2.11).

```
RReal sin(const RReal& a) {
  tape.push(a.p);
  tape.push(a.i);

  RReal c;
  c.p = sin(a.p);
  c.i = ++tape.globalStatementCounter;

  tape.push(c.i);
  tape.push(SIN);

  return c;
}
```

Figure 2.13.: Basic implementation of the overload sinus function for the reverse AD mode. It stores the required primal value, the indices for the adjoint variables and the indicator for the operation.

```
void operator_sinl_b(double* adjoint) {
  int c_i = tape.pop<int>();
  int a_i = tape.pop<int>();
  double a_p = tape.pop<double>();

  adjoint[a_i] += cos(a_p) * adjoint[c_i];
  adjoint[c_i] = 0.0;
}
```

Figure 2.14.: Basic implementation of the reverse evaluation of the sinus for the reverse AD mode. All data is retrieved from the tape and the reverse equation is then evaluated. The operation indicator is already accessed in the calling function (see Figure 2.11).

The use of the *RReal* implementation is shown with the lighthouse example (2.15). Figure 2.15 shows its implementation. The changes with respect to the forward mode are obvious. All inputs v, ω, t and γ are registered on the tape such that they have indices. The function would look like:

```
void registerInput(RReal& a) {
 a.i = ++globalStatementCounter;
}
```

The global statement counter is increased and the value retrieves the new index. Now that all input variables are marked, the function call to *lighthouse* records the operations on the tape. Because the values y_1 and y_2 are outputs, they will have indices after the function finishes. The adjoint direction of y_1 is set to one and the reverse procedure is evaluated.

The evaluation method is usually only a loop over all the statements. A simple implementation would look like:

```
void evaluate() {
 for(int curStmt = globalStatementCounter; curStmt > 0; --curStmt) {

   ID id = this->pop<ID>();
   switch(id) {
     case MUL: operator_mul_b(this->adjoints); break;
     case SIN: operator_sin_b(this->adjoints); break;
     ...
   }
 }
}
```

All statements are evaluated in the reverse order as it is required by the reverse AD mode. This will populate the adjoint vector with the sensitivities for the first output value. These sensitivities can be accessed through the indices of the input values. The functions for the setting and getting of the adjoint values would look like:

```
void setAdjoint(RReal& a, const double& value) {
 adjointVec[a.i] = value;
}

double getAdjoint(RReal& a) {
 return adjointVec[a.i];
}
```

The solution is then:

```
Position (1.05634, 2.11268)
Gradient with respect to y_1 (0.05282, 11.19582, 1.11958, -0.55607)
```

Which is the same as the solution in Section 2.3

The naive implementation of the *RReal* gives an understanding of what an operator overloading tool has to implement. All functions in the implementation are simple and straight to the point. A real reverse mode AD tool needs to implement a lot

```
template<typename Number>
void lighthouse(const Number& v, const Number& gamma, const Number& t,
    const Number& omega, Number& y1, Number& y2) {
  Number direction = tan(omega * t);
  y1 = v * direction / (gamma - direction);
  y2 = gamma * y1;
}

int main(int nargs, char** args) {
  RReal v = 20.0;
  RReal w = 0.1;
  RReal t = 1.0;
  RReal gamma = 2.0;
  RReal y1, y2;

  tape.registerInput(v, w, t, gamma); // generate indices for the input
  lighthouse(v, gamma, t, w, y1, y2);
  tape.setAdjoint(y1, 1.0);
  tape.evaluate();

  printf("Position (%.5f, %.5f)\n", y1.p, y2.p);
  printf("Gradient with respect to y_1 (%.5f, %.5f, %.5f, %.5f)\n",
      tape.getAdjoint(v), tape.getAdjoint(w), tape.getAdjoint(t),
      tape.getAdjoint(gamma));
}
```

Figure 2.15.: The application of the *RReal* reverse mode AD type to the lighthouse example. Computing the gradient with respect to the output y_1.

of additional functionality, increasing the ease of its use. Nevertheless, the general notions stay the same. Input variables of the process need to be made available to the AD tool, a function *registerInput* is used for this purpose. The output variables do not need to be treated in a special way for the naive implementation but in general they also need to be marked with a function, called *registerOutput*. It is also common to have multiple tapes indicating, that only portions of the program are recorded, as opposed to the complete program as a whole. The sections are marked with methods like *setActive* and *setPassive*.

The major difference between the naive implementation with respect to other AD tools is in the management of data. The naive implementation stores everything and does not care if the information is stored more than once. The index also increases every new statement. The variable could be used e.g. in a loop:

```
RReal c = 0.0;
for(int i = 0; i < 100; ++i) {
  RReal t = a[i]*a[i] + b[i]*b[i]
  c += sqrt(t);
}
```

Each time t is assigned, a new index is generated and the adjoint vector grows by one entry. This entry will only be used once in the reverse interpretation. It would be better if indices from the deleted variables were reused. This technique is called “index reuse”. The other indexing scheme is called “linear indexing”.

The naive implementation will be extended and improved in Chapter 6. Until then the basic theory and notion of AD, that is described in this chapter is enough to apply AD tools to large scale software and produce derivative results.

3. Derivative algorithms for fixed-point iterators

This chapter takes a look at the derivation of different procedures for the computation of sensitivities. The general framework builds onto the minimization problem

$$\begin{aligned} \min_{x \in X} \quad & f(y, x) \\ \text{s.t. } & 0 = R(y, x) \quad (\Leftrightarrow y = G(y, x)) \end{aligned} \tag{$\mathcal{P}$}$$

where $x \in X \subset \mathbb{R}^n$ is the control vector for the minimization. The minimization is described by the objective function $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$ which defines the set of optimal values for the problem. In addition, the problem is solved under the constraint $0 = R(y, x)$, which defines the state $y \in Y \subset \mathbb{R}^m$ as an implicit function of the control x with the residual operator $R : Y \times X \to \mathbb{R}^m$. The residual operator is usually generated by the discretisation of a partial differential equation (PDE). The PDE is solved when the residual operator equals to zero, that is $R(y_*, x) = 0$ where y_* describes the solution. The solution to the PDE can also be described as a fixed-point iterator $G : Y \times X \to Y$. Here, the solution is reached when the fixed-point iterator has reached its fixed-point, $y_* = G(y_*, x)$. Both formulations, the residual formulation and the fixed-point formulation, are equivalent and can be rewritten into each other.

A procedure for the solution of ($\mathcal{P}$) can be obtained via the discrete adjoint system, well known in the literature [AB99, EP97, NA99, PD10], and is given by

$$\left(\frac{\partial R(y_*, x)}{\partial y}\right)^T \psi = \left(\frac{\partial f(y_*, x)}{\partial y}\right)^T \quad , \tag{3.1}$$

where $\psi \in \mathbb{R}^m$ describes the adjoint state. With the solution ψ_* the sensitivity of the objective function with respect to the control x can then be computed by

$$\frac{df(y_*, x)}{dx} = \frac{\partial f(y_*, x)}{\partial x} - \psi_*^T \frac{\partial R(y_*, x)}{\partial x} \ . \tag{3.2}$$

The gradient vector $\frac{df}{dx}$ can be used to solve the minimization problem ($\mathcal{P}$) by applying for example the steepest descent method [NW99].

The general adjoint method is mostly used when people talk of the adjoint formulation or the adjoint method. Unfortunately the method explicitly requires that $0 \overset{!}{=} R(y_*, x)$ is fulfilled, that is the solution process for y is converged to machine precision. In most industrial applications a residual of 10^{-3} is considered to be accurate enough and the solution process is stopped at this point or it is not possible to obtain a solution where the residual is smaller. If such a disturbed solution $\hat{y}_*$ were to be used in equations (3.1) and (3.2), the adjoint solution ψ_* will also be disturbed and it can no longer be guaranteed that the gradient $\frac{df}{dx}(\hat{y}_*, x)$ will be close to the correct gradient $\frac{df}{dx}(y_*, x)$.

This chapter introduces new methods for the computation of $\frac{df}{dx}(y_*, x)$ that do not require a solution y_* to be solved to machine precision. The Black Box (BB) method is derived with the help of AD techniques and can be used to provide the gradients for arbitrarily good solutions. The other method is the Reverse Accumulation (RA) method, derived from the Lagrange formulation. It is not as general as the Black Box method but still represents a very general approach. The Reverse Accumulation method is specialized under various assumptions, such that the current state of the art methods are recovered and finally even the most specialized method will yield the formulation from equation (3.1).

3.1. Algorithmic Differentiation based algorithms

The first adjoint procedure is derived with AD. In a numerical program the objective function is computed by performing j steps of the fixed-point solver G and afterwards f is evaluated. The procedure which performs this computation is given in Algorithm 1. Each statement in the procedure can be interpreted as an elemental operation and can be differentiated with the help of AD techniques. The result is shown in Algorithm 2. In order to compute the full gradient $\frac{df}{dx}$ the adjoint of the output $\bar{z}$ is set to 1.0. The total derivative of f with respect to the control x is then described by the adjoint control $\bar{x}_0$. This method is called the *Black Box (BB)* method, and is summarized in Figure 3.1.

The Black Box method has no requirements with respect to convergence. It reverses the evaluation of the primal solver and computes the derivatives with respect to the control. Therefore it has no robustness issues nor any prerequisite on the primal computation. Nevertheless, if the primal computation is not robust enough or contains computational errors then the sensitivities with respect to the control will be numerically correct but their relevance is questionable.

Beside its robustness and exactness the Black Box approach has one drawback. The reverse sweep needs all vectors y_i in order to evaluate the Jacobi vector product. This can be handled by storing every vector y_i during the forward evaluation or with more advanced techniques like loop checkpointing from Chapter 5.

3.2. Lagrange based algorithms

Another approach is to setup the Lagrangian function

$$L(y, x, \lambda) = f(y, x) + (G(y, x) - y)^\top \lambda \tag{3.3}$$

Algorithm 1 Procedure for evaluating the objective function f for a given fixed-point iterator G.

```
Input: $y_0, x$
Output: $z, y_{j+1}$
  for $i = 0, \dots, j$ do
    $y_{i+1} = G(y_i, x)$
  end for
  $z = f(y_{j+1}, x)$
```

Algorithm 2 Procedure for evaluating the gradient of the objective function f for a given fixed-point iterator G.

```
Input: $y_0, x$
Output: $z, y_{j+1}, \bar{x}_0$
  for $i = 0, \dots, j$ do
    $y_{i+1} = G(y_i, x)$
  end for
  $z = f(y_{j+1}, x)$

  $\bar{z} = 1$
  $\bar{y}_{j+1} = \frac{\partial f}{\partial y}(y_{j+1}, x))^T \bar{z}$
  $\bar{x}_{j+1} = \frac{\partial f}{\partial x}(y_{j+1}, x))^T \bar{z}$
  for $i = j, \dots, 0$ do
    $\bar{y}_i = \frac{\partial G}{\partial y}(y_i, x)^T \bar{y}_{i+1}$
    $\bar{x}_i = \frac{\partial G}{\partial x}(y_i, x)^T \bar{y}_{i+1} + \bar{x}_{i+1}$
  end for
```

$$\begin{tikzcd}
y_0 \arrow[r, "G(y_0, x)"] & y_1 \arrow[r, "G(y_1, x)"] & y_2 \arrow[r, "\cdots"] & y_j \arrow[r, "G(y_j, x)"] & y_{j+1} \arrow[r, "f(y_{j+1}, x)"] & z \arrow[d] \\
\begin{pmatrix} \bar{y}_0 \\ \bar{x}_0 \end{pmatrix} & \begin{pmatrix} \bar{y}_1 \\ \bar{x}_1 \end{pmatrix} \arrow[l, "\frac{\partial G}{\partial y}(y_0, x)^T \bar{y}_1"', "\substack{\frac{\partial G}{\partial x}(y_0, x)^T \bar{y}_1 \\ + \bar{x}_1}"] & \begin{pmatrix} \bar{y}_2 \\ \bar{x}_2 \end{pmatrix} \arrow[l, "\frac{\partial G}{\partial y}(y_1, x)^T \bar{y}_2"', "\substack{\frac{\partial G}{\partial x}(y_1, x)^T \bar{y}_2 \\ + \bar{x}_2}"] & \begin{pmatrix} \bar{y}_j \\ \bar{x}_j \end{pmatrix} \arrow[l, "\cdots"', "\cdots"] & \begin{pmatrix} \bar{y}_{j+1} \\ \bar{x}_{j+1} \end{pmatrix} \arrow[l, "\frac{\partial G}{\partial y}(y_j, x)^T \bar{y}_{j+1}"', "\substack{\frac{\partial G}{\partial x}(y_j, x)^T \bar{y}_{j+1} \\ + \bar{x}_{j+1}}"] & 1 \arrow[l, "\frac{\partial f}{\partial y}(y_{j+1}, x)^T \bar{z}"', "\frac{\partial f}{\partial x}(y_{j+1}, x)^T \bar{z}"]
\end{tikzcd}$$

Figure 3.1.: *Black Box (BB)*: Procedure for calculating the gradient of the objective function.

for problem ($\mathcal{P}$). The equations for the optimality (KKT) [KT51] conditions can be derived. They are

$$0 \overset{!}{=} \frac{\partial L}{\partial \lambda}(y_*, x)^T = G(y_*, x) - y_* \tag{3.4}$$

$$0 \overset{!}{=} \frac{\partial L}{\partial y}(y_*, x)^T = \frac{\partial f}{\partial y}(y_*, x)^T + \frac{\partial G}{\partial y}(y_*, x)^T \lambda_* - \lambda_* \tag{3.5}$$

$$0 \overset{!}{=} \frac{\partial L}{\partial x}(y_*, x)^T = \frac{\partial f}{\partial x}(y_*, x)^T + \frac{\partial G}{\partial x}(y_*, x)^T \lambda_* \; . \tag{3.6}$$

The first equation reduces to the solution of the primal fixed-point equation

$$G(y_*, x) = y_* \; . \tag{3.7}$$

The fixed-point formulation is solved by performing the update

$$y_{i+1} = G(y_i, x) \tag{3.8}$$

until the fixed-point $y_N = y_*$ is reached. The second equation can also be transformed such that it resembles the solution of a fixed-point equation

$$\lambda_* = \frac{\partial f}{\partial y}(y_*, x)^T + \frac{\partial G}{\partial y}(y_*, x)^T \lambda_* \tag{3.9}$$

for the adjoint parameter λ. If both equations are solved for y_* and λ_*, then the sensitivities can be evaluated with

$$\frac{df}{dx}(y_*, x)^T = \frac{\partial f}{\partial x}(y_*, x)^T + \frac{\partial G}{\partial x}(y_*, x)^T \lambda_* \; . \tag{3.10}$$

The form of equation (3.9) suggests that it can be iteratively solved by a fixed-point iteration of the form

$$\lambda_{j+1} = \frac{\partial f}{\partial y}(y_*, x)^T + \frac{\partial G}{\partial y}(y_*, x)^T \lambda_j \; . \tag{3.11}$$

Equation (3.11) can be written in a shortened notation, by utilizing the definition of the shifted Lagrangian $N(y, x, \lambda) := f(y, x) + G(y, x)^T \lambda$, as

$$\lambda_{j+1} = \frac{\partial N}{\partial y}(y_*, x, \lambda_j) \; . \tag{3.12}$$

The convergence property of the equation can be derived after [Gil87] by ensuring that the spectral radius of the Jacobi matrix with respect to the iteration parameter is less than one. As G is assumed to be a strictly contracting fixed-point iterator, it holds $\left\| \frac{\partial G}{\partial y}(y, x) \right\| = \sigma < 1$ for all $y \in \mathbb{R}^m$, where $\sigma < 1$ is the contraction rate. The norm of the Jacobi matrix of (3.12) with respect to the iteration parameter λ is

$$\left\| \frac{\partial^2 N}{\partial y \partial \lambda}(y_*, x, \lambda_j) \right\| = \left\| \frac{\partial G}{\partial y}(y_*, x)^T \right\| = \left\| \frac{\partial G}{\partial y}(y_*, x) \right\| = \sigma < 1 \; . \tag{3.13}$$

$$\begin{tikzcd}
y_0 \arrow[r, "G(y_0, x)"] & y_1 \arrow[r, "G(y_1, x)"] & y_2 \arrow[r, "\dots"] & y_j \arrow[r, "G(y_j, x)"] & y_{j+1} = y_* \\
\lambda_{J+1} & \lambda_J \arrow[l, "\frac{\partial N}{\partial y}(y_*, x, \lambda_J)"'] & \lambda_{J-1} \arrow[l, "\frac{\partial N}{\partial y}(y_*, x, \lambda_{J-1})"'] & \lambda_1 \arrow[l, "\dots"'] & \lambda_0 \arrow[l, "\frac{\partial N}{\partial y}(y_*, x, \lambda_0)"']
\end{tikzcd}$$

Figure 3.2.: *Reverse Accumulation (RA)*: Procedure for calculating the discrete adjoint with a Lagrangian approach.

Equation (3.12) is a strictly contracting fixed-point iteration if and only if G is a strictly contracting fixed-point iteration. Because of the initial assumption that is always the case.

The difference between the very first adjoint formulation in equation (3.1) and the formulation given in equation (3.9), is that equation (3.1) is derived on the residual $R(y, x)$. Equation (3.9) depends more strongly upon the solution method used in the simulation and also covers the implementation details of the simulation software. Giles [Gil01] and Nielson et al. [NLPD04] state that everything that is good for the solution of the primal equation is also good for the solution of the adjoint equation. The ansatz with G in combination with AD drives this statement to the limit and yields a formulation that is always consistent with respect to the primal solution.

From the primal iteration (3.8) and the adjoint iteration (3.12), the second approach is derived for the computation of the control derivatives. It is given in Figure 3.2 and is usually referred to as the *Reverse Accumulation (RA)* procedure [Chr92, Chr94].

The advantage of this approach is that it does not need to save every y_i in order to compute the adjoint λ. The Reverse Accumulation approach on the other hand, requires that the PDE solver reaches its fixed-point $y_* = G(y_*, x)$ which is usually not the case. In practice the residual can not be reduced under a certain level e.g. 10^{-3} and never fully converges to zero or to machine precision 10^{-16}. Under this perturbation it is no longer guaranteed that the Reverse Accumulation procedure will yield a correct result. The result from this observation is, that the Reverse Accumulation procedure is less robust and exact than the Black Box procedure. This will also be seen by the numerical analysis in Chapter 7.

The fixed-point solver G is now specialized in order to reflect common solution strategies in the implementation of PDE solvers. It is defined as a quasi Newton step

$$y_{i+1} = G_{QN}(y_i, x) := y_i - P(y_i, x)R(y_i, x) \quad . \tag{3.14}$$

The preconditioner $P(y, x)$ can be chosen in several ways. For example P can represent the Newton method:

$$P_{Newton}(y, x) = \left(\frac{\partial R(y, x)}{\partial y}\right)^{-1} \tag{3.15}$$

or the backward Euler method:

$$P_{Euler}(y,x) = \left[\frac{\Omega}{\Delta t} I + \left(\frac{\partial R(y,x)}{\partial y}\right)\right]^{-1} \quad , \tag{3.16}$$

where $\frac{\Omega}{\Delta t}$ describes the space time discretisation. In order to insert equation (3.14) into equation (3.11) the derivative of G_{QN} with respect to y is required. The full derivative of equation (3.14) in an arbitrarily chosen point y is then

$$\frac{\partial G_{QN}}{\partial y}(y,x) = I - \frac{\partial P}{\partial y}(y,x) R(y,x) - P(y,x)\frac{\partial R}{\partial y}(y,x) \quad . \tag{3.17}$$

The only method to eliminate the derivative $\frac{\partial P}{\partial y}(y,x)$, requires the assumption that y and y_* are equal which yields $R(y_*,x) = 0$. But as stated above this method introduces an error to the adjoint computation as the condition $R(y_*,x) = 0$ is never reached in most real world applications. Nevertheless, if this assumption is made, then equation (3.17) will reduce to

$$\frac{\partial G}{\partial y}(y_*,x) = I - P(y_*,x)\frac{\partial R}{\partial y}(y_*,x) \quad . \tag{3.18}$$

Transposing $\frac{\partial G}{\partial y}(y_*,x)$ and multiplying by λ_j yields:

$$\frac{\partial G_{QN}}{\partial y}^T(y_*,x)\lambda_j = \lambda_j - \frac{\partial R}{\partial y}^T(y_*,x)P^T(y_*,x)\lambda_j \quad . \tag{3.19}$$

This term is then inserted into equation (3.11)

$$\lambda_{j+1} = \frac{\partial f}{\partial y}^T(y_*,x) + \lambda_j - \frac{\partial R}{\partial y}^T(y_*,x)P^T(y_*,x)\lambda_j \quad . \tag{3.20}$$

The same procedure is performed for the derivative with respect to x. It is assumed that $R(y_*,x) = 0$ holds, which gives

$$\frac{\partial G}{\partial x}(y_*,x) = -P(y_*,x)\frac{\partial R}{\partial x}(y_*,x) \quad . \tag{3.21}$$

This term can then be applied to equation (3.10), yielding

$$\frac{df}{dx}^T(y_*,x) = \frac{\partial f}{\partial x}^T(y_*,x) - \frac{\partial R}{\partial x}^T(y_*,x)P^T(y_*,x)\lambda_* \; . \tag{3.22}$$

In order to use the result from equation (3.20) in equation (3.22) it needs to be multiplied by $P^T(y_*,x)$. After some reordering, the new formulation can be written as

$$P^T(y_*,x)\lambda_{j+1} = P^T(y_*,x)\lambda_j + P^T(y_*,x)\left(\frac{\partial f}{\partial y}^T(y_*,x) - \frac{\partial R}{\partial y}^T(y_*,x)P^T(y_*,x)\lambda_j\right) \; . \tag{3.23}$$

$$\begin{tikzcd}
y_0 \arrow[r, "G(y_0, x)"] & y_1 \arrow[r, "G(y_1, x)"] & y_2 \arrow[r, "..."] & y_j \arrow[r, "G(y_j, x)"] & y_{j+1} = y_* \\
\lambda_{J+1} & \lambda_J \arrow[l, "\tilde{G}(y_*, x, \lambda_J)"'] & \lambda_{J-1} \arrow[l, "\tilde{G}(y_*, x, \lambda_{J-1})"'] & \lambda_1 \arrow[l, "..."'] & \lambda_0 \arrow[l, "\tilde{G}(y_*, x, \lambda_0)"']
\end{tikzcd}$$

Figure 3.3.: *Giles' Exact Dual (GED)*: Procedure for calculating the discrete adjoint via the Lagrangian approach and using the special structure of G defined in (3.14). The assumption $R(y_*, x) = 0$ has to hold for this iteration to be correct. $\tilde{G}$ is defined in (3.24).

The term $P^T(y_*, x)\lambda$ is now used throughout the equations (3.23) and (3.22). Both equations can be simplified by defining a new parameter $\tilde{\lambda} := P^T(y_*, x)\lambda$. The two equations then become

$$\tilde{\lambda}_{j+1} = \tilde{G}(y_*, x, \tilde{\lambda}) := \tilde{\lambda}_j + P^T(y_*, x)\left[\frac{\partial f}{\partial y}^T(y_*, x) - \frac{\partial R}{\partial y}^T(y_*, x)\tilde{\lambda}_j\right] \tag{3.24}$$

$$\frac{df}{dx}^T(y_*, x) = \frac{\partial f}{\partial x}^T(y_*, x) - \frac{\partial R}{\partial x}^T(y_*, x)\tilde{\lambda}_* \ . \tag{3.25}$$

The equations (3.24) and (3.25) now represent a simplified form of the Reverse Accumulation approach, and were obtained by exploiting the structure of the iterator G with the assumption that $R(y_*, x) = 0$ holds. Figure 3.3 shows this simplified version of the approach. It is usually referred to as the *Giles' Exact Dual* approach. This approach has several requirements that need to be met in order to qualify for an exact adjoint method. The first requirement is the special structure of equation (3.14) for G. Usually each fixed-point solver has this structure, so the requirement is not very restrictive. The other requirement involves the residual R. It has to hold that $R(y_j \equiv y_*, x) \overset{!}{=} 0$, this is used to eliminate the derivative of P with respect to y. Therefore, the exact adjoint can only be calculated if the residual can be driven to zero.

If the Newton update $P(y_*, x) = P_{Newton}(y_*, x) = \frac{\partial R}{\partial y}^{-1}(y_*, x)$ is chosen as the preconditioner, the iteration can be further simplified. It is important to emphasize that P has to be the exact Newton step and no errors or inconsistencies in the calculation of $\frac{\partial R}{\partial y}(y, x)$ can be made, otherwise the simplification will be wrong. If P is equal to P_{Newton}, then (3.24) further simplifies to

$$\frac{\partial R}{\partial y}^T(y_*, x)\tilde{\lambda} = \frac{\partial f}{\partial y}^T(y_*, x) \quad . \tag{3.26}$$

It is easy to see that equation (3.26) is exactly the same adjoint equation as (3.1). In this thesis it will be refereed to as the *General Adjoint* approach for evaluating the discrete adjoint. Figure 3.4 summarizes this approach.

$$\begin{tikzcd}
y_0 \arrow[r, "G(y_0, x)"] & y_1 \arrow[r, "G(y_1, x)"] & y_2 \arrow[r, "\dots"] & y_j \arrow[r, "G(y_j, x)"] & y_{j+1} = y_*
\end{tikzcd}$$

$$\frac{\partial R}{\partial y}^T (y_*, x)\tilde{\lambda} = f_y^T (y_*, x)$$

Figure 3.4.: *General Adjoint (GA)*: Procedure for calculating the discrete adjoint from the Lagrangian approach and using the special structure of G defined in (3.14). The assumption $R(y_*, x) = 0$ has to hold such that this iteration is correct. $P(y, x)$ has to be $\frac{\partial R}{\partial y}(y, x)^{-1}$ which is equivalent to the exact Newton method.

Four different methods for calculating the discrete adjoint have been derived. Each method is a valid adjoint calculation and can produce the exact adjoint but some require special conditions. The Black Box method has no special conditions on the primal convergence and is exact for every combination of input values, making it the most robust method with respect to the required conditions. The Reverse Accumulation and Giles' Exact Dual methods explicitly require that the fixed point iteration be at its fixed point. This means for *Giles' Exact Dual* that $R(y_*, x) = 0$ is fulfilled. This requirement makes these methods less robust. Most of the time it is not guaranteed that the steady state of the fixed-point iteration can be reached or that the residual R is close to zero. The General Adjoint method is only meaningful if the primal iteration uses a Newton step and $R(y_*, x) = 0$ holds, otherwise the error of the adjoint computation will be substantial. Table 3.1 shows a summary of the 4 different methods and their requirements.

This chapter can be concluded with the fact that several approaches for the adjoint computation are available. Nielson uses in [NA99] a similar method to the General Adjoint method in order to solve the adjoint system for turbulent flows using relatively coarse grids and simple geometries. His work successfully demonstrates the accuracy of the method. However, the scheme failed to converge for many problems, which points out the robustness issues of such algorithms. By employing a more extensive preconditioner for the Krylov algorithm in [NA02], Nielson demonstrates improved performance. However, this preconditioning strategy requires approximately five times the memory of the baseline analysis scheme. This approach is unfeasible for large-scale applications requiring grids that contain several million mesh points. As a solution to these problems, Nielsen et al. present an algorithm that preserves discrete duality in [NLPD04], following Giles' Exact Dual approach. In this algorithm the adjoint solution is solved in a similar manner to the nonlinear solver using pseudo time-stepping. From the presented papers the common agreement is that the solution method chosen for the adjoint equation is of extreme importance concerning the convergence behavior and the robustness. This emphasizes that the Reverse Accumulation procedure or Black Box procedure from this chapter are the most robust methods presented as there are no inconsistencies in either the implementation nor the modeling.

Method	Derived from	Requirements	References
Black Box(BB)	Nonlinear Solver G	-	Alg. 1, Alg. 2, Fig. 3.1
Reverse Accumulation(RA)	Lagrange function	$G(y_*, x) \overset{!}{=} y_*$	(3.8), (3.9), Fig. 3.2
Giles' Exact Dual(GED)	*RA*	$G(y, x) = y - P(y, x)R(y, x)$ $R(y_*, y) \overset{!}{=} 0$	(3.14), (3.24), Fig. 3.3
General Adjoint(GA)	*GED*	$G(y, x) = y - P(y, x)R(y, x)$ $R(y_*, x) \overset{!}{=} 0$ $P \equiv P_{Newton}$	(3.26), Fig. 3.4

Table 3.1.: List of the different methods and their requirements to be exact.

4. Differentiation of large scale software

The design and requirements of each software are different. For large scale software, special care has to be taken in the design of the data layout, structure, communication patterns etc.. This wide range of possibilities makes a set general scheme aimed at generating an optimal derivative code not possible. The approach in this thesis proposes to treat the application initially in a black box fashion, which requires no analysis on the dependency structure of the code. Hence, the application is differentiated, in its entirety, with no exceptions. This yields a differentiated code that is consistent with respect to the primal code.

After the first derivative version of the code becomes available, priority is put into obtaining a maintainable solution. This includes for every further change to the code base, an automatic generation of the derivative code ensuring that the differentiated version is always up to date.

These two steps yield a powerful foundation. Improvements to the derivative computation can be achieved by either improving the primal computation, making consistent changes to the derivative computation, and/or making inconsistent changes to the derivative computation.

The improvement of the primal in order to improve the derivative computation is not intuitive, but AD generates the derivative computation from the primal computation, any improvements to the latter will improve the former. Consistent changes to the derivative computation, like an improved storing of the tape, are covered in the next chapter. Inconsistent changes, e.g. disabling specific code areas, are application dependent and are not covered in this thesis. The selection of the adjoint algorithm can also lead to inconsistent changes. Most people start with e.g. the Giles' Exact Dual or the General Adjoint approach from Chapter 3 and if the necessary conditions for these approaches are not fulfilled, then there will be also inconsistencies in the derivative results. It is then hard to check by how much the inconsistent derivatives deviate from the correct ones. With the black box approach, there is always a consistent derivative that can be used to cross check the derivative results. With most other approaches this is not the case.

The chapter will introduce the TRACE CFD suite as an example for the features of a large scale software. It will address common problems that can occur during the differentiation process and their respective solutions. The validation of the derivatives is very important, and several techniques, that can be used to analyze inconsistencies in the derivative computation, will be presented.

4.1. Presentation of a software package

The TRACE CFD suite is being developed by the DLR Cologne (German Aerospace Center) for more than 20 years [FRKA10]. The numerical methods group under the direction of E. Kügeler is the main developer of the code and several universities and companies contribute to the development.

The main focus of the TRACE CFD suite is the simulation of the flow regime in three-dimensional compressor and turbine components. The suite consists of the main simulation code TRACE and other tools. PREP provides the preprocessing, POST the postprocessing, and INTERSEC extracts computational planes for POST.

This section will give an overview of the recent developments in the TRACE code based upon the papers [BA14, BHK10, FRKA10, YNK06]. The basic modeling approach is explained and how the discretisation of the equations is implemented. Then selected advanced topics in the simulation of a compressor or turbine are presented.

The flow regime in a turbo machine is described by the Navier Stokes equations. They formulate the conservation laws for the mass, the momentum and the energy. The equations can be written as

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\mathfrak{v}\\ \rho E\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\mathfrak{v}\\ \rho\mathfrak{v}\cdot\mathfrak{v}^T+pI-\tau\\ \rho\mathfrak{v}H-\tau\cdot\mathfrak{v}-\nabla(kT)\end{pmatrix}=\begin{pmatrix}0\\ \rho\mathfrak{f}_e\\ W_f+q_H\end{pmatrix}. \tag{4.1}$$

The conservative variable vector $W=(\rho,\rho\mathfrak{v},\rho E)^T=(\rho,\rho\mathfrak{v}_1,\rho\mathfrak{v}_2,\rho\mathfrak{v}_3,\rho E)^T$ describes the state of the system. The symbols in the equations and the conservative variables are

$\rho\in\mathbb{R}$	Density	(4.2)
$\mathfrak{v}\in\mathbb{R}^3$	Velocity	(4.3)
$E=e+\frac{\mathfrak{v}^2}{2}\in\mathbb{R}$	Total energy	(4.4)
$I\in\mathbb{R}^{3\times3}$	Identity matrix	(4.5)
$\tau=\mu\left[\nabla\mathfrak{v}+(\nabla\mathfrak{v})^T-\frac{2}{3}div(\mathfrak{v})I\right]\in\mathbb{R}^{3\times3}$	Stress tensor	(4.6)
$H=h+\frac{\mathfrak{v}^2}{2}\in\mathbb{R}$	Stagnation enthalpy	(4.7)
$p=(\gamma-1)\rho e\in\mathbb{R}$	Pressure	(4.8)
$T\in\mathbb{R}$	Temperature	(4.9)
$k=\frac{\mu c_p}{Pr}\in\mathbb{R}$	Coefficient of termal conductivity	(4.10)
$\mathfrak{f}_e\in\mathbb{R}^3$	External force	(4.11)
$W_e\in\mathbb{R}$	Work of external force	(4.12)
$q_H\in\mathbb{R}$	External heat source	(4.13)

$e = \dfrac{1}{\gamma - 1}\dfrac{p}{\rho} \in \mathbb{R}$	Internal energy	(4.14)
$h = \dfrac{\gamma}{\gamma - 1}\dfrac{p}{\rho} \in \mathbb{R}$	Enthalpy	(4.15)
$\mu \in \mathbb{R}$	Coefficient of dynamic viscosity	(4.16)
$\gamma = \dfrac{c_p}{c_{\mathfrak{v}}} \in \mathbb{R}$	Specific heat ratio	(4.17)
$Pr \in \mathbb{R}$	Prandtl number	(4.18)

In order to be fully defined, appropriate boundary conditions need to be set at the boundary of the domain $\Omega \subset \mathbb{R}^3$. From the Navier Stokes equations, the Reynolds averaged Navier Stokes (RANS) equations can be derived by defining an averaged quantity $A = \bar{A} + A'$ with

$$\bar{A}(a,t) = \frac{1}{t_{frame}} \int_{-t_{frame}/2}^{t_{frame}/2} A(a, t+\epsilon)d\epsilon \tag{4.19}$$

and a Favre-averaged quantity $\tilde{A} = \overline{\rho A}/\bar{\rho}$ with $A = \tilde{A} + A''$. The RANS equations can then be written as

$$\frac{\partial}{\partial t}\begin{pmatrix} \bar{\rho} \\ \bar{\rho}\tilde{\mathfrak{v}} \\ \bar{\rho}\tilde{E} \end{pmatrix} + \nabla \cdot \begin{pmatrix} \bar{\rho}\tilde{\mathfrak{v}} \\ \bar{\rho}\tilde{\mathfrak{v}} \cdot \tilde{\mathfrak{v}}^T + \bar{p}I - \tilde{\tau}^V - \tau^R \\ \bar{\rho}\tilde{\mathfrak{v}}\tilde{H} - (\tilde{\tau}^V + \tau^R) \cdot \tilde{\mathfrak{v}} - \nabla(k\tilde{T}) \end{pmatrix} = \begin{pmatrix} 0 \\ \rho\mathfrak{f}_e \\ W_f + q_H \end{pmatrix} . \tag{4.20}$$

The stress tensor is separated into the Reynolds stress τ^R and the averaged viscous shear stress $\tilde{\tau}^V$. The Reynolds stresses are defined as

$$\tau^R_{ij} = -\overline{\rho \mathfrak{v}''_i \mathfrak{v}''_j} . \tag{4.21}$$

Because of the unknown relations between the Reynolds stresses and the mean flow quantities the RANS equations are not closed and require a modeling of these relations.

The TRACE code implements the RANS equations with a cell centered finite volume discretisation. The integral form of a conservation law:

$$\frac{\partial}{\partial t}\int_\Omega W d\Omega + \oint_S \mathcal{F} \cdot dS = \int_\Omega Q d\Omega \tag{4.22}$$

can be used to formulate the discretised integral for each control volume Ω_j

$$\frac{\partial}{\partial t}\int_{\Omega_J} W d\Omega + \oint_{S_J} \mathcal{F} \cdot dS = \int_{\Omega_j} Q d\Omega . \tag{4.23}$$

The equation becomes fully discretised by replacing the integrals with the averaged values over the control volume

$$\frac{\partial}{\partial t}(W_J \Omega_J) + \sum_{\text{faces}} \mathcal{F}^* \cdot \Delta S = Q_J \Omega_J . \tag{4.24}$$

For each control volume the averaged quantities are updated by the numerical flux $\mathcal{F}^*$ over the cell boundaries. TRACE handles the control volumes in a block structured fashion. Each block contains a part of the domain Ω and the blocks are linked by specific boundary conditions. The blocks can contain either a structured mesh or an unstructured mesh. In order to support both meshing approaches, without to much overhead, the data layout of the blocks is arranged such that both meshing approaches are supported and most of the code can be used for both cases. This yields a consistent implementation for both meshing approaches. Both use ghost layers for the boundary definitions and the structured approach, in order to be second order accurate, uses two layers of boundary cells.

Coupling between structured and unstructured meshes is done through block interfaces. Figure 4.1 shows a schematic display of such an interface. The fluxes at the boundary can be calculated with a conservative rezoning calculation from the known fluxes of the upstream side of the boundary. The formulation is

$$(\mathcal{F}^d)_i = \sum_j (\mathcal{F}^u)_j \frac{(\Delta \bar{S})^i_j}{(\Delta S^u)_i} \tag{4.25}$$

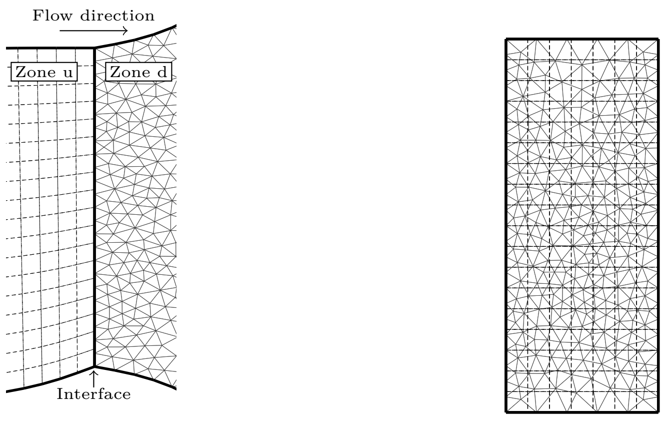
where $\mathcal{F}^d$ is the flux at the downstream side of the interface and $\mathcal{F}^u$ the flux at the upstream side. $(\Delta \bar{S})^i_j$ describes the overlapped area between the face S^d_i on the downstream side and the face S^u_j on the upstream side. $(\Delta S^u)_i$ describes the total area of zone u that overlaps the face S^d_i. The overlapped area can be computed by a modified Sutherland-Hodgman clipping algorithm [NS79]. This hybrid grid interface has the advantage that no 1-to-1 connection is required between the different blocks and can handle structured-unstructured, unstructured-structured, structured-structured and unstructured-unstructured interfaces. The results from Yang et al. in [YNK06] show that the conservative implementation yields a second order accurate interface.

The gradient reconstruction for the finite volume update is implemented in TRACE through the Green-Gauss method. The results from Becker and Heitkamp [BA14] show that this technique performs well for simple structured meshes but on unstructured and complex meshes the accuracy decreases. Therefore, the least squares approach by Barth [Bar91] is introduced to improve the reconstruction. For the quantity $\varphi_i(r) \in \mathbb{R}$ with $r \in \mathbb{R}^3$ in the cell i a Taylor-series expansion for each neighboring cell $j = 1 \ldots p$ is performed

$$\varphi_j = \varphi_i + (\nabla \varphi_i) \cdot (r_j - r_i) + \mathcal{O}(\Delta r_{ij}^2) \ . \tag{4.26}$$

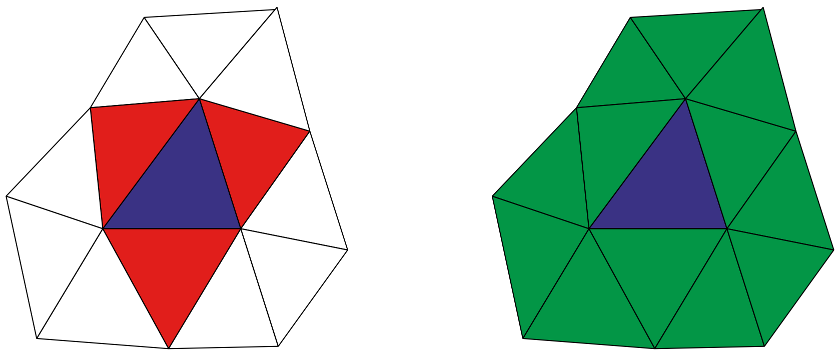
This yields an over determined system for the minimization of the error $E = Ax - b$ with

$$\begin{pmatrix} E_{i,1} \\ E_{i,2} \\ E_{i,3} \\ \vdots \\ E_{i,p} \end{pmatrix} = \begin{pmatrix} \omega_{i,1}(r_1 - r_i)^T \\ \omega_{i,2}(r_2 - r_i)^T \\ \omega_{i,3}(r_3 - r_i)^T \\ \vdots \\ \omega_{i,p}(r_p - r_i)^T \end{pmatrix} \begin{pmatrix} \frac{\partial \varphi}{\partial x} \\ \frac{\partial \varphi}{\partial y} \\ \frac{\partial \varphi}{\partial z} \end{pmatrix} - \begin{pmatrix} \omega_{i,1}(\varphi_1 - \varphi_i) \\ \omega_{i,2}(\varphi_2 - \varphi_i) \\ \omega_{i,3}(\varphi_3 - \varphi_i) \\ \vdots \\ \omega_{i,p}(\varphi_p - \varphi_i) \end{pmatrix} \ . \tag{4.27}$$



(a) Side view (b) Front view

Figure 4.1.: Schematic of hybrid grid interface. (Reproduction of the graphic in [YNK06])



(a) Face neighboring cells (b) Vertex neighboring cells

Figure 4.2.: Nearest neighbor (a) and all neighbor selection (b) for a 2D finite volume cell. (Reproduction of the graphic in [BA14])

Here $\omega_{i,j}$ describes the distance between the cell centers of the i-th and j-th element. The system is solved for the minimal error with a standard least squares method. The selection of neighbors is an important design point in reconstruction. Figure 4.2 shows the two options. The first option selects only the nearest neighbors i.e. all cells that share a face with the i-th cell. The second selects all neighbors that share a vertex with the i-th cell. Becker and Ashcroft show in [BA14] that the vertex selection yields an improvement with respect to the Green-Gauss reconstruction. The nearest neighbor selection yields mixed results depending on mesh quality. The quality of the solution, in terms of the residual, is also greatly improved for the vertex selection.

For the convective numerical flux computation Roe's TVD upwind scheme is used. The implementation is extended by Van Leer's MUSCLE [VL79] extrapolation to obtain second order accuracy in space.

The time derivative in equation (4.24) needs to be handled next. Several choices are available in the TRACE code. The first order Euler backward time integration is one example, that is implemented. Equation (4.24) can be written as

$$\Omega_J \frac{\Delta \hat{W}_J^i}{\Delta t} = \hat{R}_J^{i+1} \tag{4.28}$$

where $\hat{W}$ describes the discretised flow variables and $\hat{R}$ the flow residual. The difference between the current and next state is described by $\Delta \hat{W}_J^i := \hat{W}_J^{i+1} - \hat{W}_J^i$. The right hand side of equation (4.28) can be linearized to create a system of linear equations. The linearized fluxes are simplified such that only the first-order dependencies are taken into account. This yields a sparse matrix where only the direct neighbors of each cell have non-zero entries. It is possible to write the linearized equation after Nuernberger et al. [NES99] as

$$(L + O + D)\Delta \hat{W}^i = \hat{R}^i \tag{4.29}$$

where L describes the strictly lower triangular part of the matrix, O the strictly upper triangular part, and D the diagonal part. The system can then be solved iteratively with a symmetric Gauss-Seidel (SGS) relaxation. The forward and reverse sweeps are defined as

$$(D + L)\Delta \hat{W}_{k+\frac{1}{2}}^i = \hat{R} - O\Delta \hat{W}_k^i \tag{4.30}$$

$$(D + O)\Delta \hat{W}_{k+1}^i = \hat{R} - L\Delta \hat{W}_{k+\frac{k}{2}}^i \tag{4.31}$$

where k is a subiteration index. If only one step is evaluated, then the SGS method becomes the lower-upper symmetric Gauss-Seidel (LU-SGS).

This method can also be used to introduce a pseudo time stepping scheme for the pseudo time τ. Equation (4.24) becomes

$$\frac{\partial \hat{W}}{\partial \tau} = -\frac{\partial \hat{W}}{\partial t} - \hat{\mathcal{F}}(\hat{W}) = \hat{R}^*(\hat{W}) \; . \tag{4.32}$$

Therefore the SGS and LU-SGS schemes can be used for steady state problems and for the inner iteration loop in unsteady problems. The solution of the systems is further simplified in TRACE, such that a block implicit formulation is solved in place of a fully implicit scheme. For higher order time discretisation, multistage Runge-Kutta explicit schemes are also implemented in TRACE.

The limiting of the fluxes can be performed by the ideas of Barth. For this purpose the piecewise linear reconstruction in equation (4.26) is written with a limiter $\mathcal{L}$ as

$$\begin{aligned} \varphi_L =& \varphi_i + \mathcal{L}_i \nabla \varphi_i \cdot r_L \\ \varphi_R =& \varphi_j + \mathcal{L}_j \nabla \varphi_j \cdot r_R \quad , \end{aligned} \tag{4.33}$$

where i is the current cell and j a neighboring cell. φ_L and φ_R define the values on the left and right hand side of the cell face, and r_L as well as r_R describe the distance from the respective cell center to the center of the face. The limiter of Barth enforces monotonicity but produces the largest possible value and is defined as

$$\mathcal{L}_i(\varphi) = \begin{cases} \min(1, \frac{\Delta_{i,\min}}{\Delta_2}) & \text{if } \Delta_2 < 0 \\ \max(1, \frac{\Delta_{i,\max}}{\Delta_2}) & \text{if } \Delta_2 > 0 \\ 1 & \text{if } \Delta_2 = 0 \end{cases} \tag{4.34}$$

with

$$\Delta_{i,\min} = \min(\varphi_i, \min_{j=1\dots p}(\varphi_j)) - \varphi_i \tag{4.35}$$

$$\Delta_{i,\max} = \max(\varphi_i, \max_{j=1\dots p}(\varphi_j)) - \varphi_i \tag{4.36}$$

$$\Delta_2 = \nabla\varphi_i \cdot r_L \ . \tag{4.37}$$

Venkatarishnan proposes in [Ven93] a similar approach that does not enforce strict monotonicity but gives an improved convergence behavior for the solver. The limiter allows for a small over- or undershooting in the solution, which can be defined via the parameter

$$\epsilon^2 = (K\Delta h)^3 \tag{4.38}$$

where K is constant and Δh describes the cube-root of the cell volume in the TRACE solver. The Venkatarishnan limiter can be written as

$$\mathcal{L}_i(\varphi) = \begin{cases} \frac{1}{\Delta_2}\left[\frac{(\Delta_{i,\min}^2+\epsilon^2)\Delta_2+2\Delta_2^2\Delta_{i,\min}}{\Delta_{i,\min}^2+2\Delta_2^2+\Delta_{i,\min}\Delta_2+\epsilon^2}\right] & \text{if } \Delta_2 < 0 \\ \frac{1}{\Delta_2}\left[\frac{(\Delta_{i,\max}^2+\epsilon^2)\Delta_2+2\Delta_2^2\Delta_{i,\max}}{\Delta_{i,\max}^2+2\Delta_2^2+\Delta_{i,\max}\Delta_2+\epsilon^2}\right] & \text{if } \Delta_2 > 0 \\ 1 & \text{if } \Delta_2 = 0 \end{cases} . \tag{4.39}$$

K can be used to describe the effect of the limiter on the gradient. For K equal to zero the limiter is fully applied and for very large K the gradients are unlimited. Becker and Ashcroft [BA14] show that the Venkatarishnan limiter yields improved results with respect to accuracy and convergence.

Because of the Reynolds averaging, a modeling for the relation of the mean flow quantities and the Reynolds stresses are required. TRACE implements several turbulence modeling techniques, simpler techniques like the Spalert-Allmaras one equation model or more detailed techniques like the Wilcox k-ω model. Furthermore, the implementation and validation of the Hellsten EARSM (explicit algebraic Reynolds stress model) k-ω model is described by Franke et al. in [FRKA10], where it is shown that the Hellsten model provides an improved prediction with respect to the Wilcox k-ω model. In addition to the turbulence modeling a transition modeling can be enabled. The γ-Re_θ model introduced by Menter et. al [LML$^+$06, MLL$^+$06] is described for TRACE in [BHK10] by Becker et al.

This is a incomplete overview of the features that the TRACE code has implemented. There are further topics like the boundary conditions e.g. non reflecting boundary conditions described by Acton and Cargill [AC88], the parallelization with

```
// Start of TRACE
initialize()
$w$ = calculateMeshMetrics($x$)

for $i = 1 \dots j$ do
    // Start of G
    for $b = 0 \dots nBlocks$ do
        $(a_b, b_b)$ = updateBlock$_b(y_b, x, w)$
    end for
    $o$ = doCommunication$(b_1, \dots, b_{nBlocks}, y, x, w)$
    $y_{i+1}$ = updateSol$(a_1, \dots, a_{nBlocks}, o)$
    // End of G
end for

$z = f(y, x, w)$ // POST
```

Figure 4.3.: The conceptual structure of TRACE. The state vector y is split into the parts $y = (y_1, y_2, \dots, y_{nBlocks})$, where $nBlocks$ is the number of blocks.

MPI and PVM as well as the extensions of the RANS equations to a rotating frame of reference and additional corrections for streamline curvature. The TRACE code has also extensions for real gas computations and can be coupled with other disciplines like aerodynamics, aeroelasticity, aeroacoustics and aerothermics.

The number of features and the ongoing effort to improve and extend the TRACE code make the implementation of a consistent adjoint formulation very challenging. The ansatz that is recommended in this thesis is tailored to these challenges. One fixed-point iteration step in TRACE (G from Chapter 3) is treated as a black box. The differentiation of TRACE with AD ensures that for all features the correct and consistent derivatives are computed. This ensures that all the adjoint formulations from Chapter 3 will work with the correct derivative information and are therefore consistent with the solution of the primal problem. The global structure of TRACE is the most important information that is required in order to apply AD on the fixed-point solver. The structure of TRACE is shown in Figure 4.3. The control x describes the coordinates of the mesh that are used for the computation. The state vector y describes the flow state of the simulation. At the start of TRACE the metric terms for the mesh are precomputed and described by the metrics w. The pseudo time steps i run until the convergence criteria are fulfilled. In each pseudo time step the update for all blocks is performed and separated into the local values a_b and communication values b_b. The communication values b_b consist of data on the panels as well as other boundary structures and are communicated to the other processes. Everything inside the pseudo time loop is considered as the fixed-point iterator G. The functional f for the optimization is evaluated at the end by the post processing tool POST.

The next sections describe in detail how G and f are differentiated and how x, y and w are defined. Then the derivatives for G and f will be available and the adjoint algorithms from Chapter 3 can be implemented.

4.2. Choice of Algorithmic Differentiation tools

Chapter 2 introduces a naive implementation of the operator overloading approach for AD. If this naive implementation would be used in any software package, the result would be very slow and show a large memory consumption. It is therefore very important that the AD tools chosen, are well established and can handle the differentiation of large scale software.

Two AD tools are selected for the differentiation of the TRACE code. ADOL-C version 2.5.2 (developed by the group of Andrea Walter [WG09]), is the AD tool that has been developed for the longest time and is therefore the most tested and reliable tool available. It is published in an open source format, allowing it to be used by everybody who wants to use the differentiated version of TRACE. ADOL-C follows the advanced primal value taping from Section 6.3. It also applies an index reuse technique which enforces that the *adouble* type - provided by ADOL-C - is copied in a regular way and not by C-like memory operations. The statement level optimization from Section 6.5 is not used by ADOL-C making the memory consumption higher than it otherwise should be. However, the implementation of the *adouble* type is optimized such that its structure does not contain the floating point value. Only the member for the global index is kept in the structure and this index is used to access a global vector for the primal variables. This reduces the memory footprint of the primal computation.

dco/C++ version 3.1.4 (developed by the group of Uwe Naumann [LLN16]) is the second AD tool. It employs the Jacobi taping method and uses statement level optimizations. The indexing scheme in dco/C++ is linear and makes the *dco::a1s::type* (short *a1s*) - provided by dco/C++ - compatible with C-like memory operations. The *a1s* type implements the assign optimization and therefore uses all means to reduce the memory of the tape. The dco/C++ tool is chosen at the project start because it was the fastest AD tool available and has a very optimal memory footprint for the tape. dco/C++ however, is not open source and requires a special license. It can therefore not be used by everybody using the TRACE code.

Both tools were chosen in order to provide an optimized version and a version that can be used by everybody. A second aspect is the cross checking of the tools. Both tools should give the same results in the differentiation process. If the results differ, then one of the tools contains a bug.

4.2.1. Common interface design

Both tools require a slightly different approach for the tape evaluation process. A minimal example for both tools can be seen in the Figures 4.4 and 4.5. The *a1s* type gives the user a very direct access to the adjoint values and the interface of the *a1s* type does not require any changes to the layout of the application.

The situation is different for ADOL-C. The user needs to create arrays that hold the required information to seed the output of $\bar{y}$ and the result of the tape evaluation $\bar{x}$. This forces an order upon the input and output variables, that the user has to keep in mind when creating the array for the seeding. The first variable that is registered as an input must correspond to the first entry in the array, the second to the second etc. A common interface has to store this information and the user is required to access the variables always in the same order. The interface is shown in Figure 4.6 and the implementation example for the general interface is shown in Figure 4.7.

4.3. Differentiation of a software package

The implementation of the adjoint procedures from Chapter 3 requires the derivatives of G and f. These correspond to the numerical kernel of TRACE and the post processing tool POST. The two options available to apply AD to a code are source transformation and operator overloading.

Source transformation applies AD by parsing the source code and generating a new code that is capable of computing the derivatives. Current source transformation tools like Tapenade [HP13] perform a dependency analysis on the entire code base. With input from the user, the source transformation tool decides which functions need to be differentiated and they generate the appropriate transformed source code. This requires a code that can be parsed by the source transformation tool. Additionally, the numerical kernel needs to be separated from the bookkeeping functions like file IO. TRACE is a legacy C code that was never designed for the case that a source transformation AD tool is applied to the code base. The monolithic design structure makes it hard for the source transformation tool to analyze dependencies. A lot of annotations and extra work would be required to help the source transformation tool get the correct dependency relations. The tight interaction of the numerical kernel and the bookkeeping functions is another problem and requires special handling in several places. Other not so obvious problems like handling the special MPI communication (see Sec. 4.3.5) are manageable with operator overloading but are more complex to handle with source transformation. The majority of source transformation tools are mostly used with Fortran. This yields a good test base and a lot of experience for Fortran, that is still missing for C.

Therefore the choice is made that the TRACE code is differentiated with operator overloading AD tools. The strategy for this approach is based on two main principles:

1. All functionalities in the code are differentiated and no functionality is excluded.
2. New functionality that is added to the code is automatically differentiated and the developer does not need to know the principles of AD.

Other influences for the strategy arise from the DLR. They do not want and simply can not do major layout changes to the code. Therefore, the changes that are

```
#include <dco.hpp>

void func(const dco::a1s::type* x, const int n, dco::a1s::type* y, const
    int m){...}

int main() {
   int n = 10;
   int m = 5;

   dco::a1s::type::tape_t* tape = dco::a1s::type::tape_t::create(1000);
   dco::a1s::global_tape = tape;

   dco::a1s::type* x = new dco::a1s::type[n];
   dco::a1s::type* y = new dco::a1s::type[m];

   for(i=0; i < n; i++) {x[i] = 1.0;} // initialization of inputs

   tape->switch_to_active(); // start of recording
   // register inputs
   for(int i=0; i < n; ++i) {tape->register_input(x[i])}

   func(x, n, y, m);

   // register outputs
   for(int i=0; i < m; ++i) {tape->register_output(y[i]);}
   tape->switch_to_passive();

   // set seeding on \bar y
   for(int i=0; i < m; ++i) {dco::gradient(y[i]) = 1.0;}

   // evaluate the recorded tape
   tape->interpret_adjoint();

   std::cout << "df/dx^T*(1,..., 1)^T = (";
   for(int i=0; i<n; ++i) {
    if(i != 0) {std::cout << ", ";}
    std::cout << dco::gradient(x[i]); // read derivatives from \bar x
   }
   std::cout << ")^T" << std::endl;

   // cleanup ...
}
```

Figure 4.4.: Example for the use of dco.

```
#include <adouble.h>
#include <drivers/drivers.h>
#include <taping.h>

void func(const adouble* x, const int n, adouble* y, const int m){...}

int main() {
   int n = 10;
   int m = 5;

   adouble* x = new adouble[n];
   adouble* y = new adouble[m];

   for(i=0; i<n; i++) {x[i] = 1.0;} // initialization of inputs

   trace_on(1, 1); // start of recording, keep=1 => store primal values
   for(int i=0; i < n; ++i) {x[i] <<= x[i].value();} // register inputs

   func(x, n, y, m);

   for(int i=0; i < m; ++i) {
      double temp;
      y[i] >>= temp; // register the outputs
   }
   trace_off(); // end of recoding

   double* gradient = new double[n];
   double* adjoint = new double[m];

   for(int i=0; i < m; ++i) {adjoint[i] = 1.0;} // set seeding on \bar y

   // evaluate first order reverse for tape 1
   fos_reverse((short)1, m, n, adjoint, gradient);

   std::cout << "df/dx^T*(1,..., 1)^T = (";
   for(int i=0; i < n; ++i) {
    if(i != 0) {std::cout << ", ";}
    std::cout << gradient[i]; // read derivatives from \bar x
   }
   std::cout << ")^T" << std::endl;

   // cleanup ...
}
```

Figure 4.5.: Example for the use of ADOL-C.

```
template<typename ADType>
class ADInterface {
public:
   typedef ADType type;

   virtual void setValue(ADType& value, const double& x) = 0;
   virtual double getValue(const ADType& value) = 0;

   virtual ~ADInterface() {};
};

template<typename ADType>
class ADInterfaceReverse : public ADInterface<ADType> {
public:

   virtual void setAdjoint(ADType& value, double seed) = 0;
   virtual void updateAdjoint(ADType& value, double seed) = 0;
   virtual double getAdjoint(const ADType& value) = 0;

   virtual void registerInput(ADType& value) = 0;
   virtual void registerOutput(ADType& value) = 0;

   virtual void createTape() = 0;
   virtual void zeroTape() = 0;
   virtual void evaluateAdjoint() = 0;

   virtual void activateTape() = 0;
   virtual void deactivateTape() = 0;

   virtual bool isTapeActive() = 0;

   virtual ~ADInterfaceReverse() {};
};
```

Figure 4.6.: Definition of the common interface for the AD tools.

```
#include <ADInterfaceDco.hpp>
//#include <ADInterfaceAdolc.hpp>
//#include <ADInterfacePrimal.hpp> // empty, implementation for double

typedef ADInterfaceDco ADInterface;
typedef ADInterface::ADType Real;

void func(const Real* x, const int n, Real* y, const int m){...}

int main() {
   int n = 10;
   int m = 5;

   ADInterface* adCntl = new ADInterface();
   adCntl->createTape();

   Real* x = new Real[n];
   Real* y = new Real[m];

   for(i=0; i < n; i++) {x[i] = 1.0;} // initialization of inputs

   adCntl->activateTape(); // start of recording
   // register inputs
   for(int i=0; i < n; ++i) {adCntl->registerInput(x[i])}

   func(x, n, y, m);

   // register outputs
   for(int i=0; i < m; ++i) {adCntl->registerOutput(y[i]);}
   adCntl->deactivateTape();

   // set seeding on \bar y
   for(int i=0; i < m; ++i) {adCntl->setAdjoint(y[i], 1.0)}

   adCntl->evaluateAdjoint();

   std::cout << "df/dx^T*(1,..., 1)^T = (";
   for(int i=0; i < n; ++i) {
     if(i != 0) {std::cout << ", ";}
     std::cout << adCntl->getAdjoint(x[i]); // read derivatives from \bar x
   }
   std::cout << ")^T" << std::endl;

   // cleanup ...
}
```

Figure 4.7.: Example for the use of the common interface.

introduced by the differentiation of AD should be as minimal as possible.

The approach for the differentiation is to exchange the *Ffloat* type that is defined in TRACE on a global level. This is the best approach for legacy C codes, otherwise all functions and structures that need to be differentiated would be copied and then the type needs to be replaced. This procedure conflicts with the principle that all new functionality should be automatically differentiated. The global replacement of the *Ffloat* has the drawback that side computations are also performed with the AD type (e.g. the setup and initialization). That means these parts are slower but the ratio between the side computations and the numerical computations is more important. For highly optimized codes like the TRACE code, 99% of the calculations during the taping process are relevant for the derivative. This would make it very costly to eliminate the last percent of the side computations.

The *Ffloat* type in TRACE can already be switched between single precision and double precision. Because of that the facilities are already in place such that the AD type can be introduced on a global level. Most of the changes for the differentiation of the code are required by the switch from C to C++ compilation and by using fully defined classes instead of primitive types.

It is also beneficial that for most common tasks, preprocessor macros are defined in the TRACE code. If a problem occurs in the logic that is defined by a macro, only the macro has to be modified as opposed to every place where the logic is used.

Usually, a lot of structures are used, that contain floating point values. A common scenario requires that a certain action be performed for every floating point value in such a structure. Keeping these iterators up to date is very tedious and if not all values are reached, then errors can quite easily occur. Therefore the process for the generation of these structures is automated. A parser reads all the header files and generates iterators for all structures in these files. The iterators can be used to perform common tasks like the gathering of all the indices for the AD values. For example the function *gatherFfloatLocations* will add all *Ffloats* that are contained in the structure to a list.

Several problems can occur when a code is augmented with AD. The next sections list these problems and show solutions.

4.3.1. Memory allocation

Operator overloading tools like ADOL-C require a constructor to be called when a variable is allocated. C codes have to use *malloc* and *free* in order to acquire and release memory. If that is the case they can be replaced by *new* and *delete*. If *realloc* is used in the source code, then there is no C++ equivalent that ensures the correct calls of the constructors and destructors. The first step in solving this problem is the introduction of a global memory manager that stores the size of every created array. For *realloc* it creates a new vector and copies the old values. This approach encounters a flaw when the size of a vector is increased by one several million times. In such a situation the whole vector needs to be copied for every increase. ADOL-C stores every one of these copy operations and releases the old indices afterwards. The

```
class StorageManager {
   ...

   template<typename TYPE>
   TYPE* reallocMem(TYPE* oldPtr, const size_t newSize, const char *file,
       const int line, const char func[]) {
      DeleteInfo oldInfo;
      oldInfo.size = 0;
      if(oldPtr != NULL && !deleteInfo(oldPtr, oldInfo)) {
            SERIOUS_MEM(file, func, line, "ERROR: ...");}

      if(newSize < oldInfo.size) {
         /* delete the freed values */
         for(size_t i = newSize; i < oldInfo.size; ++i) {
            oldPtr[i].~TYPE();
         }
      }

      TYPE* newPtr = (TYPE*)realloc(oldPtr, sizeof(TYPE) * newSize);
      if(oldInfo.size < newSize) {
         /* init the new elements */
         for(size_t i = oldInfo.size; i < newSize; ++i) {
            new (&newPtr[i]) TYPE();
         }
      }

      setInfo(newPtr, newSize, deleteMemImp<TYPE>);
      return newPtr;
   }
};
```

Figure 4.8.: Implementation of *realloc* for C++ classes. It uses the placement *new* operator and explicit calls to the destructors. The presented version is abbreviated for simplicity.

second step in reconciling the problem is the use of the placement *new* operator and explicit calls to the destructor. This eliminates the overhead of explicitly copying the whole vector every time a reallocation is performed.

The final implementation of the *realloc* replacement is shown in Figure 4.8. The replacements for *malloc* and *free* are written with the same techniques. The placement *new* operator is not used often in regular software. The operator can be used to initialize a class at a specific memory location without allocating heap or stack memory. Therefore it is a very handy tool when C and C++ memory structures are used together.

4.3.2. Unions

Since C++11 custom constructors and destructors are allowed for unions that contain non plain old data structures. Only this extension to the standard allows the

```
/**
 * Additional constructor, destructor and copy operator code for unions.
 *
 * @param unionType The typename of the union.
 * @param valueName The name of the member that holds the values. Needs
 *    to be a struct.
 * @param valueType The type name of the member that holds the values.
 * @param arrayName The name of the array that represents the values.
 */
#define UNION_INJECT(unionType, valueName, valueType, arrayName) \
   unionType() : valueName() {}; \
   unionType(const unionType& value) : valueName(value.valueName) {}; \
   \
   ~unionType(){ valueName.~valueType();}; \
   \
   template <typename T> \
   unionType(std::initializer_list<T> values) { \
     std::copy(values.begin(), values.end(), arrayName); \
   } \
   \
   unionType(std::initializer_list<valueType> value) :
       valueName(*value.begin()) {}; \
   \
   unionType& operator=(const unionType& other) { \
     this->valueName = other.valueName; \
     return *this; \
   }
```

Figure 4.9.: Macro for unions that contain non plain old data types. The union needs to have the same data in all members.

proper treatment of AD types in unions. TRACE uses several unions in order to access data in an array fashion or like structures. A general macro can be used to add the additional functionality to unions that contain AD types. The macro uses the member representation of the unions to perform the constructions, destructions and assignments. The array representation is used for initializer lists. The other arguments for the macro define the member representation type and the name of the union. Figure 4.9 shows the macro and a detailed description of the arguments.

4.3.3. Variadic functions

Variadic functions like *printf* can not be used with non plain old data types. The AD types will either produce a compiler error when used in variadic functions or the output will be wrong. For C functions like *printf* this will be the case. Therefore, the AD types need to be converted to double.

If variadic functions are used throughout the source code, then the identification of every position where a special treatment for AD is necessary can become quite cumbersome. A general C++ solution would be to generate 50 functions with 1 to 50 template arguments and perform a special treatment for the AD types. The function *getValue* is called on every argument and will either return the argument or will perform a special treatment for the AD types. In C++11 variadic template functions can be used to do the same in a more general fashion. The implementation of the special treatment for the AD types can become quite involved and is discussed now.

The base implementation is

```
template<typename T>
T & getValue(T& t ) {return t;}
```

and will return the argument itself. The function is then specialized for the AD tool e.g.:

```
template<>
double getValue(const dco::a1s::type& t ) {
   return dco::value(t);
}
```

If the AD tool uses special memory reduction techniques, then the return type of an operation can become a structure. If this structure is a template class, then the specialization of *getValue* can become arbitrarily complex. If computations are performed in a *printf* e.g.:

```
printf("%d", a * a);
```

then the result of $a * a$ is no longer a *dco::a1s::type* but rather a template structure. A specialization for *getValue* is then required for all structures which can be generated by the expressions yielding millions of possible cases. Therefore, the functions need to be partly specialized, leading to the use of partly specialized classes. The implementation of *getValue* needs to be changed to

```
template<typename T>
T & getValue(T& t ) {return GetValue_Impl::getValue(t);}

template<typename T>
struct GetValue_Impl {
  static T getValue(T& t) {return t};
};
```

GetValue_Impl can now be partly specialized. dco/C++ has 4 special classes for the handling of the operators and new operators are automatically handled. Other AD tools have a different structure for each operator and need over 50 separate specializations with this approach. For every new operator a new specialization needs to be created. Even if all these structures extend from a base class, no specialization for the base class is possible. The reason behind that is that a function or class can not be specialized for sub classes. The specialization

```
template<typename Klass> struct GetValue_Impl<Expression<Klass> >{...};
```

```
// undefined base class
template< typename IN, bool SW > struct Impl_getValue_switch {};

// specialization for every other class, SW = false
template< typename IN> struct Impl_getValue_switch<IN, false> {
  typedef typename Impl_getValue<IN>::OUT OUT;

  static inline OUT& getValue(IN &value) {
    return Impl_getValue<IN>::getValue(value);
  }
};

// specialization for the Expression interface, SW = true
template< typename IN> struct Impl_getValue_switch<IN, true> {
  typedef typename Impl_getValue<Expression<IN> >::OUT OUT;

  static inline OUT& getValue(IN &value) {
    return Impl_getValue<Expression<IN> >::getValue(value);
  }
};

// selection which case of the specialization is choosen
template <typename IN>
static inline
/* return type */ typename Impl_getValue_switch<IN,
    std::is_base_of<Expression<IN>, IN >::value
  >::OUT
getValue(IN& a) {
  return Impl_getValue_switch<IN,
      std::is_base_of<Expression<IN>, IN >::value
    >::getValue(a);
}
```

Figure 4.10.: *getValue* implementation for classes that extend from a common expression class. Specializations for *const* arguments have been removed for brevity.

is not recognized by the compiler if the class *Add* extends from the class *Expression<Add>*. The solution to this uses the *is_base_of* trait from the C++11 standard and is shown in Figure 4.10. The *getValue* implementation is extended such that it checks at compile time if the template class extends *Expression*. The result is given to the class *GetValue_Switch* which is defined for the cases when the type extends from the *Expression* interface or not. If multiple AD tools are used, a hierarchy of such checks can be introduced to handle different interfaces.

4.3.4. Unstable formulations

The simple statement

```
if(vecNorm(a) > 0.0) {...}
```

can have large consequences for the tape evaluation, but the statement does not change the result of the derivative computation. If the Euclidean norm of the vector is zero, then the square root is differentiated at zero which is not defined. Usually this yields a *NaN* and in the reverse procedure the update $\bar{v}_i \mathrel{+}= jac_i * \bar{z}$ would perform the operation $NaN * 0.0$ which is still a *NaN*. This *NaN* is now propagated through the whole tape and can lead to results that only contain *NaN* values. A problem that has the same character is

```
a = 1.1;
...
c = asin(a);
z = ... * c;
if(!isfinte(z)) {
  r = 1000.0;
}
```

For a programmer, it is more convenient to check the result of an algorithm and set a default value. Checking the argument of every operation and setting a default value each time is both time consuming and less beneficial.

Legacy codes can have a lot of these pitfalls and therefore a solution in the AD tool is more appropriate. If the tool checks during the reverse evaluation that $\bar{z}$ is zero and skips the update, then the propagation of the *NaN* values, in most cases, is already stopped. A more robust solution would be to directly skip the writing of *NaN* values to the tape. This can lead to situations, where dependencies are ignored. Therefore, AD tools should contain a mode where the user is warned about arguments that are outside of the differentiable area of the operator.

4.3.5. MPI communication

The MPI communication can be handled nicely with the libraries AdjointMPI [SNHU10] or AdjoinableMPI [UHH$^+$09]. These libraries however, can only handle vectors that contain just AD data types. If structures or other types are sent together with AD types, the libraries will be unable to handle this.

These general cases can be handled in many different ways. One possible method is discussed for the TRACE communication structure which is built around general purpose data buffers that are sent between the processes. The procedure for the buffer handling is:

```
Buffer buf;
initializeBuffer(buf);

fillBuffer(buf);
MPI_Send(buf.data, buf.size, MPI_BYTE, ...);

//other process
MPI_Recv(buf.data, buf.size, MPI_BYTE, ...);
readBuffer(buf);
```

```
#if ADJOINT_COM
#  define ACCESS_BUFFER(buffer, type, length, content, mode, func)
     adAccessBufferMpi<type>(buffer, (size_t)(length),
                         content, mode, func)
#else
#  define ACCESS_BUFFER(buffer, type, length, content, mode, func)
     accessBuffer(buffer, sizeof(type) * (size_t)(length),
                content, mode, func)
#endif
/** @} */

#define BUFFER_WRITE_FFLOAT(buffer, content) ACCESS_BUFFER(
     &buffer, Ffloat, 1, &content, BUFFER_MODE_WRITE, __func__)
#define BUFFER_READ_FFLOAT(buffer, content) ACCESS_BUFFER(
     &buffer, Ffloat, 1, &content, BUFFER_MODE_READ, __func__)
...
```

Figure 4.11.: Interface for buffer access in TRACE and switch for the AD specific handling.

The routines *fillBuffer* and *readBuffer* have no restriction on the data they write into the buffer and how they do this (e.g. *memcpy*). In such a case it is impossible for AD to track the information and create the proper adjoint statements on the tape.

The first step in such a case is to define an interface that is used to access the buffers. For the TRACE buffers the methods in Figure 4.11 are now used to access them. Because TRACE is written in C99 macros are used to convey the type information. The introduction of the interface and the modification of all the *fillBuffer* and *readBuffer* routines took most of the time for the adjoint handling of the MPI communication.

After the introduction of the buffer access interface, the generation of the adjoint MPI request becomes straight forward. The five steps are:

1. Gather information about the location of the AD values in the buffers.
2. Modify the buffers such that the correct information is sent to the other process.
3. Send the buffer and create the corresponding adjoint statement.
4. Receive the buffer and create the corresponding adjoint statement.
5. Modify the buffer such that the AD values can be used in the new process.

The first step is implemented by changing the buffer access macros and storing the offset of each AD value in a list. In order to do this a function, *getValueOffset*, is generated by the parser, ensuring all structures can be handled. For ADOL-C a special routine is called that copies the structures ordinarily and not with the *memcpy* function.

```
inline void adolcBufferCopy(Ffloat* curIn, Ffloat* curOut,
                            Ffloat* temp) {
   /* reset the copied value with the temporary one.*/
   memcpy(curOut, temp, sizeof(Ffloat));

   /* reassign the Ffloats here such that they copy the value */
   *curOut = *curIn;
   curIn->~Ffloat(); /* delete the old value */

   /* move the new value to the position of the old one */
   memcpy(curIn, curOut, sizeof(Ffloat));

   /* overwrite the structure with the double value */
   double* curOutAsDouble = reinterpret_cast<double*>(curOut);
   curOutAsDouble[0] = curIn->value();
}
```

Figure 4.12.: Logic for creating a correct buffer layout for ADOL-C. *curIn* describes the value in the regular buffer, *curOut* the value in the modified buffer, and *temp* the value with the linear increasing index. *curIn* needs to be copied to *temp*, *temp* placed in *curIn* and *curOut* needs to be set to the double value.

The second step is handled right before the buffer is sent. It uses the list from the first step to access all AD values in the buffer. In the case of dco/C++ no modifications to the values are required as the *a1s* type contains the primal value. The index in the structure can be used by the other process to activate values that have been active on the sending machine (e.g. index $\neq 0$). The situation is different for ADOL-C. The *adouble* type from ADOL-C contains only the index for the value. If the buffer is sent as it is, the other process has no means to access the primal value of the AD type. It is therefore necessary to overwrite the index in the buffer with the double value. This is done for every value, even if the recording of the tape is not active. Furthermore, for ADOL-C the requirement has to be met that the indices linearly increase in the buffer. This causes an additional copy operation to be done for every floating point value. The full logic of this process is shown in Figure 4.12.

For both tools the buffer is appended with the locations of the AD values inside the buffer. This is necessary because the receiving side needs to create all of the AD values directly when the buffer is received.

The third and fourth step are handled by regular MPI calls but the modified buffers are sent.

The fifth step is performed after the buffer is received. The extended buffer contains the positions of all AD values and uses them to modify the buffer such that the AD values can be used on the new processor. For ADOL-C the value has to be read and an index needs to be created at the position. For both tools the values need to be registered on the tape. The indices of the *a1s* types from the other process can be used to check if the variable needs to be activated on the receiving processor. But the old index has to be renewed on the receiving side, otherwise it would be like using a pointer that is generated on another process.

4.3.6. Libraries

The use of libraries is quite common and a basic principle of software engineering. When AD is applied on a code the libraries need to be handled, too. Depending on the purpose of the library, the strategies differ.

A library that is not involved in the numerical computation does not need to be differentiated, but it is usually the case that floating point variables are given to functions of the library. In these cases the AD variables are not accepted by the interface and errors are generated. The usual solution is to convert the AD variables to regular floating point values. If the library is only used at a few distinct places, then the changes are quite easy. If the library is used throughout the whole software, then it can become a tedious task. A good design principle, to make such a transition as easy as possible, is to encapsulate the library access into a single translation unit. This contains library specific changes in a well defined region.

This is the work flow when the library has a well defined, type safe interface. After all compiler errors are handled the program will run correctly. If the interface is not type safe, then the errors can not be detected by the compiler. A good example is the CGNS library that is used in TRACE. It is used to write CGNS files, that contain the mesh and the flow solution. The data arrays are given directly to the library and an enumerator defines what kind of data is contained in the array. If an AD floating point array is used instead of a double array, then the CGNS library will access the wrong data. The array needs to be converted, as described above, to eliminate the error, but it can not be detected by the compiler.

In order to create a maintainable solution, a type safe wrapper should be written for the library, such that errors are detected automatically in future developments.

Another aspect for the maintainability are the types that the library defines. If the types have special meaning for the library, but are defined as common data types, then a typedef should be introduced that makes the special meaning visible to the developers. A good example are the MPI_Request objects, in some implementations they are just an int. A bad example would be the node indices in the CGNS library. These node indices are plain double types but have a totally different meaning than a floating point type. The use of double is even more particular in the context of AD. AD developers are used to change every double into an AD type which would lead to further errors.

In conclusion it can be said that libraries should be used in a type safe way and in a single translation unit.

Libraries, that are used in the numerical computations, can be handled with checkpoints (see Section 5.4) or differentiated with AD as well. The next section gives an example of such a case.

4.3.7. Differenziation of VTK

Because INTERSEC is a wrapper for VTK (the visualization toolkit) the most important point is to differentiate VTK in order to differentiate INTERSEC. VTK [ABB+10] is a general purpose library for handling CAD (computer aided design), grid and other scientific data. It contains a large library of algorithms that can e.g. modifiy the data sets or extract sub plains from the data. VTK also provides a large set of routines for displaying all the data and the results from the algorithms. The total library consists of 20 libraries that contain 2 million lines of code in 8000 files. INTERSEC uses only the libraries that are required for cutting out a plane from a mesh. These are *utilities/kwsys*, *utilities/verdict*, *common*, *filtering* and *graphics* and are differentiated with dco/C++ and ADOL-C. They still contain 400,000 lines of code in 2000 files.

VTK does not define *typedef* definitions for the floating point types. *Float* and *dobule* are directly used. Therefore, every occurrence in the libraries is changed to *BT_Float* and *BT_Double*. After that, these definitions can be used to change the floating point types to an AD type.

It is not possible to declare the single and double precision *typedef* definitions with the same type. The compiler will generate errors for all functions that are defined for both types. The other option would be to use the generalized type of dco/C++ with double and float. That would produce the types *dco::ga1s<double>::type* and *dco::ga1s<float>::type*. The problem with this approach is, that both types can not be converted into each other and each type has a separate tape. If the types could be converted into each other, then there would still be the problem about how the information of one tape is transferred to the information of another tape. It is also uncommon for AD tools to provide two different precision types using the same tape structure in the background. It is not available in dco/C++ or ADOL-C.

The solution is to extend from the given types and therefore create a "new" type. The principle is explained on a little example:

```
struct A {double a;}
struct B : public A {}

void func(A& a) {printf("A used");}
void func(B& b) {printf("B used");}
```

The structure A represents our AD type. B inherits from A but does not add any logic or data to A. From a user perspective A and B are the same. But the compiler handles both as distinct types and therefore does allow the function *func* to be overloaded.

If this technique is used, it is possible to use the member functions of the base class directly, but operators like the constructor or the assignment need to be redefined and the conversion between the new type and the base type has to be carefully defined. The compiler must not use an intermediate type like *int* for the conversion from A to B. If the compiler would perform such an conversion, the dependency information from AD would be lost. Because of that, the *static_cast* is changed to the method

```
// definition of the second AD type
class adolcExt : public adouble {
public:
  adolcExt() : adouble() {};
  adolcExt(const double &value) : adouble(value) {}
  adolcExt(const adouble &value) : adouble(value) {}
};

template<typename OT, typename IT> struct Impl_performCast {
        static inline OT performCast(IT &a) {
            return static_cast<OT>(Impl_getValue<IT>::getValue(a));
        }
};

// implementation of the conversion between the AD types
template<> class Impl_performCast<adouble, adolcExt> {
  static inline adouble performCast(const adolcExt &a) {
    return a;}};

template<> struct Impl_performCast<adolcExt, adouble> {
  static inline adolcExt performCast(const adouble &a) {
    return adolcExt(a);}};

template<typename OT, typename IT> static inline OT performCast(IT &a) {
    return Impl_performCast<OT, IT>::performCast(a);
}
```

Figure 4.13.: Implementation of a second AD type that extends from the base type. *static_cast* is changed to *performCast* in order to handle the conversion between the AD types.

performCast and this method is specialized for the conversion between the AD types. In Figure 4.13 the implementation of a second type for ADOL-C is presented. This second type is then used as the float replacement in VTK. Because both types use the same tape in the background, AD will produce the correct derivative results.

4.3.8. Differentiation and communication with external tools

External programs can become involved in two ways. Either they are called directly from the main program or the main program is part of a process chain, that consists of multiple tools. The usual process for involving external programs is a file written by the main program and read by the external program. This can be modeled by the simple assignment:

$$a_{ext} = a_{main} \tag{4.40}$$

where a_{main} describes the variables in the main program that are written to the file and a_{ext} are the variables in the external program that are read from the file. The forward and reverse mode of (4.40) are then

$$\dot{a}_{ext} = \dot{a}_{main} \tag{4.41}$$
$$\bar{a}_{main} \mathrel{+}= \bar{a}_{ext}\ . \tag{4.42}$$

The forward equation can be handled by writing a second file that contains the dot values or by extending the original file format such that it contains the primal and dot values. The reverse equation can be implemented in a similar fashion. The external program writes a file that has the same layout as the file written by the main program but this *seeding* file contains the bar values. The seeding file can then be read by the main program in order to get the update for the bar values.

A security mechanism can be included in the seeding file by adding the primal values. The main program can then check if the current state and the state when the file was written are the same. In an inconsistent state the derivatives can not be guaranteed to be correct. Checking the state can also help to detect errors that are introduced by the functions that write the files. A good example is the averaging of boundary values that are performed after the AD values are converted to floating point types. The additional computations are no longer tracked by AD, which leads to an inconsistent state. If an inconsistent state is detected, then the information flow of the application needs to be changed such that all states at the communication boundaries are correct.

This technique is used to couple the post processing tool POST to TRACE. The differentiation of POST is otherwise analog to the differentiation of TRACE.

4.4. Implementing adjoint algorithms

With a software that is differentiated, the algorithms from Chapter 3 can be implemented. They require the evaluation of the gradients $\frac{\partial G}{\partial y}$, $\frac{\partial G}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial x}$, therefore G and f need to be defined as drivers in the software and called with the appropriate settings of the AD tool. For the TRACE code, G corresponds to one iteration of the loop in Figure 4.3 and f to the call of POST. The correct implementation of the derivative algorithms is then only the correct assembly of the drivers with the correct settings. Nevertheless, there are a few things that require design decisions in the implementation in order to guarantee the correct behavior of the algorithms.

One important aspect is the definition of the state y and the control x. The mathematical modeling always assumes that they are well defined and available. If that is not the case and the state y is incomplete, then the correct behavior of the derivative algorithms can not be guaranteed. Techniques for the validation of y and x are discussed in the following sections. The complete definition of y for TRACE is shown in Table 4.1 with a short description on the variable names. y contains some surprising values including the conservative and primitive representation of the flow state. Usually only one set is added to the state and the other set is computed on need. The panel boundaries and integrals are values that are usually not required in the state. They need to be included in the state space because of the special structure of TRACE. Figure 4.3 shows that the computation of the blocks and the communication is strictly separated. All values, that are communicated, are used in the next iteration for the update of the blocks. If the values are not added to the state, the AD tool can not see the derivative dependencies of these values. A restore of the state y will leave these values untouched and therefore the restored iteration will use old values. This needs to be emphasized quite strongly.

The facilities of a simulation software, that restore the state, are called restarts, and are most of the time not implemented in an exact and consistent fashion. Small errors or inconsistencies are accepted because the primal solve will correct these after a few steps and the overall convergence progress will not change much. Usual suspects are the ghost layers where values are interpolated from the field. For AD and the derivative algorithms from Chapter 3 this can have dramatic consequences. The perturbation of the input variables can be to strong and cause the algorithms not to converge. AD requires consistent values, otherwise the execution path in the software can differ and lead to different derivative dependencies. This can be a major drawback for Reverse Accumulation, because the same tape is evaluated multiple times. Therefore the full definition of the state y should always be used to restore the state of the software in order to be fully consistent.

The handling of precomputed variables is another aspect in the implementation of the adjoint algorithms. Usually there exist some mesh metrics w, like the volume of a cell, that can be computed in advance. This increases the speed of the solver because less computations are performed during the evaluation of the fixed-point iterator G. For the derivative computation, the values are quite critical. The usual dependencies are that the control x defines the mesh metrics w and the state y is then defined by x and w. The derivative computations need to handle these dependencies correctly in order to guarantee correct results. The critical aspect for the correct handling of these values depends greatly upon how they are computed. If there exists a single function that computes all such values, then the integration with the derivative algorithms is straight forward. The function can be called in the correct places and thus the tape size will be minimal. If such a function does not exist, then the solution is more involved. A tape needs to be recorded when the mesh metrics w are computed and this tape needs to be stored during all other derivative evaluations. Figure 4.3 shows that TRACE uses precomputed mesh metrics in the computation. In the primal run they are only computed once and do not change afterwards. Therefore they have been intertwined with the initialization process and no separate function exists for the recomputation. Figure 4.14 shows the required tapes for the Reverse Accumulation process, once where the recording needs to be done at the start, and once where the mesh metrics w can be recomputed.

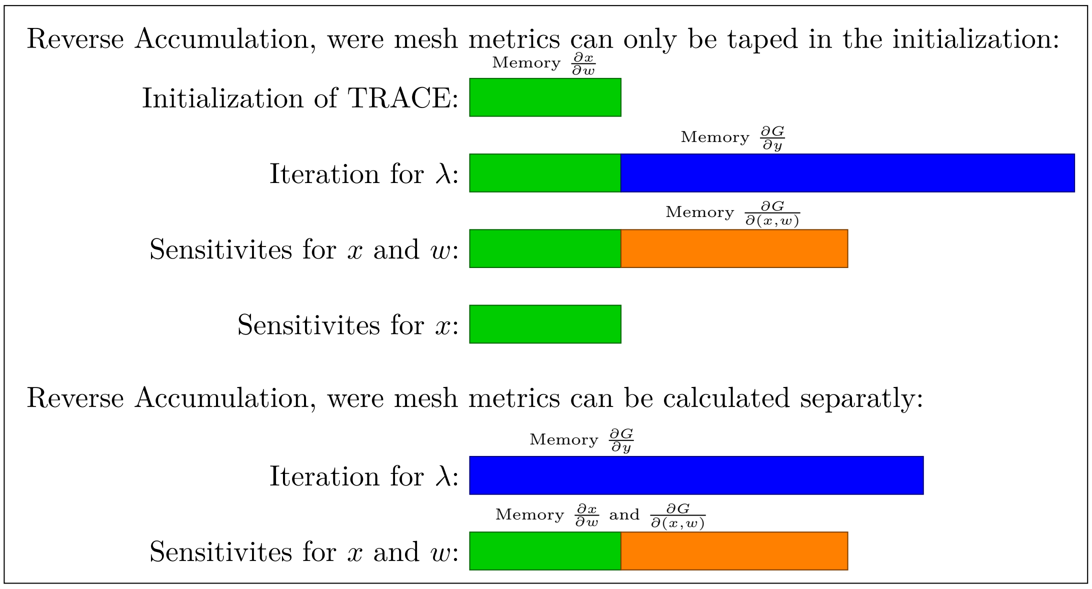
In the first setting the computation of the mesh metrics w are stored on the tape during the initialization. The tape for $\frac{\partial G}{\partial y}$ is then stored in order to perform the adjoint fixed-point iteration. When the iteration has converged, the tape for $\frac{\partial G}{\partial y}$ can be removed and the tape for $\frac{\partial G}{\partial (x,w)}$ can be stored and evaluated. The final step in the process is the propagation of the adjoint mesh metrics to the control x with the

Table 4.1.: All the arrays, panels and other values that are defined in the state vector y of TRACE. The state contains field values (values per cell center), unstructured field values and general values on panels like the mixing plane etc.

Field values	Description
P_FLOW	Conservative and primitive flow variables
P_EDDY	Viscosity variables
P_WALL_DIST	Wall distance function
P_TPDERIVED.bslF1	Derived turbulence variable
P_TPDERIVED.vorticityNorm	Derived turbulence variable
P_TRANSPORT	General transport variables used e.g. for turbulence variables
P_TRANSISTION	Variables for the transition modeling
P_MU	Coefficient for dynamic viscosity

Unstructured field values	Description
P_NODE_GRAD_PHYS_LIM_U	Limited reconstructed gradients for the finite volume scheme.
P_NODE_GRAD_TRANS_LIM_U	Limited reconstructed gradients for the finite volume scheme.
P_NODE_GRAD_PHYS_UNLIM_U	Unlimited reconstructed gradients for the finite volume scheme.
P_NODE_GRAD_TRANS_UNLIM_U	Unlimited reconstructed gradients for the finite volume scheme.

Non field values	Description
DATA_GLB.general	Global per block values e.g. CFL number
DATA_GLB.residual	Global per block residual data
P_CORNER_PHYS_U.primValues	Averaged flow values on cell boundaries. Only for unstructured meshes.
P_CORNER_PHYS_U.tp_var	Averaged transport values on cell boundaries. Only for unstructured meshes.
panel->bdIntegral	Integral values of boundary panels e.g. entry or exit
panel->interface.bandData	General communication data e.g. for mixing planes
panel->interface.faceClip	Face clip data for hybrid grid interface



Reverse Accumulation, were mesh metrics can only be taped in the initialization:
Memory $\frac{\partial x}{\partial w}$
Initialization of TRACE:
Memory $\frac{\partial G}{\partial y}$
Iteration for λ:
Memory $\frac{\partial G}{\partial (x,w)}$
Sensitivites for x and w:
Sensitivites for x:
Reverse Accumulation, were mesh metrics can be calculated separatly:
Memory $\frac{\partial G}{\partial y}$
Iteration for λ:
Memory $\frac{\partial x}{\partial w}$ and $\frac{\partial G}{\partial (x,w)}$
Sensitivites for x and w:

Figure 4.14.: Memory layout for Reverse Accumulation, where the precomputation of the mesh metrics is taped at the beginning or when needed.

initial tape $\frac{\partial w}{\partial x}$.

The second setting is a more optimized version and omits the taping of $\frac{\partial x}{\partial w}$ at the start. Less memory is required and no explicit definition of the mesh metrics w is necessary. This is the case because the tapes for $\frac{\partial x}{\partial w}$ and $\frac{\partial G}{\partial (x,w)}$ can be combined by calling the function that recomputes the mesh metrics before calling G. The recorded tape stores all of the dependencies and no special handling for w is required.

Table 4.2 gives an overview of the algorithms and the possible configurations for them. The figure also lists some additional algorithms that are used to support the adjoint algorithms and testing.

In order to have the most optimized version for the required task, the TRACE make system can produce 5 different binaries: *C99*, *C++*, *dco::t1s::type*, *dco::a1s::type* and *ADOL-C*. The *C99* version does not include any functionality from the AD module. The *C++* version contains only the logic for the primal algorithms like writing the checkpoints. The other 3 versions contain the drivers for the adjoint algorithms.

4.5. Derivative validation

After a software is differentiated in the reverse mode of AD, the derivatives of the reverse AD mode need to be validated. The two techniques available for this are the forward AD mode and the finite difference method.

In the forward AD mode all derivative computations are directly performed next to the primal computations. This process is simple and the resulting implementations are plain old datatypes, that are compatible with almost all programming techniques.

Table 4.2.: Table of all the algorithms that are implemented in TRACE. The algorithms range from the adjoint algorithms defined in Chapter 3 to some support and test algorithms.

Black Box:	The Black Box algorithm defined in Figure 3.1. No optimizations are used in this implementation. It is used for the validation of derivative results.
Black Box with checkpoints:	The Black Box algorithm defined in Figure 3.1. This version can use checkpoints of the state y such that only one iteration of G needs to be taped. If necessary the algorithm will generate the checkpoints or can use checkpoints that have been written in a previous run. The computation of $\frac{\partial G}{\partial x}$ can be disabled.
Reverse Accumulation:	The Reverse Accumulation algorithm defined in Figure 3.2. The computation of $\frac{\partial G}{\partial x}$ can be disabled or performed every i-th step.
Write checkpoints:	A simple algorithm that writes the state of y as a checkpoint every i-th step during a primal computation of TRACE. There is also a checkpoint written when the solver is finished.
Tangent mode:	Allows the computation of derivatives with the forward AD mode. Several values in y and x can be configured that are seeded with 1.0. For each point the derivative with respect to f is calculated. The algorithm is currently only used for the validation of the adjoint algorithms.
Finite differences:	Allows the computation of derivatives with a finite difference approximation. Several values in y and x can be configured and several step sizes ϵ. For each point and each step size the finite difference approximation of f is calculated. The algorithm is only used for the validation of the adjoint algorithms.
Checkpoint test:	A simple algorithm that tests if the checkpoints can restore the full state of TRACE. In the first run it writes a checkpoint after each i-th step and the state of TRACE in each step. The second run loads the checkpoints and compares the state of TRACE with the first run.

Therefore, no dependencies should be missed when the forward AD mode is used. The results produced should be the same as the results produced by the reverse AD mode up to machine precision.

Finite differences can be evaluated with any program and the program needs no modifications. The definition comes from Taylor's theorem [Apo69] and with $F : \mathbb{R} \to \mathbb{R}$, $x_0 \in \mathbb{R}$ and $\epsilon \in \mathbb{R}$ the Taylor expansion of F is:

$$F(x_0 + \epsilon) = F(x_0) + F'(x_0)\epsilon + \mathcal{R}(x_0 + \epsilon)\epsilon \tag{4.43}$$

where $\mathcal{R}(x_0+\epsilon)$ is the remainder term. For the remainder term holds $\mathcal{R}(x_0+\epsilon) = o(|\epsilon|)$ for $\epsilon \to 0$. This means that $\mathcal{R}$ is asymptotic negligible with respect to ϵ. Equation (4.43) can be written as

$$F'(x_0) = \frac{F(x_0 + \epsilon) - F(x_0)}{\epsilon} + \mathcal{R}(x_0 + \epsilon) \ . \tag{4.44}$$

The remainder term $\mathcal{R}(x_0 + \epsilon)$ gets small for small values ϵ and therefore the simplification

$$F'(x_0) \approx F'_{FD} := \frac{F(x_0 + \epsilon) - F(x_0)}{\epsilon} \tag{4.45}$$

holds for small ϵ. Formulation (4.45) can be used to approximate the derivative of a program by disturbing the parameter x. The major problem with this approach is how large the step size ϵ should be chosen. If ϵ is too large, then the term $\mathcal{R}(x_0+\epsilon)$ is still too large and the derivative approximation is of low quality. If ϵ is too small, then the floating point precision of the processor will produce rounding and cancellation errors. This leads to numerical noise for very small values of ϵ. Usually, there exists an optimal ϵ^* that minimizes the error in $\mathcal{R}$ and the numerical noise. The example in Figure 4.15 shows the comparison of finite differences approximations with analytical derivatives for the function $F(x) = x^d$ with $x \in \mathbb{R}$ and $d = 1, 2, 5$. For the nonlinear functions, the typical V-shaped curve is visible. The right branch shows the error of the remainder term R and the left branch the numerical noise. The kink of the V defines the optimal step size ϵ^* for the finite difference. This simple example shows that the optimal step size ϵ^* is different for every function. If a step size ϵ is fixed in a numerical program, then the resulting derivative can not be guaranteed to be correct. Only if multiple step sizes are evaluated for every point, an analysis can be made that will show if the derivative is correct. If this is not done, then the approximated derivatives may even have the wrong sign, which gives an optimizer a totally different intuition for the correct direction. Therefore, finite differences are only a good tool for validation and not for productive use.

The derivative validation for TRACE is performed with the feature minimal test case. It is a model of one quasi stage (stator – rotor) cut out of a high-pressure compressor.

The model of the stator is an approximation of a variable stator vane (vane pitch adjustable). The necessary gap on hub and tip is modelled as a half-gap configuration. The rotor is modelled with a fillet at the hub and a radial gap at the tip. The leakage

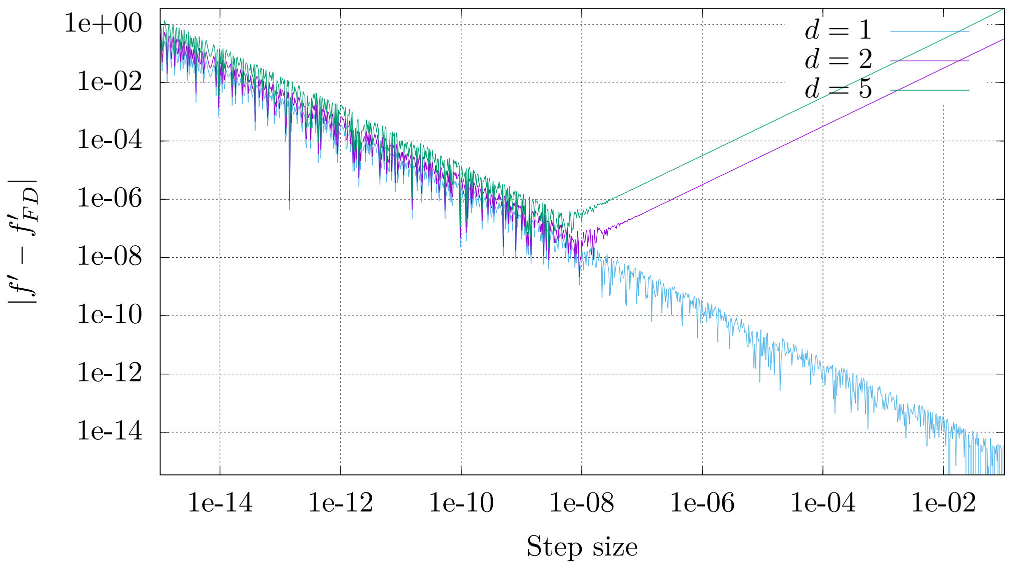


Figure 4.15.: Error of the finite differences and the analytical derivative for the function $F(x) = x^d$ with $d = 1, 2, 5$.

flow, from downstream to upstream of the stator through a so called cavity is only approximated via a downstream bleed and a corresponding leak-inlet upstream.

The quasi stage leads to a stagnation pressure ratio of roughly 1.5. While the inflow to the stator is subsonical, the inflow to the following rotor is transonic in the rotational frame of reference leading to a maximum mach number of 1.22. The inflow Reynolds number is around half a million.

The setup is chosen to contain several features that are required by the current processes at MTU Aero Engines. Figure 4.16 shows the grid of the test case. It contains only 29,000 grid points. This coarse geometry is chosen to make fast testing and evaluations possible.

The primal solution converges after 600 steps and the steps 550 to 600 are chosen for validation with finite differences. For each point in Table 4.3 the finite differences for 500 step sizes in the range from 10^{-8} to 10^{-2} are computed. The results are compared with the derivatives from the forward and the reverse mode of AD. It can be seen that both AD modes are in very good agreement. For each point the finite differences show an error that is below one percent and each of these points shows a typical V-shaped curve, as expected.

4.6. Automated testing

The process of calculating target functionals like the drag can be seen as a function call $z = h(x)$ and on this function AD is used to calculate the derivatives without

Table 4.3.: Comparison of finite differences(FD), AD forward and AD reverse derivatives for 12 grid points in each space direction (x, y, z) of the Feature Minimal test case over the steps 550 to 600.

Block	x Index	FD	AD forward	AD reverse	step size	FD rel. error
0	12741	-9.93827e-04	-9.840295342424283e-04	-9.840295342424068e-04	1.6293e-04	$< 1\%$
0	12742	-1.94559e-03	-1.926923009111530e-03	-1.926923009111562e-03	5.2000e-04	$< 1\%$
0	12743	+6.10472e-05	+6.057367886269012e-05	+6.057367886269113e-05	3.4674e-03	$< 1\%$
0	9906	-1.09503e-03	-1.084217532959289e-03	-1.084217532959235e-03	1.3932e-05	$< 1\%$
0	9907	-1.13301e-03	-1.121879194225534e-03	-1.121879194225533e-03	6.4863e-06	$< 1\%$
0	9908	+5.60042e-05	+5.656459769228783e-05	+5.656459769228634e-05	8.7902e-05	$< 1\%$
1	2514	+8.20504e-04	+8.241730026912932e-04	+8.241730026912700e-04	7.9433e-07	$< 1\%$
1	2515	+3.77915e-03	+3.784372969565374e-03	+3.784372969565362e-03	3.4041e-03	$< 1\%$
1	2516	-3.78383e-04	-3.746682593523982e-04	-3.746682593524004e-04	2.2909e-05	$< 1\%$
2	2514	-2.17008e-03	-2.188174301876196e-03	-2.188174301876189e-03	2.7040e-03	$< 1\%$
2	2515	-5.53146e-03	-5.482691123449521e-03	-5.482691123449522e-03	3.0761e-03	$< 1\%$
2	2516	+1.76672e-04	+1.784532039010828e-04	+1.784532039010839e-04	5.3951e-04	$< 1\%$
3	15441	-1.77607e-03	-1.758521048880018e-03	-1.758521048880045e-03	3.5318e-06	$< 1\%$
3	15442	+1.59102e-02	+1.607043377910989e-02	+1.607043377910983e-02	3.1915e-06	$< 1\%$
3	15443	+7.84922e-05	+7.928449550463615e-05	+7.928449550463516e-05	4.8306e-05	$< 1\%$
3	13146	+8.69935e-02	+8.704547303425419e-02	+8.704547303425446e-02	1.6904e-03	$< 1\%$
3	13147	+6.37233e-02	+6.362224649023705e-02	+6.362224649023807e-02	1.7061e-03	$< 1\%$
3	13148	-1.93677e-03	-1.956245453012186e-03	-1.956245453012188e-03	1.1912e-03	$< 1\%$
11	3651	-9.13391e-04	-9.044226218009399e-04	-9.044226218009224e-04	5.2966e-05	$< 1\%$
11	3652	+1.29016e-03	+1.300759010443960e-03	+1.300759010443999e-03	1.6444e-04	$< 1\%$
11	3653	+7.09788e-05	+7.110983560164373e-05	+7.110983560164517e-05	9.2897e-04	$< 1\%$
12	15567	+7.28853e-02	+7.362090304300935e-02	+7.362090304301001e-02	8.7902e-06	$< 1\%$
12	15568	-1.31505e-02	-1.302096850748543e-02	-1.302096850748543e-02	3.4041e-04	$< 1\%$
12	15569	-1.92571e-02	-1.906854778583585e-02	-1.906854778583594e-02	8.5507e-05	$< 1\%$
12	15060	-1.97093e-02	-1.951499719409957e-02	-1.951499719409970e-02	5.2966e-05	$< 1\%$
12	15061	+1.72878e-02	+1.746159950328999e-02	+1.746159950329003e-02	5.8076e-06	$< 1\%$
12	15062	+6.49588e-03	+6.561126741571357e-03	+6.561126741571351e-03	6.3096e-05	$< 1\%$
13	16074	-1.54926e-02	-1.564819627303708e-02	-1.564819627303704e-02	1.5849e-04	$< 1\%$
13	16075	+1.69630e-02	+1.679649406374312e-02	+1.679649406374306e-02	9.6383e-05	$< 1\%$
13	16076	+4.23409e-04	+4.192273542316657e-04	+4.192273542316629e-04	1.0471e-03	$< 1\%$
13	15567	-9.14126e-02	-9.233223191906242e-02	-9.233223191906310e-02	3.1915e-06	$< 1\%$
13	15568	+1.70553e-01	+1.688779827523305e-01	+1.688779827523305e-01	1.0471e-05	$< 1\%$
13	15569	-4.08541e-02	-4.126453830305685e-02	-4.126453830305681e-02	2.3335e-05	$< 1\%$
14	3057	+2.01789e-02	+2.038131593167954e-02	+2.038131593167954e-02	1.5136e-05	$< 1\%$
14	3058	-1.05988e-01	-1.049393389412271e-01	-1.049393389412267e-01	5.2481e-06	$< 1\%$
14	3059	+2.80998e-02	+2.838122929631948e-02	+2.838122929631952e-02	3.0761e-05	$< 1\%$

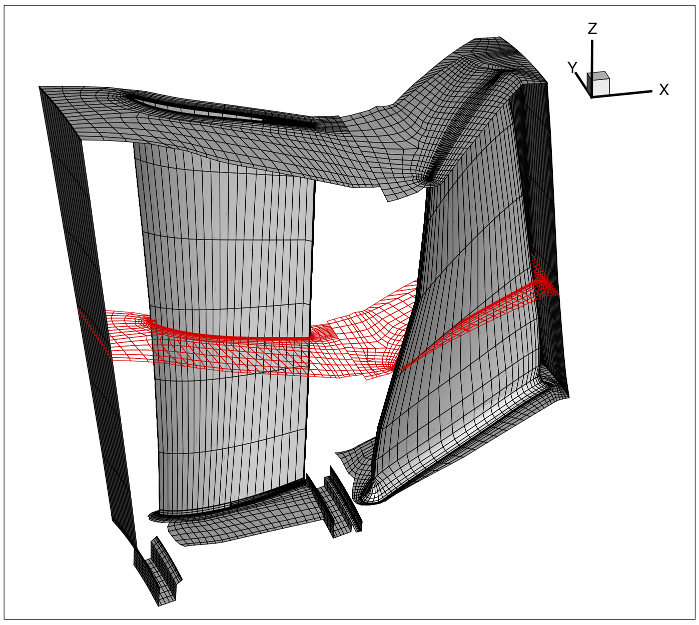


Figure 4.16.: Grid of the feature minimal test case.

exploiting any structural information or using any checkpointing technique. This process should always yield the correct results. If h is then split into its algorithmic components and problem $(\mathcal{P})$ from Chapter 3 is used as the general framework, then the schedule from the Black Box method (Alg. 2) can be used to compute the same derivatives. Before each call to $\frac{\partial G}{\partial y}$ and $\frac{\partial G}{\partial x}$ the state y has to be loaded and G is reevaluated in order to record the tape. If after this process the result from the Black Box method and the application of AD on h do not yield the same results, then this can have three reasons:

- The first reason can occur when the primal state is not restored properly and therefore a different calculation is performed.
- The second reason can occur when the adjoint state $\bar{y}_i$ is wrong and therefore the update to the previous state $\bar{y}_{i-1}$ is also wrong.
- The third reason can come into effect when using the control x in the process. If e.g. there is a function $M : X \to X' \subset \mathbb{R}^p$ with $p \in \mathbb{N}$ that computes some precomputed mesh metrics and this function is not properly handled, not all the dependencies on x will be seen in a checkpointing scheme.

The three items are discussed in the following sections.

4.6.1. Validation of the state

The theory always assumes that the state vector y is well defined and known by the developers. If for example the regular fixed-point iterator $G(y, x)$ is changed to

$G_s(p, q, x)$ with $y = (p, q)^T$ and if then a checkpointing scheme is applied to p and not to q, the full state of y is not restored and the reevaluation of G_s will yield wrong results. Why y is not fully known can have several reasons. In legacy codes not everything is documented, monolithic design structures or globally defined variables make it hard to track which variables are inputs and which are recomputed. It is possible that at the end of G some values (e.g. boundary averages) are computed that are then used in the next step of G (e.g. convergence criteria). These values come not to mind if one thinks about the state but are still necessary for a valid state vector.

The assumption in this section is that the control vector is known and an initial guess for the state vector is made.

The initial guess in this case is p and every variable in the program that corresponds to p is tagged. If 1 is used as the tag, when G_s is called, every computation that makes use of a tagged variable will tag the resulting value with 1. When the iteration of G_s finishes, the variables of p and all variables that were computed from these variables will have been tagged with 1. For the next iteration, p is tagged with 2 and G_s is called again. New variables from computations will receive the tag 2. But it can now happen, that in a computation a variable with the tag 1 will be used. Such a variable can occur when it is part of the computation and used before it is recomputed. These variables are from the unknown state q.

The process is explained again in a small example, that is:

```
void iter(double a, double b, double& a_new, double& b_new) {
 double t = a * b;
 a_new = b + t;
 b_new = a + t;
}
```

If a is tagged with 1 and *iter* is called, then t is also tagged with 1 and so are the outputs *a_new* and *b_new*. Afterwards a is tagged with 2 and *iter* is called again with the updated variables. The computation for t will now encounter the variable b, that has been tagged by the previous iteration with 1. In order to have the full state of *iter*, b needs to be added to the state. It is now possible to restore the full state of *iter*. If for example, after the 1000-th step of *iter*, the 500-th step needs to be restored, it is necessary to restore a and b. If only a is restored, then the results of the 500-th step will be wrong.

This concept is a simple technique to validate the state y of a fixed-point solver, when it is unclear if the full state y is captured. The question now is how will the variables be tagged. The solution can be quite easily implemented with the techniques of AD. Each variable in the program is replaced with an AD variable like the *RReal* and this type contains already an extra value. Additional members can be added to represent the tag and the AD operations can be adapted to update the tag.

It is now possible to tag each variable in the application and this capability can be used to identify the correct state, but it also necessary to discuss the problem of a wrong state from an AD perspective. The indices of AD are used to perform the tagging. So the application can encounter old indices from the last iteration. If a new tape is generated and starts with the index 1, the next operation will have the index 2 and so on. Eventually one of the computations will encounter an old index. Lets assume it is the last index that was used in the previous iteration and has the value $1,000,000$. The reverse interpretation of this tape will then write to the adjoint vector at the position $1,000,000$ which is clearly wrong. Yet, as the adjoint value at this position is not used again, nothing would change. In another case it could be that the old index is smaller than the currently used index. The reverse interpretation will then write to a position in the adjoint vector, which will in turn be used in an update and definitively change the result. It is very harmful if an AD tool uses indices from old iterations. The selection of the tags and the process of identifying the missing state q are now discussed.

How each index is tagged is not important and it can be done by either reserving the first or last j bits in each integer or by using a structure as the index which contains the regular index and an extra field for the tag. The assumption is that there are j bits available for the tagging.

The first option is to tag each index with the current id i from the iteration. The function

$$\text{tag} = (i \mod 2^j - 1) + 1 \tag{4.46}$$

assigns all values from 1 to $2^j - 1$ to the tag without the zero value. If the tape encounters an old tag in a right hand side expression it can throw an error. The user is then able to identify the missing value in the state of p and extend it to $\hat{p}$. If the process is repeated, all errors are eliminated step by step and p converges to the state y. This method can also be used to automatically test for incorrect states in a test suite for the application.

The next source of errors could originate from the communication of processes. The indices of each process can be interpreted as pointers in the memory and possess no meaning in any other process. The same tagging technique from above can be used to identify errors that originate from such sources. If the tagging

$$\text{tag} = (\mathit{processId} \mod 2^j - 1) + 1 \tag{4.47}$$

is used, then all values from 1 to $2^j - 1$ are used and the AD tool can detect values from other processes. It should be ensured that $\mathit{processId} < 2^j$ holds. If not, then two processors next to each other can have the same tag.

An index that is not properly initialized, will contain a random value. If this uninitialized index is used as it is, then the correct behavior of the tape can no longer be guaranteed. If the indices are tagged, then it is very unlikely that the random value contains the correct tag. Therefore, the tagging technique can also be used to detect uninitialized values in the computation.

With tagging, nearly all problems with the state vector can be analyzed or detected. If a total Black Box taping still yields different results as a Black Box scheme with reloaded states, further tests must to be developed.

```
# Comments start with a # at the beginning of the line

# for each statement that is recorded on the tape
<number of arguments> <Jacobians of all arguments> [<index of lhs>
    <indices of all arguments>] [<code position of the statement>]

# number of arguments:
#    The number of arguments on the right hand side of the statement.
#    May be already optimized such that zero Jacobies etc.
#    are not counted.
# Jacobies of all arguments:
#    For each argument on the right hand side of the statement the computed
#    Jacobie value by the tape. Have to be the same count
#    as <number of arguments>.
# index of lhs:
#    The index of the assigned value on the left hand side of the
    statement.
# indices of all arguments:
#    For each argument on the right hand side of the statement the
    associated
#    index with this variable. They should have the same order as
#    the Jacobies. Have to be the same count as <number of arguments>
# code position of the statemnt:
#    The line and file where the statement is defined.
```

Figure 4.17.: Output format for Jacobi tapes.

4.6.2. Validation of the adjoint state

If the results from the AD application on h and the Black Box method differ, then the recorded tapes from both algorithms have to be different. The tape from the Black Box method contains the evaluation of G at the point y_i for $i \in \mathbb{N}$. The tape for the AD application on h will contain all evaluations from G for all points y_0 to y_{j+1}. This tape can be split such that each part contains just one G iteration. The tape of f should be separated, too. After this procedure there should be the files *tape_bb_full_##.txt* and *tape_bb_check_##.txt* for the AD application on h, and the Black Box method respectively with ## ranging from 0 to $j+1$ where $j+1$ contains the tape for f.

A good format for Jacobi tapes is shown in Listing 4.17. It stores the minimal information and can also be used with binary data. The Jacobians can be compared in order to check if the tapes are identical. The information for the indices is not really required but can be used to traverse the file and analyze the dependencies of the statements. The code line of the statement can be used to see where the tape entry is recorded.

After the files have been written the comparison can start. The order in which the files are compared depends upon the problem. If it is important to find the first difference in the variable dependencies, then the comparison should start with the first primal iteration. On the other hand, if it is important to find the first differences in the adjoint output, then the comparison should be started with f.

The comparison process is the same for both approaches. One tape has to be declared the “master” tape. For the master the derivatives are validated and the user knows that the tape is correct. This is usually the tape from the Black Box process without any checkpointing techniques. The other tape is the “secondary” tape which is considered to contain the errors.

In order to yield the same results every statement that is on the master tape has to be on the secondary tape. This statement is proven in the following theorem, but first a definition is required for the correctness of a tape interface. In Section 4.6.1 procedures for the validation of the state were shown and a correct state was defined in the sense, that all output values need to depend upon at least one input value.

Definition 4.1

For a tape that is created by the function $y = F(x)$ with $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, the interfaces defined by the input variables x and the output variables y are *correct* if and only if for all $i \in [1, m] \subset \mathbb{N}$ there exists an index $j \in [1, n] \subset \mathbb{N}$ such that $\frac{\partial F_i}{dx_j} \neq 0$.

With this definition the theorem for the inclusion of the tapes can now be stated.

Theorem 4.2

Let $p = F(a)$ be the definition of the master tape (M) with $a \in \mathbb{R}^{n_M}$ as the input interface and $p \in \mathbb{R}^{m_M}$ as the output interface. Let $y = F(x)$ be the definition of the secondary tape (S) with $x \in \mathbb{R}^{n_S}$ as the input interface and $y \in \mathbb{R}^{m_S}$ as the output interface. Let F be in both cases the same function that is used with different sets of parameters. Without loss of generality let $x = (a, b)$ with $b \in \mathbb{R}^{n_S - n_M}$ and $y = (p, q)$ with $q \in \mathbb{R}^{m_S - m_M}$. Let the interfaces p and a of the master tape be correct.

Then $\bar{q} \neq 0$ has no influence on $\bar{a}$.

Proof. The adjoint update of F is after Theorem 2.13

$$\begin{pmatrix} \bar{a} \\ \bar{b} \end{pmatrix} = \frac{\partial F}{\partial(a,b)}^T (a,b) \begin{pmatrix} \bar{p} \\ \bar{q} \end{pmatrix} \quad \hat{=} \quad \bar{x} = \frac{\partial F}{\partial x}^T \bar{y} \; . \tag{4.48}$$

The Jacobi matrix of F can be identified as

$$\frac{\partial F}{\partial(a,b)}^T (a,b) \begin{pmatrix} \bar{p} \\ \bar{q} \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} \bar{p} \\ \bar{q} \end{pmatrix} \tag{4.49}$$

with $A \in \mathbb{R}^{n_M \times m_M}$, $B \in \mathbb{R}^{n_M \times (m_S - m_M)}$, $C \in \mathbb{R}^{(n_S - n_M) \times m_M}$ and $D \in \mathbb{R}^{(n_S - n_M) \times (m_S - m_M)}$. It is only possible that $\bar{q}$ has no influence on $\bar{a}$ if B is the zero matrix, which is proven with the assumption of its contraposition.

Let the assumption hold that $\bar{q}$ has an influence on $\bar{a}$ which implies $B \neq 0$. Therefore there exist two indices $i \in]m_M, m_S] \subset \mathbb{N}$ and $j \in [1, n_M] \subset \mathbb{N}$ such that

$B_{j,(i-m_M)}$ is not equal to zero. Because B is a submatrix of $\frac{\partial F}{\partial(a,b)}$ it holds $\frac{\partial F_i}{\partial x_j} \neq 0$. The parameter x_j is an input parameter of the master because j is in the range $[1, n_M]$ and can therefore be identified as the parameter a_j. It is now obvious, that the parameter a_j has an influence on the output value y_i because the derivative $\frac{\partial F_i}{\partial x_j}$ is nonzero. This makes the output y_i a member of the output interface of the master tape and needs to be included in the output interface p. The interfaces of the master tape are assumed to be correct and therefore the index i must be smaller or equal to m_M. This is a contradiction against the original assumption $i \in]m_M, m_S]$.

Therefore it holds, that if the interfaces of the master tape are correct, then B is equal to zero. This reduces the update (4.48) to

$$\begin{pmatrix} \bar{a} \\ \bar{b} \end{pmatrix} \mathrel{+}= \begin{pmatrix} A\bar{p} \\ C\bar{p} + D\bar{q} \end{pmatrix} \tag{4.50}$$

and shows that $\bar{q} \neq 0$ has no influence on $\bar{a}$. □

Remark 4.3

It is very important that in both tapes the same function F is used. Otherwise the proof is not possible.

The conditions of the theorem apply in our case. The master tape (M) and the secondary tape (S) are based upon the same function (F), which corresponds to the fact, that both are evaluated by the same code. The master tape can be assumed to be correct. The result of the theorem is then, that the reverse interpretation of both tapes yield only the same results for $\bar{a}$ if the sub matrix A is recorded in both tapes. This is identical to stating that every statement of the master tape has to be on the secondary tape. In addition, the theorem guarantees that every additional statement on the secondary tape will not influence the result of the reverse evaluation.

Therefore, the tape comparison process needs to ignore all statements that are not in the master tape but are in the secondary tape. The same is true for the Jacobians in the statements. All additional Jacobians that are on the secondary tape and not on the master tape need to be ignored.

Remark 4.4

In order to be able to see the dependencies in the tape files directly, it has been shown to be more efficient to disable the assign optimizations in the AD tool (see Sec. 6.9). Errors are then generated the first time a variable is copied and not the first time it is used in a statement.

4.6.3. Validation of interfaces

In this context the assumption is, that there are some additional control values w which are precomputed by the function $M : X \to X' \subset \mathbb{R}^p$ with $p \in \mathbb{N}$. The extended algorithm from (Alg. 2) with the function M would look like the listing of Algorithm 3. The tagging technique from Section 4.6.1 of this chapter can be generalized to identify interfaces and therefore the one for w.

The tag for all variables in x is set to one and then M is called. Now all values that depend on x will have the tag one. The global tag is then set to two and is set on x as well as the known parts of w. The computation is continued and when a value with tag one is used it is an indication that it belongs to the interface w. The process is also shown in Algorithm 4.

Algorithm 3 Procedure for evaluating the gradient of the object function f for a given fixed-point iterator G with some precomputed variables w.

```
Input: $y_0, x$
Output: $z, y_{j+1}, \bar{x}_0$
  $w = M(x)$
  for $i = 0, i \leq j$ do
    $y_{i+1} = G(y_i, x, w)$
  end for
  $z = f(y_{j+1}, x, w)$

  $\bar{z} = 1$
  $\bar{y}_{j+1} = \frac{\partial f}{\partial y}(y_{j+1}, x, w)^T \bar{z}$
  $\bar{x}_{j+1} = \frac{\partial f}{\partial x}(y_{j+1}, x, w)^T \bar{z}$
  $\bar{w}_{j+1} = \frac{\partial f}{\partial w}(y_{j+1}, x, w)^T \bar{z}$
  for $i = j, i \geq 0$ do
    $\bar{y}_i = \frac{\partial G}{\partial y}(y_i, x, w)^T \bar{y}_{i+1}$
    $\bar{x}_i = \frac{\partial G}{\partial x}(y_i, x, w)^T \bar{y}_{i+1} + \bar{x}_{i+1}$
    $\bar{w}_i = \frac{\partial G}{\partial x}(y_i, x, w)^T \bar{y}_{i+1} + \bar{w}_{i+1}$
  end for
  $\bar{x} = \frac{\partial M}{\partial x}(x)^T \bar{x}'_0 + \bar{x}_0$
```

4.7. Test case description

4.7.1. CRESENDO turbine

The CRESENDO low pressure turbine was created in the EU project CRESENDO (collaborative and robust engineering using simulation capability enabling next design optimization) in 2009. The 3-year research project was lead by Airbus and established a collaboration of 59 organizations from 13 countries. The aim of the project was to improve the collaboration of the companies in order to improve the efficiency of new turbine designs and to reduce the cost of the design and its construction processes.

MTU Aero Engines constructed during the project a 5 stage low pressure turbine, now called CRESENDO. The turbine contains a turning mid turbine frame (TMTF) that performs the transition of the flow from the height pressure turbine to the low pressure turbine. The vane also contains the service line, which is the support for the turbine shaft and the oil supply. In order to reduce the weight of the turbine, the

Algorithm 4 Procedure for detecting an interface with the help of variable tagging.

```
Input: $x$, func PreInterface, func PostInterface
Output: Interface $w$
  for $i = 1, ....$ do
    Set global tag to $(2 * i - 1)$
    Set tag $(2 * i - 1)$ on $x$
    call PreInterface
    Set global tag to $(2 * i)$
    Set tag $(2 * i)$ on $x$
    Set tag $(2 * i)$ on $w$
    call PostInterface
    if Tag $(2 * i - 1)$ used then
      Add variable to $w$
    else
      return $w$
    end if
  end for
```

TMTF performs a small redirection of the flow of about 40 degrees. A usual stator redirects the flow of about 90 degrees but requires an additional row and therefore extra weight. The resulting flow regime is very complex and contains strong secondary flows. This makes the case very interesting for design optimization, because several constraints need to be satisfied in order to meet the spacial requirements for the service line and to produce a stable flow. The CRESENDO test case in the thesis consists only of the first stage of the full 5-stage design. The first stage contains the TMTF and the following rotor B3 (short for blade 3). The mesh of the TMTF is very fine in order to resolve the secondary flows, this allows for a low-Reynolds modeling [WJ00] of the walls. The B3 rotor is meshed quite coarse and therefore wall functions [Röb13] are used for the wall modeling. The turbulence model is the Wilcox k-ω 1998 model [Wil88] and the multi-mode transition modeling scheme [BHK10] is used at the pressure and suction side of the TMTF. The total grid consists of 1.7 million cells (1.6 million TMTF, 0.1 million B3), the mach number of the inflow is $Ma_0 = 0.4$ and the Reynolds number is $Re = 800,000$. The case runs with 4650 rotations per minute and a total pressure ratio of $\frac{\rho_{in}}{\rho_{out}} = 1.27$.

4.7.2. THD turbine

A single compressor stage is chosen as a second test. The setup originated from an experimental investigation performed at the Technical University of Darmstadt and is referred to as THD Rotor 1. It is characterized by a maximum pre-shock Mach number of 1.4 at a design speed of 20000 rpm corresponding to a blade tip velocity of approximately 400 m/s. At design speed the blade tip operates supersonic, while the hub runs at subsonic conditions. The blades are highly cambered at the hub

Table 4.4.: Time and memory measurements for the THD turbine. First unoptimized results from the black box differentiation of TRACE.

	primal TRACE	Black Box	Reverse Accumulation
Time in seconds	1.0		
dco/C++		28.71	6.39
ADOL-C		121.13	41.10
Memory in GB (Factor)	2.51		
dco/C++		184.86 (74)	184.86 (74)
ADOL-C		226.87 (90)	226.87 (90)

and show thin airfoils at the blade tip. The mean aspect ratio of the rotor blades measures 0.94 with a hub-to-tip ratio of 0.51 [BDK08]. The computational domain used in the study is partitioned into a total of 24 blocks and $516,432$ cells are used in the computational grid ($303,772$ cells for the rotor and $212,660$ cells for the stator).

4.8. Memory and time requirement discussion

Since TRACE is now differentiated and the derivative algorithms are implemented, the memory and time performance data can be measured. The results presented here are the first results that were obtained without any optimization on the AD tools or upon the differentiated TRACE code.

The results in Table 4.4 show that the process works but the time and memory consumption costs are quite high. The column “primal TRACE” shows the data for the unmodified TRACE version. The other two columns show the data for the Black Box algorithm (Figure 3.1) and the Reverse Accumulation algorithm (Figure 3.2). The Black Box algorithm has to create a new tape for every G iteration, therefore the displayed time is the sum of the recording and the evaluation time of the tape. For the Reverse Accumulation algorithm the tape is recorded once and evaluated several times.

The required time for the taping of dco/C++ is already high, the full recording and evaluation of the tape is 28 times the factor of a primal simulation. If only the recording time is taken in consideration, then the Reverse Accumulation algorithm has a factor of 6.5 which is not sufficient. If the primal simulation takes 1 day, then the adjoint computation will take one week. The same is true for the memory. The factor with respect to the primal simulation is 74 which is, in this case unacceptable.

The time and memory measurements for ADOL-C are still higher than for dco/C++, which indicates that ADOL-C’s development focus is geared towards academic applications and not large scale applications.

These first results do not look very promising, but that is not unusual when the software is differentiated in a black box fashion. Therefore, the following chapters will discuss how the time and memory requirements of the application can be improved.

5. Memory and time reduction techniques

The previous chapter shows how an application can be differentiated with AD. Until now no assumptions about the structure of the software have been made or exploited. The black box approach from chapter 3 yields derivatives that have two properties. They are consistent and nearly maintenance free. The consistency property is derived from the differentiation of the entire code base with AD while performing no simplification. The maintenance free property can not be guaranteed but the proposed framework from chapter 3 minimizes the effort.

The drawback of the black box approach is that the requirements for memory and computation time are not optimal. This chapter will introduce techniques that can be used to reduce the memory and/or computation time.

In order to know where most of the memory and computation time are used the program needs to be analyzed. The analysis is done with the techniques of AD in order to gather memory and time metrics for AD in an automated fashion. The method for the analysis is built on the ideas of Lotz [LNM16, Lot16], modified such that it works with arbitrarily large cases. The analysis will show regions in the code that consume the greatest amount of resources, these hot spots need to be treated to reduce the recourse requirements. How this is done depends strongly on the code producing the hot spots. It is therefore hard to give general guidelines.

Two general approaches can be identified. The first is called checkpointing and defines an intrusive approach, where a special function is written for the reverse evaluation of the hot spot. This function can make several assumptions about the dependencies and apply simplifications in the derivative computations. Because this approach changes the derivative computations, the derivative results will change. If the assumptions are correct, then the error will be small. If not, the derivative information will be wrong. An example for such a case is an iterative solution process. If the process is treated as a black box then all steps in the iteration are recorded by AD. A first improvement could be the treatment of the iteration process with the Black Box method from Chapter 3. Only the state of the iteration needs to be recorded, or certain steps can be recomputed. This approach still yields exact derivative results. A second improvement could be implemented using the Reverse Accumulation method from Chapter 3, where only the last state needs to be recorded and the derivatives are computed based on the solution of the iterative process. This approach simplifies the derivative computation and if the assumptions for the Reverse Accumulation method are not fulfilled for this iteration then the derivatives can be inconsistent. It is always possible to judge these assumptions, because the black box solution is still available, yielding correct results. A drawback is, that an intrusive approach requires a special implementation which needs to be adapted each time the primal code changes. The intrusive approach breaks therefore the consistency and maintenance free properties.

The second approach is called preaccumulation and is non intrusive. Here the storage of the information for the derivative information is optimized to minimize the required memory. The consistency property of the solution is not broken as this method does not change derivative information, however a certain degree of maintenance might be required. This are most of the time variable lists describing the input and output of a code section.

The two described techniques are presented, and analyzed for their time consumption and how the change affects the memory usage. How both techniques are implemented is shown in small examples and the important aspects for different taping strategies are discussed. A special variant of the checkpointing techniques presented for TRACE, called "block checkpointing", uses the structure of TRACE in order to reduce the memory requirements in a consistent way.

5.1. Call graphs

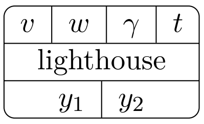
In order to be able to define for which parts of the program time and memory are measured, some basic notions about the program structure are required. Usually every numerical program consists of several functions. These functions can be separated into those that perform operations on data structures and those that drive the algorithms and steer the control flow. The second set of functions generally calls the first set, yielding a natural dependency that is represented as a call graph. The smallest unit in such a graph is a simple function call. This is defined as an evaluation node:

Definition 5.1 (Evaluation Node)
An evaluation node consists of the triplet (x, y, F) were $x \in \mathbb{R}^n$ are the input variables of the node, $y \in \mathbb{R}^m$ are the output variables of the node and $F : \mathbb{R}^n \to \mathbb{R}^m$ is a function that transforms input x into output y. This can be written as

$$y = F(x) \ .$$

The definition of the evaluation node is the definition of a mathematical function with a focused view on input and output variables.

If the function from the lighthouse example in Section 2.3 is called, then the evaluation node $L = ((v, w, \gamma, t), (y_1, y_2), lighthouse)$ would look like:



The first row displays the input variables, the second row the function name and the third row the output variables. A call graph is a collection of multiple evaluation nodes connected to one another. The output variables of one node can be the input

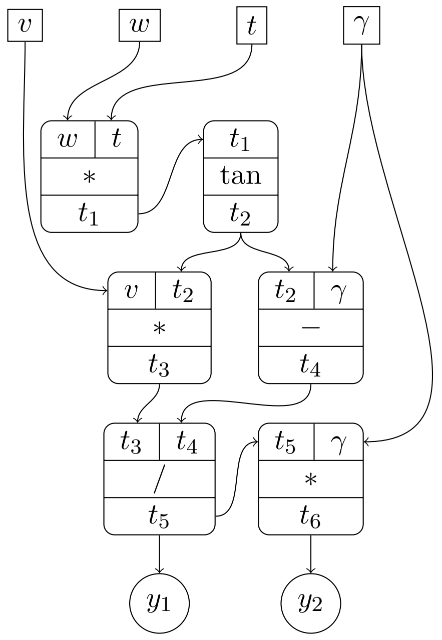


Figure 5.1.: Call graph for the lighthouse example.

variables of a different node. For the definition of a call graph a small example is introduced first. Algorithm 2.6 for the lighthouse example on page 27 is interpreted as a call graph in Figure 5.1. Each statement is interpreted as a function call and displayed as an evaluation node. The input variables are represented as rectangles and the output variables as circles. The dependencies between the nodes are marked with arrows. The definition of a call graph is now:

Definition 5.2 (Call graph)

A call graph consists of $a_1, \dots, a_p$ input vectors with $a_i \in \mathbb{R}^{n_i}, n_i \in \mathbb{N}$ for all $i = 1 \dots p$ and a set of evaluation nodes $\mathcal{N} = \{N_1, \dots N_l\}$ with $l \in \mathbb{N}$. The input $x_i \in \mathbb{R}^{n_i}$ of the evaluation node N_i for $i = 1 \dots l$ is a set which can consist of the input vectors $a_1, \dots, a_p$ and parts of the output vectors $y_j \in \mathbb{R}^{m_j}$ from the nodes N_j with $j = 1 \dots i-1$.

The output vectors $c_1, \dots, c_q$ are defined as a subset of the output vectors from the evaluation nodes, i.e. $\forall i = 1 \dots q \ \exists \ j \in \mathbb{N}$ s.t. $c_i = y_j$ holds.

Remark 5.3 (Directed Acyclic Graph (DAG))

The definition of a call graph is a specialization of a directed acyclic graph [TS92]. Each function represents a node in the graph. The inputs and outputs define the edges of the graph. Each output is the start of an edge and each input defines the end of an edge which yields the directed graph. Because each evaluation node N_i depends only upon the outputs of nodes with smaller indices, no edge can create a cycle in the graph.

One important aspect is that circular dependencies are not allowed in the call graph definition. When a program is viewed from an AD perspective all loops are considered to terminate after a finite number of steps. The loops are unrolled in the graph representation.

The connection between the evaluation nodes and the call graph is such that a call graph is a collection of multiple evaluation nodes. The reverse is also true. Each call graph can be interpreted as an evaluation node. The inputs and outputs of the call graph are the inputs and outputs of the evaluation node, respectively. The algorithm that the evaluation nodes in the call graph represent is the function of the evaluation node. This view on a call graph is a very important point. It enables an AD developer to look at different levels of granularity. The whole software can be seen as one global evaluation node corresponding to the black box approach proposed in Chapter 4. This global evaluation node can be refined into a call graph comprised of top level calls in the software. Figure 4.3 offers an example of this for the TRACE code. This refinement can be repeated until the statement level of the software is reached. If this is done for the whole software the call graph becomes too huge and it is hard to extract meaningful information from the graph. Usually only single evaluation nodes are extracted from a call graph and then only the extracted node is refined.

The lighthouse evaluation node and the call graph for the lighthouse node show this principle quite clearly. A second insight that can be gained from this example, is that each algorithm is also a call graph. The algorithm lacks the nice visual features that are usually associated with a graph, but the representation of the input and output variables are most often much more intuitive. Algorithms and call graphs are used interchangeably in the rest of the thesis.

5.2. Definitions for time and memory measurements

The next sections will examine how these nodes can be handled with special techniques. In order to examine and compare these techniques, the performance parameters of the evaluation nodes need to be identified and defined. The parameters that describe the memory and time consumption of an evaluation node are implicitly defined in Chapter 2 and consist of the total number of statements η_s, the total number of active arguments η_a (from all statements), the total number of constant arguments η_c and the total number of passive arguments η_p. The relations between these variables depends upon the evaluation node but

$$\eta_a > \eta_s \gg \eta_c \geq \eta_p$$

should hold. Each recorded statement has one or more arguments hence the total number of active arguments η_a is greater that the number of statements η_s. Only a few statements contain constant values like 4.0 and therefore η_c is much smaller. In numerical applications the number of passive variables η_p should be low. The memory for the taping process can be expressed as an action that is performed on an evaluation node (x, y, F).

Definition 5.4 ($MEM\{act\}$)
The memory required by the action *act* that is performed on an object ζ, is defined by the symbol

$$MEM\{act(\zeta)\}$$

and expressed in bytes.

Remark 5.5 (Object ζ)
The term “object” in the above definition is intentionally not specified in detail. Each action can define on which objects it can be evaluated. In the context of the thesis the “objects” will be usually functions or structures.

The action of recording a tape is simply denoted as the verb *tape* and with the variables from above the memory can be expressed in general terms as

$$\begin{aligned} MEM\{tape(F)\} =& byte\{statement\} * \eta_s + byte\{argument\} * \eta_a \\ & + byte\{constant\} * \eta_c + byte\{passive\} * \eta_p \ . \end{aligned} \tag{5.1}$$

The amount of memory required for each statement, argument, constant and passive entry is dependent on the AD tool and the implementation chosen for the AD tool. Memory requirements can vary quite strongly and can be seen in the implementations developed in Chapter 6. The four presented techniques, are summarized in Table 6.2 on page 173. The byte values in the table lead to four different memory requirements:

$$\begin{aligned} \text{Jacobi:} \quad && MEM\{tape_{Jac}(F)\} =& \eta_s + 12 * \eta_a \\ \text{Jacobi index reuse:} \quad && MEM\{tape_{JacInd}(F)\} =& 5 * \eta_s + 12 * \eta_a \\ \text{Primal value:} \quad && MEM\{tape_{Prim}(F)\} =& 16 * \eta_s + 4 * \eta_a + 8 * \eta_c + 16 * \eta_p \\ \text{Primal value} \quad && & \\ \text{index reuse:} \quad && MEM\{tape_{PrimInd}(F)\} =& 20 * \eta_s + 4 * \eta_a + 8 * \eta_c + 20 * \eta_p \ . \end{aligned}$$

The memory denoted by the *tape* action is accessed in a sequential manner and can be stored in sequential access memory (SAM).

The time for the taping process is the next critical quantity and is expressed with the same symbol as in [GW00].

Definition 5.6 ($TIME\{act\}$)
The time required by the action *act* that is performed on an object ζ, is defined by the symbol

$$TIME\{act(\zeta)\}$$

and expressed in seconds.

It is difficult to express the required time, that is taken by the action *tape* and is not investigated here. Griewank’s book [GW00] and the papers [Gri93, Gri09] make a first approach on the topic. The author of this thesis made also a first step [SAG16] to further refine the time approximation of Griewank by performing measurements for the parameters that Griewank used in his analysis. In this thesis only a simple model for the required time is assumed. The time of the taping is only correlated to the memory bandwidth mem_{speed} of the machine and therefore the general assumption

$$TIME\{tape(F)\} \geq \frac{MEM\{tape(F)\}}{mem_{speed}} \tag{5.2}$$

should always be true and gives a lower bound for the time of the taping process.

The action for the reverse evaluation process is *reverse* and the memory that is required by this action can be expressed in general terms as

$$MEM\{reverse(F)\} = byte\{double\} * \eta_i + MEM\{tape(F)\} \ . \tag{5.3}$$

The total number of indices η_i depends on the indexing scheme used. For tapes that reuse indices the number is usually quite small and it holds $\eta_s \gg \eta_i$. For all other tapes the total number of indices is equal to the number of total statements: $\eta_s \equiv \eta_i$. For the four techniques the specific numbers are:

$$\begin{aligned} MEM\{reverse_{Jac}(F)\} =& 8 * \eta_s + MEM\{tape_{Jac}(F)\} \\ MEM\{reverse_{JacInd}(F)\} =& 8 * \eta_i + MEM\{tape_{JacInd}(F)\} \\ MEM\{reverse_{Prim}(F)\} =& 8 * \eta_s + MEM\{tape_{Prim}(F)\} \\ MEM\{reverse_{PrimInd}(F)\} =& 16 * \eta_i + MEM\{tape_{PrimInd}(F)\} \ . \end{aligned}$$

The additional memory for the reversal action is accessed randomly and can be denoted as random access memory (RAM). The advantage of tapes that reuse indices can now be seen. They require more memory during the *tape* action but require less during the *reverse* action. Because the random access memory is smaller, the chance for cache misses is reduced and the overall process should be faster. The time for the reversal action is written in the same fashion as the record action. That is

$$TIME\{reverse(F)\} \geq \frac{MEM\{reverse(F)\}}{mem_{speed}} \ .$$

From the stated relations one could deduce that the time for the reverse interpretation is higher than the time for the recording, but usually the exact opposite is the case. The reason is, that in (5.2) the memory movements and computations of the primal application have been neglected.

The important result from the discussion is, that the time for the *tape* and *reverse* action are directly connected to the amount of memory that both actions generate and use. Everything that reduces the required amount of memory reduces the time required for both actions.

5.3. Hotspot analysis for Algorithmic Differentiation

Parameters defining the memory and time for an evaluation node need to be available for every function. If this is done by hand, the process is time consuming and prone to errors. This makes it necessary to automate the process of locating resource intensive hot spots.

A direct analysis of the source code is difficult and requires sophisticated software to perform the analysis. This software would need to be extended to incorporate the overhead and data for AD. As such software is not available, the implementation techniques of AD are used to gather data for the analysis.

The original idea of an analysis tape is implemented in dco/C++ [LLN16]. In order to apply it to large cases the data layout and file layout need to be changed to give reasonable runtimes. The presented implementation has only a slight overhead in memory and runtime. It can therefore be used to analyze arbitrary large cases. The proper tracking of output variables, which is not implemented in the dco/C++ version, is developed in this chapter and a novel addition to this technique.

For the analysis type the operator overloading technique is used. Section 2.6 contains a short introduction to that approach. A new type *AReal* (Analysis Real) is created with the signature:

```
structure AReal { // Analysis Real
  double p;
  Data data;
};
```

The member *Data* will contain additional items used for the analysis. The type *AReal* requires an implementation for the mathematical functions sin, cos, etc. and the binary operators $*$, $+$, etc. Before these functions can be implemented, a definition is needed for the required data.

For an evaluation node F this is the number of input variables, the number of output variables and the required memory of the tape. The memory of the tape can be described with the counts η_s, η_a, η_c and η_p from the previous section. The base implementation will address the gathering of these variables. How the number of input and output variables are detected will be discussed after the base implementation is finished.

This leaves two requirements for the online analysis

- For each function the data must be gathered
- The counts for η_s, η_a, η_c and η_p are needed

The first requirement is handled such that each function calls a function *funcEnter* immediately after the function is entered and calls a function *funcLeave* directly before it leaves the function. This can be done by using the technique *function instrumentation* which can be enabled in the gcc compiler with the flag `-finstrument-functions`. This will create the calls to *funcEnter* and *funcLeave* automatically. The implementation for the online analysis is shown in Figure 5.2. The algorithm creates an entry on the stack for each function used while the *AReal* type is active and gathers the information. When the function is finished the stack entry is removed and the data can be processed. The analysis will be done in a separate tool, therefore the data is written to a file.

```
void funcEnter() {
 tape.funcStack.push(new FuncData());
}
void funcLeave() {
 FuncData data = tape.funcStack.top();
 writeFuncData(funcName, data);

 tape.funcStack.pop();
 if(!tape.funcStack.empty()) {
   tape.funcStack.top().add(data);
 }
}
```

Figure 5.2.: Handling of the function scope for the analysis tape.

The *add* call on the parent function is currently not very intuitive but will become clear after the implementation of the overloaded functions.

The overloaded function implementations for the *AReal* type will access the top element of the stack and increase the counts for the appropriate variables. Therefore the definition of *FuncData* contains an entry for all four counters. The implementation for the overloaded functions is done exemplary for the multiplication and the sinus function in Figure 5.3. This will ensure that the correct counts for all counters are available and the execution of the program is still the same. This implementation adds an “active” member to the *AReal* structure. The member is used to track which variables are considered input variables. Inactive variables increase the statement’s passive variable count. It is also used to skip the update for statements that have no active variables.

The implementation will only record the counts for the current function that is at the top of the stack. Because the parent function needs the same updates in order to have the full counts available and not only the counts that are local to the parent, a loop over all stack elements is be required. Since, each element receives the same update the overhead from this approach can be substantial. A more efficient implementation can be reached if the counts for the variables are added to the parent function when the child function is finished. The *add* function in Figure 5.2 demonstrates this.

These definitions yield a good overview of the involved functions and the basic setup of the analysis type. Information on the call graph, input and output variables are still missing. These additions are discussed in the following sections.

5.3.1. Efficient data layout

The data structure for the call graph needs to be designed in such a way that it represents the function hierarchy nicely and is efficient for reading and writing the data. Call graphs are represented in the dot [GKN06] format or other formats like GML [Him97], GXL [WKR02], etc.. These formats are quite verbose so the focus is

```
struct FuncData {
 size_t statements;
 size_t arguments;
 size_t constants;
 size_t passives;
};

AReal operator* (const AReal& a, const AReal& b) {
 AReal c;
 c.p = a.p * b.p;
 c.setActive(a.isActive() || b.isActive());

 if(c.isActive()) {
  tape.functionStack.top().statements += 1;
  tape.functionStack.top().arguments += 2;
  if(!a.isActive()) {tape.functionStack.top().passives += 1;}
  if(!b.isActive()) {tape.functionStack.top().passives += 1;}
 }
}

AReal operator* (const AReal& a, const double& b) {
 AReal c;
 c.p = a.p * b;
 c.setActive(a.isActive());

 if(c.isActive()) {
  tape.functionStack.top().statements += 1;
  tape.functionStack.top().arguments += 2;
  if(!a.isActive()) {tape.functionStack.top().passives += 1;}
  tape.functionStack.top().constants += 1;
 }
}

AReal sin(const AReal& a) {
 AReal c;
 c.p = sin(c.p);
 c.setActive(a.isActive());

 if(c.isActive()) {
  tape.functionStack.top().statements += 1;
  tape.functionStack.top().arguments += 1;
  if(!a.isActive()) {tape.functionStack.top().passives += 1;}
 }
}
```

Figure 5.3.: Implementation of the overloaded functions for the *AReal* type.

on the trivial graph format (TGF) [RT15, Wik16] for the base of the data layout. (See the definition and example in Appendix B). File size is one important aspect when dealing with call graphs from fully featured simulation software. They can lead to graphs that contain millions of nodes, this makes verbose formats inappropriate. It is also problematic, when the graph data is stored in the RAM of the computer. All the information should be written to the disk as soon as possible.

The two required data items for each function are the *FuncData* objects and the edge information for the call graph. *FuncData* objects can be written after the function is finished. The edge information is available when a function is called. The usual TGF format requires that the edge information be written after all nodes are written to the file. This requires the storage of all edges until the end of the program. In order to avoid this overhead, the edge information is written directly to a separate file.

The handling of node data is more involved. The id for each node will be an integer, that is generated for each *FuncData* object. The data for each node is ready when the *funcLeave* method is called, which means that the data of all the nested functions are ready before the parent's function data. The TGF format does not enforce any order upon the nodes, but the human preference is an ordered set. In order to achieve this, all *FuncData* objects would need to be stored in a vector. The file of the node data could then be written at the end of the program. The nodes would then be ordered by their id. This would counteract the axiom that the memory consumption should be as minimal as possible.

If the node data is written directly after the *FuncLeave* method, then this overhead is eliminated. Another important property of this approach can be used when a node data file is read again. Each time a node is read the program can assume that all child nodes have also been read. This property can be used to directly compute additional information on the graph.

One additional file is added to the data layout. It contains information about the number of nodes that have been written and the data layout for each node. This file can be used by the reading program to speed up parsing of the files. The layout of the files is shown in Figure 5.4. The extension for the node name file is optional but the file size can be greatly reduced by this mechanism. It is also faster to generate the function names just once and to store them in a lookup table. The prefix in the specification should be the same for all files but it is not necessary. If the prefix is the same, the analysis program only needs the property file and can determine the other files by the name of the property file.

The data layout in Figure 5.4 can be written in binary form. This speeds up the reading and writing of files and reduces the required amount of memory. A further step for the data size reduction is a direct compression of the data. The lz4 algorithm [Col16] is used here.

```
<prefix>_nodes.tgf

  # Comment lines start with #
  <node_id> <name_id> <children> <data1> <data2> ... <dataN>
  ...

  # - node_id indices are created on function enter events
  # - nodes are written on function leave events
  # - Each child node is written before its parent
  # - node_id starts with the 0 index => node_id \in [0, \infinity)
  # - name_id is a name_id from names.tgf
  # - 'children' gives the number of child functions in the graph
  # - The number of data items is the same for each node
  # - The number of data items has to correspond with the entry
  #  'dataItems' in info.prop
  # - Each line contains the information for one node

<prefix>_edges.tgf

  # Comment lines start with #
  <parent_id> <child_id>
  ...

  # - parent_id and child_id are node_id indices from nodes.tgf
  # - Each line contains the information for one edge

<prefix>_names.tgf

  # Comment lines start with #
  <name_id> <name>
  ...

  # - name_id starts with the 0 index => name_id \in [0, \infinity)
  # - name_id are ordered in ascending order
  # - Each line contains the information for one name

<prefix>.prop

  # Default Java property format
  nodes=<number of nodes>
  dataItems=<number of data items per node>
  dataNames=<list of data names>
  # e.g. dataNames=Input, Output, Statements, Arguments

  # - The properties "nodes", "dataItems" and "dataNames" are mandatory
  # - The number of items in "dataNames" and the number in "dataItems"
      have to be the same
```

Figure 5.4.: Data layout for the *AReal* output containing the call graph of the program with additional AD specific information.

5.3.2. Variable tracking

The tracking of input and output variables is not required for the computation of the tape size. The tape size requires only the counts η_s, η_a, η_c and η_p. The correct input and output counts however, need to be known when a special handling for the node is applied. If for example the input values need to be stored for the special handling, then the number of input variables matter.

The tracking of input variables is simple and has already been implemented in the instrumentation tape of dco/C++. If linear increasing indices are assumed for the *AReal* type (see Section 2.6), then each function can store the current global index. This index is called the function boundary index. All variables with a smaller index have been created before the function call and are considered inputs of the function. This is illustrated here:

```
AReal func(const AReal& a) {
  AReal c = a * a; // Index of c is 1001, a is an input
  return c * c + a; // a is still an input, c is no input
}

...
 AReal a = ...; // Index of a is 1000
 AReal b = func(a); // Function boundary index is 1000
...
```

The variable *a* can be distinguished as an input because the index is smaller or equal to the function boundary index. Variable *c* is no input variable because the index is larger than the function boundary index. If a variable is used twice it would be counted as two input variables. The dco/C++ implementation adds the information to the *AReal* type which function used the variable last. This is the node index from the data layout definition generated for *FuncData* objects, calling it function index in order to avoid confusion with the index of the *AReal* variables. If the function index in the *AReal* data object is bigger or equal to the current function index, the variable has already been counted as an input. This approach yields correct results but has the drawback that arguments of the functions have to be modified. The *mutable* keyword for the corresponding member of the *AReal* can solve this. It prevents the compiler from putting any *AReal* in the constant memory section of the program and the member can always be modified even if the parent structure is marked as constant. The counting of the input variables can then be handled like:

```
void countInput(const AReal& a, FuncData& data) {
  if(a.index <= data.functionBoundaryIndex) { // a was created before the
      function call
    if(a.funcId < data.id) { // a was not counted already as an input
      a.funcId = data.id; // mutable member
      data.inputs += 1;
    }
  }
}
```

```
struct FuncData {
  ...
  int functionBoundaryIndex;
  int inputCount;
  int outputCount;
  ...

  void indexCreated(const int index) {
    outputCount += 1;
  }

  void indexReleased(const int index) {
    if(index >= functionBoundaryIndex) {
      outputCount -= 1;
    }
  }
};
```

Figure 5.5.: Output handling for the analysis tape. The count is updated each time an *AReal* is created or deleted.

This function needs to be called for every entry on the function stack as the input count is not additive.

The count of the output variables follows the same principle. Each statement creates a new index and variable. These variables are potential outputs. The distinction between temporary variables and output variables can be made by the deleted variables. Every deleted variable is a temporary variable. The destructor for the *AReal* needs to inform the tape that the variable is freed. The *FuncData* structure is extended by an *int* that counts the number of created variables in the function. When a variable is created, the count is increased by one and each time a variable is deleted decreased by one. When the function finishes, the count will contain the number of output variables of the function.

The functions for the output handling are described in Figure 5.5. The logic in these functions is very easy to understand and will generate the correct number of outputs. It is now mandatory to call the counting functions for all possibilities where an index can be generated, deleted or overwritten. For a structure in C++ this can be the constructor, the destructor and the assignment operators =, +=, etc..

If the methods in Figure 5.5 are called for every function on the stack the number of outputs is correct but the overhead will be quite large. Instead the count of output variables can be used to update the parent function after the child function is finished. All variables generated in the child function will be handled the same way in the parent function. This is the case, because the function boundary of the parent is always smaller than the child's. The output count of both functions receive the same changes, they can can be gathered in the child and added to the parent after the child has been finished. The implementation in Figure 5.5 is adapted to reflect this reasoning in Figure 5.6.

```
struct FuncData {
  ...
  FuncData* parent;
  ...

  void indexReleased(const int index) {
    if(index >= functionBoundaryIndex) {
      outputCount -= 1;
    } else if(null != parent) {
      parent->indexReleased(index);
    }
  }

  void add(const FuncData& childData) {
    this->outputCount += childData.outputCount;
  }
};
```

Figure 5.6.: Output handling for the analysis tape, which improves the speed by taking advantage of the child parent connection. All variables which are handled in the child do not need to be handled in the parent, they are added to the parent after the child finishes.

The *AReal* type can track now the performance parameters for each function and counts the number of input and output variables. Every data set for all functions can be written directly after each function finishes. This allows for minimal memory consumption, which is reflected in the designed data layout. It contains only the necessary information and can be written in binary form.

5.4. Checkpointing

Let N be an evaluation node and $F : \mathbb{R}^n \to \mathbb{R}^m$ the associated function. The graph containing N is differentiated with AD and the function F is assumed to need to be treated in a special way. This can have several reasons. F may represent a call to a library function and the AD type can not be used in the library. It can also be a function for which the analytic derivative is known and the direct computation of the analytic derivative is more efficient than the computation with AD (e.g. linear system solvers). It is also possible that F is identified as a hot spot and requires a large amount of memory when a tape is recorded. One possible treatment in such a case is that F is not recorded directly but reevaluated during the reverse sweep when memory from the tape has already been released.

First, it will be analyzed how much memory is required by cutting out F from the tape and the possible run time drawbacks. Then a closer look on the pure AD implementation is taken.

The reverse procedure for F after Theorem 2.13 is

$$\begin{aligned}\bar{x} \mathrel{+}=& \frac{dF}{dx}^T \bar{y} \\ \bar{y} =& 0 \ .\end{aligned}$$

The update formulation is chosen here because F can be considered an elemental operation after Section 2.4. In order to evaluate the equation by hand the primal x is needed and the means to access the adjoints of x and y which are respectively $\bar{x}$ and $\bar{y}$. The default reverse implementations use integers for this. The memory footprint for a function checkpoint consists of n double values, $n + m$ integers, the function that is required for the reverse evaluation and the overhead for the external function denoted by $byte\{extFunc\}$. The full memory is then

$$\begin{aligned}MEM\{check(F)\} =& byte\{double\} \cdot n + byte\{int\} \cdot (n + m) + byte\{pointer\} \\ &+ byte\{extFunc\} \ .\end{aligned}$$

This memory is the maximum amount of memory that AD needs for a function checkpoint.

Therefore, it is not free to store an external function on the tape and the required memory needs to be taken into account when the benefits of external functions are evaluated. The memory can increase when, for the evaluation of F, additional data is required. It is possible to leave data in place that is generated by F in order to use the data in the reverse model of F. These are details of implementation that can not be considered in a general framework.

The memory of the external function can be reduced if certain conditions apply. If the output vector y is not overwritten after F is evaluated then the indices of y do not need to be stored. This yields

$$\begin{aligned}MEM\{check_{no\_y}(F)\} =& byte\{double\} \cdot n + byte\{index\} \cdot n + byte\{pointer\} \\ &+ byte\{extFunc\} \ .\end{aligned}$$

The same considerations can be made for the input vector x, this yields

$$\begin{aligned}MEM\{check_{no\_x}(F)\} &= byte\{index\} \cdot m + byte\{pointer\} + byte\{extFunc\} \\ MEM\{check_{no\_x\_y}(F)\} &= byte\{pointer\} + byte\{extFunc\} \ .\end{aligned}$$

As the primal values are also still available they do not need to be stored either. With a good knowledge of the data flow in the program, these optimizations can be done easily. Another point to consider is the index layout in the vectors x and y like it is done in ADOL-C [WG09]. If the indices are linearly increasing then only the indices of x_1 and y_1 need to be stored and all other indices can be computed. The memory for the external function is then

$$\begin{aligned}MEM\{check_{lin\_ind}(F)\} =& byte\{double\} \cdot n + byte\{int\} + byte\{int\} \\ &+ byte\{pointer\} + byte\{extFunc\} \ .\end{aligned}$$

This is especially easy if the tape reuses the indices. In such a case, the indices are not changed when a value is assigned. On the other hand, it is improper if a vector does not have linear increasing indices. If the values are reassigned and become linearly increasing, additional statements need to be created on the tape. For each assignment at least 13 bytes are required. This is much more than the usual 4 bytes for an integer.

5.4.1. General checkpoints

A general procedure for creating an external function is shown in Figure 5.7. The implementation *F_check* assumes that there is a general implementation of *F_prim* and *F_adj* available. *F_prim* evaluates the function F and *F_adj* implements the reverse model. *F_b_ext* is called by the tape in the reverse sweep, extracts the data from the data structure and calls *F_adj*. The other lines in the implementation are straight forward. First a data structure is created and the data for x is stored. Then the primal values of x are extracted into the vector *x_p* for the call to *F_prim*. After that, y is made active and the generated indices are stored on the tape.

This algorithm covers the case when F has a more efficient adjoint implementation or when F is a library function.

Remark 5.7 (Registration order of external functions and output arguments)
If the tape uses linear indexing and uses the optimization that the statements do not store the index of the left hand side, then the registration of the external function has to come after the creation of the indices for the vector y. These tapes need to register a statement for each crated index and these statements reset the adjoint values of y to zero when evaluated. Since the order in the reverse AD mode is reversed, the activation has to be done before the registration of the external function. If not, $\bar{y}$ is set to zero directly before its use in the reverse interpretation. Additionally these statements increase the memory of the external function by at least $13 * m$ bytes. This can be included in the implementation of the external function such that this overhead is eliminated.

5.4.2. AD based checkpoints

The case when there is no specialized implementation for F is defined in Figure 5.8. The algorithm is nearly the same as the one in Figure 5.7 but with two differences. No longer the values from x are extracted into *x_p* and no special implementation of F is called. The AD version of F is called instead with the tape set to passive. This requires a tape implementation able to be set to passive. The advantage of this approach is that no adjoint implementation of F is required. The evaluation of F with the AD types however, takes longer. The reverse implementation of *F_b_self* is shown in Figure 5.9. This algorithm seems quite involved but is actually quite simple. Lines 5-8 perform a regular recording for F on the tape. The vector of x is restored and a vector y for the output is created in lines 2 and 3. After F is recorded

```
void F_prim(const double* x, const int n, double* y, const int m);
void F_adj(const double* x_p, const int* x_i, const int n,
        const int& y_i, const int m);
void F_b_ext(const Data& data);

void F_check(const RReal* x, const int n, RReal* y, const int m) {
  Data data;

  data.store(x, n);
  double* x_p = new double[n]; // primal values for x
  getPrimal(x, x_p, n);

  double* y_p = new double[m]; // primal values for y

  F_prim(x_p, n, y_p, m);

  setPrimal(y, y_p, m);
  tape.makeActive(y, m);
  data.storeIndices(y, m);

  tape.createExternalFunction(data, F_b_ext);
}
```

Figure 5.7.: Algorithm for an external function creation when a primal and adjoint implementation is available. No optimization for the memory consumption are used in this algorithm because no assumptions about F and the use of the data x and y can be done.

```
void F_b_self(const Data& data);

void F_passive(const RReal* x, const int n, RReal* y, const int m) {
  Data data;

  data.store(x, n);

  tape.setPassive();
  F(x, n, y, m);
  tape.setActive();

  tape.makeActive(y, m);
  data.storeIndices(y, m);

  tape.createExternalFunction(data, F_b_self);
}
```

Figure 5.8.: Algorithm for an external function creation when no primal and adjoint implementations are available. The algorithm disables the recording of the tape in this case.

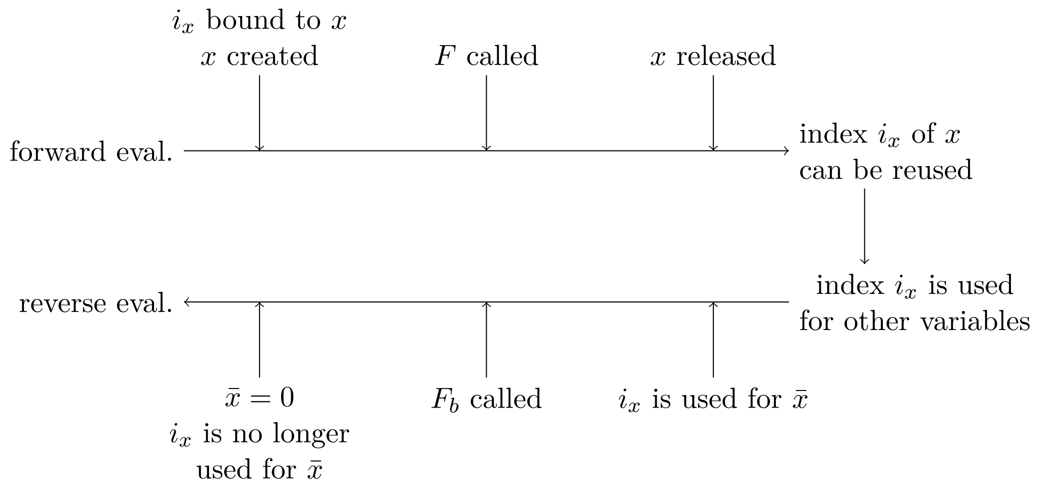
on the tape, the adjoint values of $\bar{y}$ are set in the for loop on lines 12 to 15. During the primal call of F, the output vector has been made active and the adjoint values from this vector are set to the adjoints of the current y. The evaluation call on the tape in line 17 interprets the recorded statements for F. The position argument needs to be provided so the tape knows where the recording of F started. At the end of the function the memory for the vectors and the tape is released. No special handling for x is necessary. The restored vector x has the same indices as the vector x which has been used in the primal evaluation. The tape uses these "old" indices directly in the new recording and updates the corresponding adjoint values directly. This optimization is quite advanced, but works for all index management strategies. It is also possible to register x as an input and generate new indices. If so, an update to the adjoints of x is required.

5.4.3. Checkpoints and managed index tapes

With linearly increasing indices the checkpoints will work without any issues. The situation is more complicated if indices can be reused. Two cases have to be considered, namely, the old indices from x and the new indices used during the recording of F. The indices of x might have been released after the function F was called. During the reverse evaluation of the tape the use of the indices is also reversed. Without loss of generality Figure 5.10 explains how the life cycle of x affects the index i_x that is associated with the value. At the point were x is released, the associated index i_x is also released and can be reused. At the same point during the reverse sweep the same index i_x is used for the representation of $\bar{x}$. At the point were x is created the value of $\bar{x}$ is set to zero and the index i_x is no longer associated with $\bar{x}$. Thus, stored

```
void F_b_self(const Data& data) {
  RReal* x = data.restore<RReal>(n);
  RReal* y = new RReal[m];

  tape.setActive();
  Position startPos = tape.getPosition();
  F(x,n,y,m);
  tape.setPassive();

  int* adjIndices = data.restore<int>(m);

  for(int i = 0; i < m ; ++i) {
    y[i].setAdjoint(tape.getAdjoint(adjIndices[i]));
    tape.setAdjoint(adjIndices[i], 0.0);
  }

  tape.evaluate(startPos);
  tape.reset(startPos);

  delete[] adjIndices;
  delete[] y;
  delete[] x;
}
```

Figure 5.9.: Algorithm for the reverse evaluation of an external function when no primal and adjoint implementations are available. The implementation assumes that linearly increasing indices are used. If indices are reused then the index manager needs to be reset, too. See Figure 5.11 for details.



i_x bound to x
x created
F called
x released
forward eval.
index i_x of x can be reused
index i_x is used for other variables
reverse eval.
$\bar{x} = 0$
i_x is no longer used for $\bar{x}$
F_b called
i_x is used for $\bar{x}$

Figure 5.10.: Life cycle of an input variable for a function checkpoint and how the associated index is used.

indices for x can be used without any modifications.

The indices that are used during the recording of F in the reverse sweep are depicted in figure 5.11, which gives an exemplary life cycle for an index $o \in \mathbb{N}$. The important point is that o is free at the end of the recording and is in use while the checkpoint for F is written. Then o can be assigned to an other value during the recording of F. If this is the case, then o is associated to $\bar{b}$ and $\bar{c}$ during the reverse evaluation of F. The correct result of the adjoint can no longer be guaranteed. There are several solutions in order to avoid these cases.

The first solution involves storing the state of the index manager in the checkpoint. This can increase the memory of the checkpoint by several megabytes. The second solution is to create a temporary adjoint vector for the reverse evaluation of F. The index o then has a different meaning in both adjoint vectors and resolves the clash between the two variables. The second adjoint vector can also require additional memory resources but it is only required during the reverse evaluation where some memory might already be freed. It also requires the user to update the value $\bar{x}$ from the regular adjoint vector with the value from the temporary adjoint vector. The third solution can store the maximum index i_{max} that is used when the checkpoint for F is created. During the reverse run the index manger can be configured to only use indices greater than i_{max}. The problem with this approach is the size of the adjoint vector. It could be increased by a large amount.

The last two solutions are easy to implement and have only a minimal impact on the memory for the external function.

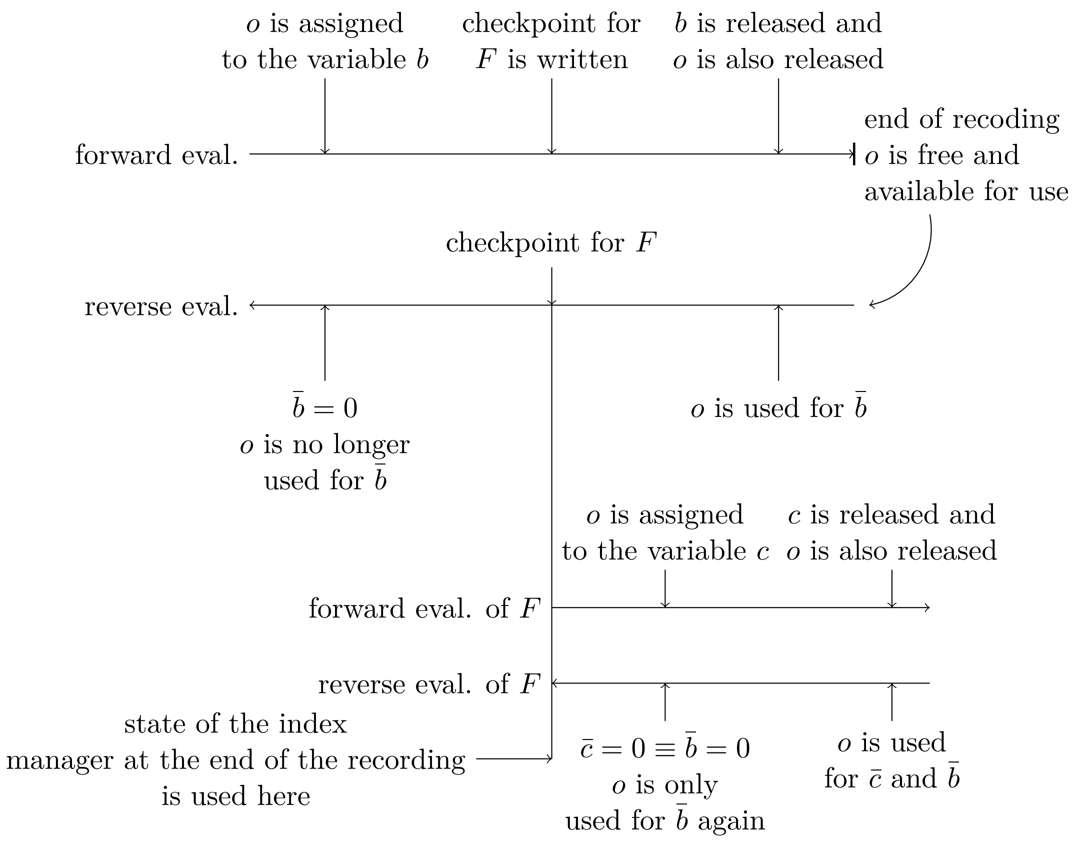

o is assigned to the variable b
checkpoint for F is written
b is released and o is also released
forward eval.
end of recoding o is free and available for use
checkpoint for F
reverse eval.
$\bar{b} = 0$ o is no longer used for $\bar{b}$
o is used for $\bar{b}$
o is assigned to the variable c
c is released and o is also released
forward eval. of F
reverse eval. of F
state of the index manager at the end of the recording is used here
$\bar{c} = 0 \equiv \bar{b} = 0$ o is only used for $\bar{b}$ again
o is used for $\bar{c}$ and $\bar{b}$

Figure 5.11.: Life cycle of an index o in a function checkpoint. The index o is available at the end of the recording and can be used when the tape is recorded for F in the function checkpoint. This can lead to a state where o is used for two adjoint variables.

5.4.4. Global memory and time analysis of checkpoints

The memory layout for the process of external functions is pictured in Figure 5.12. The recorded tape for F is cut out and added to the end. The question is now why should this save memory? If multiple calls of F or other functions are checkpointed, then the accumulated memory is not used but the maximum from all these functions. This is the case, because the memory for each external function is deleted after the function is evaluated in the reverse sweep. If the tape is evaluated only once, then an additional optimization is possible. When the checkpoint for F is reached, the tape can be reset to the point of the checkpoint. The freed memory can then be used for the tape that is recorded for F. The memory layout for this is shown in Figure 5.13. If functions $F_1, \dots, F_j$ are in the tape at the positions $\mathfrak{p}_1, \dots, \mathfrak{p}_j$ and F denotes the complete process, then $\mathit{tape}(F)|_{\mathfrak{p}_i}$ denotes the *tape* action only up to position $\mathfrak{p}_i$. The memory requirement without and with the global tape reset is

$$\begin{aligned} MEM\{\mathit{tape}_{\mathit{checked}}(F)\} = & MEM\{\mathit{tape}(F)\} \\ & - \sum_{i=1}^{j} MEM\{\mathit{tape}(F_i)\} + \sum_{i=1}^{j} MEM\{\mathit{check}(F_i)\} \\ & + \max(MEM\{\mathit{tape}(F_1)\}, \dots, MEM\{\mathit{tape}(F_j)\}) \end{aligned}$$

and

$$\begin{aligned} & MEM\{\mathit{tape}_{\mathit{checked\_reset}}(F)\} = \\ & \quad \max(\\ & \qquad MEM\{\mathit{tape}(F)\} - \sum_{i=1}^{j} MEM\{\mathit{tape}(F_i)\} + \sum_{i=1}^{j} MEM\{\mathit{check}(F_i)\}, \\ & \qquad MEM\{\mathit{tape}(F)|_{\mathfrak{p}_j}\} - \sum_{i=1}^{j-1} MEM\{\mathit{tape}(F_i)\} + \sum_{i=1}^{j} MEM\{\mathit{check}(F_i)\} \\ & \qquad\qquad + MEM\{\mathit{tape}(F_j)\}, \\ & \qquad \dots, \\ & \qquad MEM\{\mathit{tape}(F)|_{\mathfrak{p}_1}\} + MEM\{\mathit{check}(F_1)\} + MEM\{\mathit{tape}(F_1)\} \\ & \quad) \; . \end{aligned}$$

The second formulation gives a good idea for the amount of memory that can be saved. If just one function is checkpointed at the end of the tape, the reduction in memory can be quite minimal or even increase due to the overhead for the checkpoint. If the function is at the start of the tape, then the reduction in memory will be fully seen. The evaluation time for this technique is another critical factor. The time for the checkpoint is

$$TIME\{\mathit{check}(F)\} = TIME\{\mathit{passive}(F)\} + TIME\{\mathit{tape}(F)\} + 2 * \frac{MEM\{\mathit{check}(F)\}}{mem_{\mathit{speed}}} \; .$$

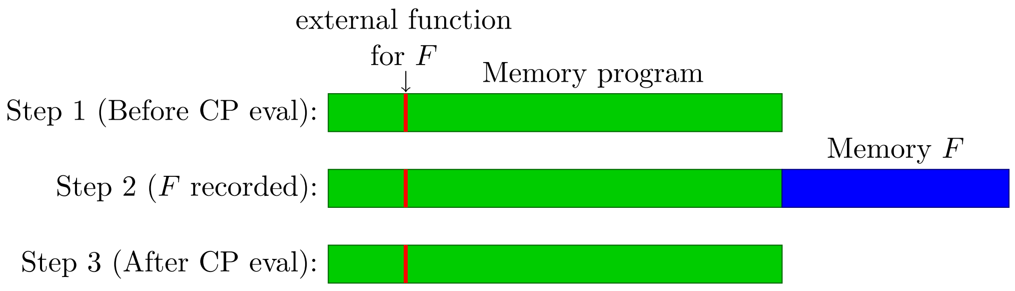


Figure 5.12.: Memory layout for function checkpoints with no global tape reset. Step 1 shows the memory before the external function is called. Step 2 shows the memory after F is recorded in the external function. Step 3 Shows the memory after the external function is evaluated.

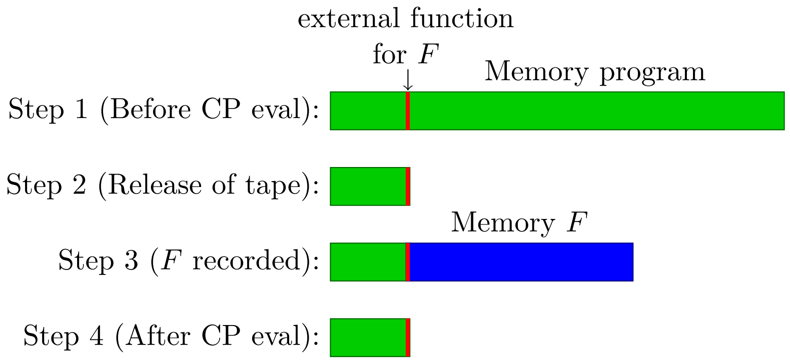


Figure 5.13.: Memory layout for function checkpoints with a global tape reset. Step 1 shows the memory before the external function is called. In step 2 the memory is reset to the function checkpoint. Step 3 shows the memory after F is recorded in the external function. Step 4 Shows the memory after the external function is evaluated.

If F is not checkpointed then $TIME\{tape(F)\}$ would be the only term that is seen. Checkpointing introduces the time for the store and restore of the data and the time for the passive evaluation of F. Because there is nothing written to the tape, $TIME\{passive(F)\} < TIME\{tape(F)\}$ holds. But if F represents a large portion of the program then the run time is increased by two.

5.4.5. Loop checkpointing

Loop checkpointing is a special kind of checkpointing. It assumes that the body of the loop is a function F, that performs an update on some variables. The algorithm looks like:

```
while(!converged) {
  yNext = F(yCur);
  yCur = yNext;
}
```

If the loop is taped, then F is recorded several times and can lead to a huge use of memory. Instead the state $yCur$ can be stored in every iteration and used to reevaluate the function F when required. It is then no longer necessary to store the tape for the loop, instead the memory for the checkpoints is required. The algorithm could then look like:

```
for(int i = 0; i < N; ++i) {
  tape.storeState(yCur);
  yNext = F(yCur);
  yCur = yNext;
}

for(int i = N - 1; i >= 0; --i) {
  tape.restoreState(yCur);

  // record tape for F and calculate the derivative
}
```

If the memory consumption of a checkpoint is large or N is big, then the memory of the system (RAM and disk space) might not be sufficient. In this case memory can be traded with time when a checkpoint is not used in every iteration. A checkpoint can be created e.g. every tenth iteration and the nine steps in between where the checkpoints are missing, can be recomputed on the fly. The optimal positions for a given maximum number of checkpoints can be computed with the revolve algorithm that Griewank and Walther present in [GW00]. It minimizes the required number of recomputations for a fixed number of checkpoints.

5.5. Preaccumulation

Like checkpointing, the preaccumulation technique tries to reduce the memory of an evaluation node. While the checkpointing technique cuts the evaluation node out of the tape, the preaccumulation technique tries to reduce the memory representation of the node. This is done by reducing the number of statements which are written to the tape. Let (x, y, F) be the triple from the evaluation node with $F : \mathbb{R}^n \to \mathbb{R}^m$. Then the reverse AD mode needs to compute

$$\bar{x} \mathrel{+}= \frac{dF}{dx}^T \bar{y} \ .$$

The memory for the taping of F is defined in equation (5.1) and looks like

$$\begin{aligned} MEM\{tape(F)\} =& byte\{statement\} * \eta_s + byte\{argument\} * \eta_a \\ &+ byte\{constant\} * \eta_c + byte\{passive\} * \eta_p \ . \end{aligned}$$

```
...

Position startPosition = tape.getPosition();
F(x, n, y, m);

double* jacobi = new double[m*n];
for(int j = 0; j < m; ++j) {
  y[j].setAdjoint(1.0);
  tape.evaluate(startPosition);

  for(int i = 0; i < n; ++i) {
    jacobi[i * m + j] = x[i].getAdjoint();
    x[i].setAdjoint(0.0);
  }
}

tape.reset(startPosition);
tape.makeActive(y,m);

Data data;
data.storeIndices(x, n);
data.storeIndices(y, n);
data.store(jacobi, m*n);

tape.createExternalFunction(data, preAccRev);

delete[] jacobi;

...
```

Figure 5.14.: Accumulation of the Jacobi matrix for F.

If $\frac{dF}{dx}$ is computed and stored $m * n * byte\{double\}$ bytes of memory will be used. In addition, the indices of x and y need to be stored requiring the use of $byte\{index\} * (n + m)$ bytes of memory. (Remark: The introduced optimizations for index storing from the function checkpointing section apply here, too.) The final amount of memory for the Jacobi matrix will be

$$MEM\{jac(F)\} = m * n * byte\{double\} + byte\{index\} * (n + m) + byte\{extFunc\}\ .$$

There will be a gain in memory if $MEM\{jac(F)\} < MEM\{tape(F)\}$. This equation holds if $m * n \ll \eta_s + \eta_a + \eta_p + \eta_c$ which means that F has few input and output variables but is difficult to calculate and needs a lot of statements.

Before further memory optimizations are developed a general algorithm for the accumulation of F is presented in Figure 5.14. The algorithm looks like the reverse evaluation for function checkpoints in Figure 5.9. It has the distinction that the algorithm does not know the seeding for $\bar{y}$. The accumulation happens in the forward evaluation and not in the reverse interpretation. $\bar{y}$ is seeded with the unit vectors and the results are stored in the Jacobi matrix. The values of $\bar{x}$ are reset to zero

```
void preAccRev(const Data& data) {

  int* x_i = data.restore<int>(n);
  int* y_i = data.restore<int>(m);
  double* jacobi = data.restore<double>(m*n);
  double* y_bar = new double[m];

  for(int j = 0; j < m; ++j) {
    y_bar[j] = tape.getAdjoint(y_i[j]);
    tape.setAdjoint(y_i[j], 0.0);
  }

  for(int i = 0; i < n; ++i) {
    double& x_bar = tape.getAdjointRef(x_i[i]);

    for(int j = 0; j < m; ++j) {
      x_bar += jacobi[i*m+j] * y_bar[j];
    }
  }

  delete[] y_bar;
  delete[] jacobi;
  delete[] y_i;
  delete[] x_i;
}
```

Figure 5.15.: Reverse evaluation of a preaccumulated Jacobi matrix.

since the update $+= \dots$ is performed on these values. If this is not done, the result would be the sum of all columns of the Jacobi matrix. $\bar{y}$ is not set to zero because this is performed by the tape in the reverse evaluation. After the full Jacobi matrix is assembled, the recorded data for F is freed and the checkpoint for the preaccumulation is stored. The activation of y after the reset of the tape is at first not very intuitive. y should already be active as it is the output of F, but the tape is reset to the state before F has been evaluated and therefore the output to y has never occurred for the tape. This invalidates the indices and they have to be renewed.

The function *preAccRev* in Figure 5.15 performs the reverse interpretation of the preaccumulated section. It consists of data unpacking and matrix vector multiplications. Please note that $\bar{y}$ is set to zero and $\bar{x}$ is updated as required by the AD theory. The weird access pattern of the Jacobi matrix in Figure 5.14 now makes sense because the data layout is optimized for the reverse evaluation.

Creation time for the preaccumulation of the evaluation node is quite high. The section of the tape for F is evaluated m times. Otherwise this section would be evaluated just once. The time for the recording increases by $m * TIME\{reverse(F)\}$

but the time for the reverse evaluation has probably decreased. The equations are

$$TIME\{jac(F)\} = TIME\{record(F)\} + m * TIME\{reverse(F)\}$$
$$TIME\{reverse\_jac(F)\} \geq \frac{MEM\{jac(F)\}}{mem_{speed}} .$$

In the *preAccRev* method there are $m * n$ multiply-accumulate operations where the reverse evaluation of the tape performs η_a multiply-accumulate operations. If the assumption holds where it is memory efficient to perform a preaccumulation, then it is faster to evaluate the checkpoint in the reverse sweep of the tape.

Further optimizations can be done for the Jacobi matrix. If $\frac{dF}{dx}$ is sparse, then this sparsity pattern can be exploited. When this pattern is fixed and known beforehand, the sparsity pattern can be stored once globally and reused several times. But if the sparsity pattern is not known or can change because of branching statements, then the sparsity pattern needs to be stored for every Jacobi separately. This can be a coordinate list (COO) [BG08] which would require two indices for each entry or if the matrix is stored in compressed sparse row (CSR) or in compressed sparse column (CSC) [DGL89] order one index is required for each entry and one index is required for the row or column. If one index is used this results in a value index pair. Such a pair is used by the Jacobi tape to store the arguments of the statements. The column index is known for each statement. This means that a sparse Jacobi matrix in a compressed sparse row format can be stored directly on the Jacobi tape. The procedure for this is shown in Figure 5.16. The non-zero elements of one row are stored with the corresponding indices from x and the number of arguments is then written. This eliminates the memory and the time overhead for the external function call and is a very elegant way to include a sparse matrix in the reverse evaluation.

Remark 5.8 (Number of arguments for Jacobi tapes)
For Jacobi tapes the assumption can be made that there are no more than 256 or 128 arguments. This is to save memory. This assumption can be easily broken by the preaccumulation technique.

Remark 5.9 (Preaccumulation on primal value tapes)
Primal value tapes store only one index per argument and need a handle function to evaluate each statement. This prevents the preaccumulation technique from being used here. The constant value storage of these tapes can be used to store the Jacobi entries and special handle functions can be programmed for the evaluation.

The memory reduction is not as good as for the Jacobi tape but is still significant.

5.6. Block checkpoints

Checkpointing can be applied to all functions and is described as a general technique. For special structures the checkpointing can be used to optimize the memory con-

```
void pushJacobiMatrix(const int* x_i, const int n, const int* y_i, const
    int m, const double* jacobi) {
  for(int j = 0; j < m; ++j) {
    int nonzeros = 0;

    for(int i = 0; i < n; ++i) {
      if(jacobi[i*m+j] != 0.0 && x_i[i] != 0) {
        tape.push(jacobi[i*m+j]);
        tape.push(x_i[i]);
        nonzeros += 1;
      }
    }

    if(0 != nonzeros) {
      tape.push(y_i[j]); // only needed for index reuse
      tape.push(nonzeros);
    }
  }
}
```

Figure 5.16.: Push a Jacobi matrix on the tape that is preaccumulated.

sumption of the tape without increasing it's run time. The structure of TRACE from Figure 4.3 on page 58 shows that during the block evaluation no data is exchanged between blocks. This can be used to optimize the memory required by the taping and reverse evaluation of G. The regular AD algorithm for lines 6 to 12 in Figure 4.3 is shown in Figure 5.17. The $\_b$ methods represent the reverse drivers of the routines and indicate which methods have to be taped. In order to perform this procedure, the inputs y, x and w are registered on the tape and G is evaluated. Afterwards y_{new} is registered as an output and the tape is evaluated. The resulting tape size is

$$\begin{aligned} MEM\{tape(G_{Full})\} =& MEM\{tape(updateSol)\} + MEM\{tape(\text{doCommunication})\} \\ &+ \sum_{i=1}^{nBlocks} MEM\{tape(\text{updateBlock}_i)\} \ . \end{aligned}$$

The memory of the communication and the solution update can be neglected in contrast to the memory of the blocks. The procedure in Figure 5.17 can be optimized to the one in Figure 5.18.

Remark 5.10 (Availability of a_i and b_i)
One could argue that the only change is that the block update is moved into the adjoint loop and that this is not possible because the local values a_i and the communication values b_i are not available in the communication and solution update. These issues are valid and will be addressed later.

After the shift, the primal and reverse methods are directly next to each other. The tape does not need to store the full procedure anymore. It can evaluate each primal and adjoint pair separately. The process for the evaluation of the algorithm in Figure 5.18 is:

1. Register $b_1, \dots, b_{nBlocks}, a_1, \dots, a_{nBlocks}, y, x, w$ as inputs.
2. Tape *doCommunication* and *updateSol*.
3. Register y_{new} as an output and set the seeding on $\bar{y}_{new}$.
4. Evaluate the tape and store the adjoints of $\bar{b}_1, \dots, \bar{b}_{nBlocks}, \bar{a}_1, \dots, \bar{a}_{nBlocks}, \bar{y}, \bar{x}, \bar{w}$.
5. For each block $i = 1 \dots nBlocks$:
   a) Register y_i, x, w as inputs.
   b) Tape updateBlock$_i$.
   c) Register a_i and b_i as outputs and set the seeding on $\bar{a}_i$ and $\bar{b}_i$.
   d) Evaluate the tape.
   e) Store and update the adjoints of $\bar{y}$, $\bar{x}$ and $\bar{w}$

The complexity and management has increased a lot but the benefit is obvious when the required memory is analyzed. The maximum memory is now

$$\begin{aligned}&MEM\{tape(G_{BlockCP})\} = \\ &\qquad \max(MEM\{tape(updateSol)\} + MEM\{tape(\text{doCommunication})\}, \\ &\qquad\qquad MEM\{tape(\text{updateBlock}_1)\}, \dots, MEM\{tape(\text{updateBlock}_{nBlocks})\}) \ .\end{aligned}$$

The biggest block defines the memory requirement for G instead of the sum of all blocks. This is an advantage if many blocks are being handled by the process. It requires more knowledge of the data structure and new interface definitions are required. The full algorithm in Figure 5.17 requires the interfaces of y, x and w. In the block algorithm in Figure 5.18 the additional interfaces of $a_1, \dots, a_{nBlocks}$ and $b_1, \dots, b_{nBlocks}$ have to be defined.

Due to the structural change in the algorithm, the values of $a_1, \dots, a_{nBlocks}$ and $b_1, \dots, b_{nBlocks}$ are not available. The values can be made available by computing the block updates before the method Blocked_G is called. Since the block updates are the most time consuming part, this will double the evaluation time.

A second option is to integrate the missing values in the checkpointing scheme. If the values of $a_1, \dots, a_{nBlocks}$ and $b_1, \dots, b_{nBlocks}$ are stored in intermediate checkpoints, then these values can be restored and no extra computation is required. If the intermediate checkpoint is used, then the computation time for Full_G and Blocked_G are equivalent. Due to the advanced memory management the memory for the tape is reduced by a significant amount. The technique is referred to as block checkpoints in the TRACE code. It can be used for the Black Box and Reverse Accumulation method but it is more appropriate for the Black Box algorithm, because the tape needs to be recorded for every G. The Reverse Accumulation process records the tape once and evaluates it multiple times.

```
**for** $i = 1 \dots nBlocks$ **do**
    $(a_i, b_i) = \text{updateBlock}_i(y_i, x, w)$
**end for**
$o = \text{doCommunication}(b_1, \dots, b_{nBlocks}, y, x, w)$
$y_{new} = \text{updateSol}(a_1, \dots, a_{nBlocks}, o)$

$\bar{y}_{new} = \dots$
$(\bar{a}_1, \dots, \bar{a}_{nBlocks}, \bar{o}) \mathrel{+}= \text{updateSol\_b}(a_1, \dots, a_{nBlocks}, o) * \bar{y}_{new}$
$(\bar{b}_1, \dots, \bar{b}_{nBlocks}, \bar{y}, \bar{x}, \bar{w}) \mathrel{+}= \text{doCommunication\_b}(b_1, \dots, b_{nBlocks}, y, x, w) * \bar{o}$
**for** $i = nBlocks \dots 1$ **do**
    $(\bar{y}_i, \bar{x}, \bar{w}) \mathrel{+}= \text{updateBlock\_b}_i(y_i, x, w) * (\bar{a}_i, \bar{b}_i)$
**end for**
```

Figure 5.17.: AD differentiation of the conceptual structure of G in TRACE from Figure 4.3. The _b methods represent the reverse drivers of the corresponding methods. They represent the methods that need to be taped.

```
$o = \text{doCommunication}(b_1, \dots, b_{nBlocks}, y, x, w)$
$y_{new} = \text{updateSol}(a_1, \dots, a_{nBlocks}, o)$
$\bar{y}_{new} = \dots$
$(\bar{a}_1, \dots, \bar{a}_{nBlocks}, \bar{o}) \mathrel{+}= \text{updateSol\_b}(a_1, \dots, a_{nBlocks}, o) * \bar{y}_{new}$
$(\bar{b}_1, \dots, \bar{b}_{nBlocks}, \bar{y}, \bar{x}, \bar{w}) \mathrel{+}= \text{doCommunication\_b}(b_1, \dots, b_{nBlocks}, y, x, w) * \bar{o}$

**for** $i = 1 \dots nBlocks$ **do**
    $(a_i, b_i) = \text{updateBlock}_i(y_i, x, w)$
    $(\bar{y}_i, \bar{x}, \bar{w}) \mathrel{+}= \text{updateBlock\_b}_i(y_i, x, w) * (\bar{a}_i, \bar{b}_i)$
**end for**
```

Figure 5.18.: Optimized AD differentiation of the conceptual structure of G in TRACE from Figure 4.3.

5.7. Optimized memory and time requirement discussion

This chapter described techniques how a differentiated code can be analyzed for hot spots and how these hot spots can be handled. An analysis of the TRACE code yields 4 functions that benefit from function checkpoints:

- *calcEulerFluxSides*: Computation of the Euler fluxes for the finite volume scheme.
- *impmatrix*: Setup of the matrix for the implicit solver.
- *ilu_decomp*: Incomplete LU factorization of the matrix from the implicit solver.
- *subforward_backward*: Forward and backward substitution of the incomplete LU factorization.

For preaccumulation a total of 16 methods have been identified that benefit from the technique. The most important are *calcEulerFluxSides*, *impmatrix* and *phi*. The preaccumulation for *calcEulerFluxSides* and *impmatrix* is done for each step in the loops over the elements.

The results from Table 4.4 in Chapter 4 on page 92 are extended by the preliminary results for function checkpoints. The column “primal TRACE” shows the data for the unmodified TRACE version. The other two columns show the data for the Black Box algorithm and the Reverse Accumulation algorithm. The Black Box algorithm creates a new tape for every G iteration, therefore the displayed time is the sum of the recording and evaluation time of the tape. For the Reverse Accumulation algorithm the tape is recorded once and evaluated several times, therefore only the evaluation time is displayed.

Table 5.1 shows the preliminary results for function checkpoints (FC in the table). If all 4 function checkpoints are enabled then the memory factor can be reduced from 74 to 35. While far from optimal, the technique demonstrates how the memory can be reduced. The time increases drastically for function checkpoints. Since, the code parts have to be evaluated with a passive tape and recorded in the reverse run. The time for the Black Box algorithm is therefore nearly doubled from a time factor of 29 to 52. The Reverse Accumulation algorithm is influenced even more by the function checkpoints. It contains now the recording of the function checkpoints and the time factor increases from 7 to 38. This is unacceptable.

Function checkpoints with a passive tape are a good technique to reduce the required memory but the impact on the time is too vast in particular for Reverse Accumulation.

Table 5.2 shows the preliminary results for preaccumulation (PA in the table). The values have been taken on a different machine with a different version of the code, therefore the baseline has changed. The memory for both methods is decreased by a large factor if all functions are enabled for preaccumulation. It reduces from 60 to 35, a gain similar to that attained through function checkpoints.

Table 5.1.: Time and memory measurements for the THD turbine. First unoptimized results from the black box differentiation of TRACE, with dco/C++ for function checkpoints.

	primal TRACE	Black Box	Reverse Accumulation
Time in seconds	1.0		
no FC		28.71	6.39
calcEulerFluxSides FC		31.19	12.20
impmatrix FC		35.13	17.92
ilu_decomp FC		35.27	14.68
subforward_backward FC			
all FC		51.48	37.34
Memory in GB (Factor)	2.51		
no FC		184.86 (74)	184.86 (74)
calcEulerFluxSides FC		136.22 (55)	150.89 (61)
impmatrix FC		134.37 (54)	163.28 (65)
ilu_decomp FC		167.88 (67)	180.20 (72)
subforward_backward FC		182.95 (73)	184.70 (74)
all FC		86.57 (35)	87.11 (35)

The time for the Black Box method increases by a large factor. This is expected because the tape is evaluated for the preaccumulated parts several times. The time for the Reverse Accumulation method decreases and goes from a factor of 9.4 to 5.5. Preaccumulation is a good technique to reduce the memory and time consumption of the Reverse Accumulation method.

The preaccumulation method looks promising because it can reduce the required memory of the Reverse Accumulation method without increasing the required time, but the memory factor is still too high for a productive use of the algorithm. The AD tools need to be further analyzed as to whether the required memory is necessary and if the time for the recording and evaluation can be improved. Further analysis and times for the final implementation are shown in Chapter 7.

Table 5.2.: Time and memory measurements for the THD turbine. First unoptimized results from the black box differentiation of TRACE, with dco/C++ for preaccumulation.

	primal TRACE	Black Box	Reverse Accumulation
Time in seconds (Factor)	0.8 (1.0)		
no PA		21.98 (27.5)	7.54 (9.4)
calcEulerFluxSides PA		26.52 (33.2)	5.88 (7.4)
impmatrix PA		59.96 (75.0)	6.48 (8.1)
phi PA		29.02 (36.3)	7.25 (9.1)
all 3 PA		70.18 (87.7)	4.72 (5.9)
all 16 PA		73.51 (91.8)	4.41 (5.5)
Memory in GB (Factor)	3.12 (1)		
no PA		186.83 (59.9)	186.83 (59.9)
calcEulerFluxSides PA		142.03 (45.5)	142.03 (45.5)
impmatrix PA		159.31 (51.1)	159.31 (51.1)
phi PA		178.31 (57.2)	178.31 (57.2)
all 3 PA		113.02 (36.2)	113.02 (36.2)
all 16 PA		110.33 (35.4)	110.33 (35.4)

6. Implementation and analysis of Algorithmic Differentiation software

This chapter will focus on the analysis of different options for memory and time optimization at the lowest level, namely the AD tools. For further optimizations the global structure of the application would have to be altered, inconsistencies in derivatives would have to be allowed, or AD tools would have to be enhanced. The first two options however, do not fit with the framework of this thesis. Therefore, in order to optimize further, the AD tools used will be analyzed.

This chapter will improve on the naive implementation from Chapter 2. The resulting implementation will be the primal value taping approach, known from ADOL-C. A different approach is presented afterwards called Jacobi taping. Instead of recording the primal values this approach records the Jacobi values. The indices associated with the reverse type of both approaches will be examined and reuse strategies for these indices are discussed. A further enhancement of the primal value taping and Jacobi taping approach is provided by shifting the implementation view from an operator level to a statement level. The efficient handling of constant and passive values will also be analyzed, as well as the optimal handling of linear subexpressions. The discussion of possible improvements will finish with the handling of reoccurring terms in one statement.

These different optimizations and choices yield different ways how an AD tool can be implemented. The possible options and their results will be compared to those of currently available AD tools for C/C++. The new AD tool CoDiPack will be introduced to allow for the implementation of all approaches discussed in this chapter. The focus lies on a framework that can be used in an optimal way for large scale industrial codes and HPC applications.

6.1. Analysis of basic software implementations

The naive implementation of the reverse AD mode is shown in Section 2.6.2. The type *RReal* is introduced there with the structure:

```
struct RReal {
 double p; // primal
 int   i; // index
}
```

The index i is used to access the corresponding bar value of each variable in the program. The implementation overloads operators like *operator ** and mathematical functions like *sin*. The naive implementation from Section 2.6.2 is

```
RReal operator * (const RReal& a, const RReal& b) {
 tape.push(a.p);
 tape.push(b.p);
 tape.push(a.i);
 tape.push(b.i);

 RReal c;
 c.p = a.p * b.p;
 c.i = ++tape.globalStatementCounter;

 tape.push(c.i);
 tape.push(MUL);

 return c;
}
```

for the multiplication and

```
RReal sin(const RReal& a) {
 tape.push(a.p);
 tape.push(a.i);

 RReal c;
 c.p = sin(a.p);
 c.i = ++tape.globalStatementCounter;

 tape.push(c.i);
 tape.push(SIN);

 return c;
}
```

for the sinus. The identifiers *MUL* and *SIN* can either be function pointers or keys for a lookup table. The reverse interpretation is then performed as:

```
void operator_mul_b(double* adjoint) {
 int c_i = tape.pop<int>();
 int b_i = tape.pop<int>();
 int a_i = tape.pop<int>();
 double b_p = tape.pop<double>();
 double a_p = tape.pop<double>();

 adjoint[a_i] += b_p * adjoint[c_i];
 adjoint[i_b] += a_p * adjoint[c_i];
 adjoint[c_i] = 0.0;
}
```

for the multiplication. The identifier is not popped in this function as it is used to call the implementation and has already been popped.

The naive primal value taping needs to store two indices and two values for the arguments of each binary operation, a pointer or lookup-table index, and the generated index for the left hand side (lhs). Indices are assumed to be 32-bit integers and floating point values are assumed to be 64-bit long. The index for the lookup

```
template<typename Number>
Number l2norm_I1(const Number* v1, const Number* v2, const size_t n) {
  Number res = 0.0;
  for(size_t i = 0; i < n; ++i) {
    Number p1 = v1[i] * v1[i];
    Number p2 = v2[i] * v2[i];
    Number sum = p1 + p2;
    res = res + sum;
  }
  res = sqrt(res);
  res = res / (Number)n;

  return res;
}
```

Figure 6.1.: Implementation of the L2-norm for AD tool comparison.

table is considered an 8-bit value based upon the assumption that there are no more than 256 distinct operators. A unary operation requires one index and one value less than a binary operation. The storage required for a binary operator is then $1 + 4 + 2 * (4 + 8) = 29$ bytes and for a unary operator $1 + 4 + (4 + 8) = 17$ bytes.

To compare the results of this chapter an example function providing a very simplistic algorithm that is easily differentiated will be introduced where all memory and operation counts can be evaluated. The function calculates the L2-norm of a two component vector. The implementation in Figure 6.1 resembles the theory in Section 2.2 and has 4 binary operations inside of the loop as well as one binary and one unary operation after the loop. For each call $4 * 29 * n + 29 + 17 = 116 * n + 46$ bytes of information are recorded on the tape for the naive primal value taping. The implementation creates a new index for each assignment. These are $4 * n + 2$ indices. Each new index increases the size of the adjoint vector by one and the method requires additional memory of $4 * 8 * n + 2 * 8$ bytes for the adjoint vector.

Without any comparison the memory and index creations are difficult to understand. All values will be compared with those of the improved implementations.

6.2. Linear index management

An improvement can be done to the naive implementation. It can be observed that the generated index for the left hand side is always increased by one before being stored directly on the tape. Lines 7 and 9 in Figure 2.13 on page 36 show this connection. In the reverse evaluation the last global index is used as the starting point and decreased by one after each statement evaluation. This yields the same value as the stored index for the left hand side and thereby the left hand side index does not need to be stored any longer. This reduces the memory of a binary operator to $1 + 2 * (4 + 8) = 25$ bytes and of an unary operation to $1 + (4 + 8) = 13$ bytes. This optimization will be used in the next improvements and is called *linear index management* or *linear indexing*. The technique has one drawback, in that the statements and adjoint variables are now tightly coupled. Each statement corresponds to an adjoint variable and therefore a statement needs to be recorded if a variable is registered as an input variable. This increases the memory slightly but for applications that write millions of statements, the gain is significant.

6.3. Primal value recording

The naive version of primal value taping stores every input of each operation. If a value is used in multiple statements, it is stored several times. If the value were not to be stored upon it's first usage, but rather at the point when it is originally assigned, then the index of the left hand side value becomes the unique identifier for each value. Therefore, the values can be stored in a vector under this unique identifier, yielding a global vector where each primal value is recorded. This vector can be used to access the primal values in the reverse evaluation of the operators.

The updated primal value scheme is called “primal value taping”. The implementations for the overloaded multiplication operator and the reverse evaluation are shown in Figure 6.2. Fewer push operations lead to less memory. Nevertheless, the memory access changes from a continuous access pattern to a random access pattern for the primal values in the reverse implementation. This can have a performance impact if the reverse evaluation is not memory bound.

A binary operation now requires 17 bytes of memory. 16 bytes less than it did before, because the primal values for the input arguments are no longer stored, and 8 bytes more are necessary for the output value. The number of bytes for the unary operation does not change and still requires 13 bytes of memory. This saves 4 bytes of memory over the naive implementation as the linear index management is now used. The memory now totals $4 * 17 * n + 17 + 13 = 68 * n + 30$ bytes for the l2norm_I1 example reducing the memory by 50 % with respect to the basic implementation.

Remark 6.1 (Computation optimizations)
Since the result of the operation is now stored, the implementation can use these values to improve the computations in the reverse evaluation methods. E.g. the squared root function needs for the computation of the Jacobi one over the square root. Because the primal result is stored under the index *c_i* it can be used in the computation, therefore the square root does not need to be computed again.

```
RReal operator * (const RReal& a, const RReal& b) {
  tape.push(a.i);
  tape.push(b.i);

  RReal c;
  c.p = a.p * b.p;
  c.i = ++tape.globalStatementCounter;

  tape.primvalValues[c.i] = c.p; // store primal in global vector
  tape.push(MUL);

  return c;
}

void operator_mul_b(const double* primal, double* adjoint, const int c_i)
    {
  int b_i = tape.pop<int>();
  int a_i = tape.pop<int>();

  adjoint[a_i] += primal[b_i] * adjoint[c_i];
  adjoint[b_i] += primal[a_i] * adjoint[c_i];
  adjoint[c_i] = 0.0;
}
```

Figure 6.2.: Implementation of the primal value taping for the multiplication. The primal values of the inputs are no longer stored. Instead the output value is stored.

6.4. Jacobi recording

Figure 2.3 on page 23 shows the reverse AD procedure and the equation for all elemental operators, which is

$$\begin{aligned}\bar{v}_j \mathrel{+}=& \frac{\partial \phi_i}{\partial v_j}(v_0, \ldots, v_{n+i-1})\bar{v}_{n+i} \quad \forall j = 1 \ldots (n+i-1) \\ \bar{v}_{n+i} =& 0 \ .\end{aligned}$$

This equation needs to be evaluated for every statement that is called in the program. The primal value taping stores the primal values, making the argument for the equation available. The gradient $\frac{\partial \phi}{\partial v}$ is computed during the reverse evaluation and used to update the bar values.

The Jacobi taping takes a different approach. Instead of storing the primal values, it directly computes the gradient $\frac{\partial \phi}{\partial v}$ and stores it onto the tape, since all primal values are available. Therefore, no primal values need to be restored during the reverse evaluation, simplifying the management.

The reverse evaluation of the tape is simplified with this method. The stored information consists of the Jacobi values for each argument, the corresponding index, and one bit which indicates a binary or unary operation. Normal overloading does not allow for more than two arguments, one bit is enough. The overloaded operators for the multiplication and the overloaded function for the sinus are implemented in Figure 6.3. The implementation for the tape evaluation covers all operators and functions $+$, $-$, $*$, $/$, *cos*, *sin*, *sqrt*, ... evaluating the full tape as opposed to one operation. Figure 6.4 shows the implementation which uses the start and end index of linear indexing as evaluation range. Since, the evaluation of the statements is done in the reverse order, the end index is decremented until the start index has been reached, as it is required by the reverse AD mode from Figure 2.3. The statements in the loop retrieve the information from the tape and perform the adjoint update. The layout of the pushes changes so that both kinds of operations can be handled identically. The memory for a binary operation is now $2 * (8+4) + \frac{1}{8} = 24 + \frac{1}{8}$ byte and for an unary operation it is $(8+4) + \frac{1}{8} = 12 + \frac{1}{8}$ byte.

Therefore the memory consumption is nearly the same as it is for the naive primal value taping. Only a small gain is achieved due to the lower amount of options for each operation, hence the Jacobi taping needs only two states. For the naive primal value taping the assumption is made that no more than 256 distinct elemental operations exist. The memory consumption for the l2norm_I1 is then $4*(24+\frac{1}{8})*n+24+\frac{1}{8}+12+\frac{1}{8} = 96*n+36+\frac{1}{2}n+\frac{1}{4}$ byte. The number of generated indices stays the same.

Remark 6.2 (Number of operations in the Jacobi taping)
If the number of addition, multiplication and nonlinear operations are compared for the primal value taping and the Jacobi taping, then the result is that the Jacobi taping evaluates the same amount of operations except for the number of multiplications. The multiplication count will be equal or higher for the Jacobi

```
RReal operator * (const RReal& a, const RReal& b) {
 tape.push(b.p); // dh/da = b
 tape.push(a.i);
 tape.push(a.p); // dh/db = a
 tape.push(b.i);

 RReal c;
 c.p = a.p * b.p;
 c.i = ++tape.globalStatementCounter;

 tape.push(true);

 return c;
}

RReal sin(const RReal& a) {
 tape.push(cos(a.p)); // dh/da = cos(a)
 tape.push(a.i);

 RReal c;
 c.p = sin(a.p);
 c.i = ++tape.globalStatementCounter;

 tape.push(false);

 return c;
}
```

Figure 6.3.: Implementation of Jacobi taping for the multiplication and the sinus. First the information for each argument is stored and then the information for the return value is updated.

```
void Tape::eval(const int end, const int start) {
 int cur = end;
 while(cur > start) {
  int c_i = cur;
  --cur;
  bool binary = pop<bool>();
  int arg_i = pop<int>();
  double arg_jac = pop<double>();
  adjoint[arg_i] += arg_jac * adjonit[c_i];
  if(binary) {
     arg_i = pop<int>();
     arg_jac = pop<double>();
     adjoint[arg_i] += arg_jac * adjonit[c_i];
  }
  adjoint[c_i] = 0;
 }
}
```

Figure 6.4.: Implementation of reverse evaluation for the Jacobi taping.

taping. The Jacobi taping loses all information about the kind of operation. For linear operations the Jacobi 1.0 has to be stored on the tape and in the reverse mode this 1.0 is multiplied by the adjoint of the left hand side. The primal value taping can optimize the functions for the reverse evaluation by omitting these operations. If only the operations in the reverse sweep are counted, than the Jacobi taping has clearly less operations. All the computations are evaluated during the recording of the tape, which increases the performance if the tape is evaluated multiple times.

6.5. Statement level evaluation

The theory for the forward and reverse mode of AD in Section 2.1 and 2.2 is based on the assumption that each computer program can be broken down into simple unary and binary operations. This restriction is never introduced into the theory. AD can handle operations with more than two arguments, but programming languages like C++ allow only the overloading of operators with one and two arguments. All mathematical functions in the standard library have one argument with few exceptions that have two. This viewpoint can be changed such that each statement is seen as a contracted elemental operation ϕ. Then the line

```
z = a + b * pow(c + d, 2.0);
```

resembles the mathematical function

$$z = h(a, b, c, d) = a + b * (c + d)^2 \ .$$

The Jacobi matrix of h can still be computed easily by hand. Algorithmic Differentiation is often also called Automatic Differentiation and this is the point were the “automatic” comes from. With the current implementations of the primal value taping and the Jacobi taping, the Jacobi matrix of h can be computed, but each operation is handled separately. The idea is to adapt both implementations allowing h to be handled as one single elemental operation ϕ by using expression templates [Vel95, ADP01]. The general theory for expression templates is explained and adapted for primal value taping and Jacobi taping.

Currently the definition for the overloaded functions are

$$\text{RReal} \circ \text{RReal} \mapsto \text{RReal} \quad \text{and} \quad F(\text{RReal}) \mapsto \text{RReal} \ .$$

The type RReal is now exchanged with a new *Expr* type based on the type that is used inside of the expression. This relation is indicated by a subscript of the used type. The new definitions for the overloaded functions are then

$$\text{Expr}_A \circ \text{Expr}_B \mapsto \text{Expr}_{A \circ B} \quad \text{and} \quad F(\text{Expr}_A) \mapsto \text{Expr}_{F(A)} \ .$$

The type RReal will be an *Expr* and the return type of each binary and unary operator will be an *Expr* that depends on the types of its arguments. With this technique an arbitrary amount of expressions can be linked together to form one big expression that contains all information about each operation and argument in that expression. The operator overloading tool sees now the whole statement and does not need to create intermediate values. If h is captured by an expression template, the resulting structure looks like:

```
ADD< RReal, MUL< RReal, POW< ADD< RReal, RReal>, RReal > > >
```

Each expression is templated with other expressions or the *RReal* type.

There are two major implications for using expression templates. The first one is the change of the return type. Because *typeof(Input)* $\neq$ *typeof(Output)* simple template functions or the ternary operator *?:* might no longer work. This has to be considered when using this technique. The second implication is, that in the overloaded operators no operations are performed. The type which is returned stores the arguments and knows which operations it needs to apply on these arguments. Not directly computing a value is also known as lazy evaluation [Wad71, HM76]. The interesting notion about this is, that not only the value can be computed, but other more advanced operations can be performed.

The implementation of the *Expr* (Expression) type will give a general concept for the unary and binary operations. Let $c = h_1(a)$ be the unary operator and $c = h_2(a, b)$ be the binary operator, then *d_a_h1*, *d_a_h2* and *d_b_h2* are the functions that calculate the derivatives of h_1 and h_2 with respect to the input variables a and b. All arguments of h_1 and h_2 are considered to be instances of a general interface that is called *Expr*. This interface has the functions *getValue* and *calcDerv*, where *getValue* returns the primal value of that expression. The other method *caclDerv* is specified later, but will be used to compute the reverse AD mode for that expression. Figure 6.5 shows the full implementation and will be discussed now step by step.

The implementation of *getValue* for h_1 and h_2 is straight forward. The function gets the values of the arguments and calls the primal function to evaluate the value of the expression.

The *calcDerv* method will be the reverse AD equivalent of the *getValue* method. It can be used to evaluate the gradient of the whole expression multiplied with a given adjoint directional derivative. The implementation and logic of the *calcDerv* method is quite involved and is explained best with an example. h_2 is still the binary operator. The arguments are declared as $A : \mathbb{R}^i \to \mathbb{R}, B : \mathbb{R}^j \to \mathbb{R}$ with $i, j \in \mathbb{N}$. The *getValue* method resembles $c = h_2(A(p), B(q))$ with $p \in \mathbb{R}^i$ and $q \in \mathbb{R}^j$. The evaluation schedule for the *getValue* method is then after the reverse AD Theorem

2.13:

$$
\begin{aligned}
a =& A(p)\\
b =& B(q)\\
c =& h_2(a,b)\\
\bar{a} \mathrel{+}=& \frac{\partial h_2}{\partial a}(a,b)^T\bar{c}\\
\bar{b} \mathrel{+}=& \frac{\partial h_2}{\partial b}(a,b)^T\bar{c}\\
\bar{c} =& 0\\
\bar{q} \mathrel{+}=& \frac{\partial B}{\partial q}^T\bar{b};\\
\bar{b} =& 0\\
\bar{p} \mathrel{+}=& \frac{\partial A}{\partial p}^T\bar{a}\\
\bar{a} =& 0
\end{aligned}
$$

The implementation of *calcDerv* is driven by this schedule. The first three equations are directly handled by lines 28 - 30 in Figure 6.5. Lines 31 and 32 implement the computation of $\frac{\partial h_2}{\partial a}^T\bar{c}$ and $\frac{\partial h_2}{\partial b}^T\bar{c}$. The two recursive calls of *calcDerv* implement the last four lines of the schedule. A and B are also *Expr* implementations and propagate the derivative values through the complete expression of the statement. The termination point in the recursion are the arguments to the elemental operator ϕ. These arguments will be of the type RReal and the call of *calcDerv* will contain the Jacobi matrix of the statement with respect to this argument. If e.g. the first argument for h_2 is of the type RReal, the seed is computed by the equation $\frac{\partial h_2}{\partial b}(a,b)^T\bar{c}$ which is the derivative of the operation with respect to the first argument. Because of the recursive implementation, the above argument holds for arbitrarily nested expressions. With this implementation all *RReal* arguments get the derivative value of ϕ with respect to the input variable they represent. This implementation allows operator overloading AD to compute the derivatives for arbitrarily long statements without recording intermediate values onto the tape. Figure 6.6 shows the implementation of the overloaded operators for the multiplication and the sinus. This technique can now be adapted to the primal value taping and the Jacobi taping.

The code example of the *l2_norm_I1* is now changed such that it requires less intermediate variables. The new implementation *l2_norm_I2* is shown in Figure 6.7. There are only two statements left, one takes 5 arguments (including res) and the other one takes two but is more complex than a simple binary operation. One could argue that the statement on line 5 has only 3 arguments. The default expression template implementation has no means to detect if two arguments are equal and therefore it is currently not possible to use this information. Ideas how this can be handled are discussed later in this chapter.

```
template<typename ExprA>
struct H_1_OP : public Expr {
  const ExprA& a;

  H_1_OP(const ExprA& a) : a(a) {}

  double getValue() const {return h1(a.getValue());}

  void calcDerv(double seed) const {
    double a_v = a.getValue();
    double c = h1(a_v);
    double a_d = d_a_h1(a_v, c) * seed;

    a.calcDerv(a_d);
  }
};

template<typename ExprA, typename ExprB>
struct H_2_OP : public Expr {
  const ExprA& a;
  const ExprB& b;

  H_2_OP(const ExprA& a, const ExprB& b) : a(a), b(b) {}

  double getValue() const {return h2(a.getValue(), b.getValue());}

  void calcDerv(double seed) const {
    double a_v = a.getValue();
    double b_v = b.getValue();
    double c = h2(a_v, b_v);
    double a_d = d_a_h2(a_v, b_v, c) * seed;
    double b_d = d_b_h2(a_v, b_v, c) * seed;

    a.calcDerv(a_d);
    b.calcDerv(b_d);
  }
};

struct RReal: public Expr {
  ...
  double getValue() const {return this->p;}

  // Jacobi taping
  void calcDerv(double seed) const {
    tape.push(seed);
    tape.push(this->i);
  }

  // primal value taping
  void calcDerv(double /*seed*/) const {
    tape.push(this->i);
  }
};
```

Figure 6.5.: Implementation of the expression creation for binary and unary functions. Also the RReal implementation of the *Expr* interface is shown.

```
double d_a_sin(double a, double res) {
  return cos(a);
}
template<typename ExprA>
H_1_Sin<ExprA> sin(const ExprA& a) {
  return H_1_Sin<ExprA>(a);
}

double d_a_mul(double a, double b, double res) {
  return b;
}
double d_b_mul(double a, double b, double res) {
  return a;
}
template<typename ExprA, typename ExprB>
H_2_Mul<ExprA, ExprB> operator *(const ExprA& a, const ExprB& b) {
  return H_2_Mul<ExprA, ExprB>(a, b);
}
```

Figure 6.6.: Specialization of the multiplication and sinus for expression templates and the implementation of the derivative functions.

```
template<typename Number>
Number l2norm_I1(const Number* v1, const Number* v2, const size_t n) {
  Number res = 0.0;
  for(size_t i = 0; i < n; ++i) {
    res += v1[i] * v1[i] + v2[i] * v2[i];
  }
  res = sqrt(res) / (Number)n;

  return res;
}
```

Figure 6.7.: Improved implementation of the L2-norm for AD tool comparison.

The Jacobi approach can use the *calcDerv* method to compute the full gradient of the statement. In order to do that the adjoint direction $\bar{c}$ is set to one. This is the *seed* argument of the function *calcDerv*. The call will propagate the derivative values to the *RReal* arguments. Each *RReal* sees the partial derivative of the full statement with respect to itself. The derivative is stored together with the index of the *RReal*. This can be seen in Figure 6.5 line 45f. For each argument this will require 12 bytes of memory, 8 for the partial derivative and 4 for the index. In addition each statement needs to know the number of arguments requiring 1 byte. The memory per statement totals $(4 + 8) * n + 1$ byte, which is only true if the maximum number of arguments is capped at 255. The memory for the function *l2_norm_I2* is $(5 * 12 + 1) * n + 12 + 1 = 61n + 13$ byte and $n + 1$ indices will be generated. It is clear that this implementation technique reduces the memory amount for the tape drastically. The memory requirement is reduced by a factor of two and is accompanied by a big reduction in the memory that is required for the adjoint vector.

The primal value taping approach can set the seed $\bar{c}$, too. When the reverse evaluation comes to the point where it wants to evaluate equation (2.13) for ϕ the value of $\bar{c}$ is known. *calcDerv* is called on the full statement and the arguments of the calls to *calcDerv* for the RReals contain the values of $\frac{\partial \phi}{\partial v_i}\bar{c}$. They can directly be used to update the adjoint values of the arguments of the statement. For each argument one index of 4 bytes needs to be stored. This can be seen in Figure 6.5 line 51. The computed seed is ignored. An improved implementation can add a method *storeIndices* which would only store the indices and skip the gradient computation. For each statement the value of the left hand side is stored which are 8 bytes and the function that performs the reverse evaluation of the statement. This function can be stored in a lookup table with 4 byte keys or directly as a function pointer with 8 bytes. The lookup table for the functions can no longer be programmed in a static or fixed way. Either a map needs to be used or a script needs to extract all the statements from the object files and generate a static lookup table from this list. Because the map introduces a runtime overhead and the other approach complicates the build process drastically, the function pointer approach is chosen here. Each statement needs $4*n+8+8$ bytes of memory. This results in $(5*4+16)*n+4+16 = 36n+20$ bytes of stored memory for the l2_norm_I2 implementation and the use of $n + 1$ indices. The same conclusion as for the Jacobi taping can be drawn here. It is interesting to note, that the constant memory for each statement is higher than that for the Jacobi taping approach but the memory for each argument is less.

6.6. Reuse index management

The generated indices have been constantly growing and have not been reused. As an example the *l2_norm_I1* can be examined. The three indices inside of the loop are only used once. If these indices are reused in every loop iteration, memory could be saved since the adjoint vector is reduced in size. The general theory for the reuse of indices will be developed first and afterwards the important changes to the primal value taping and Jacobi taping will be discussed after.

The struct *RReal* can be extended with a destructor to release the indices of temporary values. The implementation would be

```
~RReal() {
  tape.releaseIndex(this.i);
}
```

The tape can store the released index and use it elsewhere. When an old index is overwritten, it needs to be released, too. Depending on the implementation the old index can be released before or after the statement is evaluated. The tape can return this index in the evaluation of the statement, resulting in a state where the index has not changed. This has to be considered in the implementation. Furthermore, the variable on the left hand side can be used as an argument on the right hand side (rhs). This favors the release of the index after the statement has been evaluated such that the index of the left hand side always changes.

Without loss of generality a binary function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is defined. The evaluation is coded as $c = h(a, b)$ and c, a and b are *RReal* variables. c, a and b have an associated index i_c, i_a and $i_b \in \mathbb{N}$. It can be possible that $i_c \equiv i_a \equiv i_b$. If $i_c \equiv i_a$ is assumed, the formulation in the code is $c = h(c, b)$. This can be written as

$$\begin{aligned} t =& c \\ c =& h(t, b) \ . \end{aligned}$$

The forward AD mode for the procedure is after Theorem 2.10

$$\begin{aligned} \dot{t} =& \dot{c} \\ \dot{c} =& \frac{\partial h}{\partial a}\dot{t} + \frac{\partial h}{\partial b}\dot{b} \ . \end{aligned}$$

and therefore the reverse mode is after Theorem 2.13

$$\begin{aligned} \bar{b} \mathrel{+}=& \frac{\partial h}{\partial b}\bar{c} \\ \bar{t} \mathrel{+}=& \frac{\partial h}{\partial a}\bar{c} \\ \bar{c} =& 0 \\ \bar{c} \mathrel{+}=& \bar{t} \\ \bar{t} =& 0 \ . \end{aligned} \tag{6.1}$$

t is used as a temporary variable and only used once. $\bar{t}$ is 0 at the beginning of the evaluation and the update $\bar{t} \mathrel{+}= \frac{\partial h}{\partial a}\bar{c}$ becomes $\bar{t} = \frac{\partial h}{\partial a}\bar{c}$. Because $\bar{c}$ is set to 0 in the algorithm, the update of $\bar{c}$ becomes an assignment, too. Therefore the temporary value of $\bar{t}$ can be removed from the procedure in (6.1). After the removal of $\bar{t}$ it becomes

$$\begin{aligned}\bar{b} &\mathrel{+}= \frac{\partial h}{\partial b}\bar{c}_{old} \\ \bar{c}_{new} &= \frac{\partial h}{\partial a}\bar{c}_{old}\end{aligned} \tag{6.2}$$

for the adjoint version of $c = h(c, b)$. A general implementation does not know if the left hand side is also used on the right hand side. Hence, when the indices are reused, the implementation needs to assume that the left hand side is always used on the right hand side. But the general notation requires the update ($+=$) instead of the assignment ($=$). During the reverse evaluation there will be no distinction between $\bar{c}_{new}$ and $\bar{c}_{old}$ since they have the same index. In order to handle this an implementation can copy the old value of $\bar{c}$, set $\bar{c}$ to zero and use the copied value in the update. The algorithm for the example is

$$\begin{aligned}\tau &= \bar{c} \\ \bar{c} &= 0 \\ \bar{b} &\mathrel{+}= \frac{\partial h}{\partial b}\tau \\ \bar{c} &\mathrel{+}= \frac{\partial h}{\partial a}\tau \ .\end{aligned} \tag{6.3}$$

The evaluation procedure can use the adjoints of the arguments without caring if the index of the left hand side is used on the right hand side. It is now clear, that the index can be released before the statement is evaluated but an optimized implementation will avoid the release and reacquisition of the index.

The consequences for the Jacobi taping are minimal. Nearly everything can stay the same in the recording of each statement. The only change necessary is the index of the left hand side needing to be stored because it is no longer increasing monotonically. Therefore an additional 4 bytes are needed for each statement. The implementation of the reverse evaluation in Figure 6.4 needs to be slightly adapted. Figure 6.8 shows the new implementation which copies the adjoint of c and sets the value in the vector to zero.

The changes for the primal value taping are more involved. The primal value taping approach with linear indexing uses the global vector of all primal values directly, this is possible because each index is used only once in an assignment. Afterwards the index is only used on the right hand side. In the example:

```
for(int i = 0; i < n; ++i) {
  c *= a[i];
}
```

c is overwritten in each iteration of the loop. If the indices are reused, the index of c will not change after each assignment and the value in the global primal value vector is overwritten several times. This has two implications. The first being a change in the meaning of the global primal value vector. It represents now the current state of all primal values, that are currently used in the program. Beforehand it represented

```
void Tape::eval(const int end, const int start) {
 int cur = end;
 while(cur > start) {
   --cur;
   bool binary = pop<bool>();

   int c_i = pop<int>();
   double c_b = adjoint[c_i];
   adjoint[c_i] = 0.0;

   int arg_i = pop<int>();
   double arg_jac = pop<double>();
   adjoint[arg_i] += arg_jac * c_b;
   if(binary) {
      arg_i = pop<int>();
      arg_jac = pop<double>();
      adjoint[arg_i] += arg_jac * c_b;
   }
 }
}
```

Figure 6.8.: Implementation of reverse evaluation for the Jacobi taping with reused indices.

all primal values during the lifetime of the program. The second implication is, that the state of the global primal value vector needs to be reversed during the reverse sweep, too. This can only be done if the overwritten value in the global primal value vector is stored on the tape before it is overwritten. The stored value can be used to restore the old state of the global primal value vector. The implementation in Figure 6.9 reflects this with the storing of the old primal value in line 10 and the restoring of the old value in line 23. The primal vector and the adjoint vector are reduced in size because they only grow as large as the maximum number of variables that are used simultaneously in the program.

It is important to know how many indices are acquired and released in all methods. For the *l2_norm_I1* implementation 3 indices are used and released in each iteration of the for loop and one index is acquired for the return variable. Only 4 indices are used in place of the $4n + 2$ indices without reuse that were previously needed. For the *l2_norm_I2* implementation there is only one index used with the reuse of the indices, in place of the previous $n + 1$.

The memory for each statement is increased and the size of the adjoint vector is decreased. Although it is not guaranteed, it is very likely that there will be a significant gain. An other advantage of the index reuse is the evaluation speed of the reverse evaluation. The recording will take more time because the indices need to be managed. However, the reduced size of the adjoint vector makes it more likely that the indices are closer together and therefore the cpu can use the cached data more efficiently. If the maximum number of indices is low, it is possible that the whole

```
RReal operator * (const RReal& a, const RReal& b) {
  tape.push(a.i);
  tape.push(b.i);

  RReal c;
  c.p = a.p * b.p;
  c.i = tape.getIndex();

  tape.push(c.i);
  tape.push(tape.primalValues[c.i]);
  tape.primvalValues[c.i] = c.p;
  tape.push(MUL);

  return c;
}

void operator_mul_b(const double* primal, double* adjoint) {
  double c_p_old = tape.pop<double>();
  int c_i = tape.pop<int>();
  int b_i = tape.pop<int>();
  int a_i = tape.pop<int>();

  primal[c_i] = c_p_old;
  double c_b = adjoint[c_i];
  adjoint[c_i] = 0.0;
  adjoint[a_i] += primal[b_i] * c_b;
  adjoint[b_i] += primal[a_i] * c_b;
}
```

Figure 6.9.: Implementation of the primal value taping for the multiplication with reused indices.

vector will fit into the cache. With linear increasing indices this is only possible for very small programs.

Remark 6.3 (Indices in the example)
The example of the L2 norm favors the index reuse very strongly. The loop is kind of unrolled on the AD tape without the index reuse.

Remark 6.4 (Destructor and C-like memory operations)
The index reuse introduces a destructor call to the type implementation. If this destructor call is missed, the index can not be reused. Especially if arrays are created with *malloc* and released with *free* the indices will not be released.

The next problematic function is memcpy. The result of a call to memcpy will be that two variables hold the same index and if on both variables the destructor is called, the index will be freed twice. The behavior of the program is undefined in such cases. It is therefore important to ensure that no C-like memory operations are used on the *RReal* type when the indices are reused.

Remark 6.5 (Index handling)
The handling of the indices is very critical with respect to the performance of the implementation. The acquisition and release functions for the indices are called in each constructor, destructor and statement. No time consuming sorting or insertion of the indices should be done.

Remark 6.6 (Multiple tape evaluations)
In many applications of reverse AD the tape is recorded once and evaluated multiple times. The primal value taping with index reuse can still do this but needs additional memory. Since the global primal value vector is modified in the reverse sweep, a second reverse sweep would use the modified vector and the evaluation will yield wrong results. Therefore the vector needs to be copied and the reverse sweep needs to act on the copied vector.

6.7. Constant value treatment

Lets assume that a program contains the statement

$$c = 4.0 * a$$

where c and a are *RReal* types or the statement

$$c = p * a$$

where c and a are *RReal* types and p is a double. From the viewpoint of the implementation both equations are identical. The current implementations handle this situation by converting the constant values to *RReal* types. The derivative information with respect to the constant value would be stored and in the reverse evaluation the derivative information with respect to this value is computed. The aim is to eliminate this overhead and discuss possible drawbacks on the Jacobi taping or the primal value taping.

The Jacobi taping approach has a very simple solution. The Jacobi with respect to the constant or constant variable is not stored. E.g. the *operator ** can be overloaded for `RReal operator *(const double& a, const RReal& b)` and then be treated as an unary operation.

The situation is more involved for the primal value taping scheme. In the reverse evaluation the value of 4.0 or p is needed, and needs to be stored on the tape. The memory for the unary operation which is actually a binary operation changes such that it needs to store the value of the constant. Therefore 8 bytes of additional memory is needed but is still better than the required memory for the implicit conversion of the variable. Here 12 bytes would be used in addition to the memory for the binary operation.

For the primal value taping it can be more memory efficient to declare a constant value as a global variable with the type *RReal*. The same is true if a constant value is used in a loop. Then it is more efficient to declare the variable before the loop as a *RReal* value.

6.8. Recording activation and deactivation

Tape activation and deactivation might be necessary in two scenarios. If parts of the program use the *RReal* type but do not need to be differentiated then the calculations in these parts of the program should not be recorded on the tape. The second scenario arises if the developer knows that some parts of the computation do not influence the derivative result but the developer is unable to perform the computation with double variables. Figure 5.8 on page 110 showcases an example of this for function checkpoints. The tape deactivation has several implications for the primal value taping, the Jacobi taping and the index management schemes. Changes for all four approaches will be shown.

The tape can use the index 0 in the current implementation to indicate values that have no dependency to input variables. The tape is called to be *passive* if it is not recording and values with an index of 0 are called passive values. Lets assume, that for the function $c = h(a, b) = a * b$ the variables a and b are of the type *RReal* and a has the index 0 to indicate that the variable is passive. The derivative with respect to the passive variable a does not need to be calculated.

The implementation for the Jacobi taping is straight forward . It can check for the zero index and in such a case it does not store the Jacobi matrix and the index. If all of the arguments have a zero index then the whole statement can be ignored. Therefore memory can be saved while the Jacobi taping is used.

Primal value taping encounters problems when the tape contains 0 indices. Since the zero index is shared for all passive values it can not be used to store primal values. How the implementation solves this problem is highly dependent on the other optimizations used. If no indices are reused the tape can generate temporal values easily. If the indices are reused then the indices of the temporal variables need to be cleared correctly.

The first option is to create an additional assign statement for each temporal variable, which takes care of the storage of the primal values. If e.g. the following implementation of the multiplication is used:

```
RReal operator *(const RReal& a, const RReal& b) {
  ...
  if(a.i == 0) {
    RReal aTemp = a;
    tape.makeActive(aTemp);
    returns aTemp*b;
  }
  ...
}
```

Everything is correct because the index is released after the operation is evaluated. But if expression templates are used then the code would be:

```
template<typename A, teypename B>
Mul<A,B> operator *(const A& a, const B& b) {
  ...
  if(a.i == 0) {
    RReal aTemp = a;
    tape.makeActive(aTemp);
    returns Mul<A,B>(aTemp, b);
  }
  ...
}
```

The problem with this code is, that the operation is not yet evaluated and the index is already freed. If the statement $c = (((a_1 * a_2) * a_3) * a_4) * a_5$ is considered and $a_2, \ldots, a_5$ have zero indices. It can happen, that the temporary variables for $a_2, \ldots, a_5$ have all the same index. The expression templates can cover this case when the expression is stored:

```
 1  template<typename Rhs>
 2  void store(RReal& lhs, const Rhs& rhs) {
 3    RReal tempValues[Rhs::maxNumberOfValues];
 4    int nTempValues = rhs.createTempValues(0, tempValues);
 5
 6    rhs.storeIndices(0, tempValues);
 7
 8    ...
 9    for(int i = nTempValues - 1; i >= 0; --i) {
10      freeTemporary(tempValues[i]);
11    }
12  }
```

There is an additional call generating the temporal values in line four. The *store* of the indices uses this vector to replace the passive variables. At the end of the method the indices are released in line 11. This eliminates the problem, that all the temporal values may have the same index. Every temporal variable requires an additional storage of 20 bytes.

The difference between the Jacobi taping and the primal value taping have become clear. The Jacobi taping will definitely save memory. For the primal value taping, it is not clear if memory can be saved and the implementation is not trivial.

Another drawback of the primal value taping and the index reuse is that the tape can not be set to passive as easily as the Jacobi taping for parts of the code. If the indices are released during the passive code portion, then the following code parts need more memory because of the higher amount of passive values. If the indices are left in place the value vector needs to be updated and a statement restoring the overwritten value in the reverse sweep needs to be recorded. Especially that there are still statements recorded in code parts that the developer has deactivated is counter intuitive.

6.9. Inputs, outputs, constructors and assignment operators

The previous sections omitted the constructors of the *RReal* objects. This is done on purpose as the code examples would have been unclear and the implementation now is only possible because of other optimizations. The implementational details depend greatly on the chosen approach and differ for primal value taping and Jacobi taping. The index management influences the implementation, too.

With linear increasing indices every statement gets a new index. This means that every adjoint variable has a corresponding statement. If a user wants to define an input variable, then the adjoint of this variable needs an index. These two reasons lead to the conclusion that each input variable needs to create a statement upon the tape.

This can be implemented in a function *registerInput* which is analogue to the implementation of an unary or binary operator with the exception that no argument needs to be stored. The implementation is done in Figure 6.10 with the assumption that the statement level evaluation is used. From this code, the implementation of the *RReal* constructor can be derived. If passive values are allowed on the tape the index can be equal to zero the constructor simply initializes i with 0. If zero indices are not allowed, then the constructor must call *registerInput* on the constructed value. This would require for every statement like `RReal a;` that some data is written to the tape.

The statement `RReal a;` is for example used in Figure 2.13 on page 36. If the constructor would already have been defined and no passive values allowed, then the index for a would be automatically created. A second constructor could have been implemented and used in the example. For the overloaded sinus this would look like `RReal c(sin(a.p), ++tape.globalStatementCounter);`. This would have destroyed the clear coding and would have raised unnecessary issues during the discussion.

The situation changes if the indices are reused. The coupling between the indices and statements is eliminated and therefore the implementation of *registerInput* is simple. It acquires a new index from the tape and is shown in Figure 6.10.

The assignment operator requires a different handling. It can be seen as an unary function that has a Jacobi value of 1.0. The adjoint for the statement $b = a$ is

$$\begin{aligned}\bar{a} \mathrel{+}=& \bar{b} \\ \bar{b} =& 0 \ .\end{aligned}$$

The adjoint states that every update of $\bar{b}$ is added to $\bar{a}$. If linear increasing indices are used $\bar{a}$ corresponds to a fixed index and all updates to $\bar{b}$ could be directly done to $\bar{a}$. This can be achieved if the *RReal* variables of a and b share the same index. Consequently the assignment operator implemented in Figure 6.11 can copy the values from the argument. This optimization is not possible if the indices are reused. If one index is released twice, correct derivative results can no longer be guaranteed. The assignment operator for the index reuse implementation in Figure 6.11 needs to create a statement for each assignment.

Caused by this difference, it can be that an index reuse approach might need more memory than a linear increasing approach. Depending on the number of assignments the memory advantage from the reduced size of the adjoint vector is used up by the additional statements.

The outputs do not need to be handled for any combination of primal value taping, Jacobi taping, linear indexing and index reuse. No handling is required for index reuse because each variable has a unique index.

The situation is different if the assign optimization is used for the linear indexing. In the example

```
void h(const RReal& in, RReal& out1, RReal& out2) {
 out1 = in;
 out2 = in;
}
```

func copies the input to both output variables. *out1* and *out2* will have the same index. If the user sets the adjoint of only *out1* or *out2* by calling *tape.setAdjoint(out1, 1.0)*, the derivative of h will be correct. But if the user sets both adjoints to a specific value, the derivative of h will be wrong. That is due to the second call to *setAdjoint* which will overwrite the adjoint that is already set.

The origin of the problem can be found in Figure 2.3. Here the adjoints are updated as described by the $+=$ sign. By setting the adjoints explicitly the update from the theory is not performed and the result of h in the example is wrong. But this can only happen if two output variables have the same index and for both an adjoint value is set. Since, the theory assumes implicitly that each output variable has a unique index, the problem can not occur. It can be solved by introducing a new function *updateAdjoint* performing the operation $+=$. A second solution to the problem requires each output to be registered with *registerOutput* this function ensures that each output has a unique index.

```
void registerInput(RReal& value) { // linear inxreasing indices
  value.i = ++tape.globalStatementCounter;

  // primal value taping
  tape.primalValues[c.i] = value.p; // primal value vector
  tape.push(NO_OPT);         // function pointer

  // jacobi taping
  tape.push(0); // zero agruments
}

void registerInput(RReal& value) { // index reuse
  value.i = tape.getIndex();
}
```

Figure 6.10.: Register an input variable on the tape once for linear increasing indices and once for reused indices. This implementation assumes that statement level evaluation is used.

```
struct RReal {
  ...
  // Linear indexing implementation
  void operator = (const RReal& a) {
    this->p = a.p;
    this->i = a.i;
  }

  // Index reuse implementation
  void operator = (const RReal& a) {
    this->p = a.p;

    // primal value taping
    tape.push(a.i);
    tape.primalValues[c.i] = value.p; // primal value vector
    tape.push(ASSIGN_OPT);     // function pointer

    // jacobi taping
    tape.push(1.0);
    tape.push(a.i);
    tape.push(this->i); // index left hand side
    tape.push(1);  // one agrument
  }
  ...
};
```

Figure 6.11.: Implementations of the assignment operator for linear indexing or index reuse.

6.10. Optimizations for linear equations

In the Jacobi taping approach a Jacobi matrix is stored for each argument. If the equation is linear in every argument, then the Jacobi values are constant values. The computation of the adjoint does not need any primal values or the stored Jacobi values. As an example consider the equation

$$z = a + b + c + d \ .$$

All Jacobi values are just one in this statement. If the tape knows about this, then it does not need to store the Jacobi matrix and can directly update the adjoints after equation (2.13). However, if the equation is changed to

$$z = a - b + c + d \quad ,$$

three Jacobi values are one and the other one is negative. Here, the tape can not update the adjoints since one element needs to be decremented. The tape would need to store every Jacobi value of the equation again. The primal value tape could evaluate the same equation and with a specialized implementation would not need to access the primal values. As no Jacobi values need to be stored, the required memory could be reduced, and therefore the evaluation process for the Jaocbi taping can be improved. For the linear optimizations a specialized function is required that must be able to evaluate the expression by using only the right hand side indices. It is assumed that such an function exists.

Changes to the Jacobi taping include the storing of a function pointer which uses 8 bytes. It is also necessary to mark the statement for the reverse accumulation that no Jacobi values have been written to the tape. One bit of the argument count is used to indicate if the statement is purely linear. $(n-1)*8$ bytes can be saved if the right hand side of the statement is purely linear, where n is the number of input values.

The next equation to consider is

$$z = a * (b + c + d) \ .$$

If this equation is written in two statements

$$\begin{aligned} z_1 &= b + c + d \\ z_2 &= a * z_1 \quad , \end{aligned}$$

then the linear optimization can be applied to z_1 but not to z. The aim is to analyze every statement for linear subgraphs and to add a temporary value at the roots of the linear subgraphs.

The evaluation graph of a statement is known at compile time and the information if a subgraph is linear, is a compile time constant. Possible checks for linear subgraphs can be optimized by the compiler during the compilation and no checks are necessary to be performed during runtime. The storing process of statements can be changed in such a way that no overhead for nonlinear functions with no linear part is introduced. The process is divided into two steps. The first step creates the temporary variables and stores the linear subgraphs. The second step replaces the subgraphs with the temporary values. The implementation is shown in Figures 6.12 and 6.13. The store method in Figure 6.12 has 3 branches. The first branch handles a statement which is purely linear. It stores the indices of the arguments with the call to *pushIndices*. This recursive function is called through all expressions and terminates in the *RReal* arguments. They will push their indices on the tape. After that the function for the reverse evaluation is stored and the argument count is flagged, such that the reverse sweep knows that this is a linear expression. On line 11 the second branch starts. The implementation is quite involved and requires a very good understanding of C++. The branch creates a number of temporary variables for the subexpressions and calls the method that stores all of the linear subexpressions. The zero in the template argument describes the number of linear subgraphs that have already been stored on the tape. The number is a template parameter, since only then the compiler can optimize it out. It is needed to access the corresponding temporary value that is used to replace the subgraph. The implementation of the *pushLinearSubgraphs* function is only provided for a binary expression in Figure 6.13 in line 11. This function checks for each argument if it is linear. If it is, then the *store* method on the global tape is called and the first branch of the *store* method is evaluated. If the expression is not linear, then the method is called recursively on the arguments. It is important to note that for the second argument of the binary expression the index in the variable vector is increased by the number of linear subexpressions of the first argument. This ensures that no temporary value is used twice.

After all linear subexpressions are stored on the tape, the Jacobi entries can be computed. The call *calcDervWithSubgraphs* in Figure 6.12 on line 15 uses the same indexing technique as the function *pushLinearSubgraphs*. The implementation in Figure 6.13 on line 25 uses the same recursion as described in Section 6.5 for the calculation of the derivatives, but it excludes the linear subsections and uses the temporary values instead. It is important to note, that all decisions and index calculations in the methods can be evaluated at compile time and therefore the compiler can optimize them. This should yield a code were no if statements are left and no index calculations are performed.

For a statement that is already linear, the reduction of the memory is $(n-1)*8$ bytes. If the statement has n arguments and a linear subgraph with i arguments, with $n > i$, then the original memory requirement would be

$$mem = 4 + 1 + (8+4)*n \ .$$

The memory requirement for the linear subgraph is

$$mem_{sub} = 4 + 1 + 8 + 4*i \ .$$

The number of arguments for the optimized statement is now $(n-i+1)$, because the linear subgraph removes i arguments from the graph and it is replaced by a

```
template<typename Rhs>
void store(RReal lhs, const Rhs& rhs) {
 if(Rhs::isLinear) {
  lhs.i = ++tape.globalStatementCounter;

  rhs.pushIndices();

  tape.push(Rhs::linearReverseEvaluation);
  tape.push(Rhs::numberOfArguments | IS_LINEAR);
 } else if(RhS::hasLinearSubgraphs) {
  RReal tempVars[Rhs::NumberOfLinearSubgraphs];

  rhs.pushLinearSubgraphs<0>(tempVars);

  rhs.calcDervWithSubgraphs<0>(1.0, tempVars);
  lhs.i = ++tape.globalStatementCounter;

  tape.push(Rhs::numberOfArgumentsWithoutSubgraphs);
 } else {
  < default store>
 }
}
```

Figure 6.12.: The *store* method implementation for the linear subexpression optimization in a Jacobi taping setting.

new argument which holds the result of the subgraph. The total memory for the optimized equation is then

$$mem_{opt} = mem_{sub} + 4 + 1 + (8 + 4) * (n - i + 1) \ .$$

The memory of the optimized statement should be smaller than the memory of the original statement. This yields

$$\begin{aligned}
& & mem_{opt} &< mem \\
\Leftrightarrow & & (4 + 1 + 8 + 4 * i) + 4 + 1 + (8 + 4) * (n - i + 1) &< 4 + 1 + (8 + 4) * n \\
\Leftrightarrow & & 4 + 1 + 8 + 4 * (n + 1) + 8 * (n - i + 1) &< 8 * n + 4 * n \\
\Leftrightarrow & & 25 &< 8 * i \\
\Leftrightarrow & & 3.125 &< i \ .
\end{aligned}$$

The subgraph elimination yields a memory improvement only if the subgraph has 4 or more arguments. This constant factor can be added to the checks in the functions to ensure that memory will be saved. Because the factor is constant, the compiler can still optimize all *if* statements such that no checks are performed during runtime.

Remark 6.7 (Linear subgraphs and passive values)
The Jacobi taping can implement passive values such that they are not stored

```
template<typename ExprA, typename ExprB>
struct H_2_OP : public Expr {
  ...

  void pushIndices() const{
    a.pushIndices();
    b.pushIndices();
  }

  template<int tempIndex>
  void pushLinearSubgraphs(RReal* tempVars) const {
    if(ExprA::isLinear) {
      globalTape.store(tempVars[tempIndex], a);
    } else (ExprA::hasLinearSubgraphs) {
      pushLinearSubgraphs<tempIndex>(tempVars);
    }
    if(ExprB::isLinear) {
      globalTape.store(tempVars[tempIndex +
          ExprA::numberOfLinearSubgraphs], a);
    } else {
      pushLinearSubgraphs<tempIndex +
          ExprA::numberOfLinearSubgraphs>(tempVars);
    }
  }

  template<int tempIndex>
  void calcDervWithSubgraphs(double seed, RReal* tempVars) const {
    double a_v = a.getValue();
    double b_v = b.getValue();
    double c = h2(a.getValue(), b.getValue());
    double a_d = d_a_h_2(a_v, b_v, c) * seed;
    double b_d = d_b_h_2(a_v, b_v, c) * seed;

    if(ExprA::isLinear) {
      tempVars[tempIndex].calcDerv(seed);
    } else {
      a.calcDervWithSubgraphs<tempIndex>(a_d, tempVars);
    }
    if(ExprB::isLinear) {
      tempVars[tempIndex + ExprA::numberOfLinearSubgraphs]].calcDerv(seed);
    } else {
      b.calcDervWithSubgraphs<tempIndex +
          ExprA::numberOfLinearSubgraphs]>(b_d, tempVars);
    }
  }
}
```

Figure 6.13.: Expression implementation for the linear subexpression optimization in a Jacobi taping setting. a and b are the arguments of the binary expression. Their definition is shown in Figure 6.5 on page 137.

on the tape. This is not possible when a linear statement is stored. It can not be determined at compile time which variable has a passive index and no optimized function can be used. That means that all indices need to be stored on the tape. The function for the reverse evaluation will check in the reverse sweep which index is zero and skip that index.

6.11. Argument optimizations

The example implementation of the *l2_norm_I1* in Figure 6.1 on page 129 contains the statement

$$c = a \cdot a \ .$$

It is mentioned that AD can not optimize this statement. Possible avenues for handling this situation will now be discussed.

The reverse AD formulation for this statement is

$$\begin{aligned} \bar{a} &\mathrel{+}= a \cdot \bar{c} \\ \bar{a} &\mathrel{+}= a \cdot \bar{c} \\ \bar{c} &= 0 \end{aligned}$$

and could be improved to

$$\begin{aligned} \bar{a} &\mathrel{+}= 2 \cdot a \cdot \bar{c} \\ \bar{c} &= 0 \ . \end{aligned}$$

The original statement will generate two argument entries for the Jacoby taping approach since the expression templates can not detect if the arguments to the binary expression have the same index. Even if there is a special case for this in the multiplication operation, then the simple equation

$$c = sin(a) \cdot cos(a)$$

would break this improvement.

The idea is to optimize the recording of the statements in such a way, that for each variable only one entry is crated as opposed to multiple ones. Let $\phi_i : V \to \mathbb{R}$ be an elemental operator which represents a statement in the code. ϕ_i has $j \in \mathbb{N}$ arguments. Let $h : \mathbb{R}^j \to \mathbb{R}$ be the contracted form of the elemental operator. Then h can be called as

$$c = h(a_1, \ldots, a_j)$$

with $a_1, \ldots, a_j \in \mathbb{R}$ or as

$$c = h(\underbrace{a, \ldots, a}_{j\text{-times}})$$

```
struct RRealRef : public Expression {
  const RReal& ref;
  mutable double jacobi;

  RRealRef(const RReal& ref) :
    ref(ref),
    jacobi(0.0) {}

  void calcDerv(double seed) {
    jacobi += seed;
  }

  void pushLazyJacobies() {
    if(jacobi != 0.0) {
      ref.calcDerv(jacobi);
      jacobi = 0.0;
    }
  }
};
```

Figure 6.14.: Helper structure for the combination of multiple arguments. The structure delays the call to *calcDerv* in order to reduce the number of stored arguments, when a variable is used multiple times in the same statement.

with $a \in \mathbb{R}$. The first variant needs to store j arguments on the tape, the second only 1. The compiler could generate a specialized version for both cases but without the compile time knowledge about the indices, such optimizations are not possible. Because of this, the compiler will only generate the general version and there are no compile time means to detect if multiple arguments have the same index.

The brute force approach would be to search for the variables that have the same index. The search for all variables will have a complexity of $j\log(j) + j$ if the arguments are first sorted and then combined. The problem is that for each statement this overhead is introduced and would slowdown the recording. But this is the only option if the differentiated code is not modified.

If the code can be modified the solution is very efficient. The *RReal* objects have been the termination points in the expression templates until now. The structure *RRealRef* (reverse real reference) in Figure 6.14 is a new termination point. It stores a reference to a *RReal* object and has a mutable member for the Jacobi. The regular call to *calcDerv* in *RReal* (see Figure 6.5) stores the index and the seed on the tape. The *RRealRef* implementation updates the internal Jacobi instead. The call to *pushLazyJacobies* will forward the accumulated Jacobi to the *RReal* reference, which can then push this Jacobi. It will be pushed only once because it is reset after it is pushed. The *storeLazyJacboies* method needs to be implemented in all expression templates and the method needs to be called from the procedure that stores the statement information on the tape.

An example for the use of the *RRealRef* is

```
c_1 = 3*a*a*a-4*a*a+3*a-8 + sin(a);

RRealRef aR = a;
c_2 = 3*aR*aR*aR-4*aR*aR+3*aR-8 + sin(aR);
```

The statement for *c_1* will store 7 arguments on the tape whereas statement *c_2* will only store one.

The optimization can not be done for the primal value taping because no specialized function can be generated or the function would require additional data to identify which arguments in the primal call have been the same. This would increase the required memory and would defeat the purpose of reducing the required memory.

6.12. Overview on existing operator operator overloading Algorithmic Differentiation tools

The web page *www.autodiff.org* is most often the first source of information if somebody is looking for an AD tool implementation. The page contains a database of nearly all papers published on AD, and probably the most complete list of available AD tools. For C/C++, the list contains 24 entries. From these 24 items, 11 are pure forward AD tools, one is a pure reverse mode tool and 12 tools can handle the forward and reverse mode. Unfortunately, 8 tools from the 24 seem to be discontinued or abandoned. From the 9 possible reverse tools, the two source transformation tools OpenAD [Utk04] and Tapenade [HP13] are not considered. This leaves 7 reverse mode AD tools, that are to be considered by this thesis for a deeper analysis.

The **AD model builder** [FSA+12] is a special tool for the solution of non-linear statistical modeling and optimization problems. The user has to describe the problem in a special modeling language. The AD model builder provides a preprocessor that generates the appropriate source code for the solution of the problem and the generated code uses the AD tool from the AD model builder suite for the computation of the derivatives. The AD tool is the most general in the field. It implements AD for higher order structures like vectors and matrices. The taping approach is a naive primal one. The arguments of the operators are stored in 8 byte values. Each vector argument uses 16 bytes for information about the vector size and the location of the data. 16 bytes are also used for the output argument. The tool performs an operator taping approach and uses 32 bytes per operation. All memory allocations are made in sequence, so the memory is not scattered. The memory per operation is

$$8 * \text{inputValues} + 16 * \text{nArgs} + 1 + 16 + 32 \ . \tag{6.4}$$

If the vectors are large enough, then the overhead from the references, function pointers and sizes are negligible. But, if the vector size is one, then the overhead is quite large. The tool implements no assign optimization and the references resemble a linear index scheme.

FADBAD [BS96] is a traditional operator overloading tool. The user uses the FADBAD reverse type *B<double>* in the program for the derivative computation. FADBAD takes an operator taping approach with primal value taping, but does not use indices for the dependency management. A global graph structure is built and carried by each *B<double>* type. An operation adds to the graph structure and everything is allocated with *new* so the memory is scattered. Since, everything is stored, it is equivalent to an linear indexing scheme. The memory footprint for an operation is 16 bytes and additional 24 bytes for the storing of the left hand side value. This memory contains references to the adjoint vector but not yet the memory for the adjoint vector. For each argument 8 bytes are required for the reference. The memory per operation is then

$$16 + 24 + 8 * \text{nArgs} \quad , \tag{6.5}$$

38 bytes for an unary operation and 46 bytes for a binary operation. This is 25 or 29 bytes more than the minimal implementation for a primal value taping without index reuse.

Sacado [PBGH08] provides the reverse type *Sacado::Rad::ADvar<double>*. It performs a Jacobi taping approach for operators. An assign optimization is not implemented. The memory management of Sacado is tailored for AD. The tool overloads the new operator to avoid the scattering of memory in the global address space. Chunks of memory are allocated and the *ADvar<double>* types are placed in this memory. This has the advantage, that intermediate operators are kept in memory because the delete operation is empty. The types can store references to intermediate operators without caring about dangling pointers. The required memory for an argument is 32 bytes, the reference to the next Jacobi, the reference to the current Jacobi, the reference to the argument and the reference to the result. Each operator needs 16 bytes of memory but this includes the memory for the adjoint value, so only 8 bytes are counted. The memory for one operation is then

$$8 + 32 * \text{nArgs} \quad , \tag{6.6}$$

40 bytes for an unary operation and 72 bytes for a binary operation. Because everything is stored, the approach is equal to a linear indexing scheme. Therefore, Sacado needs more than three times the memory for an operation, than a Jacobi taping approach with linear indexing.

The **Stan Math Library** [CHB+15] is a traditional operator overloading tool. The type *stan::math::var* needs to be used for the derivative computation. The approach is primal value taping, that tapes the operators. The memory management is nearly the same as the one of Sacado, which means that memory is not scattered in the global address space. The arguments for each operator are stored as references taking 8 bytes. Each operator stores the primal value, the adjoint value (which is not counted) and a vtable that is used for the call to the adjoint function. Because everything is stored, the tool resembles a linear index management scheme. The memory for one operation is then

$$16 + 8 * \text{nArgs} \quad , \tag{6.7}$$

32 bytes for a unary operation and 40 bytes for a binary operation. For a tool that does primal value taping with a linear index scheme, it uses 19 bytes more per unary operation and 23 bytes per more binary operation. These high memory counts per operation are tried to be amended by specializing operators that use the *stan::mat::var* type as a template argument. There are several specializations for the linear algebra library *Eigen* [GJ+10] available, that e.g. reduce the operator count for a matrix-matrix multiplication from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^2)$.

ADOL-C [WG09] is probably the most known AD tool and the most mature. It has been under development for over 20 years. The *adouble* type implements a primal value taping approach for operators. The indices are reused and the tape is organized to store data on stacks. This yields no scattering of data and no dead memory due to padding. How much memory an operator in ADOL-C uses, is hard to judge. Every operator is implemented in a separate function, which enables optimal memory use for each operation, but e.g. the implementation of the sinus evaluates and stores the data for the cosine, too. The memory for each operation can be given as

$$1 + 4 + 8 + 4 * \text{nArgs} + \{\text{extra}\} \quad , \tag{6.8}$$

1 byte for the operator, 4 bytes for the left hand side index, 8 bytes for the left hand side value and 4 bytes for each right hand side argument. If the extra term would not be there, then ADOL-C would have the optimal memory implementation, developed in this thesis. ADOL-C reduces the memory footprint of the primal application. This is done by just storing the index in the *adouble* types. The primal value is accessed through the global primal value vector. Because this vector is required for the reverse interpretation, this introduces no memory overhead, but each primal value access requires one array lookup. The recording time is probably increased by this optimization.

CppAD [BB08] is similar to ADOL-C. The only difference is the index management. CppAD uses a linear index management scheme allowing for the optimization, that the left hand side index does not need to be stored. It also does not separate the indices form the primal values. The memory for one operation is then

$$1 + 8 + 4 * \text{nArgs} \quad , \tag{6.9}$$

1 byte for the operator, 8 bytes for the left hand side value and 4 bytes for each right hand side argument. The memory for an unary opration is 13 bytes and 17 bytes for a binary operation, which is the same as the optimal values for a primal value taping approach with linear indexing.

Adept [Hog14] is the only tool, that implements expression templates from the list on *www.autodiff.org*. The *aReal* type implements a Jacobi taping scheme, that reuses indices. For each argument the Jacobi and the index are stored, which are 12 bytes. The number of arguments for each statement is stored in an unsigned int of 4 bytes. Each statement can have $4 \cdot 10^9$ arguments. The memory for a statement is then

$$4 + 4 + 12 * \text{nArgs} \; . \tag{6.10}$$

With a more restrictive assumption for the maximum number of arguments up to 3 bytes per statement could be saved.

dco/C++ [LLN16] is not on the list of *www.autodiff.org*, but it is used in this thesis to differentiate TRACE. The *dco::a1s::type* implements a Jacobi taping on a statement level with a linear index management. The implementation also uses the assign optimization. The memory per statement used in this thesis has not been optimal. The Jacobi and index value are stored in 8 bytes and 4 bytes respectively which is optimal. Yet, these two values are stored together in one structure and due to the padding 16 bytes are used for the storing instead of 12 bytes. The number of arguments are stored in the same structure which are 16 bytes per statement. The memory per statement is then

$$16 + 16 * \text{nArgs} \; . \tag{6.11}$$

dco uses 4 bytes more per argument and 15 bytes more per statement.

From the total of 24 tools found at *www.autodiff.org* 7 tools plus a closed source tool have been analyzed. The focus of this analysis is on the memory, because the memory consumption is one of the main issues and defines also the lower bound for the time. A summary of the analysis can be found in Table 6.1. Only CppAD is implemented in an optimal way for its category. All other analyzed AD tools are not implemented in the most optimal way for memory efficiency. Since there is no tool, that implements expression templates in an optimal way, no one is a good choice for HPC specific applications. According to the developers, dco/C++ will be modified in the next release to provide nearly optimal values. The same modifications are presented in the next section. It is notable that no tool implements expression templates for a primal value taping approach. It is by far the most complex implementation, but should be in terms of memory better than any other technique. It should also be comparable in time requirements.

Table 6.1.: Overview of existing AD tools.

AD tool	Taping strategy	Index management	Memory per stmt/expr	Misc.
AD model builder	operator naive primal value	linear	8 ∗ inputValues + 16 ∗ nArgs + 1 + 16 + 32	supports matrix and array structures
FADBAD	operator primal value	linear	16 + 24 + 8 ∗ nArgs	scattered memory
Sacado	operator Jacobi	linear	8 + 32 * nArgs	custom memory manager
Stan Math Library	operator primal value	linear	16 + 8 * nArgs	custom memory manager
ADOL-C	operator primal value	reuse	1 + 4 + 8 + 4 * nArgs + {extra}	
CppAD	operator primal value	linear	1 + 8 + 4 * nArgs	
Adept	expression Jacobi	reuse	4 + 4 + 12 * nArgs	
dco/C++	expression Jacobi	linear	16 + 16 * nArgs	closed source, not on autodiff.org

6.12.1. AD tool improvement

The memory overhead is introduced by the padding of the structure:

```
struct Data {
  int index;
  double jacobi;
};
```

The structure requires 16 bytes instead of the 12 bytes of the members. Without a rewrite of dco/C++, the only option is to set the attribute *packed* on the data structure. The elimination of the padding reduces the memory consumption by 25%. The next memory overhead is generated by using the *Data* structure to store the number of arguments. No additional chunk could be easily added to dco/C++. A new global vector of the type *uint8_t* is introduced. It has the same size as the adjoint vector and needs to be grown during the taping process. In the reverse interpretation, the index of the left hand side can be used to access the number of arguments for each statement. This reduces the memory again by approximately 12%. The total memory reduction is 34%

Only now dco/C++ provides an optimal memory implementation and is usable for HPC applications.

6.13. CoDiPack

Nearly no AD tool from the previous section provides a minimal memory implementation that is required in order to use AD on HPC machines. Instead of creating yet another new AD tool, one of the existing tools could be modified in order to provide a minimal memory implementation and to implement new taping strategies. For most of the existing AD tools however, this would require a total rewrite. Most tools are designed for one specific taping strategy, but as we have seen, there exist several taping strategies with different properties. It is therefore more appropriate to create a new tool that can be specifically designed for the required purpose.

The Code Differentiation Package (CoDiPack) is a new operator overloading AD tool, that is specifically designed for HPC and industrial applications. The main goal of the tool is to implement the gained knowledge presented in this chapter and to provide a foundation for further AD based research. The basic goals for the development are

- memory efficiency
- minimal computation time

For both goals it is necessary to have a data layout optimized for HPC applications. The data structure commonly associated with this are structures of arrays [JR15]. A structure of arrays allows a good prefetching, good cacheline use and improved alignment. It avoids padding issues identified in some AD tools.

In order to achieve the minimal computation time, for the recording of the tape, it is important to keep in mind that operator overloading AD adds a large amount of logic to every statement. A simple mathematical statement as $c = a + b$, can introduce several hundreds of commands. If AD needs 1 ms for every statement that is evaluated, then a program with 1000 statements will take 1 second. It is therefore necessary, that the introduced logic of operator overloading AD will be simple and as efficient as possible.

The next goals consider the usability of CoDiPack. They describe two directions:

- simple software integration
- easy to use/debug

Integration into the software is a major issue when the software for the differentiation is already in place. If the software needs to be heavily modified for the AD tool then AD will not be used. If the AD tool can be adapted to the software and because of that, can be easily integrated, then AD becomes a valid option.

The ease of use is mostly a very imprecise statement. For a library it is very important that a specification for the API exist and that the API is narrow and well documented. There should be tutorials that explain in which order the API functions need to be called.

The final goals describe the long term support of the AD tool:

- easy to understand
- easy to extend

Mathematical software is usually written with a special mathematical model in mind. The outcome consists of variable names that are short and not descriptive. In an user friendly library each function and variable name should be self explanatory. After a user has understood how the whole or parts of the software work, he or she should be able to extend the software quite easily. It should be possible to make easy adjustments without the danger of breaking all other parts of the library.

These six design principles are the basic guidelines used for the development of CoDiPack. The following sections will highlight the most important design decisions in CoDiPack.

6.13.1. General Structure

The global structure of CoDiPack is defined in the diagram of Figure 6.15. The expression templates are self contained and provide static data and functionality for other classes. The *ActiveReal* defines a termination point for the expression templates and the interface for the user. The *ActiveReal* objects were designed to contain no logic. If they contained logic, then the logic would be tape specific and for each tape a different user type would need to be implemented. The *ActiveReal* is a pure data container that forwards nearly all calls to the tape implementation. The tapes are the classes that contain all logic and data for a specific AD approach. There is e.g. a Jacobi tape, a Jacobi tape with index management and primal value tapes.

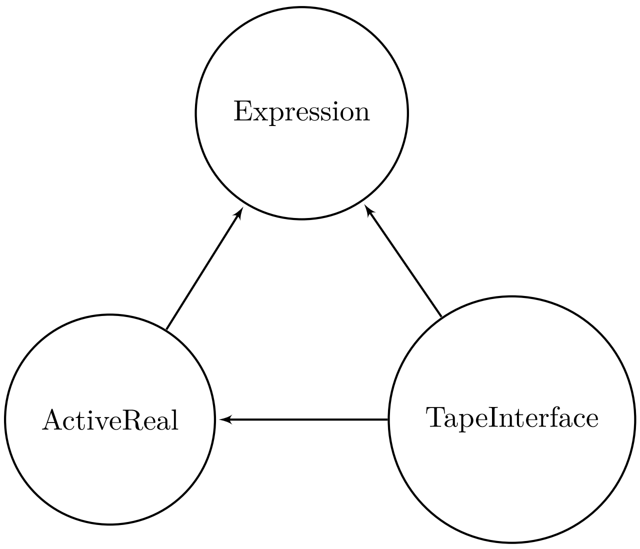


Figure 6.15.: The global structure of CoDiPack.

6.13.2. Expression implementation

The common approach for expression template implementations is the use of macros like:

```
#define CREATE_UNARY_OP(NAME, opName, primalFunc, derivFunc) \
struct NAME { \
  ... // 200 lines of code \
}; \
 \
template<typename ExprA> \
NAME<ExprA> opName(const ExprA& a) { \
  return NAME<ExprA>(a); \
}

CREATE_UNARY_OP(SIN, sin, sin(a), cos(a))
```

The user has to define the name of the structure, the name of the overloaded function, and how the primal and derivative is evaluated. The macro defines then a structure with hundreds of code lines. The problem with these macros is, that they cannot be stepped through with a debugger. Either the debugger will stop on the same line several times or will go directly to the next line. A good solution to this problem is the use of super macros [Bea04]. The idea behind a super macro is to move the generated code definition into a separate file and include the file. The problem with this technique is that an include macro does not take arguments.

The above example as a super macro would look like:

```
 1  #ifndef NAME
 2  # error Please define the name of the operator.
 3  #endif
 4  // other checks for opName, primalFunc and derivFunc
 5
 6  struct NAME {
 7    ... // 200 lines of code
 8  };
 9
10  template<typename ExprA>
11  NAME<ExprA> opName(const ExprA& a) {
12    return NAME<ExprA>(a);
13  }
14
15  #undef NAME
16  #undef opName
17  #undef primalFunc
18  #undef deriv
```

The arguments of the define are emulated in the super macro by explicitly checking for defines with the names of the arguments. Otherwise the super macro generates an error, as in line 2. These checks ensure that arguments to the super macros are not missed.

In order to be able to use the super macro multiple times, the required defines are undefined at the end of the super macro. The super macro is used by providing all necessary arguments as preprocessor defines and including the super macro file. The example for sinus implementation would look like:

```
#define NAME SIN
#define opName sin
#define primalFunc sin(a)
#define derivFunc cos(a)
#include "unaryTemplate.tpp"
```

This requires more lines than the call of the preprocessor macro in the multi line example but has the advantage, that the user can step through the macro in a well defined manner and can see all separate steps. This is a big advantage for the usability and for users that try to understand how CoDiPack works.

6.13.3. Tape interfaces

C++ provides interfaces via pure virtual classes. If, in the context of AD, the pure virtual class is used, the performance of the AD tool will be quite bad. Multiple virtual function calls would be evaluated per statement and the arguments of template functions would also need to be implemented as virtual classes, adding more virtual function calls per statement. AD tools like Adept and dco/C++ provide only the direct implementations of the tapes. The user is then confronted with implementation details of the tapes and can not see, which functions need to be implemented for a new tape and which functions are tape specific.

```
template <typename Real, typename GradientDataType, typename
    GradientValueType>
struct TapeInterface {

  typedef GradientDataType GradData;
  typedef GradientValueType GradValue;

  template<typename Rhs>
  void store(Real& lhsValue, GradData& lhsGradData, const Rhs& rhs);

  template<typename Data>
  void pushJacobi(Data& data, const Real& value, const GradData&
      gradientData);

  template<typename Data>
  void pushJacobi(Data& data, const Real& jacobi, const Real& value, const
      GradData& gradientData);

  virtual void initGradData(Real& value, GradData& gradientData) = 0;
  virtual void destroyGradData(Real& value, GradData& gradientData) = 0;

  virtual bool isGradientTotalZero(const GradData& gradientData) = 0;
  virtual void setGradient(GradData& value, const GradValue& grad) = 0;
  virtual GradValue getGradient(const GradData& value) const = 0;
  virtual GradValue& gradient(GradData& value) = 0;

  virtual bool isActive(const GradData& value) const = 0;

};
```

Figure 6.16.: The interface for a CoDiPack tape. Documentation is removed for brevity. The interface defines all functions, that are required for a new tape implementation.

CoDiPack provides a virtual class for the tape interface defining virtual functions as well as template functions. The definition can be seen in Figure 6.16. All tape implementations extend from this class. The user is able to see which functions are tape specific and which functions are part of the interface. For all definitions inside of CoDiPack, the tape interface is never used. This is a development overhead, but gives the user a distinct point where all information for a new tape is available.

For a new tape implementation the user must copy the interface file and implement all functions. In combination with the *ActiveReal* the new tape can be used without any other implementations.

6.13.4. Tape memory efficiency

The most common problems with memory are padding related or that the wrong data structures are used. Both problems can be avoided by using structs of arrays for the data. That is, the structure

```
struct Data {
  int index;
  double jacobi;
};
```

requires padding. The size of the data is 16 bytes where the size of the members is 12 bytes. The structure

```
struct DataArray {
  int* index;
  double* jacobi;
}
```

requires no padding.

The use case for AD tapes requires multiple data streams with different access frequencies. For a Jacobi tape this would be the data streams *Jacobi* (double), *index* (int) and *numberOfArguments* (uint_8). The streams *Jacobi* and *index* would have the same access frequency since one entry is stored for each argument of a statement. The *numberOfArguments* stream is accessed less often because only one value per statement is stored. The simplest option to store these streams is the allocation of data chunks. Every time a chunk is full a new one is allocated. Because of the different access frequencies, the chunks for different data streams will run out of sync. It is crucial for the reverse interpretation to access the chunks in the correct order with the correct interpretation bounds.

The implementation in CoDiPack uses two structures for the management. The data is managed in the *Chunks* structure. The basic interface from Figure 6.17 allows for re-sizing the data and check how much space is left. Specialized implementations define chunks with one, two, three, etc. data entries. For the streams of the Jacobi tape, the chunks would be *Chunk2<double, int>* and *Chunk1<uint_8>* for the arguments and the number of arguments respectively. Because the *Jacobi* and *index* streams are accessed always at the same time, they can be stored in one chunk.

The management of the chunks is done by the *ChunkVector* structure. Figure 6.18 shows the interface and consists of the functions for storing and retrieving data, defined in lines 22 to 28. This structure is a wrapper for a standard vector of chunks. The important difference is its recursive nature. Each *ChunkVector* has a child vector, that is integrated into the behaviour of the parent vector. If a position is required, then the position of the parent and the child is returned. This can be seen in the definition of the position structure in Figure 6.18 lines 6 to 11. Every time the position of the parent vector is modified the position of the child vector is modified accordingly. This has two advantages for the user. If he or she wants to execute an operation, like a reset on all vectors, then the operation only needs to be called on the root vector. The second advantage comes into play, when the user wants to add a new data stream. The user only needs to add the new stream to the hierarchy and everything else is handled by the library.

Figure 6.19 shows how the different access frequencies are handled by the nested

```
struct ChunkInterface {

  size_t size;
  size_t usedSize;

  ChunkInterface(const size_t& size);

  size_t getSize() const;
  size_t getUsedSize() const;
  size_t getUnusedSize() const;
  void setUsedSize(const size_t& usage);

  void reset();

  template<typename ... Data>
  void setDataAndMove(const Data&... values);

  template<typename ... Data>
  void dataPointer(const size_t& index, Data*... &pointer);
};
```

Figure 6.17.: The interface for data chunks in CoDiPack. Each chunk can have multiple data arrays.

structures. Each lane shows one data stream, that is stored in a *ChunkVector*. The lowest lane is the root vector and the highest lane is the terminating vector. The blue bars on the lanes show the boundaries between chunks of one data stream. If the AD tool needs now to interpret over all the data items from position $\mathfrak{p}_a$ to $\mathfrak{p}_b$, then there are 4 chunk boundaries in between. At these positions the interpretation needs to be stopped, the next chunk loaded and then the interpretation can be continued.

6.13.5. Tape modularization

The goal for CoDiPack is to implement all possible combinations of AD taping strategies. Primal vs. Jacobi taping and linear indexing vs. index reuse. It is required, that the implementations provide a fast mode for advanced users without any bound checking and with preallocation of the memory. The default mode should perform the bound checks and have an on the fly memory allocation. This gives eight different tape implementations, that differ in certain areas and are the same in other areas. Usually the common logic can be extracted in a base class and all other classes can extend from this base class. The recursive structure of the data management and the use of many templates in CoDiPack, makes this approach very complex from an implementation standpoint.

The chosen implementation uses super macros like the expression template implementation, to define each module in a separate file. The tape implementations can then include different modules. This makes the tape implementations less readable but simplifies the definition of template types.

```
template<typename ChunkData, typename NestedVector = EmptyChunkVector>
struct ChunkVectorInterface {
  typedef typename NestedVector::Position NestedPosition;
  typedef ChunkData ChunkType;

  struct Position {
    size_t chunk;
    size_t data;

    NestedPosition inner;
  };

  ChunkVectorInterface(const size_t& chunkSize, NestedVector& nested);

  void resize(const size_t& totalSize);

  void reset(const Position& pos);
  void reset();

  Position getPosition() const;

  void reserveItems(const size_t items) const;

  template<typename ... Data>
  void setDataAndMove(const Data& ... data);

  template<typename ... Pointers>
  void getDataAtPosition(const size_t& chunkIndex, const size_t&
      dataPos, Pointers* &... pointers);

  template<typename FunctionObject, typename ... Pointers>
  void forEach(const Position& start, const Position& end,
      FunctionObject& function, Pointers* &... pointers);
};
```

Figure 6.18.: The interface for the chunk vectors in CoDiPack. They manage the chunks and build a recursive structure.

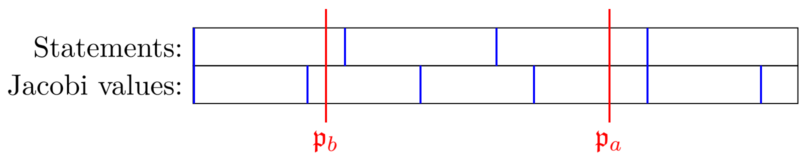


Figure 6.19.: Example for the data stream of a Jacobi tape implementation. The blue bars show the chunk boundaries and the marks $\mathfrak{p}_a$ and $\mathfrak{p}_b$ a possible region for the interpretation.

6.13.6. Customization of CoDiPack

It is quite common to enable and disable some logic in a library by preprocessor macros. A good example is the check in CoDiPack for a zero Jacobi entry. If the Jacobi is zero, then the Jacobi should not be pushed on the tape. If this functionality needs to be disabled the usual implementation would look:

```
#if CODI_ENABLE_JACOBI_ZERO_CHECK
  if(jacobi != 0.0) {
#endif
    tape.pushJacobi(jacobi);
#if CODI_ENABLE_JACOBI_ZERO_CHECK
  }
#endif
```

The preprocessor macros make the code very hard to understand and the indention of the code between the two macros is not clear. The same effect can be achieved if a constant boolean is added to the if statement. The new code is then:

```
static const bool CODI_EnableJacobiZeroCheck = true;
...
if(!CODI_EnableJacobiZeroCheck || jacobi != 0.0) {
  tape.pushJacobi(jacobi);
}
```

The variable *CODI_EnableJacobiZeroCheck* can force the if to be true. If *CODI_EnableJacobiZeroCheck* is true, then the first element of the if is false and the second element is checked. If *CODI_EnableJacobiZeroCheck* is false, then the first element of the if is always true and the second element does not need to be checked. The compiler can optimize this code such that the *if* is completely eliminated or the constant true is never evaluated.

The readability of the code has improved but the if statement is now quite hard to understand. With the introduction of a macro, the readability can be further improved:

```
static const bool CODI_EnableJacobiZeroCheck = true;
#define ENABLE_CHECK(option, condition) if(!(option) || (condition))
...
ENABLE_CHECK(CODI_EnableJacobiZeroCheck, jacobi != 0.0) {
  tape.pushJacobi(jacobi);
}
```

It is now clear that the check is only enabled if the option is true. The advantage of the approach is, that it reads like a normal if and the code structure is still intact. If several of these checks are required in a code block, then the macro approach with *#if* and *#endif* becomes in most cases unreadable.

This is a short overview of the main design decisions in CoDiPack. The implementational details will be covered in additional papers.

6.14. Conclusions on AD implementations

Table 6.2 shows the summary of the different implementations while Figure 6.20 shows the memory for the different implementations and optimizations for the *l2_norm*. The figure displays the combined memory for the tape, the adjoint vector and, if necessary, the primal vector. It can be seen that there is a jump from the operator taping to statement level taping. Reuse of the indices yields only small improvements with respect to the memory. In bigger applications where several different functions are called, the effect is more substantial since the size of the adjoint vector is greatly reduced. The figure states that the improved primal value taping approach uses less memory than the Jacobi taping approach. This can change if advanced optimization techniques are used. The primal value taping needs more memory if it is to handle constant or passive values. This can diminish the memory advantage. The Jacobi taping has the advantage of an easy implementation and most of the computations are performed when the tape is recorded. The interpretation of the tape consists only of a multiplication and update operation that makes the implementation very general and thereby easy to maintain. Advanced techniques like index reuse and passive values are also implemented in an easy way, yielding a definite reduction in memory.

The Jacobi taping is best applied to legacy codes that have not been designed for AD. The user can use AD in such a setting without knowing the details of the implementation and without any restrictions on the functions that can be used e.g. *memcpy*. If a code is specifically designed for AD then the use of the primal value taping is more favorable. Implementation guidelines for the functions can be written in such a way that the required memory will be optimal.

Here some general guidelines are presented minimizing the memory used by different implementations.

- General guidelines for AD:
  - Make statements as large as possible
  - Reduce the number of intermediate variables
- Guidelines for the primal value taping:
  - Define constant values as global variables
  - Do not use passive values
- The guidelines for the index reuse are:
  - Do not use C-like memory operations like *malloc*, *free*, *memcopy*, etc.
  - Avoid assignments of variables e.g. $c = a$;

Remark 6.8 (Code optimizations in operator overloading AD tools)
For operator overloading tools it is very hard to break out of the limitations of the

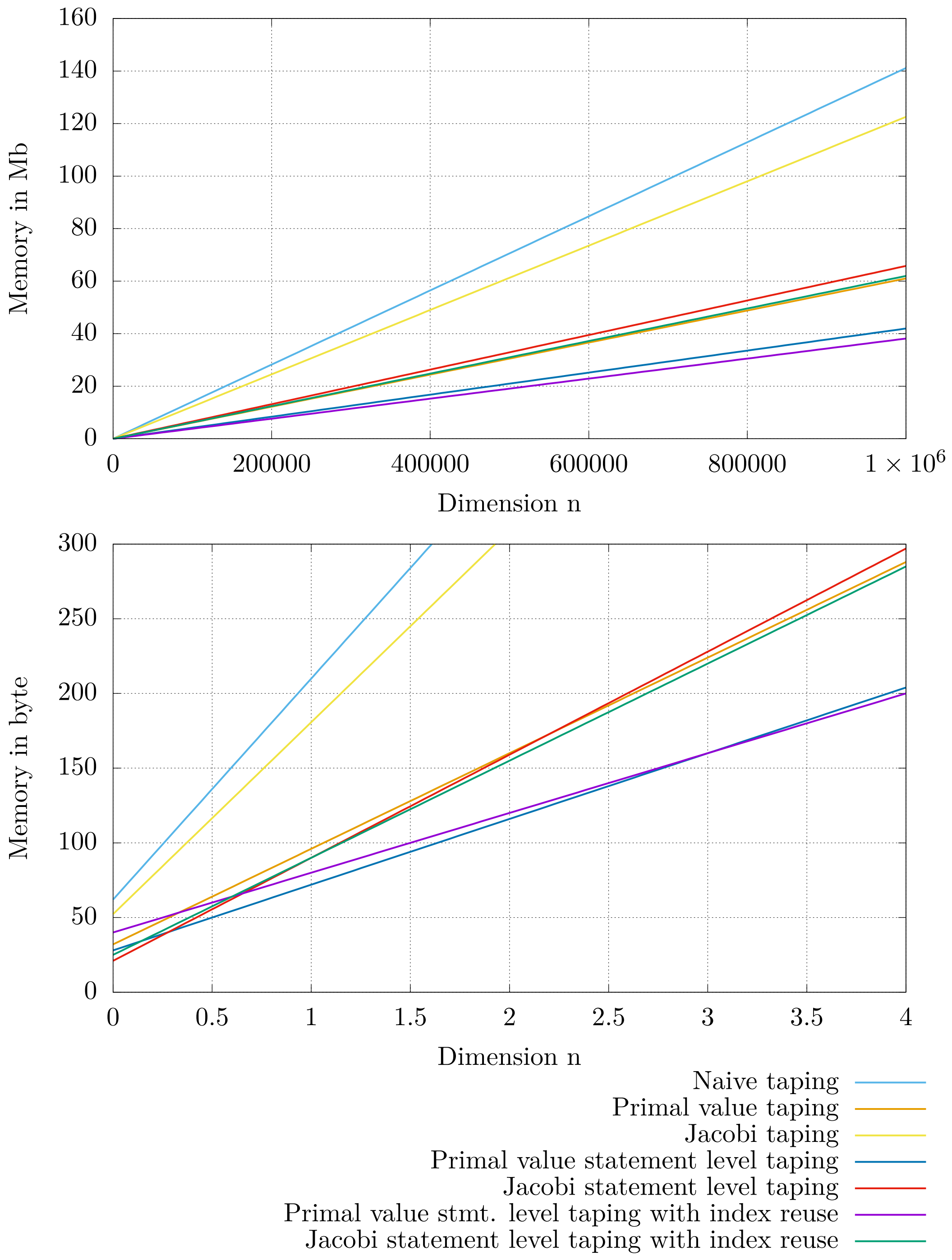


Figure 6.20.: Comparisons of different AD implementations and optimizations for the *l2_norm* example. The top graph shows the memory consumption for large vectors. The lower graph shows the same for small vectors and illustrates the crossovers between the different implementations.

language. Every technique presented in this chapter can only “see” one statement at a time and therefore all optimizations occur at the statement level. Some techniques require each statement to contain as much information as possible. If the code is written so that one operation is performed per statement, then the operator overloading tool will require much more memory. The operator overloading approach has no means to break this restriction with pure language tools. It would need to gain access to the abstract syntax tree (AST) and modify this tree in order to optimize the tape. An other option would be that the compiler knows about the operator overloading tool and will optimize the code so the required memory for the tape is minimized.

Table 6.2.: Comparison of different AD implementations and possible optimizations for them.

	Primal value	Primal value index reuse	Jacobi	Jacobi index reuse
Unary operator memory (byte)	13	17	$12 + \frac{1}{8}$	$16 + \frac{1}{8}$
Binary operator memory (byte)	17	21	$24 + \frac{1}{8}$	$28 + \frac{1}{8}$
Statement memory (byte, n arguments)	$16 + 4 * n$	$20 + 4 * n$	$1 + 12 * n$	$5 + 12 * n$
Memory for each generated index (byte)	8	16	8	8
C like memory operations (e.g memcpy)	+	-	+	-
Constant values (byte)	8	8	0	0
Passive values (byte)	9 or 16	13 or 20	-12	-12
Tape deactivation	+	-	+	+
Register input creates a statement	+	-	+	-
Assignment creates a statement	-	+	-	+
Linear optimization	already included		+	+
Multiple reverse sweeps	+	+ (incr. mem.)	+	+
Argument optimization	-	-	+	+

7. Numerical results

Global memory reduction techniques are introduced were successfully applied to the differentiated TRACE code. This can reduce memory and time requirements but the results have been mediocre. Additional improvements can be reached through the analysis of available AD tools and a comparison for memory efficiency. These steps are now combined and the performance of the solution analyzed.

The results are presented for the CRESENDO test case. On this case the two derivative methods Black Box and Reverse Accumulation are compared on a performance level and how the optimizations affect their respective performance. The two AD tools dco/C++ and ADOL-C are compared on the CRESENDO test case, too.

For the THD test case, the convergence properties of the two derivative algorithms are analyzed more closely.

7.1. CRESENDO test case

The timing and memory results for the CRESENDO case are presented in Table 7.1. The table shows the time factors for the Black Box and Reverse Accumulation algorithms. For the Black Box algorithm the time is dominated by the recording and evaluation of the tape. The time for the Reverse Accumulation algorithm mainly determined by the time for the evaluation of the tape. All time and memory consumption is measured for one iteration of the fixed point solver G. The measurements for the CRESENDO case are done with dco/C++. A comparison between dco/C++ and ADOL-C is discussed separately.

The values in Table 7.1 show that the memory and time requirements for G without optimizations are quite high. The introduction of function checkpoints reduces the required memory by 50%. Especially the computation of the Euler flux and the setup of the implicit matrix produces the greatest reduction in memory, but the time for the evaluation increases for both methods. The impact on the Black Box method is less severe because the recording of the function checkpoints is shifted to the evaluation of the tape. The impact on the Reverse Accumulation procedure is more severe. Here, during the evaluation of the tape, the function checkpoints must be recorded, slowing the process down.

The picture is reversed for the preaccumulation. Here the full Jacobi matrix is evaluated for the augmented code parts. The time for the recording of the tape increases, but through the reduction of the number of statements and Jacobi entries a faster reverse evaluation of the tape is achieved. The time factor for the Reverse Accumulation process is reduced by two counts and is 33% faster than the algorithm without any memory reduction.

Table 7.1.: Time and memory results for the CRESENDO test case. The values correspond to one evaluation of G and show the computation of $\frac{\partial G}{\partial y}^T d$ with an arbitrary $d \in Y$.'x=yes' states that additionally the sensitivities with respect to x are calculated. 'FC' is the short notation for function checkpoint and 'PreAcc' stands for preaccumulation.

	Black Box (time factor)	Rev. Acc. (time factor)	Memory in GB	Memory-factor
primal	–	–	12.85	1.00
dco x=yes start version (estim.)	–	–	617.77	48.07
dco x=yes	24.18	7.88	417.00	32.45
dco x=yes FC calcEulerFluxSides	25.58	11.45	316.00	24.59
dco x=yes FC impMatrix	26.55	11.98	301.00	23.42
dco x=yes FC ilu_decomp	25.63	9.93	379.00	29.49
dco x=yes FC subforward_backward	24.68	8.59	415.58	32.34
dco x=yes FC all 4	32.17	18.68	201.45	15.68
dco x=yes PreAcc calcEulerFluxSides	31.77	6.65	331.73	25.82
dco x=yes PreAcc impMatrix	63.44	7.03	359.78	28.00
dco x=yes PreAcc phi	29.66	7.85	413.36	32.17
dco x=yes PreAcc all 3	76.98	5.76	272.21	21.18
dco x=yes PreAcc all full	81.99	5.37	263.16	20.48
dco x=no	19.64	6.10	349.31	27.18
dco x=no FC calcEulerFluxSides	20.89	9.39	263.43	20.50
dco x=no FC impMatrix	21.97	9.93	246.58	19.19
dco x=no FC ilu_decomp	21.12	8.11	311.43	24.24
dco x=no FC subforward_backward	20.15	6.80	348.02	27.08
dco x=no FC all 4	27.62	16.45	164.62	12.81
dco x=no PreAcc calcEulerFluxSides	24.36	5.04	274.34	21.35
dco x=no PreAcc impMatrix	47.47	5.24	290.48	22.61
dco x=no PreAcc phi	22.48	6.08	345.51	26.89
dco x=no PreAcc all 3	54.93	4.15	213.37	16.60
dco x=no PreAcc all full	56.41	4.11	209.75	16.32

The typical configuration for the Reverse Accumulation process is that the control x is not taped and preaccumulation is enabled for all possible functions.

The first line in the figure is an estimation on how much memory is required if the original version of dco/C++ would still be used. Due to a careful study of the memory structures in Chapter 6 and the data layout change for dco/C++ from Section 6.12.1, the required memory is reduced by 36%.

The Black Box algorithm has no favored configuration that would result from Figure 7.1. The activation of the function checkpoints reduces the memory but increases the run time. Here, the results in Table 7.2 for the block checkpointing from Section 5.6 are promising. Table 7.2 puts the default options into relation with the results from the block checkpoints.

Because the block checkpoints perform the differentiation for each block separately, the total required memory depends on how the blocks are distributed on the processes.

Table 7.2.: Block checkpointing memory results for the CRESENDO test case. The values correspond to one evaluation of G and show the computation of $\frac{\partial G}{\partial y}^T d$ with an arbitrary $d \in Y$.'x=yes' states that additionally the sensitivities with respect to x are calculated.

	32 Processes	16 Processes
dco x=yes	417.00 Gb	417.00 Gb
dco x=no	349.31 Gb	349.31 Gb
dco x=yes Block CP	201.84 Gb	120.19 Gb
dco x=no Block CP	178.93 Gb	107.37 Gb

If each process has only one block, there would be no memory reduction. The more blocks a process computes, the greater the memory reduction will be. This can be seen very nicely with the jump from 32 process to 16 processes. Each process has to handle more blocks and therefore an additional 40% of memory can be saved. The run with the 16 processes would take twice the time but illustrates the dependence of the total memory with respect to the block distribution on the processors.

The time for the block checkpoints on the 32 processes is nearly the same as for the default configuration of the Black Box algorithm because intermediate checkpoints are used. The time increases slightly since the intermediate checkpoint has to be loaded.

The comparison of ADOL-C and dco/C++ needed to be done on a different machine of the Elwetritsch cluster. The times differ from the results in Table 7.1. The big memory node has a slower processor and the memory bandwidth is shared by more cores so the overall performance of memory limited applications is reduced. In Table 7.3 it can be seen that the memory requirement of ADOL-C is larger than the one of dco/C++. This arises from the different taping strategies. ADOL-C uses a primal value taping strategy based on operator taping and reuses indices. In contrast, dco/C++ uses a Jacobi taping approach that is based on expression templates and a linear indexing scheme. The example of the l2 norm in Figure 6.20 on page 171 shows that the primal value taping should have a slight advantage over the Jacobi taping approach. This is just an example to gain intuition as to how the different approaches behave. The real world comparison on TRACE shows that expressions can get larger and can save additional memory under the expression template approach. An additional advantage of dco/C++ is the treatment of passive values. They can be ignored for Jacobi tapes. ADOL-C has no implementation for passive values because all variables have an index.

The large time factor of the ADOL-C Black Box algorithms and Reverse Accumulation algorithms can not only be explained by the larger memory. If the memory bandwidth were the restraining factor, then the time of ADOL-C should only be two times the one of dco/C++. The other advanced techniques, function checkpoints and preaccumulation, can not be implemented for ADOL-C and are therefore not

Table 7.3.: Time and memory results for the CRESENDO test case for AD tools dco/C++ and ADOL-C. The values correspond to one evaluation of G and show the computation of $\frac{\partial G}{\partial y}^T d$ with an arbitrary $d \in Y$.'x=yes' states that additionally the sensitivities with respect to x are calculated. Computation on the big memory node of the Elwetrisch cluster.

	Black Box (time factor)	Rev. Acc. (time factor)	Memory in GB	Memory-factor
primal	–	–	12.85	1.00
dco x=yes	38.55	8.83	415.03	32.34
ADOL-C x=yes	166.31	54.16	754.90	58.83
dco x=no	24.23	6.83	346.87	27.03
ADOL-C x=no	143.75	46.36	487.32	37.98

compared. The function checkpoints in TRACE require an AD tool that can switch off the recording for parts of the code. Preaccumulation requires that parts of the tape be evaluated and that the tape be partly reset, both cases are not supported by ADOL-C.

The convergence history for the Reverse Accumulation and Black Box procedures are shown in Figures 7.1 and 7.2 respectively. The Reverse Accumulation residuals show the property from Section 3.2, that the Reverse Accumulation process inherits the convergence rate of the primal fixed-point iteration. The residuals for the Black Box algorithm are plotted "reversed" because the algorithm starts at iteration 2000 and goes back in pseudo time until it reaches time step 0. It is interesting to note that for both residuals the convergence rate is the same as for the primal after a small initial plateau of 300 steps (2000 - 17000). This property can not be derived directly, as for the Reverse Accumulation procedure, but a similar argument would give the same intuition.

It is also interesting to see the jump in the adjoint residual in step 300. At this point the primal changes from a robust initial period to the regular formulation.

7.2. THD test case

On the THD test case the properties of the Black Box (BB) and Reverse Accumulation (RA) algorithms as well as a comparison between the two is discussed.

The convergence history of the BB method is shown in Figure 7.3. After an initial build up phase, the convergence rate from the primal is also seen for the BB method. The rise in the residuals at step 2500 and 1500 do not change any significant properties of the solution either. The high rise from step 500 to 0 changes no major properties of the solution. The BB solution at step 0 is shown in Figure 7.4. It is the base line solution and considered to be the exact solution for all further comparisons. The difference for the solution at 0 and 500 can be seen in Figure 7.5. The top picture

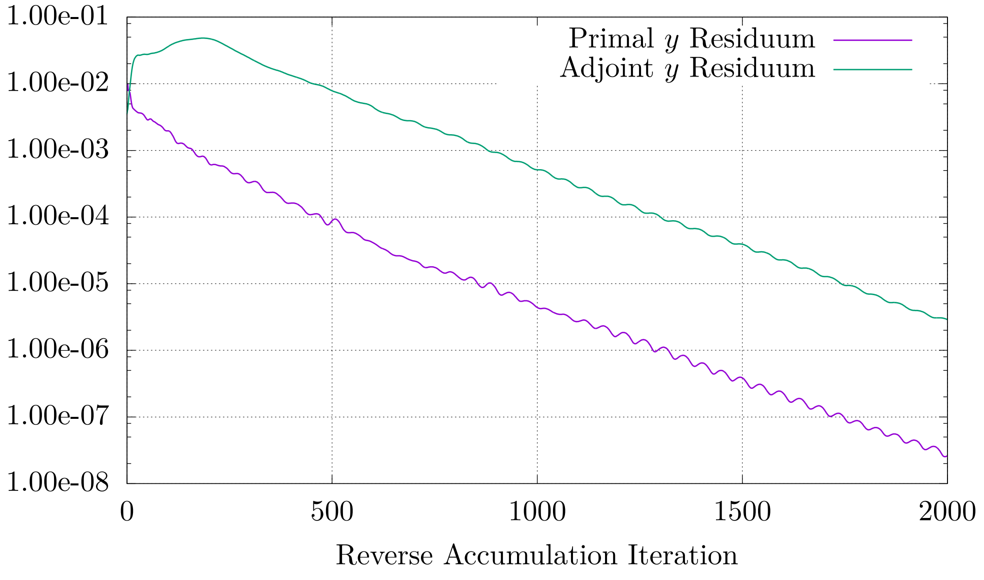


Figure 7.1.: CRESENDO Reverse Accumulation convergence history.

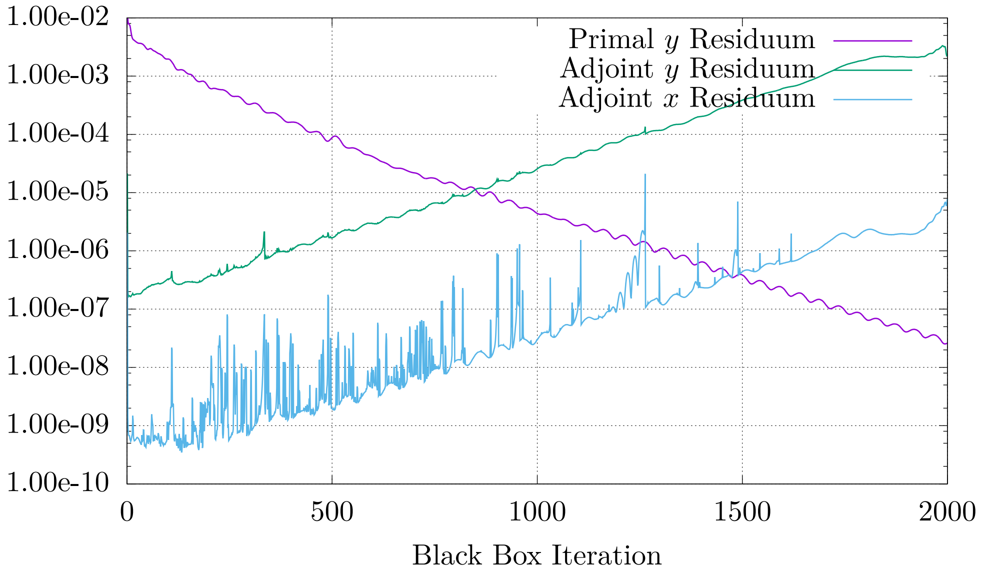


Figure 7.2.: CRESENDO Black Box convergence history.

in the figure displays the angle between the two solutions. It is calculated with the equation

$$\omega_i = \arccos(\frac{\langle \bar{x}_i^0, \bar{x}_i^{500} \rangle}{\|\bar{x}_i^0\| \, \|\bar{x}_i^{500}\|}$$

where $\bar{x}^j$ indicates the solution at the j-th pseudo time step and i runs over all nodes of the mesh. At the base of the rotor and at a few points of the stator do the directions of the solution vary strongly. The lower picture shows the length of the difference between the two solutions in relation to the base solution. The formulation that is used is

$$r_i = \frac{\|\bar{x}_i^0 - \bar{x}_i^{500}\|}{\|\bar{x}_i^0\|} \; .$$

It shows that there is a difference in the length not only where the directions differ in the two solutions. There are additional areas which have a different length scale of the direction. Especially on the suction side of the stator greater developments are still seen. But these changes are not so important since the angles of the directions are already well defined.

If the same analysis is performed for the combinations $(0, 1500)$, $(0, 2500)$ and $(0, 3500)$ the result is that from iteration 2500 on the gradient is well defined and only in some regions do larger discrepancies exist. The data for the comparison of iteration 0 and iteration 2500 is shown in Figure 7.6.

Now it is analyzed how well does the BB method behave if the solution process is stopped earlier. Figure 7.7 shows the convergence behavior of the BB method for all possible starting points from 100 to 4200 with a step size of 100. The BB process is executed as if the primal iteration would have stopped after e.g. step 3400. The figure shows only the L1 residual for the the control x but all other residuals look the same. It is interesting to note that the convergence behavior for all starts look nearly the same. If the results from all the different start locations are compared, then the results converge to the BB solution that is started at the convergence point of the primal solution. From a starting point of 3000 on, the direction is for most parts well defined. The scaling of the solution is still developing. The data is shown in Figure 7.8. Even with a starting point at iteration 4000 there are still some differences especially at the edge of the rotor.

The RA method shows the same convergence behavior as the primal solution as it is seen in Figure 7.3. The comparison of the RA solution with the BB method in Figure 7.9 shows that the direction of the RA solution is established but even with a fully converged RA method there are still some discrepancies. The scaling on the pressure side of the stator is for example off by a factor of 5.

The direction of the solution, if the RA process is terminated after 1500 iterations, is quite good. The scaling of the solution is not yet established, as it is shown in Figure 7.10. After 2500 steps everything can be considered as converged.

In the next step the same analysis for the RA method as for the BB method is performed. It is started as if the primal would have converged after e.g. 3500 steps. The results of all runs are shown in Figure 7.11. Only the runs with starting points at

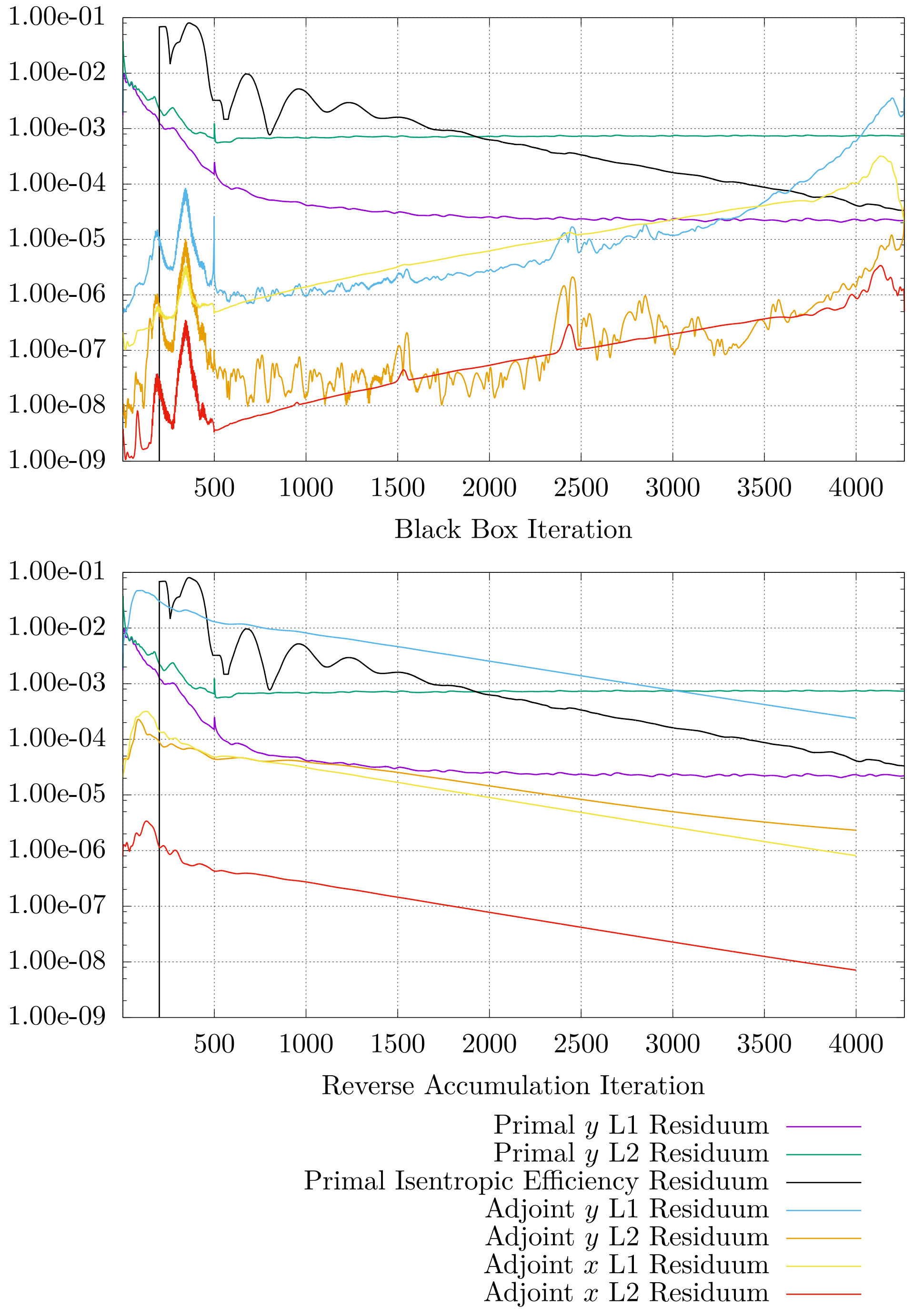


Figure 7.3.: THD Black Box and Reverse Accumulation convergence histories.

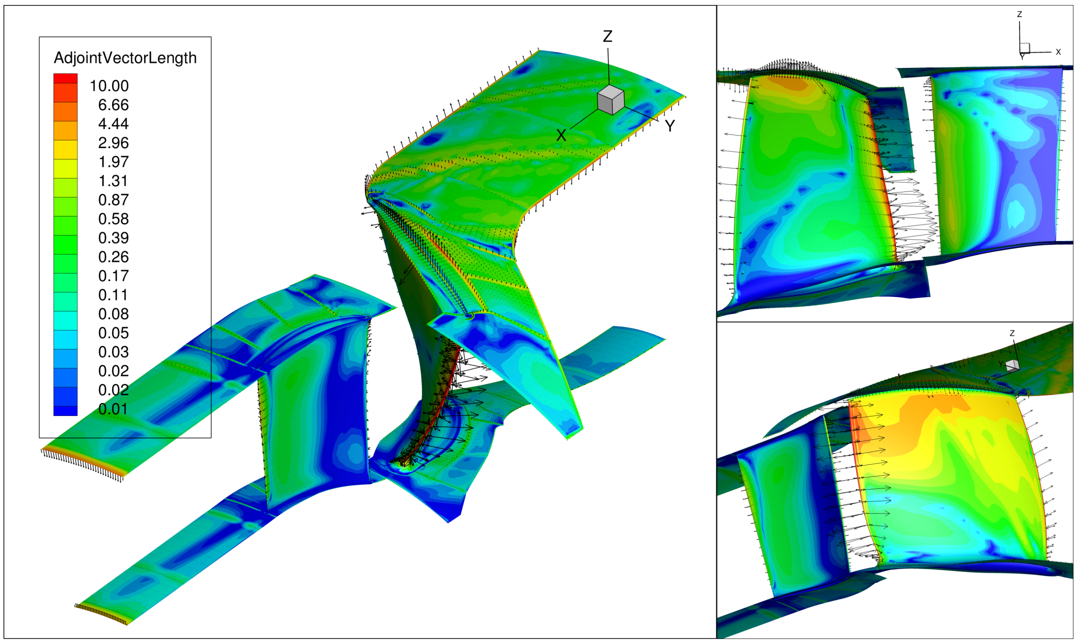


Figure 7.4.: THD Black Box solution for the control sensitivities.

1100, “2500”, 3400 and 4200 are converged. They are shown in Figure 7.12. It shows, that the RA method is only stable if the residual of the primal is small enough as it is developed in the theory of Section 3.2. The few points where the RA algorithms converge, can happen, when the state of the fixed-point solver G has no eigenvectors that disturb the result. The solutions for the 3400 and 4200 starting point look good. The solution differences for the 3400 start point is shown in Figure 7.13

From the analysis of the data it can be said that the Black Box method always yields a numerically correct result. The quality of the result depends on the quality of the primal solution. It is possible to terminate the BB process before it has reversed the entire trajectory, but as always, this would introduce an error in the derivative data.

The results for the RA process show that the primal solution needs to be converged sufficient enough. Otherwise, the convergence of the RA method can not be guaranteed. If the primal solution converges sufficiently then the RA method provides results that are comparable to the BB method.

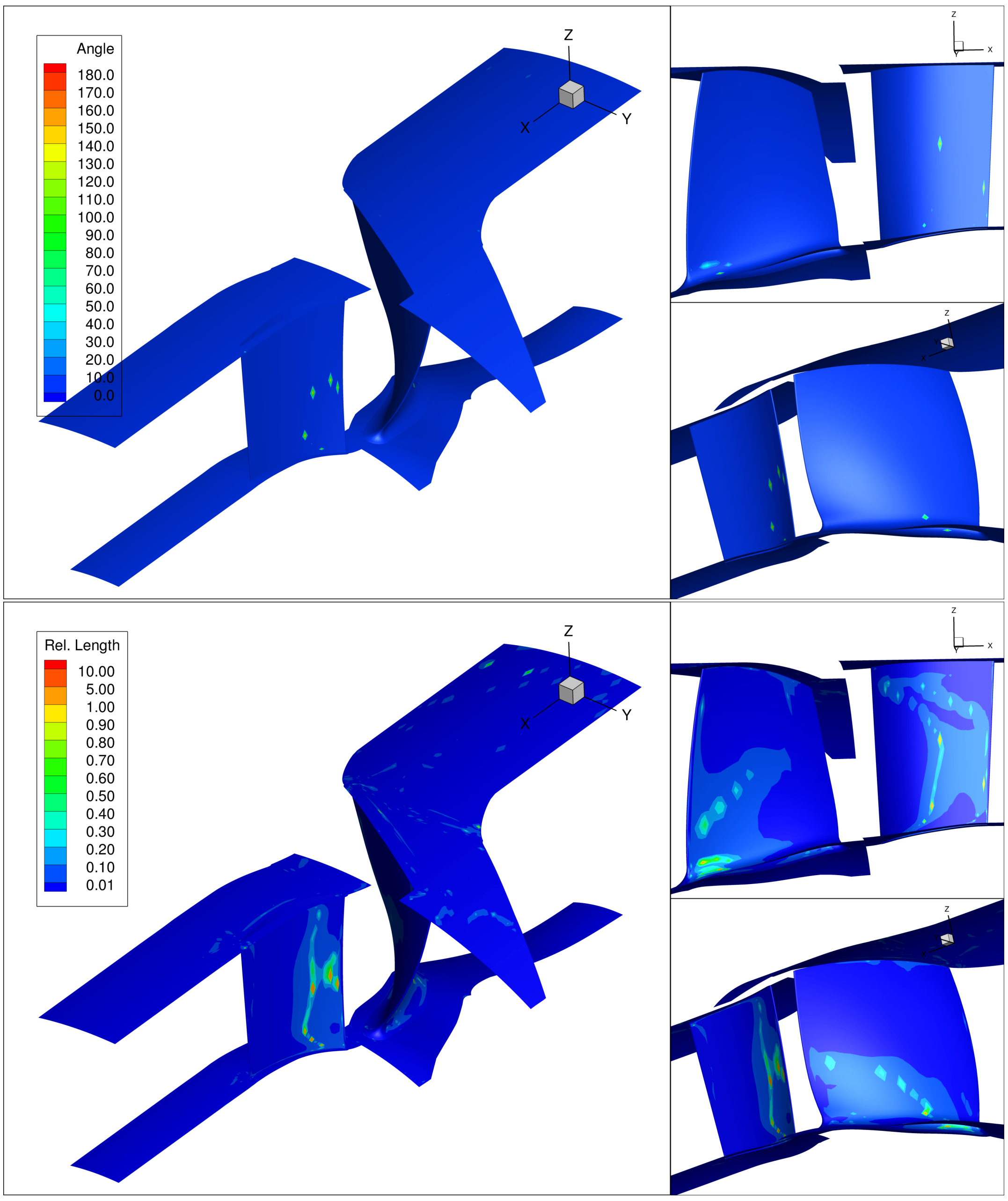


Figure 7.5.: THD difference of the Black Box solution at point 0 and 500.

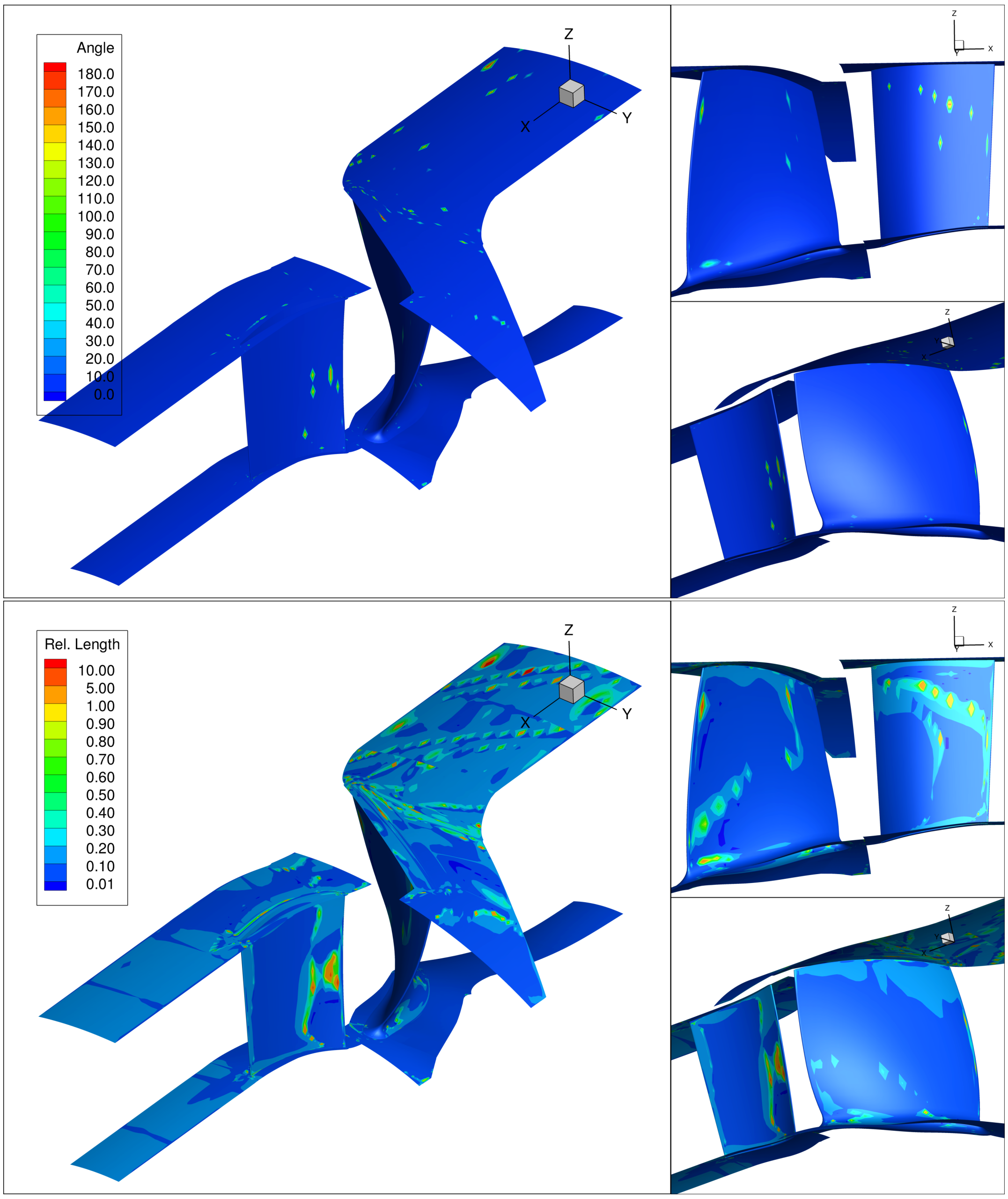


Figure 7.6.: THD difference of the Black Box solution at point 0 and 2500.

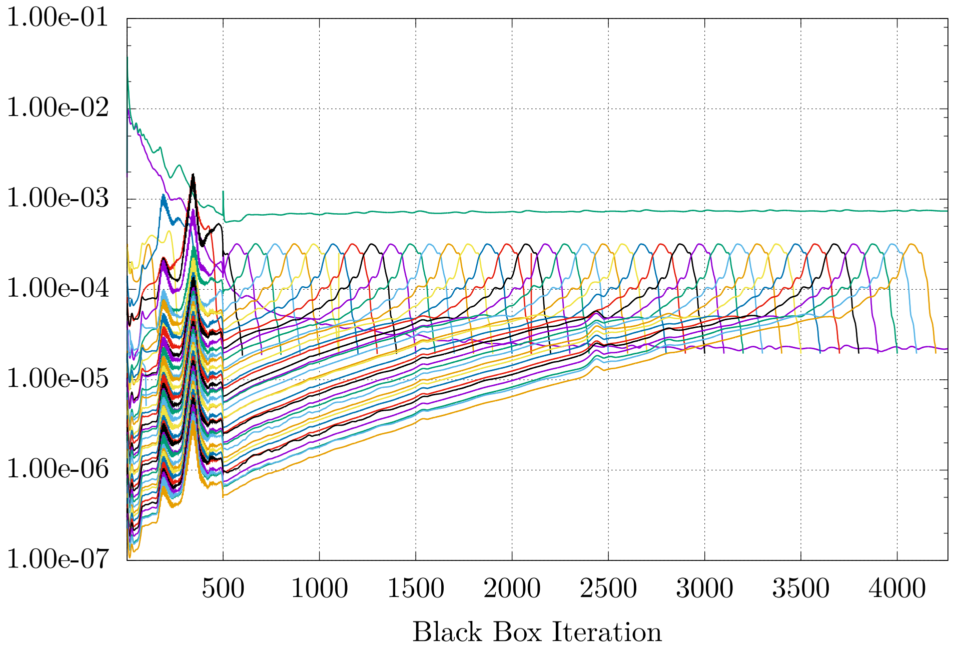


Figure 7.7.: THD Black Box convergence histories for different starting points. The L1 residuals with respect to x is shown.

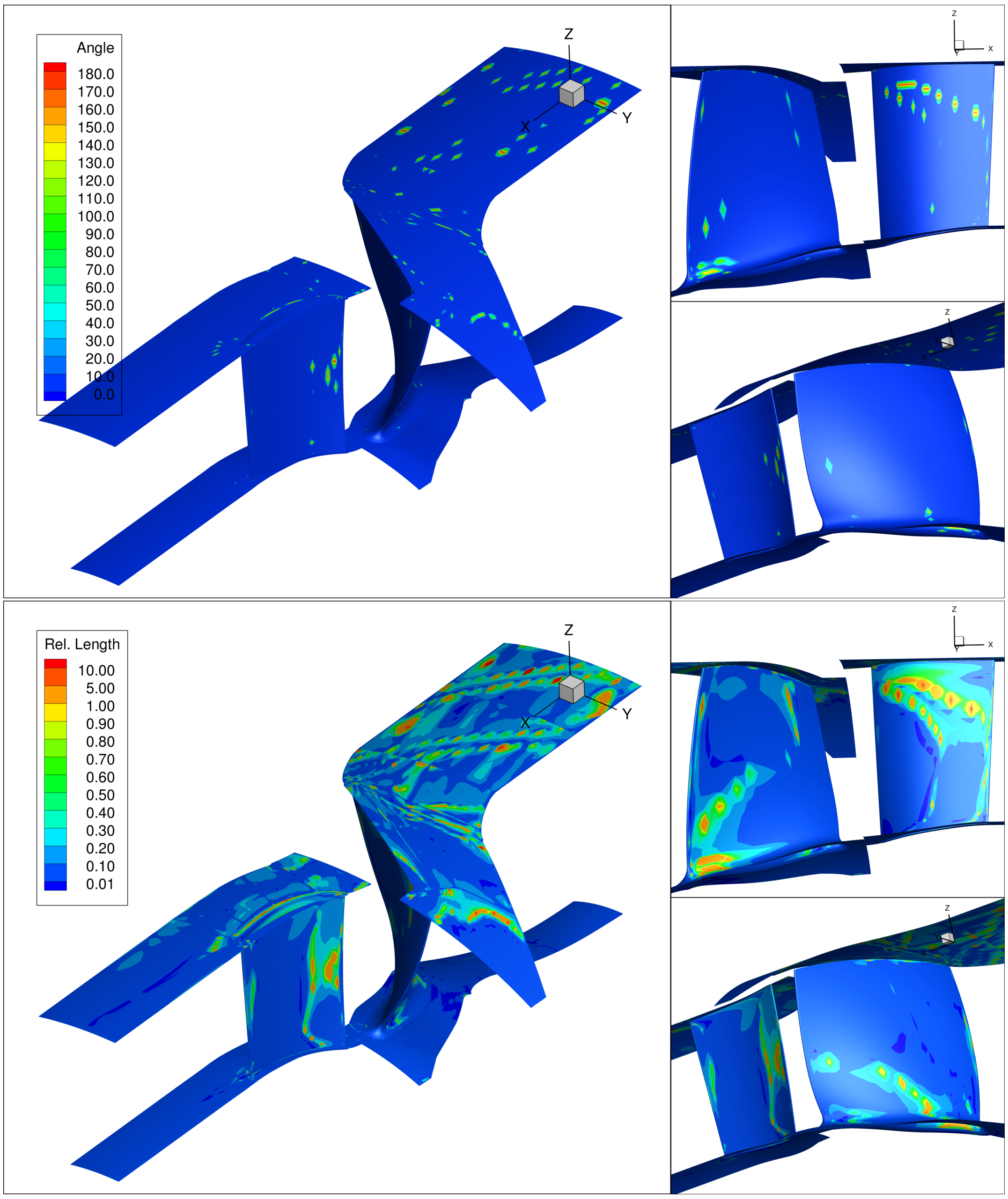


Figure 7.8.: THD difference of the base Black Box solution and the Black Box solution that is started at the primal iteration 3000.

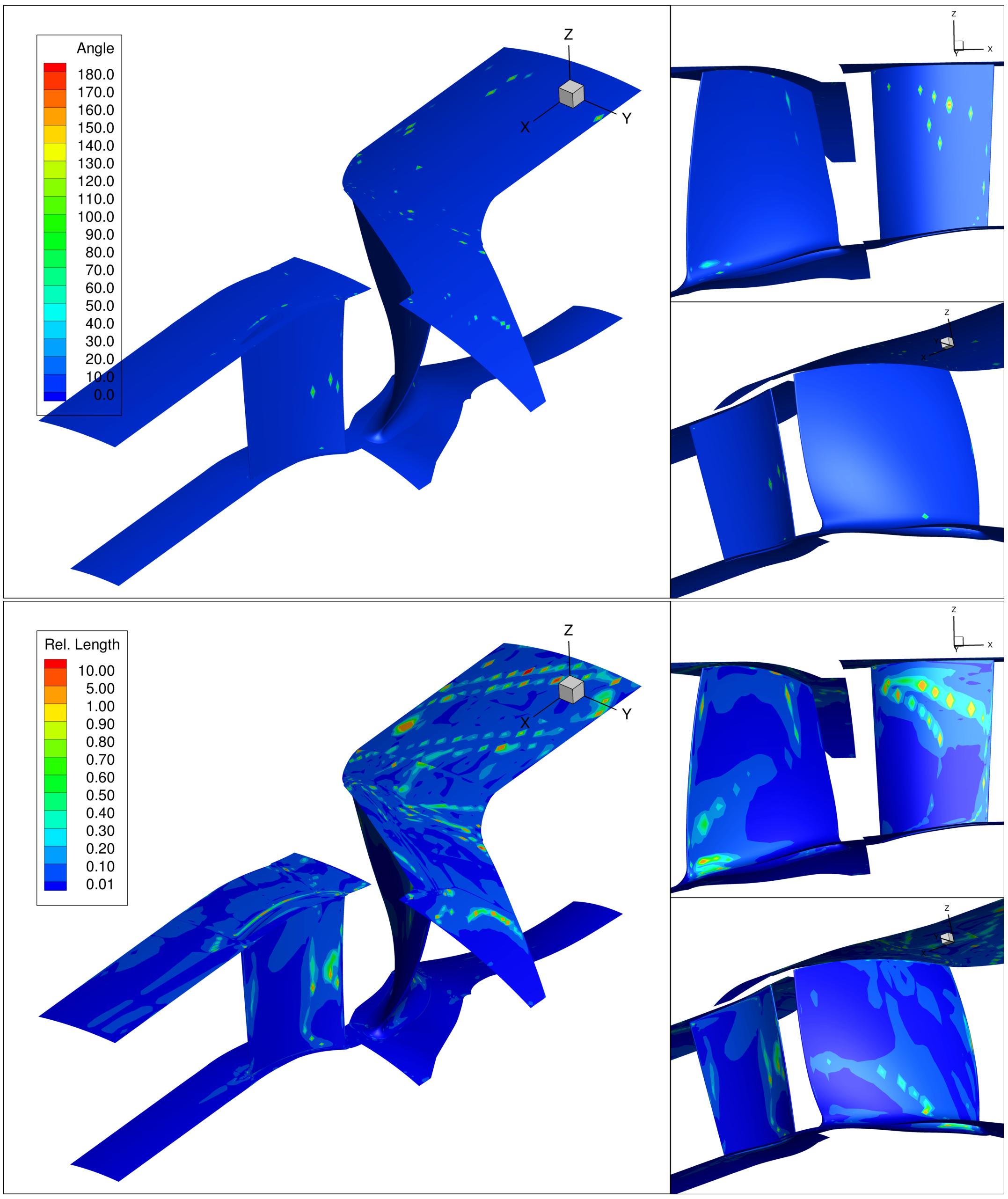


Figure 7.9.: THD difference of the base Black Box solution and the Reverse Accumulation solution.

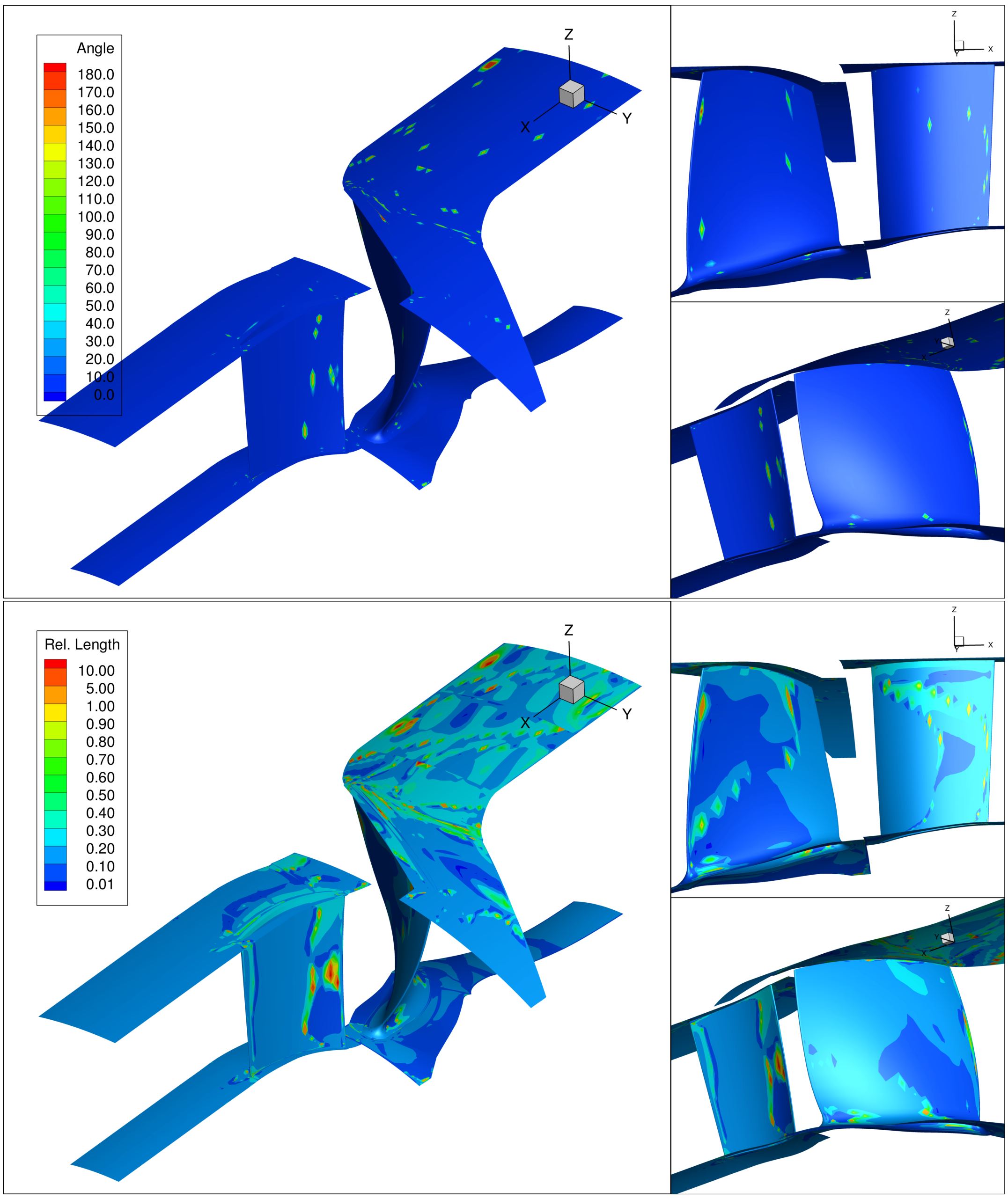


Figure 7.10.: THD difference of the base Reverse Accumulation solution and the solution after 1500 steps.

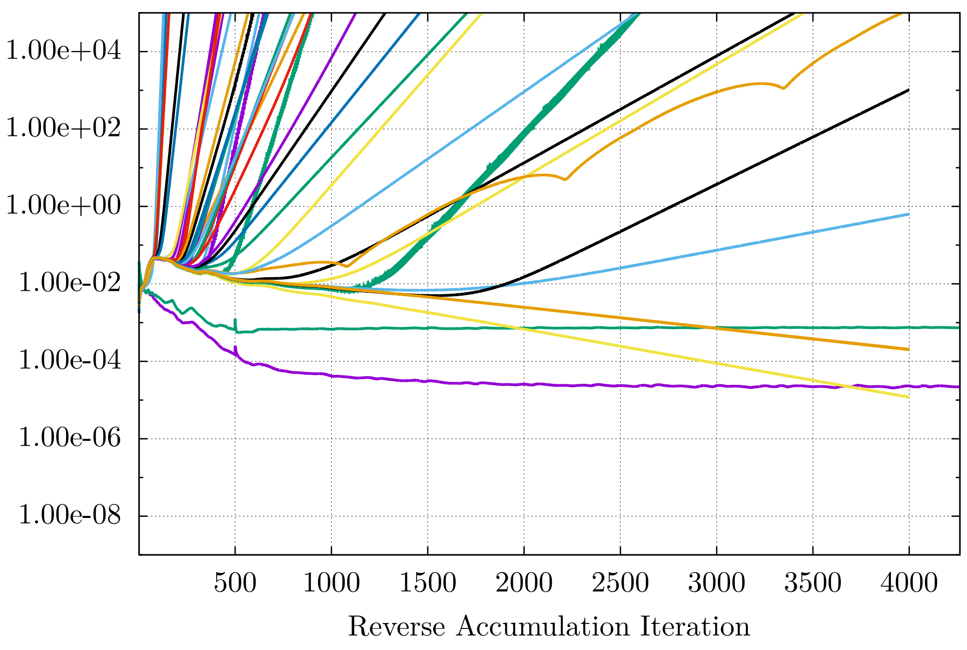


Figure 7.11.: THD Reverse Accumulation convergence evaluation

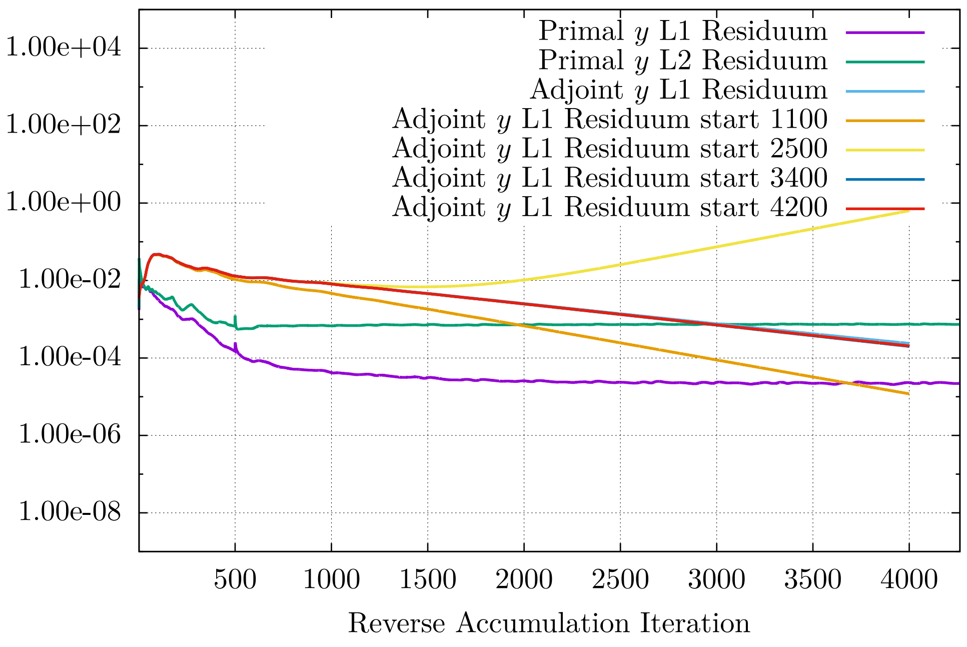


Figure 7.12.: THD Reverse Accumulation convergence evaluation (Converged starting points)

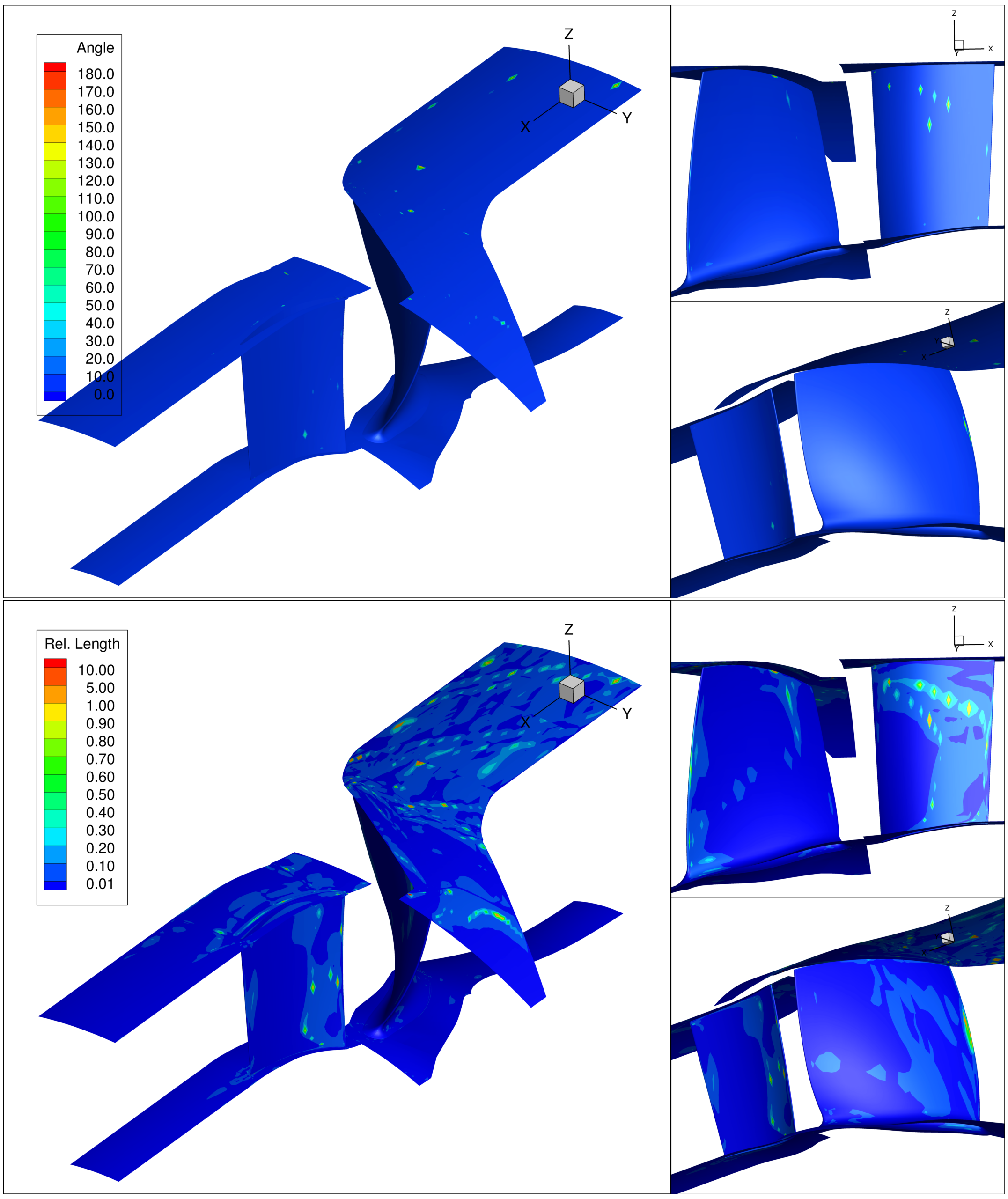


Figure 7.13.: THD difference of the base Reverse Accumulation solution and the Reverse Accumulation solution that is started at iteration 3400 of the primal.

8. Conclusion and outlook

The topic of the thesis covers the semiautomatic transition of simulation software to design software. The essential part of this process is described in Chapter 4, by postulating that the software should be treated as a black-box. With this approach every floating point value is replaced by the computation type of the AD tool. This guarantees a derivative computation, that is always consistent with the primal computation. After a successful compilation of the so modified code, the first step in the transformation of the simulation software is complete. The next important step is the verification and validation of the correct derivative results. The most important aspect is the verification of interfaces like the state. The state of the program is used to reset the program to a certain point in order to recompute some values. Without the correct definition of the state, every advanced technique similar to Reverse Accumulation or checkpointing will miss dependency information and yield wrong results. A big step forward in the automated analysis of such interfaces is made in this thesis. The tagging technique described in earlier chapters is a novel approach, that gives the AD community a new tool for the verification and analysis of AD based derivative software. The tagging allows for an automated testing and online verification of the correct states and their interfaces, yielding a correct derivative for the simulation software.

For small and medium cases the black-box approach is sufficient and will yield good enough run-time and memory requirements. For large scale applications that require high performance clusters, further optimizations need to be performed insuring feasible run-time and memory requirements are reached. It is shown in this thesis, that on all levels of detail, beginning with the derivative algorithms on the highest level and ending with the AD tool implementation on the lowest level, improvements have to be made.

The theory for the derivative algorithms is built along the assumption that for correct derivative results the iteration for the primal and adjoint should be as consistent as possible. It is the first theory that provides a completely consistent approach by using a fixed point iterator scheme that includes all implementation details of the application. For the analysis, the fixed point iterator is simplified and refined through several steps, such that all previous approaches like Reverse Accumulation, Giles' Exact Dual and the General Adjoint are covered. This shows that simplified algorithms could be employed, but only if all necessary conditions apply, that is the solution is converged "sufficient" enough. With the help of AD it is quite simple to generate all required derivatives for all of the different schemes and to employ them, even the most general Reverse Accumulation method. A further step forward is the introduction of the Black Box algorithm. The full differentiation and reversal of the full primal trajectory requires the most resources, yet always yields the correct derivative results. The method has no restrictions and should therefore always be used if the resources are available. It is also very good for verification or as a reference solution.

Improving the memory and time requirements for one step of the fixed point iterator is the next step. Existing analysis for the hot spots in a differentiated software is extended such that it is now feasible for use in large scale applications. This includes the new data layout and the correct tracking of output variables. The developer can directly focus on the improvement of the hot spots and does not need to perform a time intensive search for them. Each hot spot can either be handled by the function checkpoint or preaccumulation technique. The detailed analysis in the thesis provides for each technique how much memory is required and how the time requirements change. With this information the analysis can automatically detect which technique is the most appropriate. Both techniques are already known to the AD community, but there is no detailed analysis on the memory consumption, the time requirement, possible optimizations and implementation details for either of the different taping strategies. Other structural optimizations depend on the specific code and can not be established in a general framework. Nevertheless, for the TRACE code it is shown, how a shift in the computation algorithm can reduce the required resources dramatically.

All of the above improvements will only yield subordinate results, if the lowest level, namely the AD tool, is not optimal for the required task. The groundwork for the comparison of the AD tools is provided by a general analysis of the four major taping approaches: primal value taping, primal value taping with index reuse, Jacobi taping and Jacobi taping with index reuse. It is the most complete analysis on how an AD tool can be implemented and improved. Some of these techniques are already used in several AD tools, but the knowledge about the implementations and which properties the implementations have, is most of the time not published. Therefore, it is a big step forward, to have all of the implementation methods analyzed in a common framework and their properties compared. Also the presentation of the argument optimization provides a new possibility for low level memory optimizations. A further novel implementation technique for AD tools is the linear argument optimization. It introduces the implementation techniques of the primal value taping into the Jacobi taping approach to further reduce the memory consumption of the tapes.

The overview of the AD tools and the comparison with the presented taping approaches yields a mixed picture, some tools for use in HPC environments exist, but there is no tool that is specifically designed for HPC and implemented in an optimal way. For the Jacobi taping approach there are promising tools available and one could be modified such that a minimal memory consumption is achieved. The analysis also shows that there is a taping approach, namely primal value taping with statement level evaluation, that is currently completely unexplored and needs to be evaluated for HPC applications.

Because of this, the new AD tool CoDiPack is created. The fundamental design and layout of the tool is based on the knowledge gained by the analysis of the different taping strategies. It is therefore possible to implement all of the available strategies in one common framework and have the minimal memory consumption. The presented design patterns for CoDiPack provide, that new ideas can be implemented quite easily. This enables CoDiPack to be used in large scale software, to provide a base for a comparison of the four different taping strategies and to allow for further AD based research.

The next step for further research is the comparison of the primal value taping and Jacobi taping approach on real world applications and not only on small test problems. Here the Jacobi approach has a clear advantage during the tape evaluation. It needs less operations per statement and should therefore be faster. On the other hand, the primal value taping approach requires less memory per statement and can therefore also be competitive if the application is memory bound. Theoretically, the tape recording for the primal value tapes should be faster because the gradient for each statement does not need to be computed. Only tests with various applications will yield insights into the performance of both approaches.

These tests will raise the general question about the lower bound for the time factor of an operator overloading AD tool. This factor highly depends on the hardware architecture, the simulation software, the AD tool and the specific taping approach. It is therefore nearly impossible to give a general answer. If the simulation code is fixed, if it is not too involved and if the hardware is fixed, then it should be possible to derive a lower bound for the time factor. This lower bound can then be used to improve the AD tool implementation and function as a benchmark test for already existing AD tools and further ideas.

The availability of several different tools is important and gives the user the flexibility to choose the most appropriate. But if the user cannot easily see how the tools differ and which tool is the most appropriate for the task at hand, the choice becomes quite difficult and can even discourage the use of an AD tool. It is also obvious that over the years several different implementation optimizations have been rediscovered several times by different tool developers. These implementation aspects are mostly not published which raises three problems. First, new tool developers have no literature base for their new ideas. Second, users have no means to compare tools on a theoretical level. Third, and most importantly, the code optimization of AD tool implementations is stuck at a certain level, because a single person can not evaluate all possible avenues for such optimizations. It is therefore very important that the development of new operator overloading AD tools and existing ones be shifted to a more generalized framework, where different modules are available (e.g for index management or data management of the tapes). Only then can a more focused improvement of different aspects or modules be made possible, enhancing all tools that use these modules. The development of new tools should then be faster due to the reuse of exiting ideas.

Such a framework also allows for more experimental research with respect to accelerator cards for AD. The recording time of the tape is still the biggest bottleneck for an efficient use of the Black Box algorithm. By using the PCI Express bus, the additional memory bandwidth constraints for writing the tape data can be reduced. A next step could be the direct evaluation of the tape on the accelerator card. Especially, the Jacobi taping approach lends itself to this solution because of the simple interpretation loop. A final advantage of the accelerator cards could be the use of another level of parallelization. During the tape evaluation the processor can perform other tasks like the recomputation from a checkpoint.

All of these studies will be limited by the general operator overloading approach, namely creating a new user defined calculation type. The compilers are more conservative for the code optimization of user defined types and also the structure of a *double* and an *int* prevents the compiler from several code optimizations such as vectorization. It is therefore necessary to study the integration of AD tools into the code optimizations of a compiler. This avenue could also explore specialized code optimizations for AD tools or expression templates in general. One possible option could be the compression of several statements into one so the expressions become larger and less tape memory is required.

A step further would be to reevaluate the question if source transformation for large C++ codes is possible. The compiler landscape has changed in the last years and especially the LLVM [LA04] compiler suite makes the transformation of source code more accessible than before. At least a mixed approach where smaller code parts are transformed and directly integrated as expressions sounds very promising for primal value taping approaches.

It is safe to conclude that large scale simulation software can already be handled by AD. But, the research field of AD also offers a wide range of open questions that need to be tackled over the next years to further improve AD for HPC computers.

A. Calculus definitions

Definition A.1 (Directional derivative)
Let $F = (F_1, F_2, ..., F_m) : X \subset \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable function. Then the directional derivative of F in the direction $d \in \mathbb{R}^n$ at the point $x \in X$ is defined by

$$\frac{\partial F}{\partial d}(x) := \lim_{h \to 0} \frac{F(x + h \cdot d) - F(x)}{h} \ .$$

Definition A.2 (Partial derivative)
Let $F = (F_1, F_2, ..., F_m) : X \subset \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable function. Then the partial derivative at the point $x \in X$ is defined by the directional derivative in the directions $e_i \in \mathbb{R}^n$. e_i for all $i = 1...n$ is defined by $[e_i]_j = \delta_{ij}, j = 1...n$. This yields the directional derivative as

$$\frac{\partial F}{\partial x_i}(x) := \frac{\partial F}{\partial e_i}(x) = \lim_{h \to 0} \frac{F(x + h \cdot e_i) - F(x)}{h} \qquad i = 1...n \ .$$

Definition A.3 (Gradient)
Let $F : X \subset \mathbb{R}^n \to \mathbb{R}$ be a differentiable function. The gradient of F at the point $x \in X$ is the vector of all partial derivatives of F.

$$\nabla F(x) := grad\ F(x) = \begin{pmatrix} \frac{\partial F}{\partial x_1}(x) \\ \vdots \\ \frac{\partial F}{\partial x_n}(x) \end{pmatrix}$$

Definition A.4 (Jacobi matrix)
Let $F = (F_1, F_2, ..., F_n) : X \subset \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable function. The Jacobi matrix of F at the point $x \in X$ is defined by the partial derivatives of F.

$$F'(x) := \frac{df}{dx} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1}(x) & \frac{\partial F_1}{\partial x_2}(x) & \cdots & \frac{\partial F_1}{\partial x_n}(x) \\ \frac{\partial F_2}{\partial x_1}(x) & \frac{\partial F_2}{\partial x_2}(x) & \cdots & \frac{\partial F_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1}(x) & \frac{\partial F_m}{\partial x_2}(x) & \cdots & \frac{\partial F_m}{\partial x_n}(x) \end{pmatrix}$$

For a real valued function $F : X \subset \mathbb{R}^n \to \mathbb{R}$ there is the following relation between the Jacobi matrix and the gradient

$$\nabla F(x) = \frac{dF}{dx}^T(x) \ .$$

Definition A.5 (Chain rule)
Let $X \subset \mathbb{R}^n, Y \subset \mathbb{R}^m, Z \subset \mathbb{R}^k$ be an open set. Let $f : X \to Y$ and $g : Y \to Z$ be real differentiable functions, then the derivative of the function $F := g \circ f$ at the point $x \in X$ is defined by the chain rule

$$F'(x) = (g \circ f)(x) = g'(f(x)) \cdot f'(x) \ .$$

This covers the basic terms of the derivative calculus, that are required for AD.

B. Trivial graph format definition

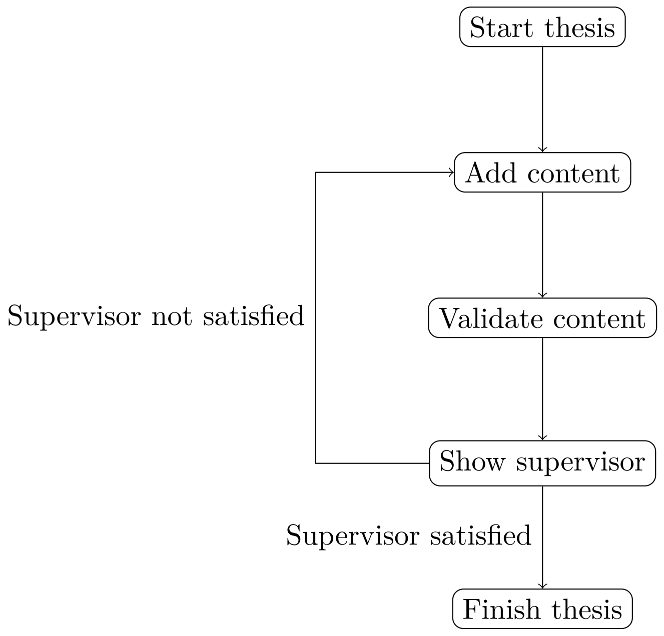
The definition for the trivial graph format (TGF) is taken from Wikipedia [Wik16]. The file has to start with a definition of all nodes. Each line defines one node where the first “word” in the line define the identifier. The reminder of the line defines the label for the node. The # character separates the node list from the edge list. It indicates that all nodes are written and that the following lines define the edges. Each line defines an edge and the first two “words” have to be identifiers that are defined in the node list. The direction of the edge is from the first identifier to the second identifier. The definition of the format is also explained in the following listing:

```
0 The label for a node
1 Node one
2 Node two
3 Node three
...
N Node N last node
#
0 1 The label for an edge
1 2 Edge from node one to node two
1 N Edge from node one to node N
2 N Edges can also have empty labels
3 N
```

An example graph from the following listing is represented by the figure in B.1.

```
1 Start thesis
2 Add content
3 Validate content
4 Show supervisor
5 Finish thesis
#
1 2
2 3
3 4
4 5 Supervisor satisfied
4 2 Supervisor not satisfied
```

Figure B.1.: Rendering of the sample TGF graph.



Curriculum vitae

Personal Data

Name:	Max Fritz Victor Sagebaum

Education

1993 - 1999	**Elementary School** Grundschule Glienicke, Glienicke Nordbahn, Germany
1999 - 2001	**Comprehensive School** Gesamtschule Glienicke, Glienicke Nordbahn, Germany
2001 - 2006	**Secondary School** Evangelisches Gymnasium Frohnau
2006 - 2011	**Studies of Mathematics** Diploma Humboldt-Universität zu Berlin, Berlin, Germany
2011 - 2014	**PhD studies Scientific Computing** RWTH Aachen, Aachen, Germany
2014 - 2018	**PhD studies Scientific Computing** TU Kaiserslautern, Kaiserslautern, Germany

Work experience

2004 - 2006	**Software development as part-time job** Fraunhofer FIRST, Berlin, Germany
2006 - 2011	**Software development as student assistant** Fraunhofer FIRST, Berlin, Germany
2011 - 2014	**Research associate** RWTH Aachen, Aachen, Germany
2014 - 2018	**Research associate** TU Kaiserslautern, Kaiserslautern, Germany

List of Publications

[ASG14] T. Albring, M. Sagebaum, and N. R. Gauger. A consistent and robust discrete adjoint solver for the Stanford University Unstructured (SU2) framework – validation and application. *Jahresbericht der Deutschen Strömungsmechanischen Arbeitsgemeinschaft STAB*, page 40–41, 2014.

[ASG15] T. Albring, M. Sagebaum, and N.R. Gauger. Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework. *AIAA 2015-3240*, 2015.

[ASG16a] T. Albring, M. Sagebaum, and N. R. Gauger. A consistent and robust discrete adjoint solver for the SU2 framework – validation and application. *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, 132:77–86, 2016.

[ASG16b] T. Albring, M. Sagebaum, and N. R. Gauger. Efficient aerodynamic design using the discrete adjoint method in SU2. *AIAA 2016-3518*, 2016.

[AZGS15] T. Albring, B.Y. Zhou, N.R. Gauger, and M. Sagebaum. An aerodynamic design framework based on Algorithmic Differentiation. *ERCOFTAC Bulletin*, 102:10–16, 2015.

[BSF$^+$17] J. Backhaus, A. Schmitz, C. Frey, M. Sagebaum, S. Mann, M. Nagel, and N. R. Gauger. Application of an Algorithmically Differentiated turbomachinery flow solver to the optimization of a fan stage. *AIAA 2017-3997*, 2017.

[LNSS13] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen. Discrete adjoints of PETSc through dco/c++ and AdjointMPI. In *Euro-Par 2013 Parallel Processing: 19th International Conference*, pages 497–507. Springer, Berlin, Heidelberg, 2013.

[NSGL17] S. A. Nørgaard, M. Sagebaum, N. R. Gauger, and B. S. Lazarov. Applications of Automatic Differentiation in topology optimization. *Structural and Multidisciplinary Optimization*, pages 1–12, 2017.

[SAG16] M. Sagebaum, T. Albring, and N. R. Gauger. Automated generation of performance values for Algorithmic Differentiation. *PAMM*, 16(1):863–864, 2016.

[SGN+13] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, and K. Leppkes. Algorithmic Differentiation of a complex C++ code with underlying libraries. *Procedia Computer Science*, 18:208–217, 2013.

[SOG14] M. Sagebaum, E. Özkaya, and N. R. Gauger. Challenges in the Automatic Differentiation of an industrial CFD solver. *EUROGEN 2013*, 2014.

[SOG+17] M. Sagebaum, E. Özkaya, N. R. Gauger, J. Backhaus, C. Frey, S. Mann, and M. Nagel. Efficient Algorithmic Differentiation techniques for turbomachinery design. *AIAA 2017-3998*, 2017.

Bibliography

[AB99] W. K. Anderson and D. L. Bonhaus. Airfoil design on unstructured grids for turbulent flows. *AIAA Journal*, 37(2):185–191, 1999.

[ABB$^+$10] L. S. Avila, S. Barre, R. Blue, B. Geveci, A. Henderson, W. A. Hoffman, B. King, C. C. Law, K. M. Martin, and W. J. Schroeder. *The VTK User's Guide*. Kitware Clifton Park, 2010.

[AC88] E. Acton and M. Cargill. Non-reflecting boundary conditions for computations of unsteady turbomachinery flow. In *Proceedings, 4th Int. Symp. Unsteady Aerodynamics and Aeroelasticity of Turbomachines and Propellers*, pages 211–228, 1988.

[ADP01] P. Aubert, N. Di Césaré, and O. Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3(4):197–208, 2001.

[AJ10] J. D. Anderson Jr. *Fundamentals of aerodynamics*. Tata McGraw-Hill Education, 2010.

[Alf09] G. Alfonsi. Reynolds-averaged Navier-Stokes equations for turbulence modeling. *Applied Mechanics Reviews*, 62(4):040802, 2009.

[Apo69] T. M. Apostol. *Calculus, Volume 2*. John Wiley & Sons, Inc., 1969.

[BA14] K. Becker and G. Ashcroft. A comparative study of gradient reconstruction methods for unstructured meshes with application to turbomachinery flows. In *52nd AIAA Aerospace Sciences Meeting*, 2014.

[Bar91] T. J. Barth. A 3-D upwind Euler solver for unstructured meshes. In *10th Computational Fluid Dynamics Conference*, page 1548, 1991.

[BB08] B. M. Bell and J. V. Burke. Algorithmic Differentiation of implicit functions and optimal values. In *Advances in Automatic Differentiation*, pages 67–77. Springer, 2008.

[BBL$^+$01] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and J. W. Risch. On the use of a differentiated Finite Element package for sensitivity analysis. In *Computational Science – ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 795–801, Berlin, 2001. Springer.

[BBL$^+$03] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and J. W. Risch. Extending the functionality of the general-purpose finite element package SEPRAN by Automatic Differentiation. *International Journal for Numerical Methods in Engineering*, 58(14):2225–2238, 2003.

[BC05] J. T. Betts and S. L. Campbell. Discretize then optimize. *Mathematics for industry: Challenges and frontiers*, pages 140–157, 2005.

[BDK08] G. A. Brignole, F. Danner, and H. P. Kau. Time resolved simulation and experimental validation of the flow in axial slot casing treatments for transonic axial compressors. In *Proceedings of ASME TurboExpo*, June 2008.

[Bea04] S. Beal. Supermacros: Powerful, maintainable preprocessor macros in C++. `http://wanderinghorse.net/computing/papers/supermacros_cpp.pdf`, 2004. [Online; accessed 21-December-2016].

[BG08] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[BHK10] K. Becker, K. Heitkamp, and E. Kügeler. Recent progress in a hybrid-grid CFD solver for turbomachinery flows. In *Proceedings Fifth European Conference on Computational Fluid Dynamics ECCOMAS CFD*, 2010.

[BHN08] C. H. Bischof, P. Hovland, and B. Norris. On the Implementation of Automatic Differentiation Tools. *Higher-Order and Symbolic Computation*, 21(3):311–331, 2008.

[BS96] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for Automatic Differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[CGBN94] A. Carle, L. L. Green, C. H. Bischof, and P. A. Newman. Applications of Automatic Differentiation in CFD. Technical report, NASA Langley Technical Report Server, 1994.

[CHB$^+$15] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, and P. Li. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *arXiv*, 1509.07164:1–96, 2015.

[Chr92] B. Christianson. Automatic Hessians by Reverse Accumulation. *IMA Journal of Numerical Analysis*, 12:135–150, 1992.

[Chr94] B. Christianson. Reverse Accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.

[CKS00] B. Cockburn, G. E. Karniadakis, and C.-W. Shu. The development of discontinuous Galerkin methods. In *Discontinuous Galerkin Methods: Theory, Computation and Applications*, pages 3–50. Springer, Berlin, Heidelberg, 2000.

[Col16] Y. Collet. lz4. `https://github.com/Cyan4973/lz4`, 2016. [Online; accessed 21-December-2016].

[DGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM TOMS*, 15(1):1–14, 1989.

[DS58] N. Dunford and J.T. Schwartz. *Linear Operators: General theory*. Linear Operators. Interscience Publishers, 1958.

[EP97] J. Elliot and J. Peraire. Practical 3D aerodynamic design and optimization using unstructured meshes. *AIAA Journal*, 35(9):1479–1485, 1997.

[FE02] S. A. Forth and T. P. Evans. *Aerofoil optimisation via AD of a multigrid cell-vertex Euler flow solver*, pages 153–160. Springer New York, New York, NY, 2002.

[FRKA10] M. Franke, T. Röber, E. Kügeler, and G. Ashcroft. Turbulence treatment in steady and unsteady turbomachinery flows. In *V European Conference on Computational Fluid Dynamics, ECCOMAS CFD*, pages 14–17, 2010.

[FSA$^+$12] D. A. Fournier, H. J. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. N. Maunder, A. Nielsen, and J. Sibert. AD Model Builder: Using Automatic Differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software*, 27(2):233–249, 2012.

[Gil87] J. R. Giles. *Introduction to the analysis of metric spaces*, volume 3. Cambridge University Press, 1987.

[Gil01] M. B. Giles. On the iterative solution of adjoint equations. In *Automatic Differentiation: From Simulation to Optimization*, pages 434–443. Springer-Verlag, 2001.

[GJ$^+$10] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GKE$^+$09] R. Giering, T. Kaminski, B. Eisfeld, N. Gauger, J. Raddatz, and L. Reimer. Automatic Differentiation of FLOWer and MUGRIDO. In *MEGADESIGN and MegaOpt*, pages 221–235. Springer, Berlin, Heidelberg, 2009.

[GKN06] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. Technical report, AT&T Research, 2006.

[GNH96] L. L. Green, P. A. Newman, and K. J. Haigler. Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via Automatic Differentiation. *Journal of Computational Physics*, 125(2):313–324, 1996.

[Gri93] A. Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. *Complexity in Nonlinear Optimization*, pages 128–161, 1993.

[Gri09] A. Griewank. *Complexity of gradients, Jacobians, and Hessians*, pages 425–435. Springer US, Boston, MA, 2009.

[GW00] A. Griewank and A. Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM TOMS*, 26(1):19–45, March 2000.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives, second edition.* SIAM, 2008.

[Har04] S. Harris. *An introduction to the theory of the Boltzmann equation.* Dover books on physics. Dover Publications, 2004.

[HBU15] A. Hück, C. Bischof, and J. Utke. Checking C++ codes for compatibility with operator overloading. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 91–100, Sept 2015.

[Him97] M. Himsolt. GML: A portable graph file format. `https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf`, 1997. [Online; accessed 29-April-2017].

[HL97] X. He and L.-S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Phys. Rev. E*, 56:6811–6817, Dec 1997.

[HM76] P. Henderson and J. H. Morris, Jr. A Lazy Evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL, pages 95–103, New York, NY, USA, 1976. ACM.

[Hog14] R. J. Hogan. Fast reverse-mode Automatic Differentiation using expression templates in C++. *ACM TOMS*, 40(4):26:1–26:24, 2014.

[HP13] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, model, and specification. *ACM TOMS*, 39(3), 2013.

[HUB16] A. Hück, J. Utke, and C. Bischof. Source transformation of C++ codes for compatibility with operator overloading. *Procedia Computer Science*, 80:1485 – 1496, 2016.

[Irv93] C. Irving. *Wide-body: The triumph of the 747.* William Morrow & Co, 1993.

[JMP98] A. Jameson, L. Martinelli, and N. A. Pierce. Optimum aerodynamic design using the Navier-Stokes equations. *Theoretical and Computational Fluid Dynamics*, 10(1):213–237, 1998.

[JR15] J. Jeffers and J. Reinders. *High performance parallelism pearls volume two: Multicore and many-core programming approaches.* Morgan Kaufmann, 2015.

[Jue91] D. W. Juedes. A Taxonomy of Automatic Differentiation Tools. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, PA, 1991.

[KT51] H. W. Kuhn and A. W. Tucker. Nonlinear Programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LeV07] R. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations.* SIAM, 2007.

[Lin76] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[LLN16] K. Leppkes, J. Lotz, and U. Naumann. Derivative code by overloading in C++ (dco/c++): Introduction and summary of features. Technical Report AIB-2016-08, RWTH Aachen University, September 2016.

[LMHF13] G. C. Lee, S. B. Mohan, C. Huang, and B. N. Fard. A study of US bridge failures (1980–2012). MCEER-13–0008, State University of New York, 2013.

[LML$^+$06] R. B. Langtry, F. R. Menter, S. R. Likki, Y. B. Suzen, P. G. Huang, and S. Völker. A correlation-based transition model using local variables—Part II: Test cases and industrial applications. *Journal of Turbomachinery*, 128(3):423–434, 2006.

[LNM16] J. Lotz, U. Naumann, and S. Mitra. Mixed integer programming for call tree reversal. In *Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 83–91. SIAM, 2016.

[LNSS13] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen. Discrete adjoints of PETSc through dco/c++ and AdjointMPI. In *Euro-Par 2013 Parallel Processing: 19th International Conference*, pages 497–507. Springer, Berlin, Heidelberg, 2013.

[Lot16] J. Lotz. *Hybrid approaches to adjoint code generation with dco/c++*. PhD thesis, RWTH Aachen University, Aachen, 2016.

[Mic91] R. Mickens. *Difference equations.* CRC Press, 1991.

[MLL$^+$06] F. R. Menter, R. B. Langtry, S. R. Likki, Y. B. Suzen, P. G. Huang, and S. Völker. A correlation-based transition model using local variables—Part I: Model formulation. *Journal of turbomachinery*, 128(3):413–422, 2006.

[NA99] E. J. Nielsen and W. K. Anderson. Aerodynamic design optimization on unstructured meshes using the Navier-Stokes equations. *AIAA Journal*, 37(11):957–964, 1999.

[NA02] E. J. Nielsen and W. K. Anderson. Recent improvements in aerodynamic design optimization on unstructured meshes. *AIAA journal*, 40(6):1155–1163, 2002.

[Nau12] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* Software, Environments, and Tools. SIAM, 2012.

[ND17] E. J. Nielsen and B. Diskin. High-performance aerodynamic computations for aerospace applications. *Parallel Computing*, 64:20 – 32, 2017. High-End Computing for Next-Generation Scientific Discovery.

[NES99] D. Nuernberger, F. Eulitz, and S. Schmitt. Effiziente Berechnung der instationären Strömung in Turbomaschinen mittels impliziter Zeitintegration. In *Proc. DGLR-Jahrestagung*, 1999.

[NLPD04] E. J. Nielsen, J. Lu, M. A. Park, and D. L. Darmofal. An implicit, exact dual adjoint solution method for turbulent flows on unstructured grids. *Computers and Fluids*, 33(9):1131–1155, 2004.

[NS79] W. M. Newman and R. F. Sproull. *Principles of interactive computer graphics.* McGraw-Hill, Inc., 1979.

[NW99] J. Nocedal and S. Wright. *Numerical Optimization.* Springer Series in Operational Research, 1999.

[PBGH08] E. T. Phipps, R. A. Bartlett, D. M. Gay, and R. J. Hoekstra. Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via Automatic Differentiation. In *Advances in Automatic Differentiation*, pages 351–362. Springer, 2008.

[PD10] J. E. V. Peter and R. P. Dwight. Numerical sensitivity analysis for aerodynamic optimization: A survey of approaches and applications. *Computers & Fluids*, 39:373–391, 2010.

[PP12] E. Phipps and R. Pawlowski. Efficient expression templates for operator overloading-based automatic differentiation. In *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 309–319. Springer, Berlin, 2012.

[Ran06] D. A. Randall. Reynolds averaging. *Colorado State University, Fort Collins*, 2006.

[RJL$^+$15] D. Roggenkamp, W. Jessen, W. Li, M. Klaas, and W. Schröder. Experimental investigation of turbulent boundary layers over transversal moving surfaces. *CEAS Aeronautical Journal*, 6(3):471–484, 2015.

[Röb13] T. Röber. Continuous formulation of wall function with adverse pressure gradient. In *New Results in Numerical and Experimental Fluid Mechanics VIII*, pages 411–418. Springer, 2013.

[RT15] M. Roughan and J. Tuke. Unravelling graph-exchange file formats. *arXiv preprint arXiv:1503.02781*, 2015.

[SAG16] M. Sagebaum, T. Albring, and N. R. Gauger. Automated generation of performance values for Algorithmic Differentiation. *PAMM*, 16(1):863–864, 2016.

[SG03] M. Sinapius and H. Giese. Experimental modal analysis of the Airbus-A380 wind tunnel model 855. Technical report, DLR, 2003.

[SGN$^+$13] M. Sagebaum, N.R. Gauger, U. Naumann, J. Lotz, and K. Leppkes. Algorithmic Differentiation of a complex C++ code with underlying libraries. *Procedia Computer Science*, 18:208–217, 2013.

[SNHU10] M. Schanen, U. Naumann, L. Hascoët, and J. Utke. Interpretative adjoints for numerical simulation codes using MPI. *Procedia Computer Science*, 1(1):1825–1833, 2010.

[Spa88] P. R. Spalart. Direct simulation of a turbulent boundary layer up to RΘ 1410. *Journal of Fluid Mechanics*, 187:61–98, 1988.

[Ste88] S. Stewart. *Air Disasters.* Arrow Books, 1988.

[Tem84] R. Temam. *Navier-Stokes equations*, volume 2. North-Holland Amsterdam, 1984.

[TS92] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and algorithms.* John Wiley & Sons, Inc., 1992.

[UHH$^+$09] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward adjoinable MPI. In *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC'09.* IEEE, 2009.

[Utk04] J. Utke. OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 2004.

[Utk05] J. Utke. Flattening Basic Blocks. In *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 121–133. Springer, 2005.

[Vel95] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[Ven93] V. Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. In *31st Aerospace Sciences Meeting*, page 880, 1993.

[VL79] B. Van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of computational Physics*, 32(1):101–136, 1979.

[VM07] H. K. Versteeg and W. Malalasekera. *An introduction to computational fluid dynamics: The finite volume method.* Pearson Education, 2007.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the Lambda-calculus.* PhD thesis, University of Oxford, 1971.

[Wer82] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City*, pages 762–770. Springer, Berlin, Heidelberg, 1982.

[Wer94] P. J. Werbos. *The roots of backpropagation: From ordered derivatives to neural networks and political forecasting*, volume 1. John Wiley & Sons, 1994.

[WG09] A. Walther and A. Griewank. Getting started with ADOL-C. In *Combinatorial scientific computing*, pages 181–202, 2009.

[WGNP07] B. J. N. Wylie, M. Geimer, M. Nicolai, and M. Probst. Performance analysis and tuning of the XNS CFD solver on Blue Gene/L. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 107–116. Springer, 2007.

[Wik16] Wikipedia. Trivial Graph Format — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 5-February-2017].

[Wil88] D. C. Wilcox. Reassessment of the scale-determining equation for advanced turbulence models. *AIAA Journal*, 26(11):1299–1310, 1988.

[WJ00] S. Wallin and A. V. Johansson. An explicit algebraic Reynolds stress model for incompressible and compressible turbulent flows. *Journal of Fluid Mechanics*, 403:89–132, 2000.

[WKR02] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Software Visualization*, pages 324–336. Springer, 2002.

[YNK06] H. Yang, D. Nuernberger, and H.-P. Kersken. Toward excellence in turbomachinery computational fluid dynamics: A hybrid structured-unstructured Reynolds-averaged Navier-Stokes solver. *Journal of turbomachinery*, 128(2):390–402, 2006.

[Zao08] F. Zaoui. Large electrical power systems optimization using Automatic Differentiation. In *Advances in Automatic Differentiation*, pages 293–302. Springer, 2008.

[ZTZ13] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The finite element method: Its basis and fundamentals.* Butterworth-Heinemann, Oxford, seventh edition, 2013.