Implementing Algorithmic Differentiation Capabilities in the Universal Laminar Flame Solver

Bachelor-Thesis von Sebastian Kreutzer aus Weinheim
Tag der Einreichung:

1. Gutachten: Prof. Christian H. Bischof
2. Gutachten: Alexander Hück

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Institut für Scientific Computing

Implementing Algorithmic Differentiation Capabilities in the Universal Laminar Flame Solver

Vorgelegte Bachelor-Thesis von Sebastian Kreutzer aus Weinheim

1. Gutachten: Prof. Christian H. Bischof
2. Gutachten: Alexander Hück

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. Mai 2017

(S. Kreutzer)

Abstract

In scientific computing applications, the efficient and accurate computation of derivatives is key. Typically, the evaluated functions are very complex which makes differentiation by hand infeasible. A commonly used approach is numerical differentiation which approximates the derivatives by evaluating the function with perturbed input variables. However, due to poor performance characteristics and limited accuracy, this method is not always applicable.

Algorithmic Differentiation (AD) is a method that applies the chain rule of differential calculus on the evaluation function to compute partial derivatives alongside the primary function variables. In contrast to finite difference methods, AD yields derivatives accurate up to machine precision. AD provides two modes of operation: A forward mode, which scales with the number of input variables in terms of run time, and a reverse mode, which scales with the number of output variables.

The Universal Laminar Flame Solver (ULF) is a C++ application that allows the simulation of chemically reacting gas mixtures on orthogonal grids. The behavior of the system is described with differential equations. ULF provides different strategies to solve them, using variations of the Newton method which require the Jacobian of the system. Since the Jacobian must be reevaluated regularly during the solving process, the efficiency of the differentiation method is a key factor for the overall performance. Subject of this research is the implementation of AD capabilities in ULF using the CoDiPack library, an AD tool based on operator overloading. As such, this thesis documents the process of performing the global type change required for AD. Furthermore, we describe our approach to implementing a general type conversion mechanism to make communication with external libraries operating on built-in floating point types possible. Following the introduction of CoDiPack, we added new Jacobian evaluation procedures based on AD. Using two model problems which focus on the evaluation of transport and chemistry coefficients respectively, we evaluated the performance of our methods with respect to run time and memory consumption. The application of AD on the heat equation yielded similar performance as the existing implementations. For the homogeneous reactor, which models the reaction of an evenly distributed gas mixture, we were able to able to achieve a speedup of 3.6 compared to the finite difference method.

Zusammenfassung

Für Anwendungen im Bereich Scientific Computing ist die effiziente und genaue Berechnung von Ableitungen von zentraler Bedeutung. Typischerweise sind die betrachteten Funktionen sehr komplex und können deshalb nicht analytisch differenziert werden. Eine gängige Lösung ist die Verwendung von numerischen Differenzierungsverfahren, welche die Ableitungen durch die Evaluierung der Funktion mit gestörten Eingabevariablen approximieren. Allerdings ist diese Methode wegen schlechter Performanz und Genauigkeit nicht immer anwendbar.

Algorithmisches Differenzieren (AD) ist eine alternative Differenzierungsmethode, die auf der Anwendung der Kettenregel auf den einzelnen Operationen der Evaluationsfunktion basiert um Ableitungen parallel zu den Funktionsvariablen zu berechnen. Anders als die numerischen Finite-Differenzen-Methoden, erlaubt AD die Berechnung von Ableitungen mit Maschinengenauigkeit. Es gibt zwei verschiedene AD Modi: Einen Vorwärtsmodus, der bezüglich der Laufzeit mit der Anzahl der Eingabevariablen skaliert, und einen Rückwärtsmodus, der mit der Anzahl der Ausgabevariablen skaliert.

Der Universal Laminar Flame Solver (ULF) ist eine C++ Software für die Simulation von chemisch reagierenden Gasgemischen auf orthogonalen Gittern. Das Verhalten der Systeme die mit ULF gelöst werden können wird mittels Differentialgleichungen beschrieben. ULF bietet verschiedene Strategien um diese zu lösen und verwendet dabei Variationen des Newtonverfahrens, welches die Jacobi-Matrix des System benötigt. Da die Jacobi-Matrix während des Lösungsvorgangs regelmäßig neu ausgewertet werden muss, ist die Effizienz der Differenzierungsmethode wichtig für die Performanz des gesamten Verfahrens. Im Rahmen dieser Arbeit wurde AD in ULF eingeführt, unter der Verwendung der CoDiPack Bibliothek, einem AD Tool welches auf Operator Overloading basiert. Diese Thesis dokumentiert die Einführung eines neuen globalen Typs, der für die Verwendung von AD benötigt wird. Weiterhin beschreiben wir unseren Ansatz für einen Mechanismus zur Typumwandlung, den wir eingeführt haben um die Kommunikation mit externen Bibliotheken zu ermöglichen die mit `double` arbeiten. Im Anschluss an die Integration von CoDiPack wurden neue Funktionen zur Evaluation der Jacobi-Matrizen basierend auf AD implementiert.

Unter der Verwendung von zwei Modellproblemen, welche auf die Auswertung von Transport- und Chemiekoeffizienten fokussiert sind, evaluieren wir die Performanz unserer Methoden im Bezug auf Laufzeit und Speicherverbrauch. Die Anwendung von AD auf die WÃd'rmeleitungsgleichung hat zu Ãd'hnlichen Laufzeiten wie die existierenden Implementierungen gefÃijhrt. Für den homogenen Reaktor, welcher die Reaktion eines gleichmäßig verteilten Gasgemischs modelliert, konnten wir dabei signifikante Verbesserungen mit einem Speedup von 3.6 im Vergleich zu dem numerischen Verfahren feststellen.

Contents

1 Introduction 6

2 Related Work 9
2.1 ADOL-C . . . 9
2.2 ADIC . . . 9
2.3 ADiMAT . . . 9
2.4 Adept . . . 9
2.5 Adjoinable MPI . . . 10

3 Algorithmic Differentiation 11
3.1 Definitions . . . 11
3.2 The Forward Mode of AD . . . 12
3.3 The Reverse Mode of AD . . . 12
3.4 Overhead . . . 14
3.5 Vector Mode . . . 14
3.6 Evaluating Whole Jacobians . . . 14
3.7 AD in Practice . . . 15
3.7.1 Source-to-Source Transformation . . . 15
3.7.2 Operator Overloading . . . 16
3.8 CoDiPack . . . 16
3.8.1 Expression Templates . . . 17

4 The Universal Laminar Flame Solver (ULF) 18
4.1 Solving Strategies and the Significance of Jacobians . . . 18
4.2 Design Principles . . . 19
4.2.1 Modularity . . . 19
4.2.2 Extensibility . . . 20
4.3 Important Data Structures in ULF . . . 20

5 Introducing AD to ULF 21
5.1 Code Style . . . 21
5.2 Refactoring the Data Structures . . . 22
5.3 Performing the Global Type Change . . . 23
5.3.1 Introducing `ulfScalar` . . . 23
5.3.2 Replacing Occurrences of `double` . . . 23
5.4 Solving Type Compatibility Issues . . . 24
5.4.1 Extracting Primal Values from the Active Type . . . 24
5.4.2 Copying Data with `memcpy` . . . 25
5.5 Enabling active types in the assert system . . . 25

6 Jacobian Evaluation in ULF 27
6.1 Jacobian Characteristics . . . 27
6.2 Compression Techniques . . . 28
6.3 Existing Jacobian Evaluation Methods . . . 29
6.3.1 Numerical Evaluation . . . 29
6.3.2 Analytical Evaluation . . . 31
6.4 Jacobian Evaluation with AD . . . 32
6.4.1 Forward AD Algorithm . . . 33

6.4.2 Reverse AD Algorithm . . . 34
6.4.3 Vector mode AD . . . 35

7 Application on Model Problems 38
7.1 Reaction Mechanisms . . . 38
7.2 Methods . . . 38
7.3 Heat Equation Problem . . . 39
7.3.1 Jacobian structure . . . 39
7.3.2 Jacobian Accuracy . . . 40
7.3.3 Run Time Measurements . . . 40
7.3.4 Scaling with Jacobian Size . . . 42
7.3.5 Memory Consumption . . . 42
7.4 Homogeneous Reactor . . . 45
7.4.1 Jacobian structure . . . 45
7.4.2 Jacobian Accuracy . . . 46
7.4.3 Run Time Measurements . . . 46
7.4.4 Reasons for the Discrepancy between Forward and Reverse Mode . . . 47
7.4.5 Performance of the Vector Mode . . . 50
7.4.6 Scaling with Jacobian Size . . . 50
7.4.7 Memory Consumption . . . 51

8 Conclusion and Outlook 52

A Source Code of `recast` 54

B Source Code of `ulf_memcpy` 55

C Source Code of variadic `printf` 56

D Homogeneous Reactor Plots 57

E Heat Equation Plot 58

1 Introduction

In many scientific applications, the accurate and fast computation of derivatives is crucial. For optimization problems and sensitivity analyses, it is necessary to determine Jacobians or even Hessians of the equations that govern the system. These equations are usually defined in the form of an evaluation procedure and are often very complex and non-linear. Therefore, it is generally not possible to differentiate these functions analytically[1].

Finite difference methods avoid this problem by using a black-box approach. The simplest form is the forward difference quotient, which is defined as

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \tag{1}$$

This method has the advantage of being easy to implement, but gives only an approximation of the real derivative value. To get adequate results, the step size h must be chosen appropriately. Because of the limited floating point accuracy on real-world machines, too small values for h lead to cancellation errors. On the other hand, if h is too big, higher order terms of the underlying function introduce truncation errors.

Approximation errors aside, in some cases the use of finite difference approximation is simply not possible due to its poor performance characteristics. In scientific applications it is not uncommon to have functions that depend on thousands of input variables. Computing a complete Jacobian of such a function with the finite difference method requires one evaluation for each of these variables, which is infeasible for non-trivial equations.

In these cases, Algorithmic Differentiation (AD), also called Automatic Differentiation, provides a viable alternative [1][2]. AD is based on the fundamental assumption that every program, no matter how complex, is composed of a number of elementary operations. These can be algebraic operations such as $*, +, -$, etc. or more complex intrinsic functions like `sin` and `sqrt`. Algorithmic Differentiation exploits the fact that the derivatives of these primitive building blocks are known. Using the chain rule shown in equation 2, AD combines the derivatives of the individual operations to construct an evaluation code for the differentiation of the whole function. Since there is no approximation or tuning of parameters required, AD provides derivatives which are accurate up to machine precision.

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x} \tag{2}$$

AD has two different modes of operation, called the forward and the reverse mode. The forward mode propagates derivatives in parallel with the function values according to Equation 2. Similar to the finite difference method, in order to construct the whole Jacobian, the forward mode requires one evaluation of the function for each independent variable.

The reverse mode employs an alternative way to compute derivatives. In an initial step called the forward sweep, the function is evaluated normally and intermediate results are stored. The subsequent reverse sweep reverses the flow of control and computes the so called adjoint for each recorded variable, which is then used to construct the derivative. Although being the more complex of the two modes, the reverse mode makes gradient computation cheap, which allows efficient differentiation of scalar functions with a high number of independent variables.

AD is usually introduced by augmenting the original code using source-to-source transformation or by exploiting operator overloading capabilities of the programming language. In C++ environments, the first option is rarely used because of the complexity of the language. Operator overloading has the benefit of being less intrusive, as the original function code remains largely unchanged. However, this method typically requires a global type change from the primitive scalar to the active AD type, which can be a challenging task from a software engineering standpoint[3][4].

The Universal Laminar Flame solver (ULF), developed by the chair of numerical thermo-fluid dynamics (NTFD) at the Technical University Bergakademie Freiberg, is a simulation framework that solves problems involving chemically reacting gas mixtures on orthogonal grids, described by systems of differential equations. Given a starting configuration, ULF employs time-stepping methods to numerically integrate these equations over time on a one-dimensional mesh.

The ODE solvers used by ULF rely on variants of the Newton method, which require a good approximation of the Jacobian of the equation system. During the course of the solving process, the Jacobian has to be reevaluated multiple times. Therefore, an efficient differentiation method is central to the overall performance.

Previously, ULF provided numerical and analytical differentiation methods.

The numerical evaluation procedure is based on a finite difference approach similar to Equation 1. As discussed above, this approach can be applied to every problem without the need for customization and is therefore ULF's default method for Jacobian computation. However, because it requires multiple costly equation evaluations, it is rather slow.

As an alternative, ULF provides the option to utilize PyJac [5] and TChem [6], two analytical differentiation frameworks. Based on a mechanism description, which contains information about the reactions of the involved species, these libraries supply an evaluation procedure to compute the derivatives. In contrast to the finite difference method, analytical evaluation promises exact Jacobians without approximation errors. In theory, this reduces the run time of the solver, as improved quality of the derivatives can lead to faster convergence. Experiments have shown that both TChem and PyJac result in a shorter run time compared to the numerical method.

Unfortunately, the analytical methods have some major limitations that reduce their usefulness. The biggest problem is that these approaches only operate locally on individual grid points and do not take the interactions between neighboring points on the mesh into considerations. This means that effects like diffusion and convection, which typically appear in realistic flame simulations, are not covered by these methods and have to be treated separately.

As both of the existing approaches lack in certain regards, Algorithmic Differentiation promises to be a good alternative. What makes AD so attractive in this context, is the fact that it combines positive aspects of both approaches. Like the analytical methods, AD computes the derivatives up to machine precision. At the same time, AD does not require any additional knowledge about the system and is therefore not bound to a specific type of problem.

This thesis describes the process of introducing AD to ULF. We chose the AD tool CoDiPack for this task, as it employs modern C++ features and promises good performance [7]. The contributions of this thesis are:

- Introducing AD to ULF with a global type change
- Documenting issues of the type change and proposing code changes to take care of the problematic code constructs, thus guiding similar future efforts
- A thorough performance evaluation, comparing the existing finite differences and analytical approaches in ULF to the AD implementation with respect to run time and memory consumption

The thesis is structured as follows: Section 2 gives an overview of related work in the field of AD. Section 3 gives a detailed introduction to AD and discusses the characteristics of the different modes. An explanation of the functionality of ULF and its architecture is given in Section 4. Here, we describe the steps of a problem solve run and introduce the most important data structures that are relevant in the context of the type change. Section 5 documents the execution the global type change to the active type provided by CoDiPack. Subsequently, we present the issues that occurred during that process and propose solutions to each one. The code that we implemented to evaluate the Jacobians is presented in Section 6. In this part, we discuss how the seeding and the collection of the derivatives differs for the forward and the reverse mode. Additionally, we show how the evaluation procedure has to be modified to work with

CoDiPack's vector types, which allow the simultaneous propagation of multiple derivatives. We analyze and compare the new evaluation methods with the existing numerical and analytical approaches in Section 7. First, we investigate the performance of the heat equation problem, which does not involve any chemical reactions. Secondly, we consider the homogeneous reactor, a problem configuration that simulates the reaction of an evenly distributed gas mixture on a single grid point. For both problems we analyze how each evaluation method impacts the run time and memory consumption, and identify bottlenecks. Finally, Section 8 gives an overview of the results of this research and provides an outlook on possible future work.

2 Related Work

This section aims to introduce the reader to related work in the field of AD.

2.1 ADOL-C

ADOL-C is an AD tool for C and C++ based on operator overloading that provides implementations for the computation of Jacobians, Hessians as well as Jacobian-Vector and Hessian-Vector products [8][9]. ADOL-C introduces the active type `adouble`, which replaces the primitive double type in the differentiated code sections.

Both forward and reverse mode are supported. For the reverse mode, ADOL-C uses a tape that stores intermediate results from the forward sweep for the subsequent adjoint computation. For Large-scale applications the tape can become very large and has to be written out to the hard drive, which can significantly impact performance. Newer versions of ADOL-C introduced a tapeless forward mode which avoids this issue and can be used to compute first-order derivatives.

Furthermore, ADOL-C supports the integration of the graph coloring library ColPack [10] that allows the exploitation of sparse Jacobians. ColPack helps identifying columns of the Jacobian that correspond to variables that are independent from each other. Since these columns can then be evaluated simultaneously, the number of required function evaluations to construct the whole Jacobian can be reduced. For ODEs, ADOL-C provides specialized routines that evalutate the Taylor coefficients with respect to the current state vector. ADOL-C supports MPI through the Adjoinable MPI framework [11].

2.2 ADIC

ADIC is a tool that allows the integration of AD to ANSI-C programs by using source-to-source transformation[12]. Since C does not support operator overloading, this approach is the only option to apply AD. ADIC uses a modular structure and allows the language independent development of AD modules that implement specific differentiation functionalities.

For the analysis of the program, the C code is first transformed to the automatic differentiation intermediate form (AIF). The AIF code ist then augmented with derivative computations using the selected derivative module. The derivative module implements this process based on the required derivative order and is implemented for first and second-order differentiation. The final derivative code is then constructed from multiple transformed code sections.

In 2010, the the follow-up project ADIC2 was introduced[13]. It applies the same principles as ADIC, but is extended to support a subset of C++ as well.

2.3 ADiMAT

ADiMAT is an AD tool for the MATLAB language[14]. In contrast to most other tools, ADiMAT does not rely solely on operator overloading or source-to-source transformation. Instead, it uses a hybrid method that takes advantage of both approaches. Similar to conventional source transformation tools, ADiMAT first analyzes the code and computes data-dependencies. Using the differentiation rules, the updates for the derivative variables are subsequently inserted into the code. The individual operators, however, are overloaded to work both for the primary variable as well as the derivative objects, which are implemented using cell-arrays of dynamic sizes. This approach has several benefits. For one, directly inserting the derivative updates into the code eliminates the need to define overloaded versions of the elemental functions, which in MATLAB reside in individual files. Additionally, the use of cell-array allows the simultaneous computation of an arbitrary number of derivatives.

2.4 Adept

In previous tools, the reverse mode was usually very inefficient, with execution times of the adjoint code 10-35 times slower than the original program The authors of [15] introduce Adept, an AD tool that

significantly improves performance by using expression templates. Instead of evaluating each elemental operation independently, expression templates introduce templated structures for each of these operations. This approach allows the construction of large expression trees, which can then be evaluated without the creation of temporary variables. As a result, the derivative code is much more efficient. The authors report speedups of 2.6 - 9 over other AD tools.

2.5 Adjoinable MPI

In [11], the authors introduce a framework that allows the generation of adjoint code for message-passing programs. MPI is a standardized interface that defines a set of communication operations between processing nodes. The application of the reverse mode always requires a reversal of the control and data flow of the differentiated program. Because of the transfer of data between processors, in programs using MPI the computation of the adjoints requires data that may not be stored at the same node. Therefore, relying on standard AD tools to apply the reverse mode on MPI programs is not sufficient. The *Adjoinable MPI* framework solves this problem by providing adjoint versions of common MPI idioms. During the reverse sweep, additional communication operations are introduced to retrieve the data needed for the adjoint computation. *Adjoinable MPI*, as well as the alternative framework *Adjoint MPI* [16] can be integrated into existing AD tools and have been successfully applied in large-scale applications[7].

3 Algorithmic Differentiation

This section formally introduces the principles of Algorithmic Differentiation and discusses the two modes of operation, using a notation that is based on [1]. This is followed by a description of how AD is implemented in practice, elaborating on advantages and disadvantages of source-to-source transformation and operator overloading approaches. Subsequently, the evaluation of whole Jacobians, which usually requires multiple AD passes with specific initial derivative values, is discussed. Finally, the AD tool CoDiPack, which was used for the integration of AD in ULF, is introduced.

AD allows the differentiation of functions which are defined by an evaluation procedure. In theory, the principles behind AD can applied for the manual transformation of the code. However, for larger applications, adding the derivative computations manually is too complex and error-prone. Specialized AD tools automate this process.

3.1 Definitions

To properly introduce the methods behind the two modes of AD, a formal description of the evaluation code is required. Basis for the differentiation methods is a vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ which is defined by an evaluation procedure that takes input variables $x_1, \ldots, x_n$ and returns the output variables $y_1, \ldots, y_m$. In the context of AD, the variables x_i are usually referred to as independent variables and the variables y_i are referred to as dependent variables. The evaluation code can then be expressed as a sequence of continuously differentiable elemental functions ϕ_i which compute a set of intermediate variables $v_1, v_2, \ldots, v_l$.

For the sake of simplicity, it is assumed that the evaluation function is a single-assignment code. In other words, each variable can only appear once on the left side of an assignment and may not be overwritten in subsequent operations. Each of these assignments has the following form:

$$v_i = \phi_i(v_j)_{j \prec i}$$

Here, ϕ_i is an elemental function with known derivative that takes a set of program variables v_j and computes a value that is subsequently stored in v_i. The relation $j \prec i$ expresses that v_i is directly dependent on v_j.

The full program can then be expressed with the following pattern:

$v_{i-n} = x_i$	$i = 1 \ldots n$
$v_i = \phi_i(v_j)_{j \prec i}$	$i = 1 \ldots l$
$y_{m-i} = v_{l-i}$	$i = m-1 \ldots 0$

In the following explanation of the forward and reverse mode, this scheme will serve as a base which will be extended with the derivative computations introduced by AD.

Both modes rely on the chain rule to propagate derivatives. Applying it to function F results in the following general derivative formula:

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial y_i}{\partial u_1}\frac{\partial u_1}{\partial u_2} \cdots \frac{\partial u_{k-1}}{\partial u_k}\frac{\partial u_k}{\partial x_j} \tag{3}$$

Here, $u_1, ..., u_k$ with $\{u_j\} \subseteq \{v_i\}$ correspond to the program variables that are involved in the computation of y_i.

3.2 The Forward Mode of AD

The fundamental idea behind the forward mode is that given initial derivatives for the independent variables, the chain rule can be applied to propagate derivative values, called tangents, alongside the function variables. To this end, for every variable v_i a new tangent variable $\dot{v}_i$ is introduced.

Using the chain rule from Equation 3, the derivatives are then evaluated starting at the innermost subexpression:

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial y_i}{\partial u_1}\left(\frac{\partial u_1}{\partial u_2}\cdots\left(\frac{\partial u_{k-1}}{\partial u_k}\left(\frac{\partial u_k}{\partial x_j}\right)\right)\right) \tag{4}$$

Following this rule, for each assignment in F a tangent update of the following form is added to the evaluation code:

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial \phi_i(v_j)_{j \prec i}}{\partial v_j} * \dot{v}_j \tag{5}$$

The resulting evaluation procedure including the derivative computations has the following structure:

$v_{i-n} = x_i$ $\dot{v}_{i-n} = \dot{x}_i$	$i = 1 \ldots n$
$v_i = \phi_i(v_j)_{j \prec i}$ $\dot{v}_i = \sum_{j \prec i} \frac{\partial \phi_i(v_j)_{j \prec i}}{\partial v_j} * \dot{v}_j$	$i = 1 \ldots l$
$y_{m-i} = v_{l-i}$ $\dot{y}_{m-i} = \dot{v}_{l-i}$	$i = m-1 \ldots 0$

The evaluation of the above code yields both the primary function value $y = F(x)$ as well as the tangent $\dot{y} = \dot{F}(x, \dot{x}) = F'(x)\dot{x}$.

Using this formulation, we can now compute any Jacobian-vector product by evaluating $\dot{F}$ with the appropriate initial values. Based on this, the partial derivative of F corresponding to the i-th column of the Jacobian $F'(x)$ can be computed with

$$\frac{\partial F(x)}{\partial x_i} = F'(x)_{*,i} = F'(x)e_i,$$

where e_i is the i-th Euclidian basis vector.

The process of initializing $\dot{x}$ is often referred to as *seeding*. In general, the evaluation of the whole Jacobian requires multiple runs with different values for $\dot{x}$. The details of how this can be achieved as well as optimization techniques are discussed in Section 3.6.

3.3 The Reverse Mode of AD

As presented in the previous Section, the forward mode of AD allows the computation of one partial derivative equating to a single column of the Jacobian $F'(x)$ during each evaluation of the derivative code. Similarly to the finite difference method, the run time of evaluating the whole Jacobian thus scales with the number of independent variables. If this number is very high, neither of these methods can be applied effectively.

In this case, the reverse mode of AD is often the only viable differentiation strategy. Because of the associativity of the chain rule, the derivatives terms can be evaluated in the opposite direction of the forward mode.

Applied to Equation 3, the outer subexpressions are evaluated first:

$$\frac{\partial y_i}{\partial x_j} = \left(\left(\left(\frac{\partial y_i}{\partial u_1}\right)\frac{\partial u_1}{\partial u_2}\right)\dots\frac{\partial u_{k-1}}{\partial u_k}\right)\frac{\partial u_k}{\partial x_j} \tag{6}$$

For the computation of these terms, the reverse mode is required to start with the evaluation of the dependent variables, propagating gradients instead of tangents. In order to apply the reverse mode to function F, for each variable v_j a so called adjoint $\bar{v}_j$ is introduced which holds a directional derivative of the dependent variables with respect to v_j. The adjoints are computed as follows:

$$\bar{v}_j = \sum_{i \succ j} \bar{v}_i \frac{\partial \phi_i(v_j)_{j \prec i}}{\partial v_j} \tag{7}$$

For an in depth derivation of this formula, please refer to [1].

The computation of the adjoints is performed in the direction opposite to the control flow of the original function, starting with the dependent variables. This however, means that it is impossible to evaluate the derivative alongside the original function, since the computation of the adjoints requires intermediate results that are evaluated later on in the code.

To solve this issue, the adjoint code is divided into two parts: In the initial *forward sweep*, the primary function values are computed and intermediate results are stored. Subsequently, the *reverse sweep* evaluates the adjoints using the recorded values from the forward sweep.

The general derivative code structure looks as follows:

$v_{i-n} = x_i$	$i = 1 \dots n$
$v_i = \phi_i(v_j)_{j \prec i}$	$i = 1 \dots l$
$y_{m-i} = v_{l-i}$	$i = m-1 \dots 0$
$\bar{v}_{l-i} = \bar{y}_{m-i}$	$i = m-1 \dots 0$
$\bar{v}_j = \sum_{i \succ j} \bar{v}_i \frac{\partial \phi_i(v_j)_{j \prec i}}{\partial v_j}$	$i = l-m \dots 1-n$
$x_i = v_{i-n}$	$i = n \dots 1$

The evaluation of the above code yields the gradient $\bar{x}^T = \bar{F}(x, \bar{y}) = \bar{y}^T F'(x)$. Similar to the forward mode, this can be used to compute rows of the Jacobian:

$$\nabla F(x)_i = F'(x)_{i,*} = e_i^T F'(x)$$

Consequently, the number of evaluation required for the construction of the whole Jacobian scales with the number of dependent variables. Functions with $n \gg m$ can thus be differentiated a lot more efficiently compared to the forward mode and the finite difference method.

To elaborate on the importance of this property, consider a scalar function $f : \mathbb{R}^{1000} \to \mathbb{R}$. The computation of the gradient $\nabla f(x)$ for input x with the forward mode then requires 1000 function evaluations. In contrast, only a single function evaluation is needed when the reverse mode is employed instead.

As this example demonstrates, the performance of the two modes is highly dependent on the underlying function. As a general rule, functions with more dependent than independent variables, that is $m \gg n$, should be differentiated using the forward mode. In the opposite case that $m \ll n$, the reverse mode should be used instead.

3.4 Overhead

In both forward and reverse mode, the computation of $\dot{F}$ and $\bar{F}$ respectively introduces an overhead compared to the original function F. This is caused by the additional operations that are required to update the tangents or adjoints in the respective mode. As a consequence, the cost of evaluating both the original function and its derivative is a small multiple of the original run time. More precisely,

$$TIME\{F(x), F'(x)\dot{x}\} \leq \omega_{tang} TIME\{F(x)\}$$

for the forward mode and

$$TIME\{F(x), \bar{y}F'(x)\} \leq \omega_{grad} TIME\{F(x)\}$$

for the reverse mode. According to [1], this constant is equal to $\omega_{tang} \in [2; 2.5]$ and $\omega_{grad} \in [3; 4]$. Here, the increased run time in the adjoint code is caused by additional memory movements. In practice, the overhead is greatly dependent on the optimality of the derivative code and can vary significantly for different AD tools.

Regarding memory consumption, the forward mode needs to keep track of additional variables to store the derivatives. As a result, the memory consumption approximately doubles compared to the original function.

The reverse mode introduces additional memory requirements due to the storage of intermediate variables and is therefore less memory efficient than the forward mode [17]. As a general rule, the memory requirements thus grow proportionally to the number of elemental operations in the function code, which can be an issue for large programs. Therefore, many current AD tools use checkpointing techniques [18], splitting up the adjoint code into multiple smaller parts.

3.5 Vector Mode

Thus far, the presented methods all use scalar valued derivative variables. Because of this, only one tangent or gradient can be computed during each function evaluation. In vector mode, the derivative variables are replaced with vectors of size p. Each of the vector entries are updated using the same differentiation rules introduced in Equation 5 and 7 for the forward and the reverse mode respectively. However, each of these entries propagates derivatives with independent seeding. In the forward mode, the vector $\dot{x}$ is replaced with a seed matrix $\dot{X} \in \mathbb{R}^{n \times p}$. As a result, the evaluation of the derivative code then computes the tangent matrix $\dot{Y} = F(x)\dot{X}, \dot{Y} \in \mathbb{R}^{m \times p}$. Similar modifications are made for the reverse mode. Here, $\bar{y}$ is replaced with the seed matrix $\bar{Y} \in \mathbb{R}^{m \times p}$. The derivative code then computes the gradient matrix $\bar{X}^T = \bar{Y}^T F'(x), \bar{X} \in \mathbb{R}^{n \times p}$. The vector mode can help reducing overhead when more than one derivative needs to be computed. In contrast to the scalar mode, where the function values are recalculated during each evaluation, the same intermediate function variables are used for multiple derivatives.

One drawback of the vector mode is the increased memory requirements imposed by the additional derivative variables. As a general rule, the memory consumption grows linearly with p [1].

3.6 Evaluating Whole Jacobians

For the sake of brevity, this section elaborates on Jacobian evaluation techniques based on forward mode AD only. The concepts introduced here are with small modifications equally applicable to the reverse mode.

As discussed in Section 3.2, the evaluation of the tangent-linear code yields the result of the Jacobian-vector product $\dot{y} = F'(x)\dot{x}$. Therefore, arbitrary directional derivatives can be computed without knowledge of the whole Jacobian $F'(x)$.

Nonetheless, some applications require the explicit evaluation of $F'(x)$. One example is the Newton method, were the linear system $F'(x) * \Delta x = -F(x)$ needs to be solved to find the appropriate Newton step.

The direct computation of $F'(x)$ without any optimizations requires one evaluation of the derivative code for each independent variable. For each of these variables x_i, the function is evaluated with the seeding $\dot{x} = e_i$. The resulting tangent $y' = \dot{F}(x, \dot{x})$ can then be used to fill the i-th column in the Jacobian.

For sparse Jacobians, it is possible to optimize this process by reducing the number of required function evaluations. This optimization is based on the fact that the computation of two columns $u = \frac{\partial F}{\partial x_i}$ and $v = \frac{\partial F}{\partial x_j}$ with no non-zero entries in the same row, that is $\nexists k : u_k \neq 0 \wedge v_k \neq 0$, is completely independent from each other. Columns with this property are called *structurally orthogonal*. This can be exploited in the seeding process by setting $\dot{x} = e_i + e_j$. Provided that it is known which entries are non-zero for each column, u and v can then be computed with a single evaluation of $\dot{F}(x, \dot{x})$. The same technique can be applied for multiple columns, as long as any two columns in the set are structurally orthogonal. This way, the computation of a Jacobian with disjoint column subsets $c_1, ..., c_l$ that each adhere to this property can be reduced from n to l function evaluations.

Applied to whole Jacobians, the goal is to find a seed matrix $S \in \{0,1\}^{n \times l}$ for the computation of $B = F'(x)S$ such that every nonzero entry of $F'(x)$ is present in B. Here, the matrix B is referred to as the compressed Jacobian. The optimization problem to find a minimal l is then equivalent to a NP-complete graph coloring problem [10]. The typical approach relies on heuristics which promise the computation of close-to optimal column partitionings within limited run time. Naturally, if the structure of the Jacobian is known beforehand, a fitting seed matrix can be determined manually instead.

To give an example, consider a Jacobian A with the following sparsity pattern:

$$\begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,4} \\ a_{2,1} & a_{2,2} & 0 & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix}$$

It can easily be seen that columns 1 and 3 as well as 2 and 4 are each structurally orthogonal. Hence, we can compute the compressed matrix for the partitioning $c_1 = \{1, 3\}, c_2 = \{2, 4\}$ as follows:

$$\begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,4} \\ a_{2,1} & a_{2,2} & 0 & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,4} \\ a_{2,1} & a_{2,2} \\ a_{3,3} & 0 \\ a_{4,3} & a_{4,4} \end{pmatrix}$$

For the reverse mode, the same principles apply with the only major difference that the roles of rows and columns are switched.

3.7 AD in Practice

In practice, tools that implement the principles of AD to existing program codes rely on one of the following approaches.

3.7.1 Source-to-Source Transformation

Derivative code compilers using source-to-source transformation analyze the original function evaluation code and insert the operations necessary for the derivative computation. During this process, an intermediate representation of the program code is generated. Because the derivative computations are added directly into the program, the compiler can perform additional optimizations.

Furthermore, the possibility to analyze the code in a bigger context allows the application of hierarchical approaches. Based on the computational graph representing the data dependency relations of the differentiated function[19], the code can be partitioned into substructures which allow the generation of a more efficient derivative code [20].

However, source-to-source transformation has the drawback of being difficult to implement for complex languages[13]. This is the reason why for C++, this approach is rarely used, as it is very difficult to cover the whole standard. Furthermore, the maintenance of the derivative code requires some additional work, since the derivative code must be updated whenever changes to the original function are made.

3.7.2 Operator Overloading

Some programming languages such as C++ provide a feature called operator overloading, which allows the developer to redefine common operators for custom data structures. AD tools that make use of this paradigm introduce so called active types which, in addition to the primary value, hold a second variable that is used to store the derivative. These active types provide implementations for the elemental functions required by AD. Besides emulating the behavior of the primitive type for the primary value, the overloaded functions update the derivatives according to the differentiation rules of the respective mode.

For the forward mode, this can be implemented in a fairly straightforward manner. For each operation that involves the active type, the secondary value, which in this mode is used to propagate the tangent, is updated according to Equation 5. After the evaluation, the output variables hold the derivatives of the function.

The reverse mode is more difficult to implement. Since the function evaluation must be split into a forward and a reverse sweep, the derivatives can not be updated at the same time as the primary variables. Consequently, a mechanism must be introduced to store the intermediate results that are required later. Usually, this is achieved by the use of a so called tape. Inside the differentiated code sections, the execution of each elemental operation causes the tape to store an entry that contains information about the type of operation and the involved variables. Outside of the active sections, the tape remains unchanged and only the primal values are updated. After the function has been executed, the information needed for the computation of the gradients is recorded. The tape can then be evaluated, which corresponds to the reverse sweep. During this process, the secondary variable of the active type is used to store the adjoints, which can now be computed based on the tape entries using Equation 7.

The operator overloading approach has some advantages as well as drawbacks compared to source-to-source transformation. Unfortunately, AD with operator overloading is not applicable to all languages, since some languages simply do not support this feature. Furthermore, the reliance on the active type necessitates a type change to replace the primitive type. This can lead to many problems that are hard to predict and may be difficult to solve. Finally, operator overloading applies AD on a statement level. Consequently, it is not possible to perform global optimizations that require a larger context. However, the fact that the original function code remains unchanged makes the target program easy to maintain after the initial type change. Furthermore, the operator overloading approach is not affected by the complexity of the language. As a consequence, new features in languages such as C++ can be used without the need to extend the AD tool.

Instead of using source-to-source transformation or operator overloading exclusively, some tools use a combination of both approaches to optimize performance and usability [14].

3.8 CoDiPack

CoDiPack is a relatively new tool written in C++ that focuses on the application of AD in HPC software [21]. It is based on operator overloading and provides implementations for both the forward and reverse mode of AD for first as well as higher-orders derivatives. In comparison to older tools, CoDiPack improves the performance of the adjoint evaluation by using expression templates, as proposed in [15]. Research

on the performance of CoDiPack on large-scale scientific applications shows that it performs significantly better than alternative tools [7]. For documentation and further information on the usage of CoDiPack, please refer to [22].

3.8.1 Expression Templates

Previous implementations of AD with operator overloading evaluate each statement independently [8]. For the scalar type T, binary expressions are then represented by functions

$$T \circ T \to T$$

For chained expressions with multiple operations, each individual operation is evaluated on its own. The computation of the expression

$$y = (sin(a+b) * (a-b))/c \tag{8}$$

therefore causes the creation of the following intermediate variables for the evaluation of each subexpression.

$$t_1 = a+b, \quad t_2 = sin(t_1), \quad t_3 = a-b, \quad t_4 = t_2 * t_3, \quad y = t_4/c$$

In CoDiPack, the operations are implemented in such a way that instead of the numeric result of the calculation, an expression structure is returned which stores the provided operands. Binary functions are therefore expressed with

$$E_A \circ E_B \to E_{A \circ B}$$

, where E_x is the expression type that holds information about x. For each overloaded operation, an expression structure is introduced that defines the way the primary value as well as the derivative is evaluated. Applied to the example in 8, CoDiPack constructs an expression tree with the following structure:

```
Div< Mult< Sin< Add<T, T> >, Sub<T, T> >
```

This large expression can then be evaluated in one go, avoiding the creation of additional temporaries. This is especially beneficial for the reverse mode because the number of tape entries can be severely reduced. For the given example, only a single entry in the tape has to be created, as opposed to 5 without the use of expression templates.

4 The Universal Laminar Flame Solver (ULF)

This section gives an introduction to the Universal Laminar Flame Solver (ULF).

ULF is a simulation framework developed at the chair of Numerical Thermo-Fluid Dynamics (NTFD) at the TU Bergakademie Freiberg. Its primary use is the simulation of chemically reacting, laminar flows on orthogonal grids. While currently limited to one-dimensional grids, in its final state ULF will eventually also be able to solve higher-dimensional problems. As a library, ULF can be integrated into other projects. However, in the scope of this research, we focus on the use of the software as a standalone tool.

Problems to be solved with ULF are specified in configuration files, which use similar syntax and structure as the popular CFD library *OpenFOAM*[23]. These files are highly customizable and allow the user to specify the complete problem without having to modify the code.

4.1 Solving Strategies and the Significance of Jacobians

ULF provides solution strategies for two basic problem types: steady and transient problems.

Steady problems are described by a system of differential algebraic equations (DAE) of the form

$$0 = f(x)$$

, where x is the state vector and f defines a residual function that has to be minimized in order to find the steady state solution.

For this thesis, we will focus on transient problems, which describe how the variables of a system change over time. These problems are based on a system of ODEs or PDEs of the form

$$\frac{dx}{dt} = f(x)$$

, but may also include DAEs to define boundary conditions. Following the convention employed by ULF, the term ODE will from here on used to refer to both ODEs and PDEs.

Transient problems are solved using time-stepping algorithms. For non-stiff equations, it is in some cases sufficient to apply explicit Runge-Kutta methods. For reacting gas mixtures, a subset of the equations describing the reaction rates of the species is typically very stiff. ULF therefore provides a number of more sophisticated solvers such as CVODE [24], which are applicable for those ODEs as well.

While the solver implementations differ in their overall strategy, most of them are based on a variant of Newton iteration, which requires the Jacobian of the system. Usually, this Jacobian does not have to be exact in order for the solver to converge, as long as the quality of the derivatives is “good enough”. Consequently, the number of Jacobian evaluations can be reduced by using an outdated version and only recomputing the derivatives when necessary. Nonetheless, for problems such as the homogeneous reactor, which is discussed in Section 7.4, the Jacobian may have to be recomputed hundreds of times during the solving process. Depending on the type of problem and the complexity of the equations, the Jacobian evaluation can thus make up a significant portion of the run time. For that reason, speeding up the procedure responsible for computing the Jacobian has the potential to significantly improve the overall performance.

Preceding the introduction of AD, ULF provided two different evaluation methods:

1. Numerical differentiation using finite differences

2. Analytical differentiation with TChem and pyJac

In the context of this research, we implemented additional evaluation strategies based on AD. Both the existing and the new implementations are described in detail in Section 6.

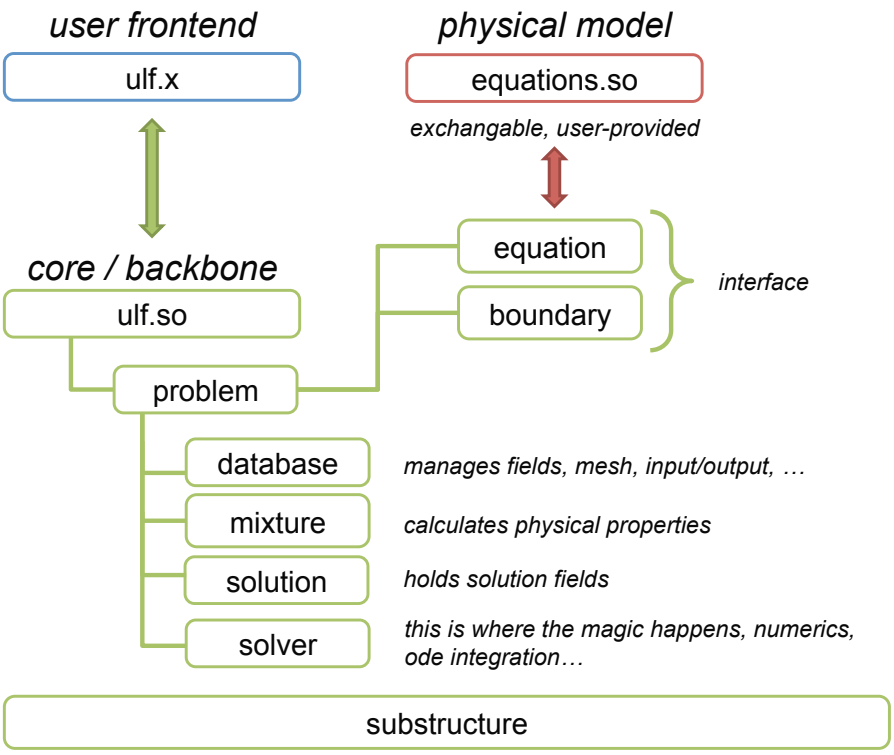

Figure 1: Overview of the Code Structure (from the ULF User Guide)

4.2 Design Principles

This sections explains important characteristics of ULF and gives an overview of the structure of the code base.

4.2.1 Modularity

One of the key design principles of ULF is modularity. As shown in Figure 1, ULF can be divided into three basic parts: The user frontend, the physical model and the core library.

ULF's core makes up the largest part of the code base. Here, the main functionality is implemented, which is responsible for interpreting configuration files, reading and writing data, allocating and initializing the data structures required by the problem and most importantly, controlling the solving process.

The executable `ulf.x` represents the front end of ULF and acts as an interface between the user and the core library. Aside from parsing the command-line options and instantiating the main `problem` object, this part of the code implements little functionality. The fact that everything required to solve a problem is contained in the core library makes it easy to integrate ULF into other projects.

ULF is not specialized on certain types of problems, but can solve arbitrary equation systems. It is not sensible to include the evaluation functions for the differential equations in the core library, as the addition of new functions would require the user to recompile the whole framework. Therefore, the physical model is provided in form of a shared library. The ULF core provides the classes `equation` and `boundaryCondition`, which allow easy access to the problem state from the equation library. As a result, the user does not need detailed knowledge of ULF's code base too extend the list of available equations and can do so without modifying ULF directly.

As a starting point, ULF provides a default library that contains implementations for common equations in thermo-fluid dynamics.

4.2.2 Extensibility

Because ULF aims to cover a wide variety of problem types, the principle of extensibility is important to facilitate future development.

An example for this is the implementations of solvers. A base class `solver` is implemented which defines virtual functions to control the solving process and access relevant data. Deriving from `solver`, several child classes are available that either directly implement a specific solving method or act as an interface to one of the external solvers.

A similar approach is chosen for the `mixture` class, which calculates the chemical properties of the gas mixture. Depending on the problem type, the mixture object must be able to handle the evaluation of chemistry and transportation coefficients. Frameworks like *Cantera* [25] and *EGLib* [26] provide well tested and highly optimized implementations for these computations. These libraries are integrated with subclasses of `mixture` that handle calling the external functions.

New solvers, mixtures and other subclasses can be easily made available to the user by adding an appropriate keyword and a reference to the class name to the parser of the configuration files. ULF will then choose the correct implementation based on the selected method in the problem specification.

4.3 Important Data Structures in ULF

The central data structure in the ULF code base is the `fieldData` class. It serves a a general vector type and is used to store all numeric data. In addition to the vector data, each `fieldData` object has a name. This can be used to identify different variables of the equation system that is being solved. The class supports a wide variety of arithmetic operations which simplify the implementation of equations. Recently, `fieldData` was extended to use template arguments that define the internal scalar and vector types. This is not only important for the integration of external math libraries like *Eigen*[27] and *Blaze*[28] which have their own specialized vector implementations, but allows the replacement of `double` with the respective CoDiPack types for AD.

The `field` class inherits from `fieldData` and stores additional information. While `fieldData` can be used to store any type of numeric data, `field` requires a `mesh` object for context, which determines the coordinate system on which the chemical reaction takes place. Problem variables such as temperature, pressure and the concentration of different chemical species are all represented by `field` objects. Similar to `fieldData`, `field` is a template class.

5 Introducing AD to ULF

This section describes the process of introducing AD to ULF and documents the issues that had to be resolved.

For the introduction of the operator overloading functionalities, CoDiPack provides specialized active types. In the parts of the code that are involved in the evaluation of the system functions, referred to as the active sections, they have to replace the primitive floating point types.

Different types are available for the various AD modes:

- `RealForward`: The active type for the forward mode
- `RealReverse`: The active type for the reverse mode
- `RealForwardVec<N>, RealReverseVec<N>`: Vector versions of the forward and reverse types that allow the simultaneous propagation of N derivatives

The introduction of the active type is typically achieved with a global type change. For C++, this means that a central `typedef` statement is introduced that defines the basic scalar type for the active code sections. With the help of preprocessor directives, the user can then choose a specific type when building the application.

The global type change necessitated several modifications to the the basic data structures of ULF. Before the introduction of the type change, new templated versions of these classes were implemented. While the author was not directly involved in these code changes, they represent an important prerequisite to the subsequent work and are therefore described in Section 5.2.

After the new data classes were in place and the type change had been performed, we had to resolve a list of issues caused by incompatibilities of the new types with the ULF code base. One of the major obstacles was the communication between ULF and external libraries, which required the implementation of a general type conversion mechanism. Other issues were caused by inconsistencies in the code style of ULF. These aspects are discussed in the following sections.

5.1 Code Style

ULF has been in development for several years.

While the software is written C++, a majority of the code base uses a “C with classes” style. Newer C++ features are generally avoided and C functions are often used in favor of the corresponding implementations in the C++ standard library, e.g. `malloc` and `free` as opposed to `new` and `delete` for dynamic memory management.

As the project evolved, the code was increasingly refactored to adopt a more modern coding style. Nonetheless, a lot of the early design decisions still have impact on the current version of ULF. This was one of the most challenging aspects of performing the type change, as the mixture of C and C++ conventions can lead to problems. During the introduction of AD, we had to pay special attention to the affected code parts.

One of these conventions is the use of `printf` instead of `std::cout`. This is especially relevant in the context of the assert system which utilizes this function very frequently. The modifications required to enable its use with the CoDiPack types is discussed in Section 5.5.

Another of these problematic code construct is the use of `memcpy` to transfer data for the use by external functions. For the types that we employed in ULF, the use of `memcpy` is allowed, as specified in the CoDiPack manual. However, this is only possible if the type of the source and destination data matches. Since this is not always the case, we had to integrate a suitable type conversion mechanism that is presented in Section 5.4.

5.2 Refactoring the Data Structures

As described in Section 4.3, ULF uses the `field` class and the underlying `fieldData` and as the primary containers for numerical data. Previously, these classes used `double` as the underlying scalar type and provided no ability to switch to user-defined types. Being able to compile ULF with different types provides value beyond just the use of AD. For example, this feature can be used to introduce `float` instead of `double` as the primitive floating point type. When the higher precision of `double` is not needed, this can be a way of reducing the memory requirements of the application. For this reason, even before the developments with respect to AD started, these classes were refactored to make use of templates.

In the current version of ULF, `field` and `fieldData` are parameterized by two template arguments: `_Scalar` and `_Vector`. `_Scalar` determines the type of the scalar primitive and replaces the `double` of the previous implementation. When introducing CoDiPack, this is where we applied the active type. `_Vector` sets the vector implementation that is used to store the raw data in the `fieldData` class. By default, the `std::vector` class of the standard library is used. The option to switch out the vector type is not directly relevant with respect to AD. However, this allows to use implementations of specialized math libraries which can improve the performance of vector operations.

Besides replacing the types, some additional changes to the methods that implement mathematical operations had to be made. For example, consider the following code that shows the `fieldData::sqrt()` method before the introduction of templates:

```
1 fieldData sqrt() const
2 {
3   fieldData temp = *this;
4   double* & values = temp.values_;
5   for(int i = 0; i < size_; i++)
6   {
7     values[i] = ::sqrt(values[i]);
8   }
9   return temp;
10 }
```

For comparison, this is the current implementation:

```
1 template <class _Vector, class _Scalar>
2 fieldDataTmpl<_Vector, _Scalar> fieldDataTmpl<_Vector, _Scalar>::sqrt() const
3 {
4     _Vector tmp(this->vector());
5     using std::sqrt; // Allows overloading
6     std::transform(std::begin(tmp), std::end(tmp), std::begin(tmp),
7                    [](_Scalar val) { return sqrt(val); });
8
9     return fieldDataTmpl<_Vector, _Scalar>(
10         std::move(tmp));
11 }
```

Both versions first create a temporary container to hold the result, which is then followed by the computation of the square root of each element in the vector. The result is then returned.

In the given example the following changes have been made:

- While the old implementation creates a new `fieldData` object that operates directly on a raw `double` array, the new function uses the `_Vector` type instead
- The C-style math operation `::sqrt()` has been replaced by a call that allows the use of custom types. The addition of `using std::sqrt` in line 5 is needed in order for the compiler to find the standard implementation for primitive types.

- Instead of `for`-loops, the new implementation makes use of `std::transform` and lambda functions.
- The new version uses `std::move` to efficiently transfer the data to the result object.

Similar modifications were made for all mathematical operations.

5.3 Performing the Global Type Change

The global type change was performed in two stages: First, we introduced the global type `ulfScalar` and configured the basic data containers to use it. Secondly, we worked through the ULF code base and replaced occurrences of numeric primitives with this new type.

5.3.1 Introducing `ulfScalar`

In order to allow the user to specify the global scalar type, we introduced the CMake options `USE_AD` and `AD_REVERSE_MODE`. When active, these options cause the definition of a preprocessor macro of the same name.

As an entry point for the type change, we created the `typedefScalar.h` header. This file introduces the type `ulfScalar` which in the current version of ULF, is used as the default floating point type and replaces `double` as the `_Scalar` template argument for the `field` and `fieldData` implementations.

A shortened version of the code that handles setting the correct type looks as follows:

```
1  #ifdef USE_AD
2
3  #include "codi_extensions.h"
4
5  namespace ulf {
6  #ifdef AD_REVERSE_MODE
7  // Scalar values are represented by the CoDiPack type for reverse mode AD
8  using ulfScalar = codi::RealReverse;
9  #else // AD_REVERSE_MODE
10 // Scalar values are represented by the CoDiPack type for forward mode AD
11 using ulfScalar = codi::RealForward;
12 #endif // AD_REVERSE_MODE
13 }
14
15 #else // USE_AD
16
17 namespace ulf
18 {
19 // Scalar values are represented by doubles
20 using ulfScalar = double;
21 }
22
23 #endif // USE_AD
```

If `USE_AD` is defined, first the CoDiPack headers are included in line 3. Subsequently, the preprocessor checks if the reverse mode should be used. Based on the selected mode, `ulfScalar` is then set to the correct type provided by CoDiPack. In the case that ULF is compiled without AD, `ulfScalar` is defined as `double` instead.

5.3.2 Replacing Occurrences of `double`

Following the introduction of the global scalar type, we had to work through the ULF code base and replace occurrences of floating point types in relevant sections of the code.

For this taks, we classified ULF's code into the following groups:

1. Passive code sections
2. Active code section
3. Active code sections calling non-differentiated external functions

Determining which parts of the code base belongs to which category was not an easy task, as it required detailed knowledge of the inner workings of ULF.

For category 1, usually no modifications were required. This group includes classes such as `ulf::matrix` which is used to store certain chemical properties, but is never directly involved in any computations during the function evaluation. Only a minority of the classes fell under this category.

Regarding category 2, the type change was usually accomplished with a search and replace approach, although in most cases, this had to be followed up with a manual inspection of the changes in order to resolve minor issues. For example, the direct usage of `printf` requires the conversion to primitive types, as discussed in Section 5.5.

Category 3 was more involved in comparison. Classes and functions that belong to this category communicate with external libraries, which typically operate on built-in floating point types. For these code sections, we first identified the parts of the code that could be included in the type change. In a second step, we converted the `ulfScalar` variables to `doubles` in order to supply the correct input for the function calls. Similarly, the results of the function calls are converted back to `ulfScalar` after completion.

5.4 Solving Type Compatibility Issues

As illustrated in the previous section, not all parts of the code base operate on the AD type. The passive code sections are not completely isolated and need to communicate with the active code parts to exchange data. In order to ensure compatibility, we had to find a strategy that allows for the conversion between the involved types.

As a general approach to these compatibility issues, a template function was introduced with a default implementation to solve the specific issue for the primal build. Using partial template specialization, we then added dedicated implementations for the CoDiPack types. With this approach, the same function can be used regardless of the actual type of `ulfScalar`, as the compiler automatically selects the right implementation in each case. Not only does this method allow the integration of CoDiPack, but ULF can easily be extended to use additional types, as only these specialized template functions have to reimplemented. This opens up the possibility for future work involving additional types, such as the integration of an alternative AD tool.

5.4.1 Extracting Primal Values from the Active Type

The transfer of data between active and passive sections generally requires the conversion from one type to another. For CoDiPack, the conversion from a primitive to the active type happens implicitly. However, extracting the primal double value from an active variable `x` requires an API call to the function `getValue()`.

The simplest approach is to insert the `getValue()` calls directly into the code whenever the `double` value of an `ulfScalar` is needed.

This however, works only when the CoDiPack types are always enabled, as illustrated by the following example:

```
1  int convert_to_int(ulfScalar x)
2  {
3    int x_int = (int) x.getValue(); // Error, if ulfScalar = double
4    return x_int;
5  }
```

Since `getValue()` is not defined for primitive types, this leads to a compile time error in the primal build. To resolve this issue, we introduced a general casting method that is based on the presented template specialization technique.

To this end, we implemented a templated `value()` function which in the default implementation simply returns the provided argument. A specialized implementation for CoDiPack types invokes `getValue()` in order to extract the primal floating point value. For convenience, we also implemented a `recast` function that first retrieves the `double` value of the given variable and then uses a `static_cast` to convert the resulting `double` to the desired type. The full code is provided in Appendix A. These new casting functions establish the base for all other conversion operations.

Applying `recast` to the sample code from above then results in the following code, which works for both primitive and active types:

```
1  int convert_to_int(ulfScalar x)
2  {
3    int x_int = recast<int>(x);
4    return x_int;
5  }
```

5.4.2 Copying Data with `memcpy`

At various places in the code base, ULF makes use of `memcpy` to copy arrays of primitive data types. According to the documentation of CoDiPack, all of the active types that are used in ULF support this operation. Consequently, applications of `memcpy` as in the following example require no further modifications.

```
1    void copy(ulfScalar* from, ulfScalar* to, unsigned int size)
2    {
3      memcpy(to, from, size); // Works with both double and active types
4    }
```

However, in some cases `memcpy` is used to transfer data between active and passive code sections:

```
1    void copy_active(ulfScalar* from, double* to, unsigned int size)
2    {
3      memcpy(to, from, size); // Incorrect behavior if ulfScalar != double
4    }
```

Here, relying on the default `memcpy` implementation leads to incorrect results, as the active types store an additional `double` value to hold the derivative.

Our solution, again, uses the partial template specialization approach. We introduced the function `ulf_memcpy`, which in the default implementation copies each entry individually using a `for` loop. For the correct type conversion, we use the previously introduced `recast` function. For calls of `ulf_memcpy` with identical source and target types, we added a specialized implementation which simply invokes `memcpy` without any modifications. This way, we ensure the correct conversion of all scalar types, while simultaneously avoiding any unnecessary casting operations. The corresponding source code is provided in Appendix B.

5.5 Enabling active types in the assert system

ULF utitlizes `printf` instead of `std::cout` to write to standard output. The format specifiers of `printf` do not support user-defined types. Calling `printf` with the active types introduced by CoDiPack thus results in an error, as illustrated by the following example:

```
1  void print_first(const field& T)
2  {
3    ulfScalar t0 = T[0];
```

```
4     printf("T[0]=%f\n", t0); // Error, if ulfScalar != double
5   }
```

A solution to this problem is to introduce a `recast`, as described in the previous section:

```
1   void print_first(const field& T)
2   {
3     double t0 = recast<double>(T[0]);
4     printf("T[0]=%f\n", t0);
5   }
```

Direct calls to `printf` in the release build of ULF are relatively rare. As such, the `recast` calls were added manually.

However, ULF employs a macro based assert system, that is used in thousands of code locations. These assert macros use `printf` to output a message to the user in case of an error. Because the number and type of the format arguments varies for each use, it is impossible to handle the casting of the active types directly in the macro definition. At the same time, performing the casting by hand before every call to the assert system would have taken a considerable amount of time and effort. For this reason, a `printf` wrapper was developed which can handle user-defined types. We then replaced every `printf` instance in the affected assert macros with a call to this new function.

Our implementation is based on variadic templates which were introduced in C++11 and allow the developer to specify templates with varying numbers of arguments. We used this feature to write a templated `convert_printf` function, which transforms every argument with a call to `convert` before passing them to `printf`. Depending on the type of the argument, `convert` performs different operations. Primitive arguments are directly returned, as there is no need for conversion. For complex types, `convert` applies a `recast<double>` in order to extract the underlying floating point value. The full implementation of `convert_printf` is included in Appendix C.

As a result of the automatic type conversion, the developer is able to treat complex types the same as `doubles` when writing assertion code. Since `convert_printf` makes use of the `recast` system, integrating new types into the assert system can simply be done by implementing a corresponding recast function.

One drawback of this method is that the performance might be worse compared to the manual casting approach due to additional function invocations. However, in our case this is not an issue because all assert statements are removed in the release build.

6 Jacobian Evaluation in ULF

The previous section focused on the steps that were necessary to integrate CoDiPack into ULF. In the following, we present our evaluation methods for Jacobians using the newly introduced AD functionalities.

For the evaluation and storage of derivatives, ULF implements a `jacobian` class. The solver object holds an instance of `jacobian` and can trigger its reevaluation during the solving process whenever the current derivatives are outdated. To do this, the `jacobian` stores a function pointer `eval_` which is configured based on the problem file to invoke the selected evaluation method. The function signature looks as follows:

```
1  void (jacobian::*eval_)(double t, fieldData* x, fieldData* residual, double rdt);
```

Here, `t` is the current solver time, `x` is the state vector of the system, `residual` is the residual vector of the solver and `rdt` is the reciprocal of the time step. The `rdt` parameter is not used in the current implementations and is not relevant for the presented methods.

Central part of each evaluation method is the call to the differentiated function, which in this case computes the differential equations based on the given state vector. In ULF, this function has the following signature:

```
1  int computeResiduals(double t, fieldData* x, fieldData* residual);
```

The function takes the current solver time `t`, the state vector `x` and a pointer to the result vector `residual`. After the equations have been evaluated, the results are then stored in `residual` and a status flag is returned. In the context of AD, `x` thus holds the independent variables and `residual` holds the dependent variables. Consequently, these are the variables whose derivatives values need to be initialized during seeding and hold the entries for the Jacobian after the evaluation.

6.1 Jacobian Characteristics

In ULF, the computations of the differential equations at grid point i are typically dependent on

1. The temperature T_i
2. The mass fractions $Y_{i,j}$ for species $1 \dots k$

The state vector of the system is then made up of a concatenation of these variables for all points in the mesh.

A problem configuration involving the reaction of k species on a mesh of n points thus results in the state vector

$$X = (X_1, \dots, X_n),$$

where

$$X_i = (T_i, Y_{i,1}, \dots, Y_{i,k}).$$

Because there is exactly one differential equations for each variable in the state vector describing the behavior of that variable over time, the evaluation of this system is performed with a function $F : \mathbb{R}^N \to \mathbb{R}^N$, where $N = n * (k+1)$ is equal to the number of independent as well as the number of dependent variables. Therefore, the Jacobian $J_F = F'(X) \in \mathbb{R}^{N \times N}$ corresponding to the partial derivatives of F with respect to the state vector X is a square matrix.

Currently, ULF only supports 1-dimensional grids. Because of this, the equations governing the behavior of the variables at point i are only dependent on the state of the point itself as well as its neighbors X_{i-1} and X_{i+1}. All other points have no influence on these equations. Therefore, the derivatives for point i with respect to variables from non-adjacent points are always zero. As a consequence, J_F has a tridiagonal block structure, as displayed in Figure 2.

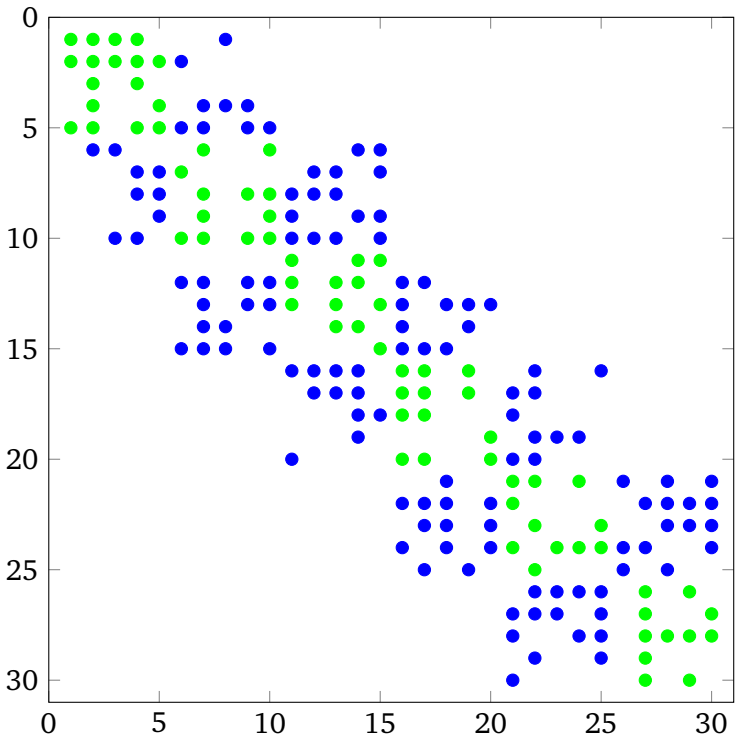

Figure 2: Example for the sparsity pattern of ULF Jacobians

Each block row corresponds to the derivatives of one grid point. The entries of the central block of each triple, displayed in green, are determined by the chemistry of the system, as they contain the derivatives of the reaction rates with respect to the species concentrations inside each point. The neighboring blocks, displayed in blue, are determined by the transportation of mass and energy between adjacent points. For point i, the left block describes the influence of point $i-1$, while the right block describes the influence of point $i+1$. Typically, these derivatives are controlled by convection and diffusion effects.

6.2 Compression Techniques

Because the block structure described in the previous section is the same regardless of the specific problem, the Jacobian compression techniques introduced in Section 3.6 can be applied.

It is easy to see that the columns corresponding to the pairings of grid points (1, 4), (1, 7), (2, 5) and so on fulfill the conditions for structural orthogonality since they are guaranteed to have no rows with overlapping non-zero entries. Consequently, the derivatives of points which have no common neighbor, that is they are three points apart, can be computed simultaneously. The corresponding column partitioning for a Jacobian with block size $m = k+1$ (k being the number of species) and n grid points is then defined by

$$c_{1,i} = \{i + m*3*j \mid j = 0, ..., \lfloor \frac{n-1}{3} \rfloor\},$$
$$c_{2,i} = \{i + (m+1)*3*j \mid j = 0, ..., \lfloor \frac{n-2}{3} \rfloor\},$$
$$c_{3,i} = \{i + (m+2)*3*j \mid j = 0, ..., \lfloor \frac{n-3}{3} \rfloor\},$$

with $i = 1, \ldots, m$.

This optimization reduces the number of required function evaluations from $n*m$ down to $3*m$. Consequently, the run time complexity of the Jacobian evaluation is largely unaffected by the size of the mesh.

Applying the corresponding seeding on the example Jacobian from above yields the following compressed Jacobian:

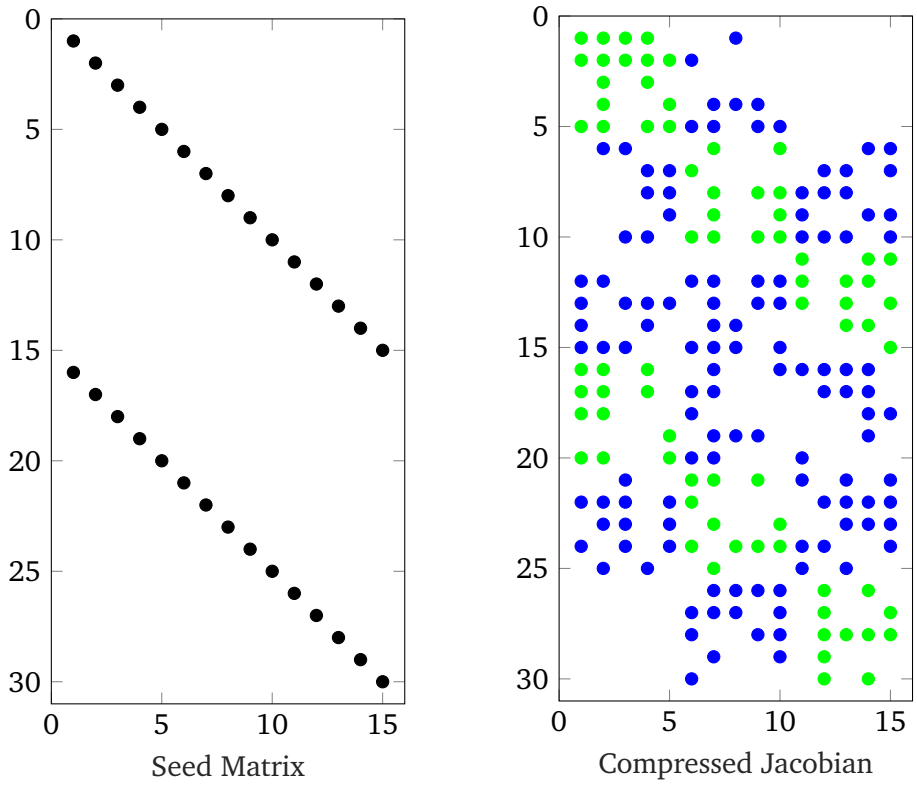

Figure 3: Compression of the example Jacobian

Because of the simplicity of the seed matrix, explicitly solving the equation system $B = J_F \cdot S$ is not necessary. The uncompressed Jacobian J_F can easily be constructed by copying the entries of each column to their respective location. As such, our implementation does not explicitly store B. The derivatives are written directly to the main Jacobian after each tangent evaluation.

The described compression technique is independent of the actual evaluation method and therefore applicable for both the evaluation with finite differences and with forward mode AD. For the reverse mode, the seeding is performed with regard to the transposed Jacobian J_F^T. Because the block structure of the Jacobian is symmetrical, the approach of the forward mode can be used without any modifications.

6.3 Existing Jacobian Evaluation Methods

Prior to the implementation of AD, ULF provided three different Jacobian evaluation methods:

1. Numerical evaluation using finite differences
2. Analytical evaluation with pyJac
3. Analytical evaluation with TChem

6.3.1 Numerical Evaluation

For the numerical Jacobian computation, ULF employs a simple finite difference method based on Equation 1.

For the independent variables X_i, the procedure looks as follows:

(1) Choose an appropriate step-size in order to minimize truncation and rounding errors. Save for each mesh point.

(2) Perturb the entries of the input vector belonging to independent points with chosen step size.

(3) Evaluate the equation system.

(4) Compute the derivatives of the equation of the selected points and their neighbors with respect to X_i using the forward difference quotient.

(5) Write the results to the corresponding columns of the Jacobian.

Figure 4 shows the corresponding function code. In line 7, `J_`, the variable which stores the Jacobian matrix, is reset so that previous evaluations have no influence on the result. Subsequently, the variable `rdx_` is declared, which is used to store the inverse of the perturbation for each solution entry.

The variable `stencilSize` is set to 3 by default and corresponds to distance between points that can be evaluated simultaneously. This means that the loop in line 15 is able to processes all mesh points in three iterations.

The subsequent loop iterates over all input variables. For each entry, the temporary solution vector is first reset in line 23. Then, every entry of the currently selected points in the solution vector is perturbed using a value based on a relative tolerance to minimize rounding and truncation errors. The inverse of this value is stored afterwards in line 30, for further use in the computation of the difference quotient.

The subsequent call to `computeResiduals(t, xtmp, rtmp_)` in line 34 causes the evaluation of the equation system using the perturbed input values of `xtmp`. The results are written to `rtmp_`.

The block from line 37 onwards performs the computation of the forward difference quotient. Here, the outermost loop in line 37 iterates over the points which have been perturbed in the current iteration. For every of the corresponding blocks in the Jacobian, one column is filled, corresponding to the derivatives of the equations with respect to the perturbed input variable. Line 43 performs the computation of the finite difference quotient. After computing each entry, a `recast` is applied, because the underlying matrix class operates on `doubles`.

```
1  void ulf::jacobian::evalFDJac(double t, fieldData* x0, fieldData* residual)
2  {
3      using std::fabs;
4
5      // reset jacobian
6      J_->resetDecomposition();
7      J_->fill(0.0);
8
9      int k, l, n, m, ipt = 0, iloc; // index variables
10     double dx;                     // perturbation variable
11
12     fieldData rdx_ = fieldData(size_, 0.0);
13
14     // construct jacobian by numerical perturbation of the solution vector
15     for (k = 0; k < stencilSize_;
16          k++) // we need stencilSize_ runs, perturb only independent columns
17     {
18         for (n = 0; n < nSolutions_; n++) {
19             // perturbation, initially all zero
20             rdx_ = (ulfScalar)0.0;
21
22             // reset temporary solution vector
23             *xtmp_ = *x0;
24
25             // perturb every third column in xtmp_
26             for (l = k; l < nPoints_; l = l + stencilSize_) {
```

```
27                  ipt = l * nSolutions_ + n;
28                  dx = AbsTol_ + std::fabs(recast<double>((*xtmp_)[ipt])) * RelTol_;
29                  (*xtmp_)[ipt] = (*x0)[ipt] + dx;
30                  rdx_[ipt] = 1.0 / dx;
31              }
32
33              // calculate perturbed residual
34              computeResiduals(t, xtmp_, rtmp_);
35
36              // compute columns of Jacobian
37              for (l = k; l < nPoints_; l = l + stencilSize_) {
38                  ipt = l * nSolutions_ + n;
39                  for (int i = l - nDepNeighbors_; i <= l + nDepNeighbors_; i++) {
40                      if (i >= 0 && i < nPoints_) {
41                          iloc = i * nSolutions_;
42                          for (m = 0; m < nSolutions_; m++) {
43                              ulf::ulfScalar temp =
44                                  ((*rtmp_)[m + iloc] - (*residual)[m + iloc]) * rdx_[ipt
                                        ];
45                              (*J_)(m + iloc, ipt) = recast<double>(temp);
46                          }
47                      }
48                  }
49              }
50          }
51      }
52
53      for (n = 0; n < size_; n++) {
54          (*ssdiag_)[n] = (*J_)(n, n);
55      }
56
57      age_ = 0;
58  }
```

Figure 4: The code of the function that evaluates the numerical Jacobian

6.3.2 Analytical Evaluation

Numerical methods provide a way to approximate the Jacobians of arbitrary systems of differential equations and are thus universally applicable to every problem type. However, the computation of the finite differences requires the repeated evaluation of the underlying functions, which can be costly.

Analytical differentiation, unlike the finite difference method, relies on knowledge about the differentiated function to compute derivatives with high accuracy. Because they do not require the evaluation of the function, analytical approaches can be more efficient than numerical methods, especially when the computation of multiple partial derivatives is required.

The downside of this approach is that it requires detailed knowledge of the system. For very simple functions, a generalized symbolic approach may be sufficient. With growing complexity, however, this becomes increasingly inefficient. Therefore, the analytical differentiation procedures have to be implemented specifically for each system. To do this, the targeted function must first be differentiated by hand, which can be difficult.

In the context of flame simulation, part of the system's behavior is governed by the equations that describe the chemical reactions of the gas mixture. Because the underlying chemistry behaves identical regardless of the specific problem, these equations can be differentiated with the same analytical methods.

ULF integrates pyJac[5] and TChem[6], two libraries that implement analytical differentiation. pyJac is a Python-based project that can be used to generate specialized C or CUDA source code for analytical

Jacobian evaluation. pyJac bases the derivative code on a reaction mechanism which is provided in a specific file. Using the information about the reactions specified in the mechanism file, pyJac generates efficient Jacobian evaluation code in the selected target language. Since this code is specific to the involved reactions, it is not possible to apply the same code to problems using different mechanisms. Thus, switching between multiple mechanisms while using pyJac can be tedious.

TChem is a toolkit for numerical simulations of chemical reaction that provides functions to compute analytical Jacobians. In contrast to pyJac, TChem is written in C and computes the derivatives directly, without the intermediate step of generating specialized code. This allows the user to switch between mechanism dynamically and thus makes the usage with multiple problem configurations easier.

As illustrated, analytical approaches have the general limitation of only working for specific functions. In the context of ULF, the analytical differentiation with TChem and pyJac is limited to the equations describing the chemical reactions, which make up the central blocks of the Jacobian displayed in Figure 2.

However, most real-world problems additionally take diffusion and convection into account. These effects govern the blocks left and right of the central chemistry block in the Jacobian. For different types of problems, the transport terms that describe these effects may not be the same and can thus not be covered by a general analytical differentiation code.

Therefore, in most cases ULF can not rely on the analytical approach alone. Any problem that includes transport terms is thus treated with a two-step approach:

1. Compute the central blocks of the Jacobian, determined by the system's chemistry, analytically

2. Use numerical methods to differentiate the transport terms, filling the entries left and right of the central chemistry block

In the second step, ULF sets a flag that causes the system to only evaluate transport terms, thus avoiding the unnecessary overhead of evaluation the chemistry again.

6.4 Jacobian Evaluation with AD

Following the global type change, we added new implementations for the Jacobian evaluation based on AD.

As explained in Section 5.3.1, the type of `ulfScalar` is determined by the CMake options `USE_AD`, `AD_REVERSE_MODE` and `AD_VECTOR_MODE`. The CoDiPack types can only be used for one respective mode, which means that the AD mode cannot be changed at run time and is fixed for each build configuration. Based on the specified type, the `eval_` function pointer is automatically set to the appropriate evaluation function.

The following table lists the implemented evaluation functions with their respective `ulfScalar` type and CMake options.

AD Mode	`AD_REVERSE_MODE`	`AD_VECTOR_MODE`	Type of `ulfScalar`
Scalar Forward Mode	`OFF`	`OFF`	`codi::RealForward`
Scalar Reverse Mode	`ON`	`OFF`	`codi::RealReverse`
Vector Forward Mode	`OFF`	`ON`	`codi::RealForwardVec<AD_VECTOR_DIM>`
Vector Reverse Mode	`ON`	`ON`	`codi::RealReverseVec<AD_VECTOR_DIM>`

Table 1: List of the supported AD modes and the corresponding active types.

In vector mode, the parameter `AD_VECTOR_DIM` controls the number of the derivatives that are propagated simultaneously.

6.4.1 Forward AD Algorithm

The evaluation functions for the finite difference method and forward mode AD both compute the Jacobian in a column-wise fashion, evaluating the equation system once for each independent variable. As a consequence, these functions are structurally very similar, with two major differences:

1. The perturbation of the independent variables is replaced by the seeding process
2. Instead of calculating the finite difference quotient, the derivatives are directly harvested from the tangents of the dependent variables

The code of the evaluation function is shown in Figure 5. Again, the derivatives of points with no common neighbors are computed simultaneously. The corresponding seeding process is handled in line 17. Here, the derivative value of the currently differentiated dependent variable is initialized to `1.0` for each of the points. This corresponds to one column of the compressed seed matrix displayed in Figure 3.

The call `computeResiduals(t, xtmp_, rtmp_)` in line 21 subsequently invokes the evaluation of the system equations with input variables `xtmp_`, with the results stored in `rtmp_`. The tangents contained in `rtmp_` then equate to one column of the compressed Jacobian. The code from line 22 to 33 collects these entries and writes them to the correct location in the matrix `J_`. Note that the `if` statement in line 25 is required in order to correctly handle the points at the borders of the mesh, as they only have one neighbor each.

```
1  void ulf::jacobian::evalAD(double t, fieldData* y, fieldData* residual, double rdt)
2  {
3      problem_->setJacobianUpdateType(STD_JAC_UPDATE);
4
5      nEvals_++;
6
7      J_->resetDecomposition();
8      J_->fill(0.0); // reset jacobian
9
10     *xtmp_ = *y;
11
12     int k, i, j, n, l;
13
14     // Derivatives of independent points are computed simultaneously
15     for(k = 0; k < stencilSize_; k++) {
16         for (n = 0; n < nSolutions_; n++) {
17             for (i = k; i < nPoints_; i += stencilSize_) {
18                 // Set derivatives of current input variables to 1.0
19                 (*xtmp_)[i * nSolutions_ + n].setGradient(1.0);
20             }
21             computeResiduals(t, xtmp_, rtmp_);
22             for (i = k; i < nPoints_; i += stencilSize_) {
23                 for (j = i - nDepNeighbors_; j <= i + nDepNeighbors_; j++) {
24                     if (j >= 0 && j < nPoints_) {
25                         for (l = 0; l < nSolutions_; l++) {
26                             double g = (*rtmp_)[j * nSolutions_ + l].getGradient();
27                             (*J_)(j * nSolutions_ + l, i * nSolutions_ + n) = g;
28                         }
29                     }
30                 }
31             }
32             for (i = k; i < nPoints_; i += stencilSize_) {
33                 // Reset derivatives
34                 (*xtmp_)[i * nSolutions_ + n].setGradient(0.0);
35             }
36         }
37
```

```
38      }
39
40      for (n = 0; n < size_; n++) {
41          (*ssdiag_)[n] = (*J_)(n, n);
42      }
43      age_ = 0;
44
45      problem_->setJacobianUpdateType(NO_JAC_UPDATE);
46
47  }
```

Figure 5: The code of the evaluation function corresponding to the forward mode of AD

6.4.2 Reverse AD Algorithm

In contrast to both the finite difference and the forward evaluation method, the evaluation method for reverse mode AD fills the Jacobian in a row-wise fashion. This is caused by the fact that reverse mode AD computes gradients as opposed to tangents. However, the seeding is essentially the same as for the forward mode due to the symmetry of the block-structure of the Jacobian.

Figure 6 shows our implementation. In line 12 and 13 the global tape is fetched and reset. This is done in order to delete the records from previous evaluations. We do not have to worry about deleting data that is required elsewhere, because this function is currently the only place where the tape is accessed.

The code from lines 17 to 30 handles the forward sweep. First, `tape.setActive()` is called, which causes CoDiPack to record all following computations on the tape. Subsequently, all input variables are registered with calls to `tape.registerIput()`. After the equations are evaluated with `computeResiduals(t, xtmp_, rtmp_)`, the results of the computation, now stored in `rtmp_`, are registered as output variables in a similar fashion with `tape.registerOutput()`. All intermediate results are now stored on the tape, which is then returned to the passive state again, concluding the forward sweep.

In the subsequent reverse sweep, the gradients of the dependent variables are first initialized according to the seeding scheme shown in Figure. 3. The following call to `tape.evaluate()` performs the computation of the adjoints. In this step, CoDiPack iterates over the tape entries in reverse order, updating the gradients of every variable appearing on the right side of a recorded operation. After this process is complete, the derivatives are stored in the gradient data of the independent variables. Similar to the forward mode evaluation function, the code from lines 42 to 51 then collects these derivatives and inserts them into the correct location in the Jacobian. A subtle difference is the fact that here, the matrix coordinates are switched around, since the Jacobian is computed row-wise instead of column-wise. After each tape evaluation and copying of the derivatives to the Jacobian, `tape.clearAdjoints()` is called. This step ensures that the adjoints of the previous evaluation have no impact on the subsequent computations.

```
1  void ulf::jacobian::evalAD(double t, fieldData* y, fieldData* residual, double rdt)
2  {
3      problem_->setJacobianUpdateType(STD_JAC_UPDATE);
4
5      nEvals_++;
6
7      J_->resetDecomposition();
8      J_->fill(0.0); // reset jacobian
9
10     *xtmp_ = *y;
11
12     codi::RealReverse::TapeType& tape = codi::RealReverse::getGlobalTape();
13     tape.reset();
14
15     int k, i, j, n, l;
16
```

```
17      tape.setActive();
18      // Register input variables
19      for (i = 0; i < nPoints_ * nSolutions_; i++) {
20          tape.registerInput((*xtmp_)[i]);
21      }
22
23      // Compute RHS with active tape
24      computeResiduals(t, xtmp_, rtmp_);
25
26      // Register output variables
27      for (i = 0; i < nPoints_ * nSolutions_; i++) {
28          tape.registerOutput((*rtmp_)[i]);
29      }
30      tape.setPassive();
31
32      // Derivatives of independent points are computed simultaneously
33      for(k = 0; k < stencilSize_; k++) {
34          for (n = 0; n < nSolutions_; n++) {
35              // Seeding
36              for (i = k; i < nPoints_; i += stencilSize_) {
37                  (*rtmp_)[i * nSolutions_ + n].setGradient(1.0);
38              }
39              // Tape evaluation
40              tape.evaluate();
41              // Retrieve derivatives and write to jacobian
42              for (i = k; i < nPoints_; i += stencilSize_) {
43                  for (j = i - nDepNeighbors_; j <= i + nDepNeighbors_; j++) {
44                      if (j >= 0 && j < nPoints_) {
45                          for (l = 0; l < nSolutions_; l++) {
46                              double g = (*xtmp_)[j * nSolutions_ + l].getGradient();
47                              (*J_)(i * nSolutions_ + n, j * nSolutions_ + l) = g;
48                          }
49                      }
50                  }
51              }
52              // Clear adjoints
53              tape.clearAdjoints();
54          }
55
56      }
57
58      for (n = 0; n < size_; n++) {
59          (*ssdiag_)[n] = (*J_)(n, n);
60      }
61      age_ = 0;
62
63      problem_->setJacobianUpdateType(NO_JAC_UPDATE);
64  }
```

Figure 6: The code of the evaluation function corresponding to the reverse mode of AD

6.4.3 Vector mode AD

The Jacobian evaluation methods from above can be modified to work with the vector mode introduced in Section 3.5. Instead of propagating a single derivative alongside the primary value, this mode uses vector-valued derivatives of arbitrary size.

The vector mode is activated with the option `AD_VECTOR_MODE` and allows the user to specify the vector dimension in `AD_VECTOR_DIM`. It should be noted that the CoDiPack vector types are only compatible with

types of the same size. Mixing vector types of different sizes will lead to errors during compilation. The vector size of `ulfScalar` is therefore fixed and can not be changed at run time.

Technically, it is possible to compute large Jacobians with a single function evalatution, provided the vector dimension is chosen appropriately. However, there are also downsides to employing large derivative vectors.

When AD is applied with a global type change, a bigger vector size does not always equate to better performance. If the vector size is selected too big, overall performance may actually decrease. Although the evaluation of the Jacobian will get faster due to the reduced number of function evaluations, the code outside of the differentiated function operating on the active type will run slower, because additional operations are required to update all of the derivative values. Therefore the optimal vector size depends on the type of problem and performance requirements and must be chosen accordingly. The concrete effects of the vector mode on model problems is discussed in Section 7.

Listing 6.4.3 displays the code performing the Jacobian evaluation in reverse mode using derivative vectors. The modifications needed to make the scalar version work with vector types are very similar in the forward mode. Therefore, only the vector version of the reverse mode is discussed here.

The section of the code which is responsible for the forward sweep is virtually identical to the scalar version. The first real difference occurs at line 34. Here, instead of handling one dependent variable at a time, multiple gradients are computed simultaneously.

Different problem configurations operate on different numbers of variables and thus lead to varying Jacobian dimensions. As `AD_VECTOR_DIM` is fixed at run time, it can not be guaranteed that the Jacobian dimension is a multiple of the vector size. Consequently, not all of the slots provided by the gradient vector might be needed during the last tape evaluation. To account for this occurrence, line 35 determines the actual number of rows of the Jacobian that need to be computed during each iteration.

The process of seeding does not change much except for the additional loop in line 38, which now sets the derivative values of multiple variables to `1.0`. This equates to `AD_VECTOR_DIM` columns of the seed matrix in Figure 3.

The same goes for the subsequent tape evaluation and the copying of the results to the Jacobian. Like in the seeding code, the loop in line 49 is introduced in order to write all of the computed derivatives to the Jacobian.

```
1 void ulf::jacobian::evalAD(double t, fieldData* y, fieldData* residual, double rdt)
2 {
3     problem_->setJacobianUpdateType(STD_JAC_UPDATE);
4
5     nEvals_++;
6
7     J_->resetDecomposition();
8     J_->fill(0.0); // reset jacobian
9
10    *xtmp_ = *y;
11
12    codi::RealReverseVec<AD_VECTOR_DIM>::TapeType& tape = codi::RealReverseVec<
          AD_VECTOR_DIM>::getGlobalTape();
13
14    int k, i, j, n, l, m;
15    tape.reset();
16
17    tape.setActive();
18    // Register input variables
19    for (i = 0; i < nPoints_ * nSolutions_; i++) {
20        tape.registerInput((*xtmp_)[i]);
21    }
22
23    // Compute RHS with active tape
24    computeResiduals(t, xtmp_, rtmp_);
```

```
25
26      // Register output variables
27      for (i = 0; i < nPoints_ * nSolutions_; i++) {
28          tape.registerOutput((*rtmp_)[i]);
29      }
30      tape.setPassive();
31
32      // Derivatives of independent points are computed simultaneously
33      for(k = 0; k < stencilSize_; k++) {
34          for (n = 0; n < nSolutions_; n += AD_VECTOR_DIM) {
35              int numEvals = n + AD_VECTOR_DIM <= nSolutions_ ? AD_VECTOR_DIM :
                    nSolutions_ - n;
36              // Seeding
37              for (i = k; i < nPoints_; i += stencilSize_) {
38                  for (j = 0; j < numEvals; j++) {
39                      (*rtmp_)[i * nSolutions_ + n + j].gradient()[j] = 1.0;
40                  }
41              }
42              // Tape evaluation
43              tape.evaluate();
44              // Retrieve derivatives and write to jacobian
45              for (i = k; i < nPoints_; i += stencilSize_) {
46                  for (j = i - nDepNeighbors_; j <= i + nDepNeighbors_; j++) {
47                      if (j >= 0 && j < nPoints_) {
48                          for (l = 0; l < nSolutions_; l++) {
49                              for (m = 0; m < numEvals; m++) {
50                                  double g = (*xtmp_)[j * nSolutions_ + l].getGradient()[m
                                        ];
51                                  (*J_)(i * nSolutions_ + n + m, j * nSolutions_ + l) = g;
52                              }
53                          }
54                      }
55                  }
56              }
57              // Clear adjoints
58              tape.clearAdjoints();
59          }
60
61      }
62
63      for (n = 0; n < size_; n++) {
64          (*ssdiag_)[n] = (*J_)(n, n);
65      }
66      age_ = 0;
67
68      problem_->setJacobianUpdateType(NO_JAC_UPDATE);
69  }
```

Figure 7: The code of the evaluation function corresponding to the forward mode of AD, using vector-valued derivatives

7 Application on Model Problems

To examine the performance characteristics and accuracy of the implemented evaluation methods, we applied AD to two different model problems: The heat equation and the homogeneous reactor.

7.1 Reaction Mechanisms

The parameters for the source terms of the differential equations that describe the chemistry of the system are based on previously established reaction mechanisms. These mechanism are based on experimental data and describe the reactions that are active for a fixed set of species. Different mechanisms have been established by various research groups. Although they describe the same overall behavior, the level of detail may differ. As a consequence, some mechanisms include a larger number of species than others. Some of these species have no impact on the simulation for common combustion problems and can thus be ignored. However, there are special cases in which these aspects become relevant. Similarly, there are reaction paths that are rarely active in most simulations. These reactions paths are excluded in some mechanisms.

Because the size of the Jacobian is directly determined by the number of species, the complexity of the mechanism heavily impacts performance. Smaller mechanism usually perform better, but may lead to less accurate results.

For the performance measurements, we used the following mechanisms:

Mechanism	Number of Species
ch4_smooke [29]	17
gri_30_113 [29]	28
GRI-Mech [30]	53
C3 CRECK [31]	84
USC-Mech II [32]	111

Table 2: The mechanism we used for the performance evaluation

For a majority of the performance measurements, we relied on the GRI-Mech mechanism. For the examination of the homogeneous reactor with regard to Jacobian size, all of the above listed mechanisms were investigated.

7.2 Methods

All performance and memory measurements were performed on the Lichtenberg Cluster of the TU Darmstadt[1], using a single compute node with two Intel Xeon E5-2680 v3 processors. The programs were compiled with optimization level `O2` and run on a single core with a fixed frequency of 2.5 GHz.

In order to evaluate and compare the performance of the different ULF builds, we implemented a timing system using the steady clock of the `std::chrono` API introduced in C++11. If timing is enabled with the `USE_TIMING` flag, ULF records time stamps at specific points during the execution, which are tagged with an identifier. After completion, these time stamps are written out to the hard drive in plain-text format for post-processing.

To get a good impression of the average performance as well as examine the distribution of our measurements, every problem configuration was run 10 times. The plots and tables presented here use the median of the recorded timings. To ensure that the correctness of the results produced by the newly introduced methods, the problem solutions were compared to reference data.

For memory measurements, we relied on data provided by the operating system which can be read out from the pseudo-file `/proc/self/status` during execution. This method allows to track the virtual and

[1] `http://http://www.hhlr.tu-darmstadt.de/hhlr/index.de.jsp`

physical memory consumption over the course of the execution. Additionally, CoDiPack provides status information about the tape size which was used to augment the memory data.

During the measurements, only one of these system was enabled at a time in order to avoid skewing the results.

In the performance analysis of both problems we compare the following build configurations:

1. *Primal Build*: Uses `double` as the default scalar type
2. *Forward Mode Build*: Uses the CoDiPack type `ForwardReal`
3. *Reverse Mode Build*: Uses the CoDiPack type `ReverseReal`

The primal build will be the base for comparisons with the newly introduced methods.

For both model problems, we used CVODE of the SUNDIALS suite as the ODE solver[24].

7.3 Heat Equation Problem

The heat equation is a second-order differential equation that describes how the temperature of the grid, given an initial heat distribution, will develop over time. Other problems, such as the diffusion effects in a mixture of gases, can be described with equations of similar structure, making this equation a good indicator of how these problems perform in general.

The one-dimensional heat equation is defined by

$$\frac{\partial T}{\partial t} = \frac{\lambda}{c_p * \rho}\left(\frac{\partial^2 T}{\partial x^2}\right), \tag{9}$$

where T denotes the temperature, λ the thermal conductivity, c_p the specific heat capacity and ρ the mass density of the mixture.

In order to completely eliminate the necessity to simulate any chemistry, we used a constant value for $\frac{\lambda}{c_p * \rho}$ with a mixture of non-reacting N_2 gas. This not only increases the performance of the simulation, but also allows us to isolate the evaluation of transport terms, resulting in a more accurate assessment of the performance of AD for this type of problem.

For this problem configuration, we can only compare the modes of AD with the numerical evaluation method. The analytical methods provided by PyJac and TChem are only able to differentiate the chemical source terms of one point, but can not deal with transport terms of problems such as the heat equation. Therefore, they can not be applied here.

A plot displaying the solved heat equation problem using our configuration is attached in Appendix E.

7.3.1 Jacobian structure

The state vector of the heat equation simply contains the temperature variable of all points on the grid. Since there is no reaction happening, there is no need to derive the mass fractions of the gas mixture. As a result, the block size of the Jacobian is 1, reducing it a simple tridiagonal matrix (shown in Fig. 8).

Each row stores the derivatives of a specific mesh point. For point k, the first and third entries in each triple hold the derivatives with respect to the adjacent points $k-1$ and $k+1$ respectively. The central entry determines how the temperature of the point changes with regard to itself.

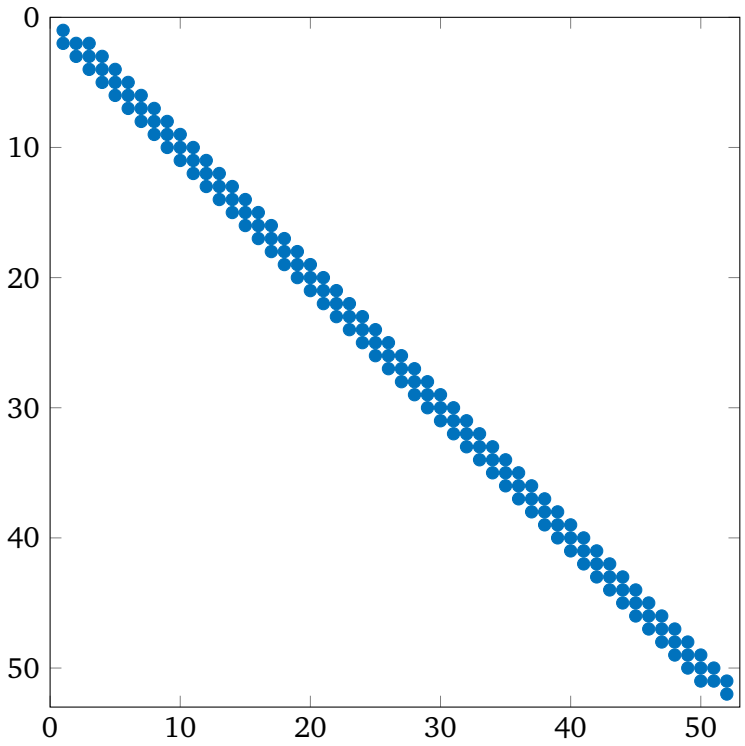

Figure 8: Structure of the heat equation jacobian

7.3.2 Jacobian Accuracy

One of the major benefits of AD is the improved accuracy over the finite difference method. In theory, the application of AD should provide us with exact derivatives equal to the analytical solution up to machine precision. However, the comparison of the numerical and AD Jacobians showed that the results provided by these methods is identical up to 6 decimal places. We could not observer any improvements in run time performance due to increased derivative accuracy. On a mesh with 100 points, the heat equation requires 14 total Jacobian evaluations regardless of the selected differentiation method.

7.3.3 Run Time Measurements

As improving overall performance was one of the main motivations behind the project, the effects of AD on the problem solve time is possibly the most important aspect of this examination. Table 3 and Figure 9 present the measurements of the various builds on a mesh with 100 points.

Because of the simplicity of the equation, the run time for this problem is quite short. Despite this fact, the standard deviation is less than 2% of the median for all configurations.

	Init [ms]	Jacobian Evaluation [ms]	Total Problem Solve Time [ms]	S_{Num}
Numerical	3.66	0.32	99.48	1.00
Forward Mode AD	3.34	0.30	101.85	0.98
Reverse Mode AD	3.63	0.36 (0.08 / 0.28)	101.80	0.98

Table 3: Time Measurements of the heat equation. For the reverse mode, the Jacobian evaluation is split into forward and reverse sweep. The last column represents the speedup of the problem solve time with respect to the numerical method.

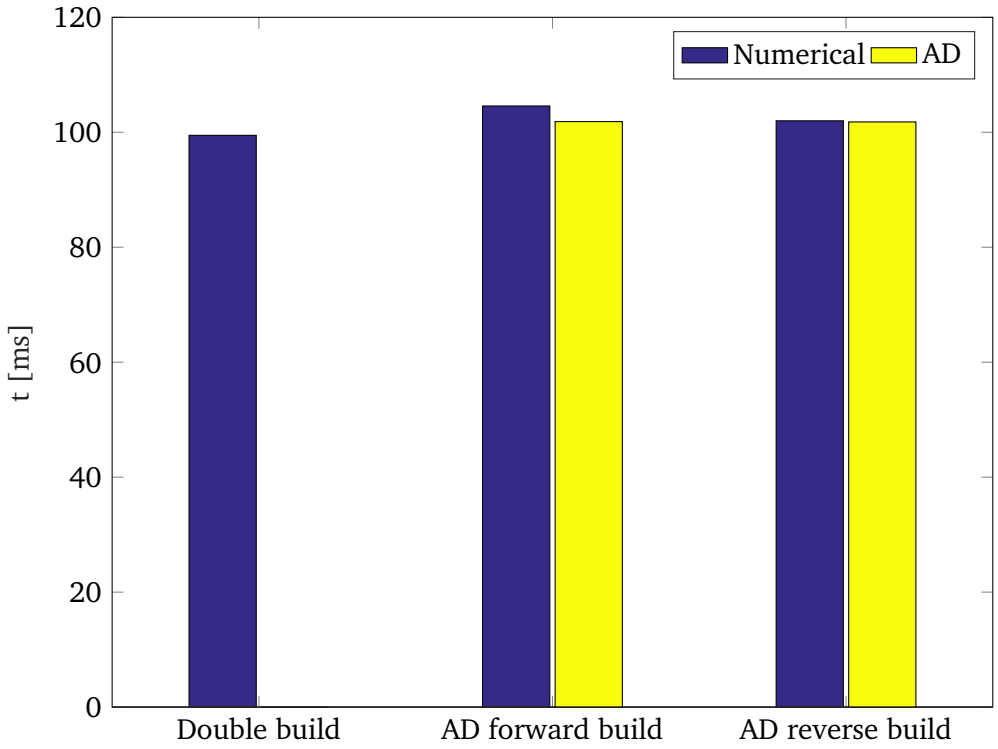

Figure 9: Problem solve time for various builds with different evaluation methods. Measurements for the numerical evaluation in the forward and reverse build are included to illustrate the overhead of AD.

Our measurements show little impact of the evaluation method on the overall run time. Both forward and reverse mode perform similar to the primal build. While the forward mode is a little bit faster, the difference between the two modes is quite small.

Figure 10 presents a breakdown of the total run time: The internal solver code dominates the run time, while the Jacobian evaluation, due to the simplicity of the heat equation, contributes less than 5%. This explains the similar performance of the examined methods.

Figure 11 shows the median time of the Jacobian evaluations during the solving process. The fastest implementation is the version using the forward mode, while the reverse mode is the slowest. This is likely caused by the fact that the necessity to set up and evaluate the tape, as well as clearing the adjoints between evaluations, leads to some overhead. Due to the simplicity of the heat equation, this factor plays a bigger role than the efficiency with which the tape is evaluated.

Despite having the fastest Jacobian evaluation procedure, the configuration using the forward mode yields the worst overall performance. This is due to the fact that other code sections operating on the active type are also affected by the overhead. Different from the reverse mode, where switching the tape to the inactive state prevents performing unnecessary operations, in the forward mode CoDiPack always computes the tangent updates, even if the operations are not part of the differentiated function.

The overhead introduced by AD is evident from the increased run time of the numerical evaluation in comparison with the primal build. The forward mode build is approximately 5.1% slower than the primal build. The reverse mode build performs better, with a speed down of 2.5%. The fact that the overhead is relatively low in both modes is likely due to over 90% of the run time being spent in the solver, which operates mostly on `double` and is therefore largely unaffected by the type change.

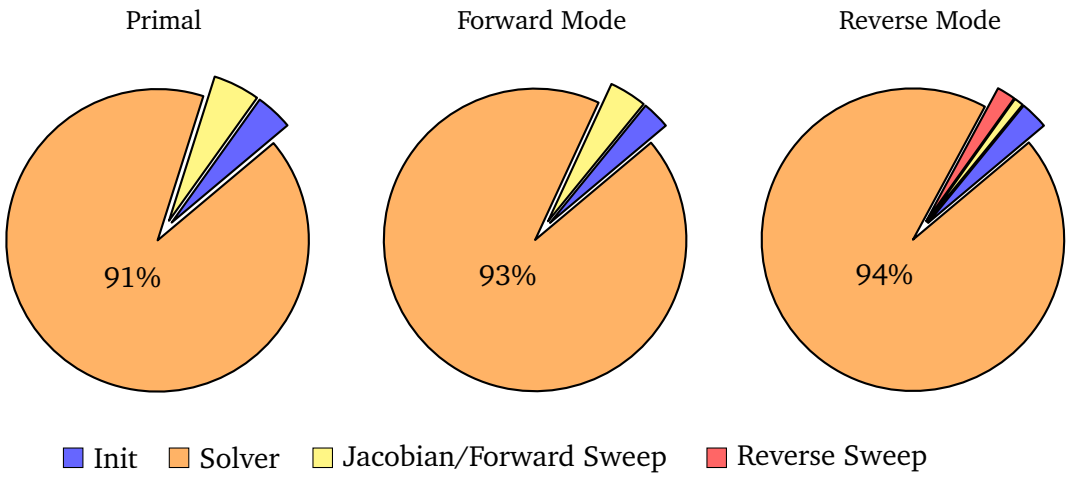

Figure 10: Breakdown of the total run time for the heat equation problem. For the reverse mode, the Jacobian evaluation time is broken up into forward and reverse sweep.

7.3.4 Scaling with Jacobian Size

In order to assess the scaling properties of the heat equation, we examined the effects of the mesh size on the performance. To this end, we measured the median Jacobian evaluation time for various configurations with between 50 and 20000 grid points. The results are displayed in Figure 12. Is is evident that all evaluation methods have very similar scaling characteristics. The numerical evaluation method performs best even for larger mesh sizes, however the difference between the modes remains rather small. This is in line with what one would expect from a theoretical standpoint because the number of function evaluations increases proportionally with the size of the mesh for every mode. The fact that the actual run time does not scale linearly may be caused by worsened cache efficiency due to the increased memory consumption.

7.3.5 Memory Consumption

Since AD augments the primal value of each active variable with a derivative, it is to be expected that the memory consumption effectively doubles in comparison to the primal build. To investigate this effect, we polled the current physical memory consumption multiple times during various steps of the execution. We then recorded the maximal values for each build types. The results are displayed in Figure 13.

Surprisingly, there is very little difference between the build configurations. The maximal memory consumption using the forward mode is only about 1% higher than in the primal build. As further investigation with the heap profiling tool *massif* [33] revealed, only a very small fraction of the total memory is used for the storage of numerical data. Over 95% percent is made up by external libraries and parts of the core that are not directly involved in the solving process. As these parts are not affected by the increased size of `ulfScalar`, most of the memory requirements remain constant across different builds. The result is that the use of AD has almost no impact in this regard.

For the reverse mode, the increase is a little higher. This is to be expected, as CoDiPack requires additional memory to record the results of the forward sweep. Upon further inspection, it was revealed that only a small part of the tape is actually in use. This is caused by the fact that CoDiPack allocates the tape memory in chunks to avoid unnecessary memory operations. The memory consumption of the reverse mode can therefore be further reduced by lowering the default chunk size.

For a much larger mesh with 20000 points, the memory consumption reaches close to 10 GB in the primal build. Similar to the smaller mesh configuration, the overhead of AD is only a small fraction, with less than 10 MB in both modes.

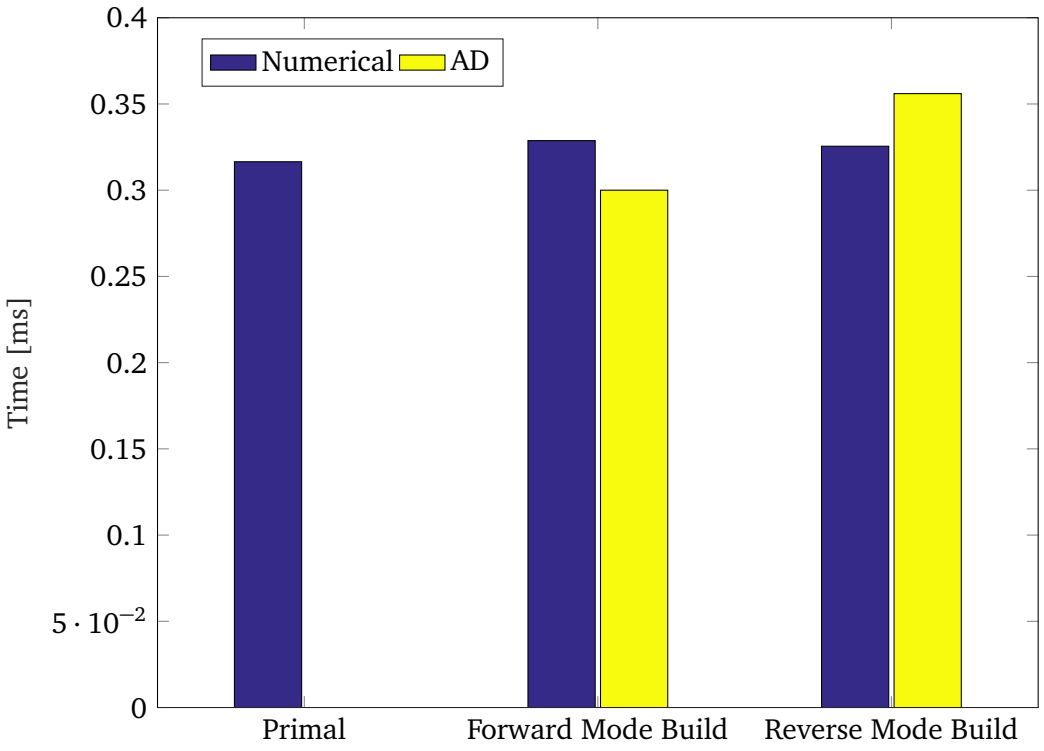

Figure 11: Jacobian Evaluation Time

Unfortunately, these observations are not very useful, as they give little indication of the behavior of AD for real-world problems. For more insight, the investigation of a problem is required in which the system variables make up a much larger part of the total memory consumption.

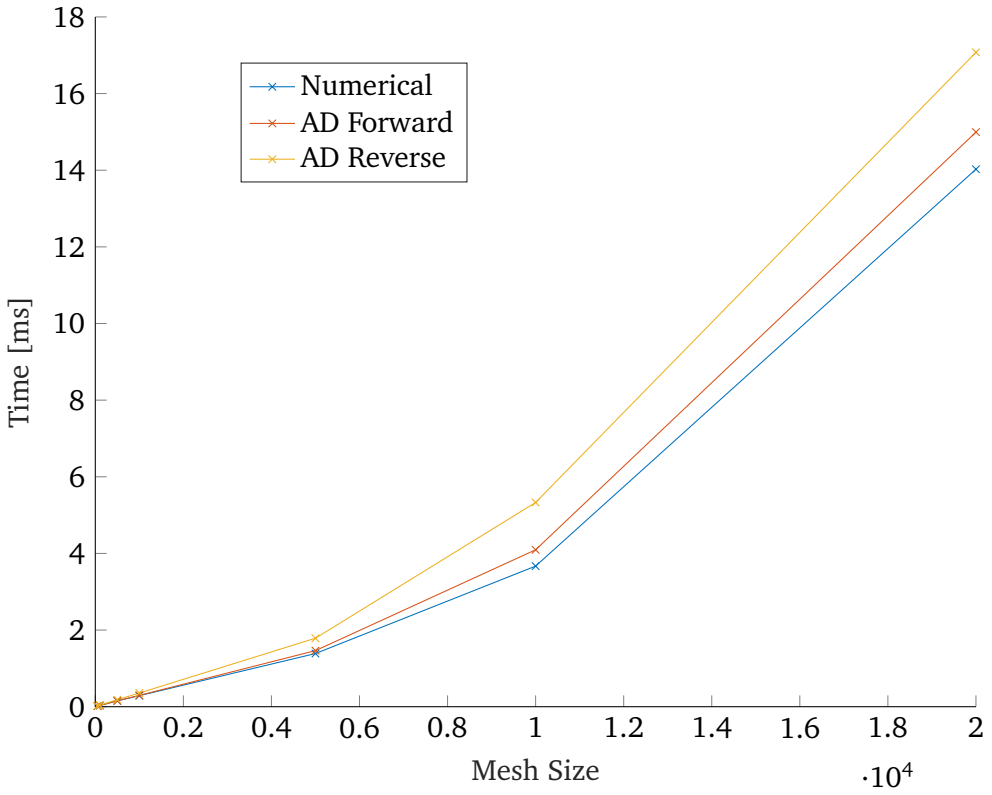

Figure 12: Jacobian Evaluation Time of the Heat Equation with Varying Mesh Size

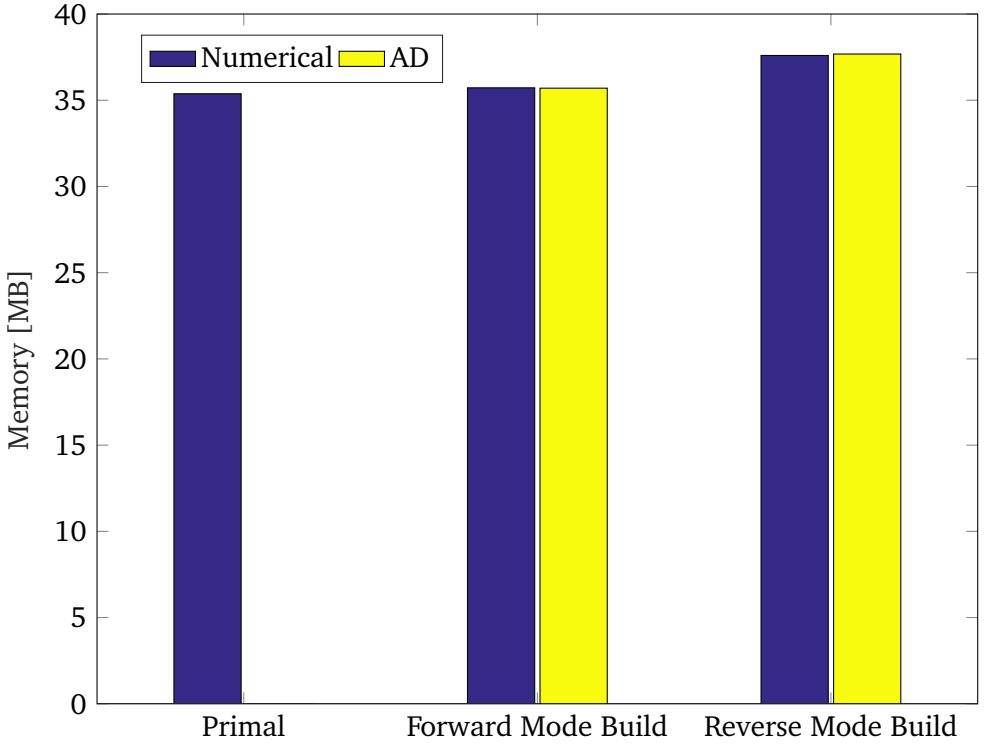

Figure 13: Memory Consumption of the heat equation problem with 100 grid points

7.4 Homogeneous Reactor

The homogeneous reactor is the simplest case of a problem involving a chemical reaction that can be solved with ULF. Starting from a configuration with given mass fractions for each involved species, the homogeneous reactor simulates the reaction of this gas mixture. As the name suggests, it is assumed that the gases are mixed completely evenly and no such effects as convection and diffusion take place. Therefore, no transport terms have to be evaluated. This also means that the homogeneous reactor can be solved on a single grid point, since without the transfer of heat or matter between adjacent points the simulation on a larger mesh is redundant.

For a gas mixture of N species, the homogeneous reactor is defined through the following set of equations

$$\frac{\partial T}{\partial t} = -\frac{1}{c_p \rho} \sum_{k=1}^{N} h_k \dot{m}_k \tag{10}$$

$$\frac{\partial Y_i}{\partial t} = \frac{\dot{m}_i}{\rho} \tag{11}$$

where T denotes the temperature, Y_i the mass fraction of species i, c_p the heat capacity of the mixture, ρ the density, h_k the enthalpy of species k and $\dot{m}_k$ is the source term of species k.

In addition to the numerical and AD evaluations methods, the homogeneous reactor supports the computation of Jacobians with analytical methods. Therefore, PyJac and TChem are included in the performance evaluations.

Plots displaying the solution of the homogeneous reactor problem with GRI-Mech are included in Appendix D.

7.4.1 Jacobian structure

The structure of the Jacobian belonging to the homogeneous reactor differ from the generally assumed block-tridiagonal structure. With a single mesh point, the only remaining part of the matrix is what would normally be the central of the three blocks corresponding to one mesh point, which is determined by the chemical reactions at that location. Therefore, the Jacobian is a dense square matrix. Figure 14 shows such a Jacobian from a problem run with a stoichiometric mixture of H_2 and air using the GRI-Mech mechanism. Apart from the first row, which contains the derivatives of the temperature Equation in 10, each consecutive row holds the derivatives of the reaction rate of a species, based on Equation 11. The non-zero entries, indicated by blue dots, show the species which are involved in the reaction. Depending on the type of mixture, some reaction paths may not be active at all.

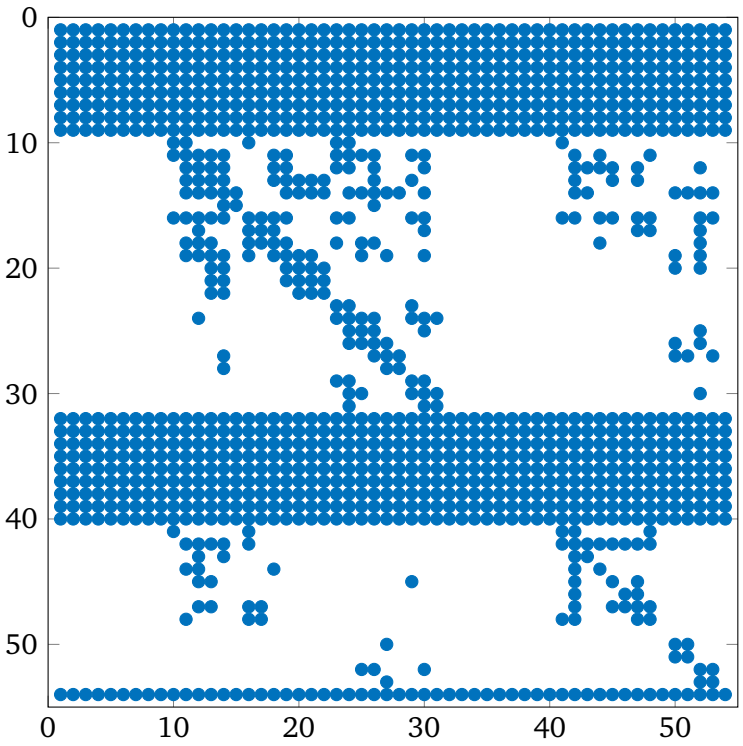

Figure 14: Sparsity Pattern of the numerical Homogeneous Reactor Jacobian at t=0.5 ms with GRI-Mech

7.4.2 Jacobian Accuracy

The comparison of the numerical and the AD Jacobian revealed that the maximum relative error of individual entries is 6%. Additionally, the finite difference method produces a small number of non-zero entries that do not exist in the analytical and AD Jacobians. Due to the increased accuracy, the solver converges with fewer Jacobian evaluations. While for AD, 111 evaluation are needed, the numerical method requires 113 evaluations for the same problem. Due to the small difference, it is unlikely that this has a significant impact on the overall performance.

7.4.3 Run Time Measurements

The run time measurements of the homogeneous reactor are shown in Table 4. A bar graph visualizing the results is provided in Figure 15. In contrast to the heat equation, the Jacobian evaluation method has a significant impact on the performance. The reverse mode yields the best performance of of the investigated methods, with a speedup of 1.68 and 3.64 over the PyJac and numerical configurations respectively. PyJac and TChem have fairly similar run times, both outperforming the numerical method. This is to be expected, since the evaluations of the system equations are avoided. With the use of forward mode AD, there is no improvement over the previous methods. Compared to the numerical evaluation method in the primal build, the run time is increased by a factor of 1.22.

In comparison to the heat equation, the overhead introduced by the active type is much more noticeable. The configuration using the numerical method is about 23% slower in the forward mode build compared to the primal build. Again, for the reverse mode this effect is not as prominent. The increased overhead is likely caused by the fact that a much larger portion of the run time is spent in the active code sections.

Figure 16 shows the run time breakdown of the investigated configurations. In comparison to the heat equation, a much larger percentage of the total time is spent for the computation of the Jacobians. For the primal build, the Jacobian evaluation procedure makes up over 75% of the total run time. Between the numerical and forward mode AD evaluation, the overall composition of the run time has not changed

	Init [ms]	Jacobian Evaluation [ms]	Problem Solve Time [s]	S_{Num}
Numerical	316.86	69.96	10.06	1.00
TChem	315.35	23.39	4.79	2.10
PyJac	315.08	22.62	4.65	2.16
Forward Mode AD	308.12	86.78	12.35	0.81
Reverse Mode AD	301.72	5.10 (0.67 / 4.43)	2.76	3.64

Table 4: Time Measurements of the homogeneous reactor with GRI-Mech. For the reverse mode, the Jacobian evaluation is split into forward and reverse sweep. The last column represents the speedup of the problem solve time with respect to the numerical method.

much. The total time spent in the Jacobian evaluation procedure stays mostly the same. In the reverse mode AD build, this part of the program has been significantly sped up. Together, the forward and reverse sweep only make up about 20% of the total run time, leaving the solver-internal computations as the biggest time-consumer.

The initialization time does not vary significantly between the different builds, despite the increased memory requirements of AD.

The mean run time of the Jacobian evaluation procedures is shown in Figure 17. In this visualization, the efficiency of the reverse mode is even more apparent. The evaluation procedures of the forward mode and numerical method have almost the exact same run time.

7.4.4 Reasons for the Discrepancy between Forward and Reverse Mode

From a theoretical standpoint, one would expect that forward and reverse mode perform approximately equally well. Since for the problems in ULF the number of independent variables is always the same as the number of dependent variables, the reverse mode should not be inherently faster. However, there is a crucial difference in the implementation of the two modes. In the Jacobian evaluation procedure, the forward mode requires one evaluation of the equations for each input variable. In contrast, the reverse mode only needs to call the evaluation function once in the forward sweep. The subsequent computation of the adjoints in the reverse mode is based on the information that is stored on the tape. While the number of required tape evaluations is equal to the number of function evaluations in the forward mode, our measurements show that evaluating the tape is over 10 times faster than one function evaluation. The evaluation function thus poses a significant bottleneck which can be bypassed in the reverse mode. The reason for the inefficiency of the evaluation function lies in the call to `mixture::update()`. This function is responsible for updating the chemical properties of the mixture and must be invoked before the actual differential equations can be evaluated. It is difficult to pinpoint the exact cause of the poor performance of the mixture update function. However, further inspection of the source code as well as the investigation of the call graph generated with *callgrind* [33] has led us to believe that this is partly due to inefficient memory access patterns, caused by repeatedly reading, copying and modifying a large number of fields.

The efficiency of the reverse sweep is a consequence of the way CoDiPack is implemented. The inspection of the statistics provided by CoDiPack showed that the memory consumption the tape is low and few entries are stored. As a consequence, the whole tape likely fits into the cache, making memory accesses in subsequent evaluations very cheap. It is difficult to say why exactly the tape is so small compared to the many operations in the mixture update. Part of the reason may be the reduced number of intermediates due to the use of expression templates. However, this requires further investigation.

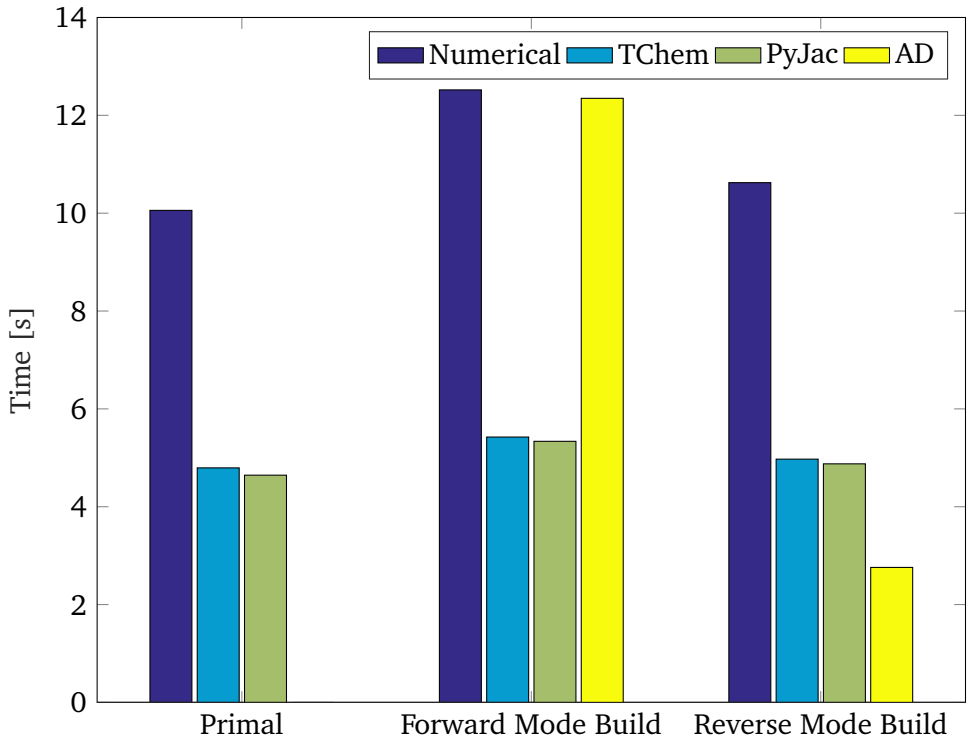

Figure 15: Problem Solve Time of the Homogeneous Reactor Jacobian with GRI-Mech. Measurements for the numerical and analytical evaluations with the forward and reverse build are included to illustrate the overhead of AD.

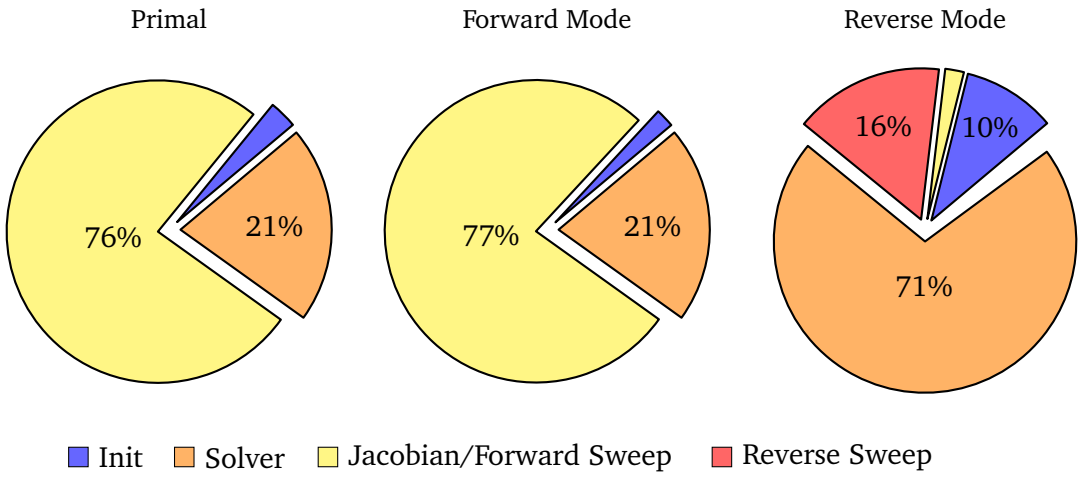

Figure 16: Breakdown of the total run time for the homogeneous reactor with GRI-Mech. For the reverse mode, the Jacobian evaluation time is broken up into forward and reverse sweep.

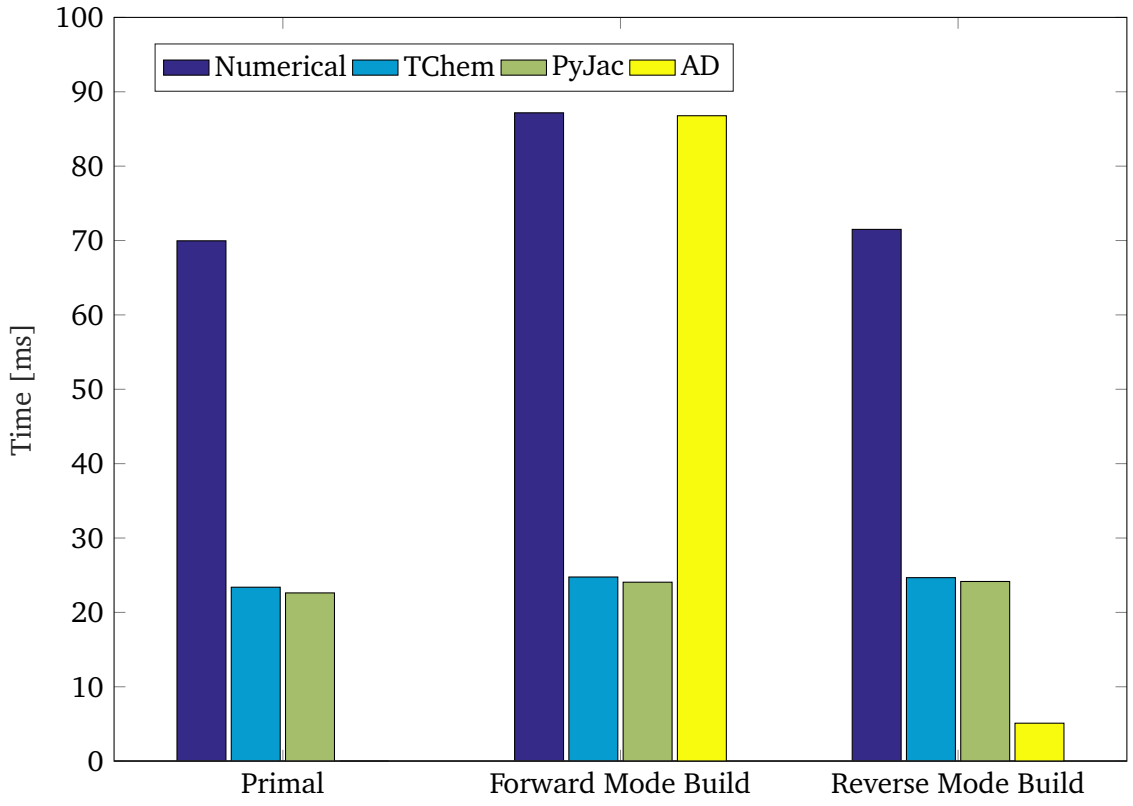

Figure 17: Jacobian Evaluation Time of the Homogeneous Reactor Jacobian with GRI-Mech

7.4.5 Performance of the Vector Mode

As established in the previous section, the mixture update called during each evaluation of the system equations is a major bottleneck. The vector mode introduced in Section 3.5 reduces the number of these function calls and thus promises better performance.

The overhead of the vector mode grows linearly with the vector size and affects all code sections negatively which operate on the active type but are not part of the differentiated functions. Therefore, too large vector sizes can lead to a decrease in performance.

We investigated this aspect by measuring the run time of the homogeneous reactor with various vector sizes between 2 and 16. The results are displayed in Figure 18.

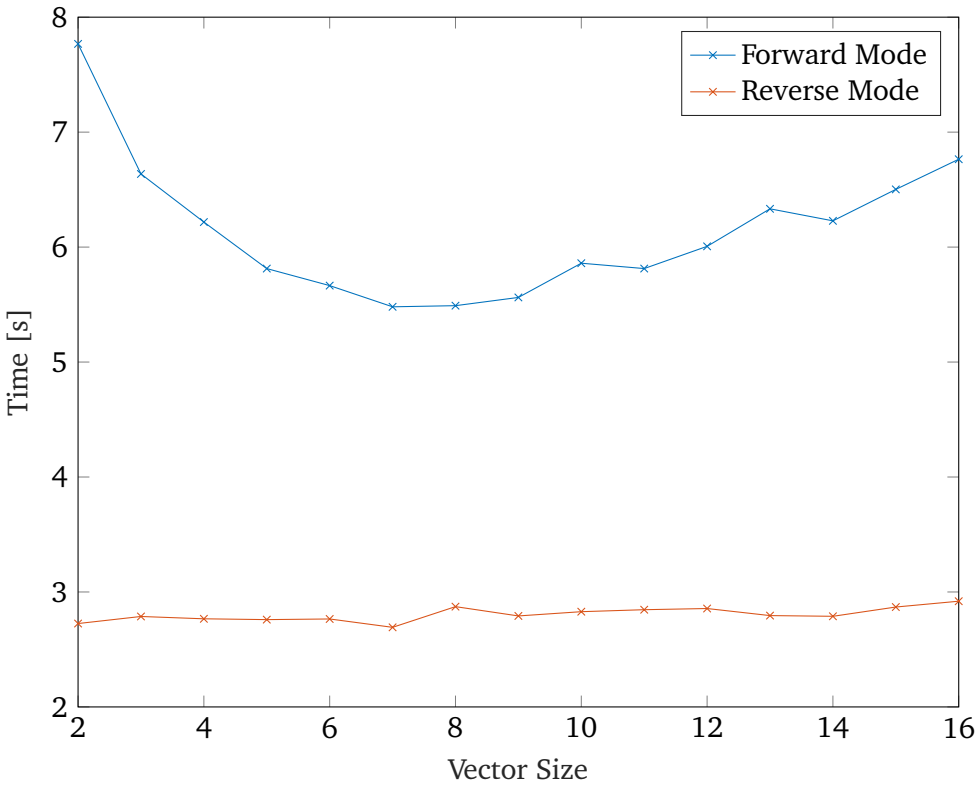

Figure 18: Run time of the homogeneous reactor using AD with varying vector sizes.

The forward mode performs best with vector size 7, taking 5.48 seconds to solve the problem. This equates to a speedup of 2.25 over the scalar forward mode. In this configuration, the number of mixture updates during each Jacobian evaluation is reduced from 54 to 8.

In contrast, the reverse mode is hardly effected at all by the vector size. Because the equations are not directly evaluated during the reverse sweep, the bottleneck of the mixture update is already eliminated. The minute time savings due to the reduced number of the already very fast tape evaluations are thus canceled out by the increased overhead.

7.4.6 Scaling with Jacobian Size

The reactions examined for the performance evaluations are very simple and thus require only relatively small mechanisms. For real-world application in the context of laminar flames, the mechanisms can become much larger, as they require the inclusion of more and bigger molecules. With increasing mechanism size, the size of the Jacobian grows as well, thus impacting the run time of the evaluation procedure. To investigate this aspect, we measured how the run time of the differentiation methods scales with varying numbers of species. The results are displayed in Figure 19. Both the numerical and

the forward mode AD evaluation method have relatively poor scaling characteristics. The two analytical approaches behave virtually identically, performing significantly better than those methods. Because of incompatibility of the analytical methods with larger mechanism, only three mechanisms were investigated. Again, the reverse mode AD evaluation yields the best results, with approximately linear scaling. It is therefore to be expected that AD performs well with larger mechanism and should be a lot faster than the previous methods when applied to more complex problems.

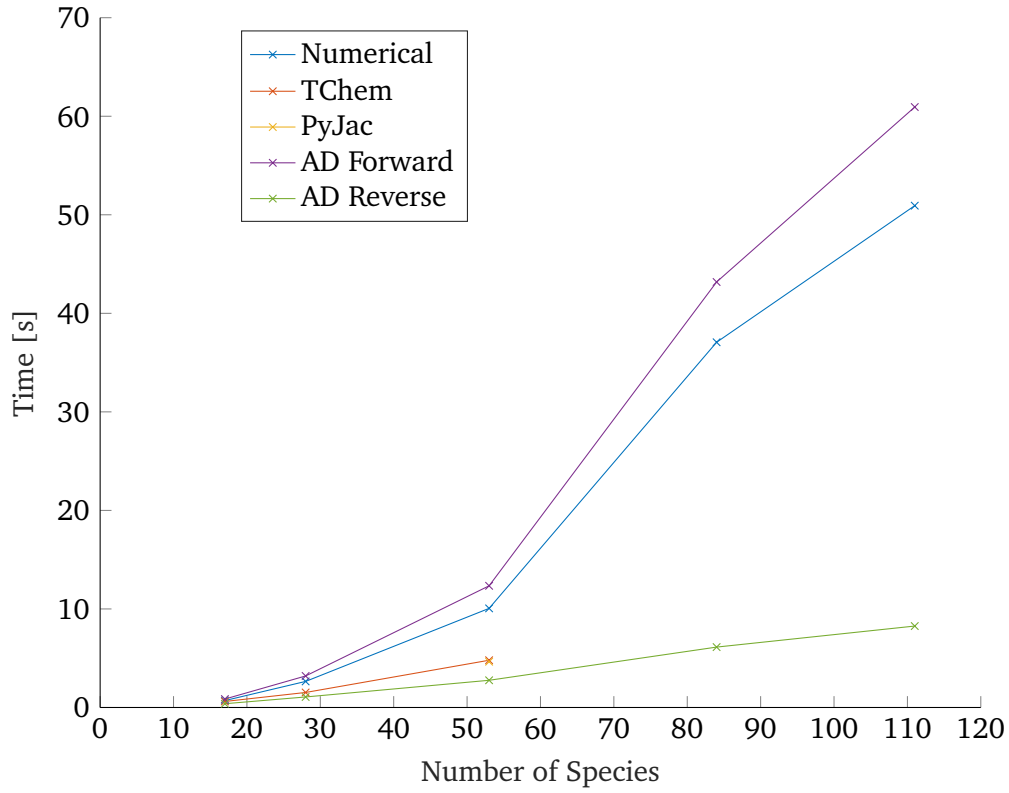

Figure 19: Performance of the Homogeneous Reactor with Different Mechanisms

7.4.7 Memory Consumption

In terms of memory consumption, the situation is similar to the heat equation in that there is hardly any difference between the build configurations. As Figure 20 shows, the memory requirements of the forward mode are only marginally increased over the primal build. Again, the reverse mode requires more memory due to the tape implementation, which allocates its memory in chunks of fixed size.

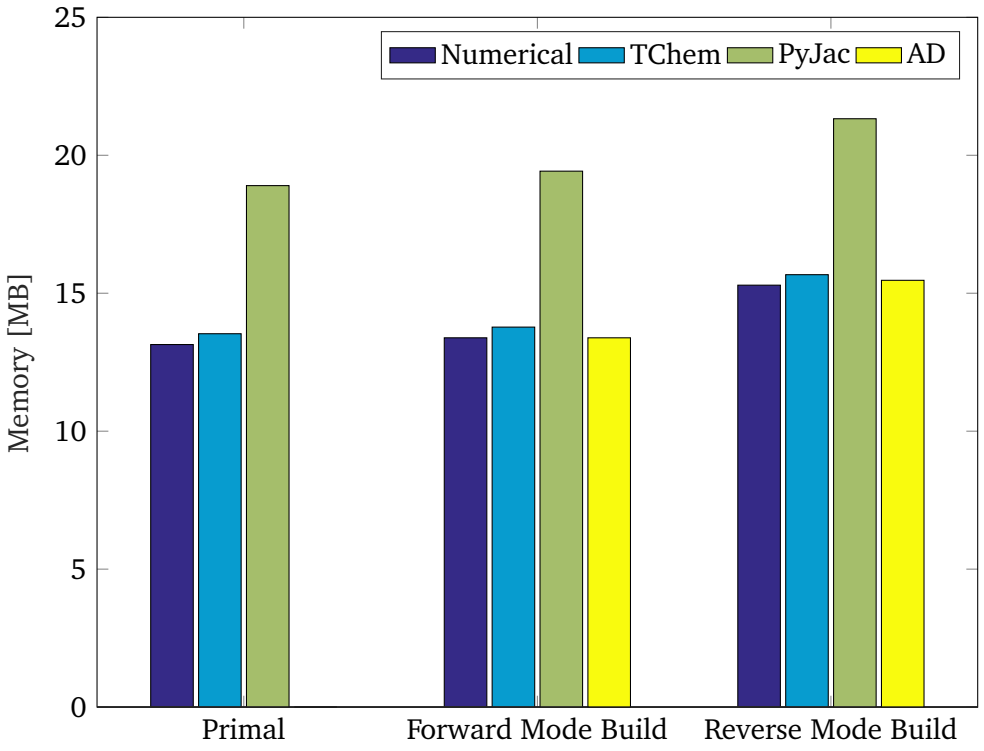

Figure 20: Memory Consumption of the Homogeneous Reactor

8 Conclusion and Outlook

In this thesis, we described the introduction of AD to ULF. We then used AD to implement a number of alternative Jacobian evaluation functions. To evaluate the performance of our evaluation functions, we then examined two model problems. For the heat equation, we observed that AD was roughly as effective as the numerical method regarding overall run time. Although we made improvements in the performance of the Jacobian evaluation function, the percentage of time spent there is too small to compensate for the overhead introduced by the new types. Nonetheless, we provide a viable alternative to the finite difference approach.

Examining the Homogeneous Reactor, we were able to significantly improve the performance compared to the existing differentiation methods. With the reverse mode, we observed a speed up of 3.6 compared to numerical Jacobian evaluation with respect to total run time. Regarding Jacobian evaluation time, the reverse mode performed even better with a speedup of over 9. Furthermore, applying larger reaction mechanisms showed that AD scales better than any of the other methods. With the forward mode we were not able to directly improve the run time. However, we showed that the use of vector mode AD allows for significant improvements in this regard.

For the chemical aspect of problems, our investigation of the homogeneous reactor shows clear improvements with AD. To make a definitive statement on the effectiveness of AD applied to transport problems, more complex problems need to examined.

So far, we have only looked at model problems that isolate certain aspects of flame simulation. The next step is to apply AD to real-world problems. Before this can be done, one remaining issue has to be resolved. The simulation of flames typically involves a strategy that first solves the problem on a coarse grid and then incrementally adds more grid points, yielding a more detailed solution. During each of these remeshing events, the properties of the newly introduced points are computed through linear interpolation of adjacent points. However, some of the underlying differential equations are highly unlinear, due to e.g. the high reaction rates of radicals. When using the finite difference methods, this is usually not a problem. Because of approximation errors, the resulting Jacobian is not exact, which

dampens the effects of these unstable terms. However, with the use of AD and analytical differentiation, the derivatives are "too accurate", which results in very large or very small Jacobian entries. As a consequence, the solver is not able to converge using these Jacobians.

This problem is not directly caused by AD but rather by the fact that the initial values for new grid points can not be chosen optimally. Consequently, there are no changes required regarding the AD mechanisms. Instead, the problem can be solved by adjusting the remeshing process to include a relaxation step after the initial interpolation. The relaxation causes the extremely large and small values to be adjusted. With the resulting Jacobian, solver convergence is then possible. This relaxation functionality has not yet been implemented. However, it is too be expected that after this has been done, we will be able to use AD to solve a wide variety of flame setups.

Our current implementation of the vector mode defines the vector size globally for all `ulfScalar` variables. As we have observed with the homogeneous reactor, this introduces additional overhead and thus reduces the effectiveness of the vector mode. An alternative approach is to apply the vector mode locally on the differentiated functions. This requires maintaining a second mixture object that operates on the corresponding AD type and is only used for the evaluation of Jacobians. Other parts of the code are then unaffected by the vector mode overhead, thus increasing overall performance.

Currently, we have only used AD to compute Jacobians for the ODE solver. In addition to this, AD can be applied in a larger context to perform a sensitivity analysis on a selected flame model. This allows to investigate the effects of the specific parameters on the behavior of the system. In order to do this, ULF must be further modified to include AD for the solver code.

It its current state, ULF is a strictly sequential application. In future versions, there is the possibility that ULF will be extended to support message-passing parallelism using MPI. As mentioned in Section 2.5, adjoint code for the communication operations necessary for the transfer of data between processing nodes can not be generated with standard AD implementations. Therefore, such a change would require further adjustments. Fortunately, CoDiPack already provides an interface for the Adjoint MPI framework. Because of this, we expect that the changes required to make ULF work with AD and MPI will be relatively minor.

```
#ifndef RECAST_H
#define RECAST_H

namespace detail
{
template <typename T>
struct ForPartialSpecialization {
    static auto value(const T& val) -> decltype(val) { return val; }
};
} // namespace detail

template <typename T>
auto value(const T& val) -> decltype(detail::ForPartialSpecialization<T>::value(val))
{
    return detail::ForPartialSpecialization<T>::value(val);
}

template <typename To, typename From>
To recast(const From& val)
{
    return static_cast<To>(value<From>(val));
}

#if USE_AD
#include "codi_extensions.h"

namespace detail
{

template <typename Tape>
struct ForPartialSpecialization<codi::ActiveReal<Tape>> {
    static double value(const codi::ActiveReal<Tape>& val) { return val.getValue(); }
};
}

#endif

#endif // RECAST_H
```

Figure 21: Source code of recast

B Source Code of ulf_memcpy

```
#ifndef TYPECONVERSION_H
#define TYPECONVERSION_H

#include "parallel.h"
#include "typedefFieldData.h"
#include "recast.h"

namespace ulf
{

  namespace memcpyDetail
  {
    template <class T1, class T2>
    struct ForPartialSpecialization {
      static void ulf_memcpy(T1* out, const T2& in, int size, int offset = 0){
        FOR(i, 0, size){out[i] = recast<T1>(in[i + offset]);
        }
      };
    };

    template <class T>
    struct ForPartialSpecialization<T, T> {
      static void ulf_memcpy(T* out, const T& in, int size, int offset = 0)
      {
        memcpy(out, &in + offset, size);
      };
    };
  }

  template <class T1, class T2>
  void ulf_memcpy(T1* out, const T2& in, int size, int offset = 0)
  {
    memcpyDetail::ForPartialSpecialization<T1, T2>::ulf_memcpy(out, in, size, offset);
  };

}

#endif // TYPECONVERSION_H
```

Figure 22: Source code of ulf_memcpy

C Source Code of variadic `printf`

```
#ifndef CONVERT_H
#define CONVERT_H

#include "recast.h"

namespace ulf
{

template <class T>
auto convert(T arg) -> typename std::enable_if<!std::is_class<T>::value, T>::type
{
    return arg;
};

template <class T>
auto convert(T arg) -> typename std::enable_if<std::is_class<T>::value, double>::type
{
    return recast<double>(arg);
};

/**
 * Use instead of regular fprintf to avoid problems with user-defined types.
 * @t_param Args Types of printf arguments
 */
template <class... Args>
void convert_fprintf(FILE* stream, const char* fmt_string, const Args&... args)
{
    // Convert arguments and forward to fprintf
    fprintf(stream, fmt_string, convert(args)...);
}

/**
 * Use instead of regular printf to avoid problems with user-defined types.
 * @t_param Args Types of printf arguments
 */
template <class... Args>
void convert_printf(const char* fmt_string, const Args&... args)
{
    // Convert arguments and forward to fprintf
    printf(fmt_string, convert(args)...);
}

/**
 * Use instead of regular sprintf to avoid problems with user-defined types.
 * @t_param Args Types of sprintf arguments
 */
template <class... Args>
void convert_sprintf(char* s, const char* fmt_string, const Args&... args)
{
    // Convert arguments and forward to fprintf
    sprintf(s, fmt_string, convert(args)...);
}
} // namespace ulf

#endif // CONVERT_H_H
```

Figure 23: Source code of the variadic `printf`

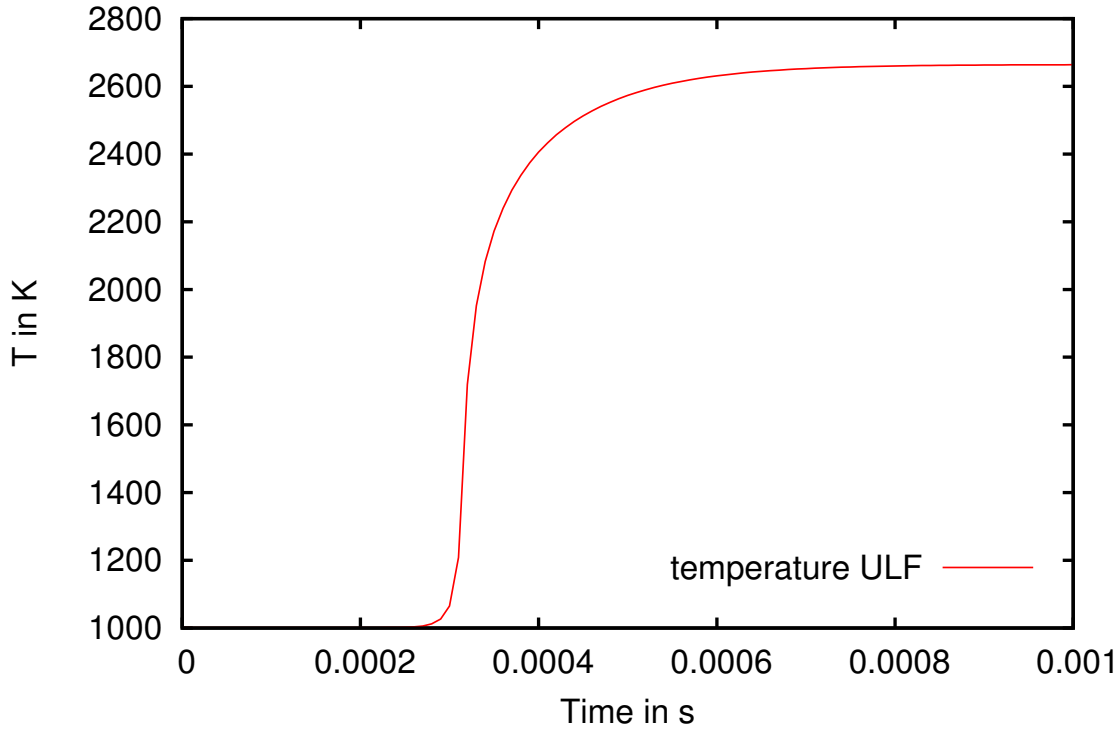

Figure 24: Homogeneous Reactor Solution for T

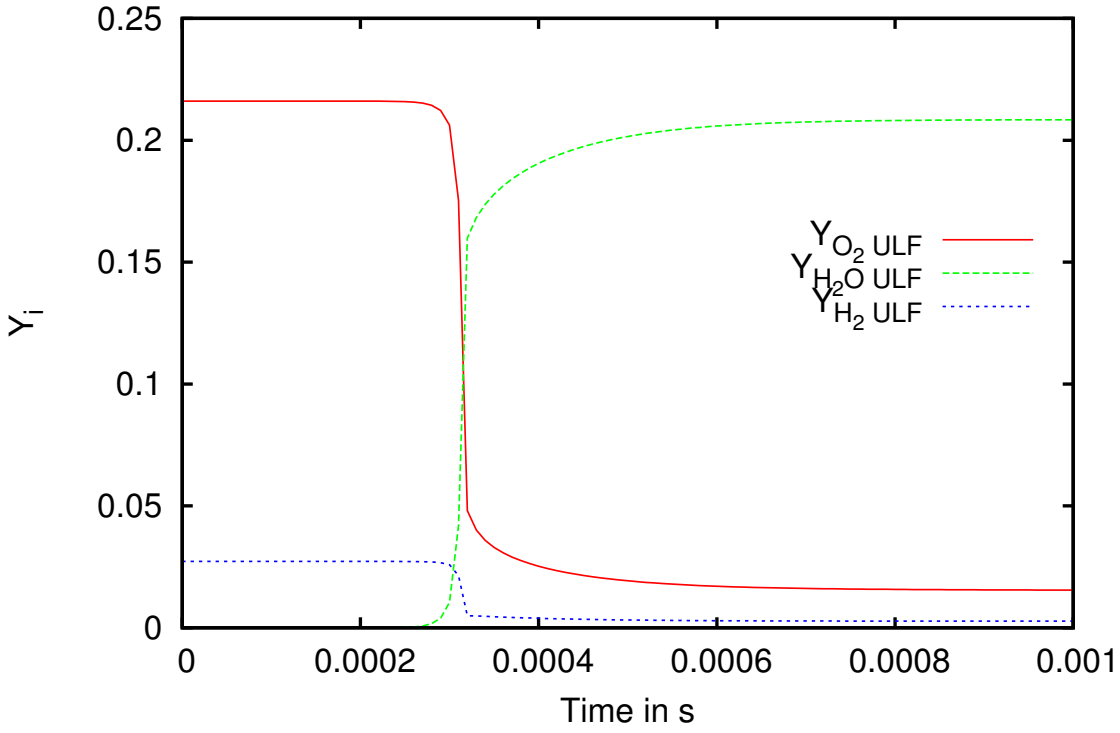

Figure 25: Homogeneous Reactor Solution for Y_i

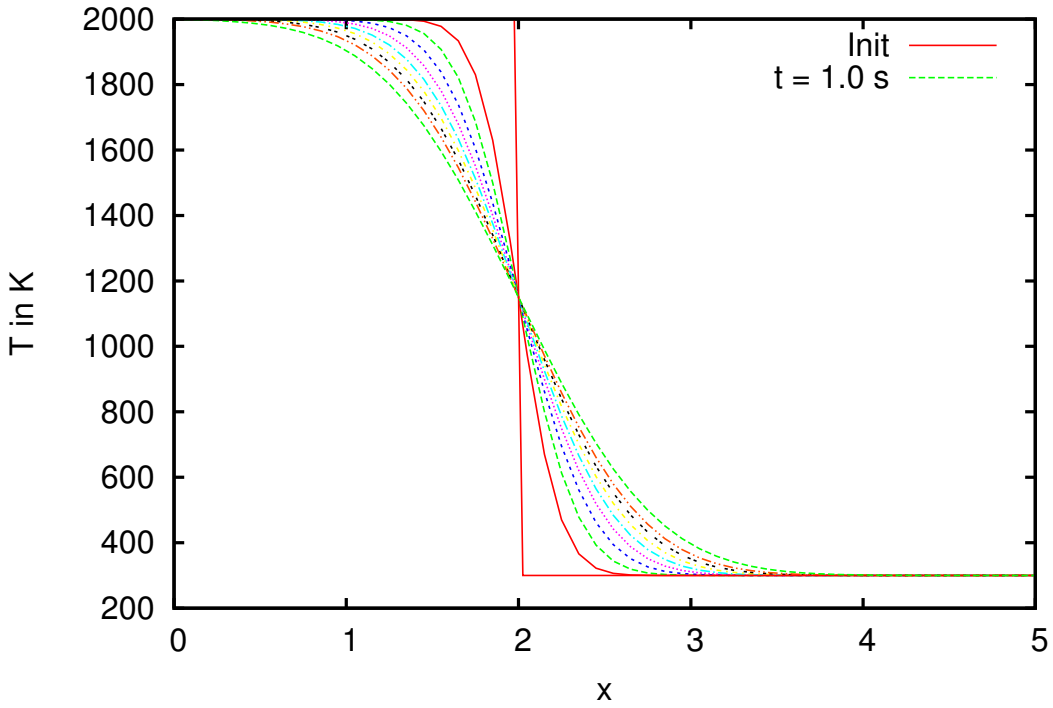

Figure 26: Heat Equation Solution

References

[1] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[2] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[3] Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes. Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries. *Procedia Computer Science*, 18:208 – 217, 2013.

[4] Alexander Hück, Jean Utke, and Christian Bischof. Source Transformation of C++ Codes for Compatibility with Operator Overloading. *Procedia Computer Science*, 80:1485 – 1496, 2016.

[5] Kyle E. Niemeyer, Nicholas J. Curtis, and Chih-Jen Sung. pyJac: Analytical Jacobian generator for chemical kinetics. *Computer Physics Communications*, 215:188–203, 2017.

[6] Cosmin Safta, Habib N. Najm, and Omar M. Knio. TChem-a software toolkit for the analysis of complex kinetic models. *SANDIA REPORT, SAND2011-3282, Unlimited Release*, 2011.

[7] Alexander Hück, Christian Bischof, Max Sagebaum, Nicolas R. Gauger, Benjamin Jurgelucks, Eric Larour, and Gilberto Perez. A Usability Case Study of Algorithmic Differentiation Tools on the ISSM Ice Sheet Model. *Submitted for Review*, 2017.

[8] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2):131–167, 1996.

[9] A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[10] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Rev.*, 47(4):629–705, April 2005.

[11] Jean Utke, Laurent Hascoet, Patrick Heimbach, Chris Hill, Paul Hovland, and Uwe Naumann. Toward adjoinable MPI. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[12] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[13] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845 – 1853, 2010.

[14] Christian H. Bischof, H. Martin Bücker, Bruno Lang, Arno Rasch, and Andre Vehreschild. Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 65–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[15] Robin J. Hogan. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:24, jun 2014.

[16] Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. Interpretative adjoints for numerical simulation codes using MPI. *Procedia Computer Science*, 1(1):1825 – 1833, 2010.

[17] Bert Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.

[18] Andrea Walther and Andreas Griewank. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In *Numerical mathematics and advanced applications*, pages 834–843. Springer, 2004.

[19] Friedrich L. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

[20] Christian H. Bischof, Mohammad R. Haghighat, et al. Hierarchical approaches to automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94, 1996.

[21] Tim Albring, Max Sagebaum, and Nicolas R. Gauger. Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework. In *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 3240, 2015.

[22] Max Sagebaum. CoDiPack Documentation. `http://www.scicomp.uni-kl.de/codi/`. Accessed: 2017-05-01.

[23] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. OpenFOAM: A C++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20. IUC Dubrovnik, Croatia, 2007.

[24] Scott D. Cohen, Alan C. Hindmarsh, Paul F. Dubois, et al. CVODE, a stiff/nonstiff ODE solver in C. *Computers in physics*, 10(2):138–143, 1996.

[25] David G. Goodwin, Harry K. Moffat, and Raymond L. Speth. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes. `http://www.cantera.org`, 2017. Version 2.3.0.

[26] Alexandre Ern and Vincent Giovangigli. Fast and accurate multicomponent transport property evaluation. *Journal of Computational Physics*, 120(1):105–116, 1995.

[27] Benoît Jacob. Eigen. `http://eigen.tuxfamily.org/`. Accessed: 2017-05-01.

[28] Klaus Iglberger. Blaze. `https://bitbucket.org/blaze-lib/blaze/overview`. Accessed: 2017-05-01.

[29] R. Zeißler. *Modellierung der Gasphasenreaktion bei der autothermen katalytischen Erdgasspaltung unter hohen Drücken*. PhD thesis, TU Bergakademie Freiberg, 2005.

[30] Gregory P. Smith, David M. Golden, and Michael Frenklach. GRI-Mech 3.0. `http://www.me.berkeley.edu/gri_mech/`.

[31] E. Ranzi, A. Frassoldati, R. Grana, A. Cuoci, T. Faravelli, AP. Kelley, and CK. Law. Hierarchical and comparative kinetic modeling of laminar flame speeds of hydrocarbon and oxygenated fuels. *Progress in Energy and Combustion Science*, 38(4):468–501, 2012.

[32] Hai Wang, Xiaoqing You, Ameya V. Joshi, Scott G. Davis, Alexander Laskin, Fokion Egolfopoulos, and Chung K. Law. USC Mech Version II. High-Temperature Combustion Reaction Model of H2/CO/C1-C4 Compounds, 2007.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.