Hybrid Approaches to Adjoint Code Generation  
with dco/c++

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

Diplom-Ingenieur

Johannes Lotz

aus São Paulo, Brasilien

Berichter: Universitätsprofessor Dr. rer. nat. Uwe Naumann
Universitätsprofessor Dr. rer. nat. Nicolas R. Gauger

Tag der mündlichen Prüfung: 03. März 2016

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.
1 Introduction and Motivation

Simulation software has become an indispensable instrument for research and development in the academic as well as the industrial environment. In many cases computer simulations are advantageous over experimental approaches not only for financial reasons, but also for opening the possibility to gain insight into systems that are inaccessible for experiments – provided, of course, that the physical and technical laws that determine the system are well known. Nonetheless, the advantages of simulating certain processes are partly compensated by several problems in writing the respective simulation code. A crucial part of simulation codes – especially of those arising from discretized partial differential equations – is usually the numerical kernel including complex solution algorithms. Many of these algorithms require derivative tensors of the model formulation, like e.g. Newton’s method for solving nonlinear systems. This setting can be summarized as derivatives for numerical algorithms. With growing computational power and further developments, the transition from pure simulation to optimization emerged as highly desirable and most notably reachable. Since a lot of numerical algorithms for optimization are also based on derivative information of the underlying problem, efficient gradients of objective functions are required. For complexity reasons those gradients should usually be computed using adjoints. This setting on the other hand can be summarized as derivatives of numerical algorithms and is the concern of this thesis.

Especially for the latter case, long-lasting effort has usually been invested into simulation codes, which will now turn into the basis for an optimization problem. Developing the corresponding adjoint simulation code is a challenging and therefore cost-intensive task. To ease this task, Algorithmic (or Automatic) Differentiation (AD) techniques can be used. With restrictions described in more detail later, AD is able to generate the adjoint simulation code automatically from the primal (original) one. This is done with a statement-by-statement transformation of the program exploiting the chain rule of differential calculus. AD therefore generates code that computes the so-called discrete or algorithmic adjoint, since differentiation is applied after all discretization already took place, i.e. on the algorithmic level. The counterpart to the algorithmic adjoint is the continuous or symbolic adjoint where differentiation takes place before discretization (or in general before the application of numerical methods), see Figure 1.1. Developing the symbolic adjoint formulation typically requires considerably more mathematical knowledge of the specific problem and implementing the resulting formulation typically requires again programming effort. Whether the algorithmic or the symbolic approach is the better method has been already discussed over the last decades with no final answer (see for example [PZG05; NJ07; Gau+15]). Hybridization, i.e. coupling of both methods, therefore seems the way to go. This leads to some parts of the code being differentiated algorithmically and other parts being differentiated symbolically. Such hybridization translates into an interlacing of AD and hand-written derivative code in the course of this thesis.

Generally speaking, AD tools have the advantage of being more automatic\(^1\) than hand-coding,

\(^1\)the formerly wide-spread perception in the community, that AD is fully automatic has changed with good reasons over the last decades
1 Introduction and Motivation

Figure 1.1: Algorithmic (discrete) and symbolic (continuous) differentiation. Based on the set of primal equations $F$ in the upper left, an algorithm for computing its adjoint (or its sensitivities in general) in the lower right can be derived via the algorithmic or the symbolic differentiation path. The algorithmic approach differentiates the solution algorithm for $F$, i.e. after the application of numerical methods (from lower left to lower right). The symbolic approach on the other hand first differentiates the primal set of equations (from upper left to upper right) and applies the numerical methods afterwards.

but on the other hand they usually yield less efficient code than an experienced and mathematically well-versed developer could write by hand. Nonetheless, when hand-coding an adjoint simulation program, an incremental software development process is hard to achieve, which is particularly inconvenient for prototyping. Based on this observation, a reasonable code development strategy is to begin with an AD approach only, and enhance step by step exactly those parts of the code, where the greatest benefit is expected. This benefit could be related to memory requirement, runtime complexity, or even numerical effects like convergence behavior. In particular the memory requirement is a key problem for adjoint AD.

As already indicated, the pragmatic details of applying AD to a given code are not trivial and fundamentally depend on the chosen programming language and the AD technique used. AD can principally be applied by source transformation or by operator overloading. While source transformation usually has the advantage of generating more efficient code, the overloading approach naturally supports the complete programming language standard. The latter point is especially relevant for C++ and since it is used for roughly a fifth of all codes\(^2\) including also simulation programs like OpenFOAM or libraries for linear algebra like Eigen\(^3\), the focus in this thesis is put on C++. A simulation code that is written in C++ can be of high complexity including various numerical algorithms, library calls, interfaces to other programming languages, and parallelism. In addition, modern C++ includes thread-safety, exception-safety, and similar techniques. Those requirements together with the former observations about adjoint codes motivate the core of this thesis: Efficient hybrid adjoints based on AD by overloading in C++.

Besides this hybridization, the mentioned problem of memory requirements in adjoint AD can be approached by checkpointing techniques. Checkpointing is a wide-spread technique and yields a potentially optimal tradeoff between operations count due to recomputations and memory requirement while still being feasible on a given machine. Restriction of checkpoints to arguments of function calls yields the Call Tree Reversal Problem (CTR). CTR is proven to be NP-complete and has usually been approached by guessing a solution or by heuristics. In this thesis an integer programming approach is taken.

\(^2\)the actual fraction strongly depends on the various statistics and the data they are based on
\(^3\)see http://openfoam.com/ and http://eigen.tuxfamily.org
Objective and Approach  During the course of this thesis an AD overloading tool for C++ was to be developed featuring the flexibility for handling a complex simulation code as described above. This includes in particular the possibility of coupling hand-written derivative codes with the AD tool. Parallelism is left aside here, though approachable through similar techniques. The AD tool itself should be as efficient as possible and in compliance with modern C++. In addition, efficient tangent and adjoint projections of higher order were to be developed for common numerical algorithms like linear and nonlinear solvers in different contexts (e.g. in matrix-free solvers). Applicability as well as efficiency gains with respect to runtime and memory of those achievements were to be demonstrated. Apart from this, the Call Tree Reversal Problem was to be reformulated to approach the problem of high memory consumption from a different direction. Here, the focus was on a general solution approach to avoid the heuristics used so far – or at least to be able to measure their quality.

To achieve those objectives, the AD tool dco/c++ was further developed. Its efficiency was ensured by a newly designed AD test suite. The capability of arbitrary-order derivative computation was realized by exploiting template programming techniques in C++, shifting the recursive nature of derivatives into the tool. A flexible interface for coupling the tool with handwritten codes was developed. In addition, formulations for computing arbitrary-order derivative projections for linear solvers as well as first-order projections for nonlinear solvers in different contexts were derived including complexity estimations. Applicability of the proposed software development strategy was shown in an extensive case study. Apart from this, the Call Tree Reversal Problem was reformulated into an integer programming problem which can be solved using general mixed-integer solvers.

Review of Related Prior Work  This paragraph gives an overview over related articles and books. More detailed citations are given in the introductions of the respective chapters and sections later.

Since AD is in its core an application of the chain rule, it could be argued that it originated with Gottfried W. Leibniz or Guillaume de L’Hospital. Nonetheless, the actual technique and corresponding implementations got more attention with growing computational capabilities and was reinvented a few times in the twentieth century. Significant contemporary textbooks are by A. Griewank and A. Walther [GW08] (“Evaluating Derivatives”) with emphasis on theoretical aspects and by U. Naumann [Nau12] (“The Art of Differentiating Computer Programs”) looking deeper into practical issues with source transformation and overloading tools. In addition, the field of AD is vivid with one international conference every four years. Successful applications and current research activities in the field of AD can be found in the proceedings [Cor+02; Büc+06; Bis+08; For+12]. Since the introduction of C++, different AD tools using operator overloading techniques were developed, of which the most widely known is ADOL-C [WG12]. Nonetheless, progress in the programming language standard (C++0x/11/14) and in the research of AD has led to new software developments techniques. For example [Hog14] described the use of modern C++ techniques for AD by overloading and presented an efficient tool implementation called Adept, where the emphasis is on plain adjoint mode AD.

Adjoint methods in general are used in many research areas, e.g. aerodynamic shape optimization [GB02], optimal control [Han12], or computational finance [GG06]. Concerning the combination of symbolic and discrete approach, e.g. [BWB08] looked at forward differentiation in the context of a finite volume C++ code. They also used AD and template programming
1 Introduction and Motivation

techniques and had the goal of combining implementation effectiveness and computational efficiency. Research concerning continuous and discrete adjoint approaches was done e.g. by [NJ07], [Gau+08], or [Aso+08] in the field of aerodynamic shape optimization and by [PZG05] in the area of turbomachinery applications. Also a recent Dagstuhl seminar has been organized on continuous and discrete adjoint methods [Gau+15].

Related work in the field of checkpointing and the Call Tree Reversal Problem is presented in Chapter 5.

Overview of the Thesis This thesis is organized as follows. In Chapter 2 the fundamentals of AD and the theory of hybridization thereof with externally differentiated parts is shown. This is followed by an introduction to dco/c++ in Chapter 3, where first- and higher-order differentiation is introduced and the interface for hybridization is demonstrated. Afterwards, efficient derivative projections of linear and nonlinear solvers are investigated in Chapter 4. Though the emphasis in this thesis is mainly put on the adjoint model, in most cases both models are considered there. In this chapter the reader is especially pointed to the extensive case study, in which all previously introduced techniques are applied. Chapter 5 covers the Call Tree Reversal Problem and includes the derivation for a corresponding integer programming formulation. Chapter 6 collects the results and in Chapter 7 the thesis closes with conclusions and an outlook.
2 Algorithmic Differentiation

In the previous chapter the use of Algorithmic Differentiation (AD) was motivated in general and it was argued to be a crucial basis for the hybridization approach proposed in this thesis. For that reason, a short general introduction to AD is given in this section followed by theoretical underpinnings for the hybridization technique. In addition, at the very beginning, the notation used throughout this thesis is introduced. Though the AD tool dco/c++ is primarily presented in the next chapter, a few related aspects are already mentioned here.

For the well-versed reader in the field of AD, the first section (Foundations, 2.1) will certainly not contain anything new. Nonetheless, the first paragraph introduces the notation briefly and may therefore be valuable to be read anyway. The fundamental derivative models of AD including those of higher order are given and the transition to the computational graph is carried out. Especially the notation and definition for the higher-order models are required later on. The second section (Hybridization, 2.2) describes on the basis of computational graphs why and when hybridization techniques are valid. This introduces the reader also into the author’s way of thinking concerning hybridization and in addition is often referenced by later statements. Notwithstanding, with an algorithmic intuition the later chapters are still comprehensible without reading Section 2.2.

2.1 Foundations

Throughout the thesis the following notation will (mostly) be used. Vectors are lowercase letters written in bold (e.g. \( \mathbf{x} \in \mathbb{R}^n \)), matrices are uppercase letters (e.g. \( A \in \mathbb{R}^{n \times m} \)), scalar-valued functions are lowercase letters (e.g. \( f : \mathbb{R}^n \to \mathbb{R} \)), and vector-valued functions are uppercase letters (e.g. \( F : \mathbb{R}^n \to \mathbb{R}^m \)). For referencing the \( i \)-th element of a vector or a vector-valued function, square brackets and subscript \( i \) are used (e.g. \( [x]_i \)) and similarly for higher-order tensors (e.g. \( [A]_{ij} \)). The index ranges start with 1 for mathematical symbols and with 0 in source codes (based on the C++ standard). The bold symbols \( \mathbf{0} \) or \( \mathbf{1} \) denote vectors, where each element has the respective value. Formally introduced later in Definitions 3 and 4, each variable has corresponding tangent and adjoint components as defined in [Nau12, Chapters 2.1 and 2.2]. In this regard, we follow the notation from [Nau12], i.e., tangents are denoted by the differentiation level as parenthesized superscript, e.g. \( x^{(1)} \) for first-order tangents of \( x \), and adjoints are denoted by the differentiation level as parenthesized subscript, e.g. \( x_{(1)} \) for first-order adjoints of \( x \). Higher-order tangents and adjoints are denoted by combinations of superscript and subscript, e.g. \( x^{(2)}_{(1)} \) are tangents of adjoints of \( x \).

Remark. The corresponding notation used by Griewank and Walther in [GW08] for tangent and adjoint components is \( \dot{x} \equiv x^{(1)} \), \( \bar{x} \equiv x_{(1)} \), and e.g. \( \ddot{x} \equiv x^{(2)}_{(1)} \).

Definition 1. Let \( f : \mathbb{R}^n \to \mathbb{R} \) be continuously differentiable on \( \mathbb{R}^n \). The partial derivative of
$y = f(x)$ at point $x_0$ with respect to $[x]_j$ is denoted by
\[
\frac{\partial f}{\partial [x]_j}(x_0) \in \mathbb{R}.
\]
(2.1)

The vector
\[
\frac{\partial f}{\partial x}(x_0) \equiv \left( \frac{\partial f}{\partial [x]_1}(x_0), \ldots, \frac{\partial f}{\partial [x]_n}(x_0) \right) \in \mathbb{R}^n
\]
is a row vector and contains all partial derivatives of $f$ at $x_0$.

**Remark.** The transpose of $\frac{\partial f}{\partial x}(x_0)$ is usually called the gradient of $f$ at point $x_0$ and denoted by $\nabla f$.

**Definition 2.** Let
\[
F : \mathbb{R}^n \to \mathbb{R}^m
\]
be continuously differentiable on $\mathbb{R}^n$. The matrix
\[
\frac{\partial F}{\partial x}(x_0) \equiv \begin{pmatrix}
\frac{\partial [F]_1}{\partial x}(x_0) \\
\vdots \\
\frac{\partial [F]_m}{\partial x}(x_0)
\end{pmatrix} \in \mathbb{R}^{m \times n}
\]
contains the transposed gradients of each of the $m$ components $[F]_i$ of $F$ as rows and it is called the Jacobian of $F$ at point $x_0$.

We now introduce the major two first-order models of AD and their combinations for higher-order derivative computations. More discussion about so-called cross-country modes and elimination techniques can be found in [GW08, Chapter 9] or [Nau08b]. Nonetheless, the hybridization introduced in the next section can be seen as some kind of cross-country mode. For the basic models, remarks concerning complexity and default use cases will be given in the next chapter when introducing dco/c++.

In the following, only the more general case of a multi-variate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ is considered, since this function degenerates to the multi-variate scalar function $f : \mathbb{R}^n \to \mathbb{R}$ for $m = 1$.

**Definition 3.** The Jacobian $\frac{\partial F}{\partial x} = \frac{\partial F}{\partial x}(x)$ induces a linear mapping defined by
\[
x^{(1)} \mapsto \frac{\partial F}{\partial x} \cdot x^{(1)}.
\]
(2.5)

The function $F^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}^{2m}$, defined as
\[
\begin{pmatrix}
y \\
y^{(1)}
\end{pmatrix} := F^{(1)}(x, x^{(1)}) \equiv \begin{pmatrix}
F(x) \\
\frac{\partial F}{\partial x} \cdot x^{(1)}
\end{pmatrix},
\]
is referred to as the first-order tangent model of $F$. The first-order tangents of $x$ and $y$ are denoted by $x^{(1)}$ and $y^{(1)}$, respectively.
2.1 Foundations

**Definition 4.** The Jacobian $\frac{\partial F}{\partial x} = \frac{\partial F}{\partial x}(x)$ induces a linear mapping $\mathbb{R}^m \rightarrow \mathbb{R}^n$ defined by

$$y(1) \mapsto \left(\frac{\partial F}{\partial x}\right)^T \cdot y(1).$$

The function $F(1) : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^{m+n}$, defined as

$$\begin{pmatrix} y \\ x(1) \end{pmatrix} := F(1)(x, y(1)) = \begin{pmatrix} F(x) \\ \left(\frac{\partial F}{\partial x}\right)^T \cdot y(1) \end{pmatrix}, \quad (2.7)$$

is referred to as the first-order adjoint model of $F$. The first-order adjoints of $x$ and $y$ are denoted by $x(1)$ and $y(1)$, respectively.

**Remark.** Here, the adjoint model is formulated in its non-incremental form. We will use the incremental formulation later in the context of hybridization. Nonetheless, both versions are identical in terms of the projections that are computed.

Both first-order models defined above are extendable to the so-called vector models also defined in [Nau12, Chapter 2]. Those are very straightforward to derive and are therefore introduced later in Chapter 3 when needed for presenting implementation details.

Higher-order models are derived by a recursive application of one of the first-order models to itself or the other. From now on, all functions are assumed to be $d$-times continuously differentiable for derivative models of order $d$. For higher-order derivatives, the (multi-)linear mappings include tensors of higher dimensions. Therefore the usual multiplication is not defined anymore. Whenever this occurs, Einstein notation is used, i.e., when an index variable appears twice in a single term it implies summation of that term over all the values of the index.

**Definition 5.** Higher-order tangent models are derived by recursive application of the tangent model to itself. For $D = \{1, \ldots, d\}$, $d \geq 1$, the function $F^{(D)} : \mathbb{R}^{2^{d-1}n} \rightarrow \mathbb{R}^{2^{d-1}m}$ computes the function values $y$ and its $2^d - 1$ output tangents

$$y^{(K)} \in \mathbb{R}^m \quad \forall K \subseteq D, K \neq \emptyset \quad (2.8)$$
from the inputs $x$ and its $2^d - 1$ tangents

$$x^{(K)} \in \mathbb{R}^n \quad \forall K \subseteq D, K \neq \emptyset \quad (2.9)$$
and is referred to as the $d$-th-order tangent model of $F$.

**Example 1.** The derivative tensor $\frac{\partial^2 F}{\partial x^2}$ induces a bi-linear mapping defined by

$$x^{(1)}, x^{(2)} \mapsto \frac{\partial^2 F}{\partial [x]^i \partial [x]^j} \cdot [x^{(1)}]_i \cdot [x^{(2)}]_j \quad (2.10)$$
and the function $F^{(1,2)} : \mathbb{R}^{4,n} \rightarrow \mathbb{R}^{4,m}$ given by

$$
\begin{pmatrix}
  y \\
  y^{(1)} \\
  y^{(2)} \\
  y^{(1,2)}
end{pmatrix} = F^{(1,2)}(x, x^{(1)}, x^{(2)}, x^{(1,2)})
$$

(2.11)

is derived by applying the tangent model to itself. $F^{(1,2)}$ is referred to as the second-order tangent model of $F$.

Due to the symmetry of higher-order derivative tensors, all combinations of $d$-times recursive applications of tangent and adjoint models where at least one adjoint model is involved are mathematically equivalent as stated in [GW08, Chapter 5] and [Nau12, Chapter 3.1]. Mathematically, we therefore only consider higher-order adjoint models, where tangent models are applied recursively to a first-order adjoint model. There are, however, differences when looking at the implementation in dco/c++ and its respective efficiency. Those differences are discussed in Section 3.2.3 and Section 6.2.2.

**Definition 6.** Higher-order adjoint models are derived by recursive application of the tangent model to the first-order adjoint model. For $D = \{2, \ldots, d\}$, $d \geq 2$ the function $F^{(D)} : \mathbb{R}^{2d-1,(n+m)} \rightarrow \mathbb{R}^{2d-1,(n+m)}$ computes the function values $y$, its $2^{d-1} - 1$ output tangents

$$
y^{(K)} \in \mathbb{R}^m \quad \forall K \subseteq D, K \neq \emptyset ,
$$

(2.12)

and the $2^{d-1}$ input adjoints

$$
x^{(K)}_{(1)} \in \mathbb{R}^n \quad \forall K \subseteq D
$$

(2.13)

from the inputs $x$, its $2^{d-1} - 1$ tangents

$$
x^{(K)} \in \mathbb{R}^n \quad \forall K \subseteq D, K \neq \emptyset ,
$$

(2.14)

and the $2^{d-1}$ output adjoints

$$
y^{(K)}_{(1)} \in \mathbb{R}^n \quad \forall K \subseteq D ,
$$

(2.15)

and is referred to as the $d$-th-order adjoint model of $F$.

**Example 2.** The derivative tensor $\frac{\partial^2 F}{\partial x^2}$ induces a bi-linear mapping defined by

$$
\begin{array}{c}
x^{(2)}, y_{(1)} \rightarrow \left[ y_{(1)} \right]_i \cdot \left( \frac{\partial^2 [F]_{ij}}{\partial x \partial \left[ x \right]_k} \right)^T \cdot \left[ x^{(2)} \right]_k
\end{array}
$$

(2.16)
and the function \( F^{(2)}_{(1)} : \mathbb{R}^{2(n+m)} \to \mathbb{R}^{2(n+m)} \) given by

\[
\begin{pmatrix}
y \\
x^{(1)} \\
y^{(2)} \\
x^{(2)}
\end{pmatrix} :=\begin{pmatrix}
F(x) \\
(\frac{\partial F}{\partial x})^T \cdot y^{(1)} \\
\frac{\partial F}{\partial x} \cdot x^{(2)} \\
[y^{(1)}]_i \cdot \left(\frac{\partial^2 F}{\partial x^2} \cdot [x^{(2)}]_k + \left(\frac{\partial F}{\partial x}\right)^T \cdot y^{(2)}\right)
\end{pmatrix}
\]

(2.17)

is derived by applying the tangent model to the adjoint model. \( F^{(2)}_{(1)} \) is referred to as the second-order adjoint model of \( F \).

In the following, an implementation of the function \( F \) is assumed to decompose into a single assignment code (SAC) given as

\[
v_{j} = \phi_{j}(v_{i}) \quad \text{for} \quad j = n + 1, \ldots, n + q + m ,
\]

(2.18)

for given inputs \((v_1, \ldots, v_n) = x\), the outputs \( y = (v_{n+q+1}, \ldots, v_{n+q+m}) \) and \( q \) intermediate variables. The notation \( i \prec j \) denotes a direct dependence of \( v_j \) on \( v_i \). The result of each elemental function \( \phi_j \) is assigned to a unique auxiliary variable \( v_j \). The functions \( \phi \) are assumed to be continuously differentiable with respect to their arguments at all points of interest.

The directed acyclic graph (DAG)

\[
G = (V, E) \quad \text{with} \quad V = (1, \ldots, n + q + m) \quad \text{and} \quad (i, j) \in E \iff i \prec j \quad (2.19)
\]

is induced by the SAC. The vertices are sorted topologically with respect to variable dependence, i.e. \( i < j \) for all \( i, j \in V : (i, j) \in E \). The sets

\[
X = \{1, \ldots, n\} , \; Y = \{n + q + 1, \ldots, n + q + m\} , \; \text{and} \; Z = V \setminus (X \cup Y) \quad (2.20)
\]

correspond to input vertices, output vertices, and intermediate vertices, respectively. The linearized DAG is obtained by attaching values

\[
c_{i,j} = \frac{\partial \phi_j}{\partial v_i} \quad (2.21)
\]

to the respective edges \((i, j)\) in \( G \).

**Example 3.** Given a function

\[
F : \mathbb{R}^2 \to \mathbb{R}^2 , \quad \begin{pmatrix}
y_1 \\
y_2
\end{pmatrix} = F(x) = \begin{pmatrix}
[x]_1 \cdot [x]_2 + [x]_1 \\
\exp([x]_2)
\end{pmatrix}
\]

(2.22)

\[1\] All unary and binary operations and all intrinsic functions of C++.
composed of one nonlinear and one linear binary operation as well as one nonlinear unary function, its decomposition into a possible SAC and the induced linearized DAG with corresponding edge labels are given by:

\[
\begin{align*}
  v_1 &= [x]_1 \\
  v_2 &= [x]_2 \\
  v_3 &= v_1 \cdot v_2 \\
  v_4 &= v_3 + v_1 \\
  v_5 &= e^{v_2} \\
  [y]_1 &= v_4 \\
  [y]_2 &= v_5 \\
  c_{1,3} &= v_2 \\
  c_{2,3} &= v_1 \\
  c_{1,4} &= 1 \\
  c_{3,4} &= 1 \\
  c_{2,5} &= e^{v_2}
\end{align*}
\]

For such a given graph \( G \), the tangent and adjoint models can be formulated as follows.

**Theorem 7.** The tangent model for a graph \( G = (V, E) \) is given as

\[
v_j = \varphi_j(v_{i \prec j}) \quad \text{and} \quad v_j^{(1)} = \sum_{i : i \prec j} c_{i,j} \cdot v_i^{(1)}
\]

for \( j \in V \setminus X \) in increasing order, i.e. \( j = n+1, \ldots, n+q+m \). The tangent model on \( G \) implements the tangent projection from Definition 3 with input tangents \( (v_1^{(1)}, \ldots, v_n^{(1)}) = x^{(1)} \) and output tangents \( y^{(1)} = (v_{n+q+1}^{(1)}, \ldots, v_{n+q+m}^{(1)}) \).

**Proof.** See [Nau12], Theorem 2.6.

As can be seen, the tangent model has the same data flow as the original function in Equation (2.18), since the graph vertices are traversed in increasing order. From an implementation point of view, the computation of tangents is therefore performed in one forward propagation step alongside with the function evaluation itself. The graph is not required to be stored in memory, since the local partial derivatives \( c_{i,j} \) can always be computed in place when needed. Consequently, additional memory requirement is restricted to the tangent components \( v_j^{(1)} \) of each variable \( v_j \). dco/c++ implements such a graph-free tangent mode, but exploits assignment-level preaccumulation as described later. Since the data flow is unchanged, hybridization can be implemented straightforwardly as shown in Section 3.3.

**Theorem 8.** The incremental adjoint mode is given as

\[
v_j = \varphi_j(v_{i \prec j}) \quad \text{and} \quad v_i^{(1)} = c_{i,j} \cdot v_j^{(1)} \quad \forall i < j 
\]

for \( j \in V \setminus X \) in decreasing order, i.e. \( j = n+q+m, \ldots, n+1 \). The incremental adjoint mode implements the adjoint projection from Definition 4 with the output adjoints \( (v_{n+q+1}^{(1)}, \ldots, v_{n+q+m}^{(1)}) = y^{(1)} \) and the input adjoints \( x^{(1)} = (v_1^{(1)}, \ldots, v_n^{(1)}) \).

**Proof.** Follows from [Nau12], Theorem 2.14, as stated here [Nau12, p. 60].

10
2.2 Hybridization

As opposed to the tangent model, the adjoint model reverses the data flow, since the graph vertices are traversed in decreasing order. Implementation of the adjoint model therefore requires a forward section and a reverse section, since \( c_{i,j} \) depend on the function values \( v_k \) for \( k < j \) (at least for nonlinear functions \( \varphi_j \)). For operator overloading tools this implies, that alongside with the function evaluation the graph is stored during the forward section in an internal data structure called the tape. The tape is then used for the incremental reverse propagation of adjoints (the reverse section). The reverse propagation in \( \texttt{dco/c++} \) exactly implements the incremental adjoint mode from Equation (2.24). Consequently, a huge amount of additional memory is required for storing the tape in addition to the adjoint variables \( v^{(1)}_j \) for \( v_j \). Due to the data flow reversal, hybridization is more complex since an interface to the tape is required.

Higher-order models can be formulated on the linearized DAG by recursive application of the model implementations above. The tangent or adjoint code Theorem 7 and Theorem 8, respectively, can again be decomposed into a SAC inducing a higher-order linearized DAG. This in turn has a corresponding implementation for computing tangents or adjoints thereof. The notion of this recursion is exactly reproduced in \( \texttt{dco/c++} \) as shown later in Section 3.2.3.

2.2 Hybridization

Here, we motivate the need for hybridization from a pragmatic viewpoint in two scenarios. In the first scenario, the simulation code is coupled with a library where the source code is unavailable. Since application of AD requires the source code, hybridization techniques are essential for regaining feasibility of applying AD at all. In the second scenario, a routine is called which implements an expensive numerical algorithm but with known mathematical meaning, e.g. a nonlinear solver. In both cases, but specifically in the latter, a huge benefit can be obtained by hybridization in terms of memory consumption and runtime of the resulting tangent or adjoint code. In addition to such scenarios, hybridization can be used in terms of graph transformations. This is mainly beneficial for rather small components of the graph and will be used for assignment-level preaccumulation and user-defined local gradients later. All of those things are brought together in a formal description by establishing the link between hybridization and the linearized DAG introduced in the previous section.

For a given vertex set \( \hat{V} \) we consider the induced subgraph \( \hat{G} = (\hat{V}, \hat{E}) = G[\hat{V}] \) with \((i, j) \in \hat{E} \Leftrightarrow (i, j) \in E \land i, j \in \hat{V} \). The subgraph has input sets and pure output sets
\[
\hat{X} := \{ v \in \hat{V} \mid P(v) \cap \hat{V} = \emptyset \} \quad \text{and} \quad \hat{Y} := \{ v \in \hat{V} \mid S(v) \not\subseteq \hat{V} \land P(v) \subset \hat{V} \},
\]
respectively, where \( P(v) \) the set of direct predecessors and \( S(v) \) the set of direct successors of \( v \) in \( G \). In addition, \( \hat{Z} = \hat{V} \setminus (\hat{X} \cup \hat{Y}) \) is the set of intermediate vertices. Descriptively, we only consider vertices as inputs if all predecessor are outside of \( \hat{G} \) and we only consider vertices as pure outputs if at least one successor is outside of \( \hat{G} \) and all predecessors are inside of \( \hat{G} \).

**Definition 9.** \( \hat{G} \) is considered to be valid for hybridization in the following if
\[
\forall v \in \hat{Z} : (P(v) \cup S(v)) \subset \hat{V}.
\]

**Remark.** A similar but more restricted definition is chosen by Griewank and Walther to describe local preaccumulations in [GW08, Chapter 10.2].
Figure 2.1: Three example subgraphs. The left subgraph is valid for hybridization in accordance with
Definition 9. The subgraph in the middle is invalid, since vertex 3 has a predecessor outside of $\hat{G}$. The
right subgraph is invalid due to the same reason, but shown to make clear that 3 is not allowed to be an
output, since it is not pure, i.e., it has a predecessor outside of $\hat{G}$.

This restriction ensures that intermediate vertices are not connected to vertices outside of $\hat{G}$. This is
descriptively the absence of side effects and is required for being able to cut out the
subgraph in a clean and simple way in the tangent and adjoint propagation Theorems 7 and 8. Though
more general subgraphs than those in Definition 9 are possible in principle, we keep to
the restriction to avoid nonessential complexity and edge cases in the upcoming descriptions and
definitions. One valid and two invalid subgraphs, where the different vertex sets are visualized,
are shown in the example Figure 2.1.

In the following, we will call the variables corresponding to the elements in $\hat{X}$ the local inputs
$\hat{x} \in \mathbb{R}^n$ for $\hat{n} = |\hat{X}|$ and the variables corresponding to the elements in $\hat{Y}$ the local outputs
$\hat{y} \in \mathbb{R}^m$ for $\hat{m} = |\hat{Y}|$.

**Gaps in the Graph** The scenarios described previously can now be defined by gaps created
through elimination of subgraphs, which represented the computations performed in library
routines or in the nonlinear solver. A visualization in the context of the overall linearized DAG is
shown in Figure 2.2, where $\hat{G}$ is either a black box (library call) or a white box (nonlinear solver)
with a huge amount of vertices and edges. Elimination of those subgraphs results in a clean
extraction of respective assignments in the tangent and adjoint projections from Equation (2.23)
and Equation (2.24). For the tangent model we get a decomposition of

$$v_j^{(1)} = \sum_{i: i \prec j} c_{i,j} \cdot v_i^{(1)} \quad \text{for} \quad j \in \hat{S}_k, k = 1, 2, 3 \text{ and } j \text{ in increasing order}$$

(2.27)
2.2 Hybridization

Figure 2.2: \( \hat{G}_1 \) represents a black-box library routine and \( \hat{G}_2 \) a complex part of the program. \( \hat{X}_i \) and \( \hat{Y}_i \) are the respective in- and outputs of the subgraphs, see Equation (2.25).

into the three sets \( \hat{S}_1 = \{ j : j \in V \setminus (X \cup \hat{Y} \cup \hat{Z}) \text{ and } j \leq \max(\hat{X}) \} \), \( \hat{S}_2 = \hat{V} \setminus \hat{X} \) and \( \hat{S}_3 = \{ j : j \in V \setminus \hat{V} \text{ and } j > \max(\hat{X}) \} \). \( \hat{S}_2 \) corresponds to the tangent computations that are required to be performed for the subgraph \( \hat{G} \). This is preceded and succeeded by the tangent computations topologically prior to and after \( \hat{G} \), respectively. For the incremental adjoint model we get a decomposition of

\[
v_{(i)k}^+ = c_{i,j} \cdot v_{(1)j} \quad \forall i < j \quad \text{for } j \in \hat{S}_k, \ k = 1, 2, 3 \text{ and } j \text{ in decreasing order} \quad (2.28)
\]

into the three sets \( \hat{S}_1 = \{ j : j \in V \setminus (X \cup \hat{V}) \text{ and } j > \min(\hat{Y}) \} \), \( \hat{S}_2 = \hat{V} \setminus \hat{X} \) and \( \hat{S}_3 = \{ j : j \in V \setminus (X \cup \hat{V}) \text{ and } j < \min(\hat{Y}) \} \). Here, \( \hat{S}_2 \) corresponds to the adjoint computations that are required to be performed for the subgraph \( \hat{G} \). After the elimination of \( \hat{G} \), the local tangent and adjoint projections need to be computed externally by

\[
\hat{y}^{(1)} = \frac{\partial \hat{y}}{\partial \hat{x}} \cdot \hat{x}^{(1)} \quad \text{and} \quad \hat{x}^{(1)}_+ = \left( \frac{\partial \hat{y}}{\partial \hat{x}} \right)^T \cdot \hat{y}^{(1)} , \quad \text{respectively,} \quad (2.29)
\]

with local input tangents \( \hat{x}^{(1)} \in \mathbb{R}^{\hat{n}} \), local output tangents \( \hat{y}^{(1)} \in \mathbb{R}^{\hat{m}} \), local output adjoints \( \hat{y}^{(1)}_+ \in \mathbb{R}^{\hat{n}} \), and local input adjoints \( \hat{x}^{(1)}_+ \in \mathbb{R}^{\hat{n}} \). The input adjoints need to be incremented since the respective vertices possibly have outgoing edges to a vertex \( v \in V \setminus \hat{V} \) (see Figure 2.1, leftmost graph).

The external routines for computing the projections Equation (2.29) may or may not compute the output tangents or input adjoints exactly. Whenever possible, exact projections should be preferred since this is usually the expected behavior. But already in the scenarios given
earlier, there may exist good reasons to do otherwise. Related further discussions as well as details concerning memory and runtime benefits for linear and nonlinear solvers are considered in Chapter 4. In Section 3.3.1 the respective interface of dco/c++ for extracting $\hat{G}$ from $G$ is shown.

Valid Transformations on the Graph  
Associativity of the addition can be exploited to do transformations on the graph while preserving validity of the tangent and adjoint projections. There is a huge number of theoretical problems concerning optimal transformations on graphs in the context of AD, for example analysis of scarcity [GV05; LSU12], the optimal Jacobian accumulation problem [Nau04; Nau08b], or the minimum edge count problem [MN11]. In the following, we consider all valid transformations, since the actual transformation performed is defined externally by the user, as seen later.

Definition 10. Given a graph $\hat{G}$ valid for hybridization, a valid transformation transforms $\hat{G}$ into $\tilde{G} = (\tilde{V}, \tilde{E})$ with input vertices $\tilde{X} = \hat{X}$, output vertices $\tilde{Y} = \hat{Y}$, and arbitrary intermediate vertices $\tilde{Z}$ that satisfy the condition in Equation (2.26) (for $\tilde{Z}$ instead of $\hat{Z}$). A globally correct renumbering of vertices is assumed if $|\hat{V}| \neq |\tilde{V}|$. The edges $\tilde{E}$ and their corresponding edge labels are assumed to fulfill the property of preserving the validity of the tangent and adjoint projections in (2.23) and (2.24).

Definition 11. A full preaccumulation transforms $\hat{G}$ to $\tilde{G}$ where $\tilde{Z} = \emptyset$ and

$$\text{for } i, j \in \hat{V}, i \neq j \text{ and } j \text{ reachable from } i \text{ in } \hat{G} \iff (i, j) \in \tilde{E} \text{ with } c_{i,j} = \frac{\partial v_j}{\partial v_i}, \quad (2.30)$$

where a vertex $j$ is reachable from vertex $i$ if there exists at least one path from $i$ to $j$.

$\tilde{G}$ from Definition 11 represents the local Jacobian $\frac{\partial \hat{y}}{\partial \hat{x}}$ and is therefore a valid transformation following the chain rule of differential calculus.

Example 4. Given a graph with subgraph $\hat{G}$, valid for hybridization, in Figure 2.3, the full preaccumulation decreases the number of edges from 5 to 3. Input and output sets $\hat{X}, \hat{X}$ and $\hat{Y}, \hat{Y}$ are unchanged though a global renumbering took place. The edge labels in $\tilde{G}$ are accumulated derivatives of the local outputs with respect to the local inputs connected by an edge, e.g., $c_{1,3} = \frac{\partial v_3}{\partial v_1}$.

Figure 2.3: Example for a full preaccumulation corresponding to Definition 11. Please note that the edge labels have different values left and right, though the same symbols are used. This is due to the global renumbering.
A valid transformation should always be performed on $\hat{G}$, if either the memory consumption or the runtime is expected to be substantially reduced. In case of an adjoint model, the order of complexity of the memory consumption corresponds to the number of vertices and edges in the resulting subgraph $\tilde{G}$. The memory consumption is reduced substantially, if

$$|\tilde{V}| + |\tilde{E}| \ll |\hat{V}| + |\hat{E}|.$$  

(2.31)

A full preaccumulation may or may not be a good choice as discussed and analyzed in detail with the minimum edge count problem [MN11]. The runtime on the other hand not only depends on the number of vertices and edges, but also on the computational complexity of the edge labels itself. This holds for tangent and adjoint models, since both properties determine the cost of the local projections Equation (2.23) and Equation (2.24), respectively. dco/c++ always performs full preaccumulation at the assignment level in tangent and adjoint modes as explained in Section 3.2.2. For adjoint modes, it additionally provides an interface for doing arbitrary transformations shown in Section 3.3.2. For the latter it is the user’s responsibility to perform valid transformations only.
3 dco/c++

Hybrid differentiation techniques based on Algorithmic Differentiation (AD) by overloading require as a reliable foundation a robust and efficient AD overloading tool. In the course of the authors work for the institute for Software and Tools for Computational Engineering (STCE) a small team including the author intensely worked on the AD tool dco/c++. Continuous and productive work has not only led to a robust and efficient tool that is rich in interesting and useful features but it also led to a growing number of users. A lot of valuable remarks and suggestions concerning new features, efficiency issues in existing features, but also bug reports were submitted by the users who have proven to be very proficient and qualified in what they do.

A brief but profound introduction to the features of dco/c++ is given in this chapter. This is done for two reasons. First, illustrating the applicability of hybridization for adjoint code generation of numerical simulation programs is an essential interest of this thesis. Since details matter in programming, this includes the presentation of code listings within the thesis as well as a software package attached to it – this introduction is meant to support the reader in understanding listings and attached code. Second, nontrivial effort of the author has flowed into the generic capabilities of dco/c++, especially useful for higher-order derivative computations, as well as the efficient and yet flexible interface for high-level intrinsic functions. This interface facilitates the hybridization with dco/c++. The insights gained during this work are assessed to be of value in discussions with computer scientists as well as software developers working in the field of AD research.

The first section (Overview, 3.1) gives a summary about the development and the features of dco/c++. If the reader is familiar with the principles of a tape-based AD overloading tool the second section (First- and Higher-Order Modes, 3.2) may not provide anything substantially new, though the interface of dco/c++ could at least stimulate the interest of programmers who are practically working with other AD tools. Additionally, vector and higher-order modes are presented, the latter of which brings out interesting results in Section 6.2.2. Very closely related to the main point of this thesis is the third section (High-Level Intrinsics, 3.3) and therefore recommended for every reader. The last section (AD Test Suite, 3.4) presents the AD test suite to benchmark the performance of dco/c++ and other AD overloading tools for C++. Most interesting point there is the common interface for all tools.

The functionality of dco/c++ is illustrated by showing source listings as well as figures. To keep the link between those illustrations and their description as close as possible, detailed comments are embedded in the captions. Therefore also crucial pieces for the overall reasoning may appear in captions. I encourage the reader to keep that in mind especially in this section. A documentation for the dco/c++ version used in this thesis can be found in attached_code/dco_cpp_documentation and the source code of all examples in attached_code/dco_cpp_examples.
3 dco/c++

3.1 Overview

dco/c++ is an AD overloading tool written in C++ for computing first- and higher-order tangent and adjoint projections of C++ programs. The software development process started in the form of teaching activities of Prof. Uwe Naumann at the RWTH Aachen about ten years ago. With strong support of Klaus Leppkes (student researcher) and Jan Riehme (research associate) the kernel of dco/c++ was implemented in today’s shape around 2010. In the last five years, the reliable kernel served as the basis for improvements and extensions pushed forward by an active support of a growing number of users and by the collaboration with NAG Ltd.\textsuperscript{1} that was getting closer in the field of AD by overloading in C++.

In those five years, in prolific joint work with the persons already mentioned (U. Naumann, K. Leppkes, and J. Riehme), the author worked on further improvements concerning a C++ compliant way of a maximum level of exploiting generic programming and associated adjustments of the interface. Those generics spread over all features of the software to enable arbitrary derivative nesting. The logic of coupling arbitrarily obtained derivative projection codes with dco/c++ as well as an efficient and flexible user interface along with it was key ingredient of the work done by the author.

The software has been successfully applied to a rising number of applications in financial engineering, atmospheric physics, or fluid mechanics. During various collaborative research and development projects, it was possible to generate fast adjoints for real world applications. A more detailed description of the most important joint projects is given in the results section of this thesis, Section 6.1.

The main features of dco/c++ are data types for computing first- and higher-order tangent and adjoint projections. This is done by operator and function overloading techniques in C++. For both modes, scalar and vector-valued projections are supported. The adjoint mode uses an internal data structure called the tape which facilitates the data flow reversal (see Section 2.1). The tape usually allocates memory in RAM and grows dynamically chunk-wise, but it can also switch to disc when getting too large. For efficiency reasons in terms of runtime and memory, activity analysis and assignment-level preaccumulation is implemented, see Section 3.2.2 for more details. Apart from those basic features, high-level intrinsics can be added to dco/c++ via different interfaces. This is mainly interesting for the adjoint modes and can be used for a variety of different scenarios, e.g., symbolic adjoint approaches, a coupling with compiler generated code, for checkpointing, or for preaccumulating small Jacobians and directly add them to the tape. Those things are looked at in more detail in Section 3.3. dco/c++ is also involved in research on the generation of discrete adjoints of simulation codes using parallel environments, in particular OpenMP\textsuperscript{2} and MPI [For94]. Adjoining OpenMP codes usually require manual effort on the basis of suitable data types that respect the thread-safety property. Those suitable data types are provided by dco/c++. MPI codes on the other hand can be approached by using the Adjoint MPI library [Sch15a]. The interface between Adjoint MPI and dco/c++ is successfully applied for example in discrete adjoint OpenFOAM [TSN15]. Nonetheless, adjoining parallel codes with dco/c++ is not part of this thesis.

\textsuperscript{1}Numerical Algorithms Group, http://www.nag.co.uk
\textsuperscript{2}http://openmp.org/
3.2 First- and Higher-Order Modes

dco/c++ is written on the basis of a wide range of C++ features\(^3\) and is fully C++98 compliant. It runs successfully under Linux, Mac OS, and Windows operating systems and a variety of C++ compilers among which are GNU’s gcc\(^4\), clang\(^5\), Intel’s icc\(^6\), and Microsoft’s Visual Studio\(^7\). Besides an extensive use of the C preprocessor, the dco/c++ source uses in particular operator and function overloading, class and function templates, template specialization, and expression templates. Expression templates are a general C++ technique [Vel95]. They are applied and already proven to be beneficial in various forms in the context of AD overloading tools [AD02; PP12; Hog14]. In addition to the C++98 features used in dco/c++, the examples shown in this thesis are making use of a moderate amount of C++11 features. Those include the auto keyword for automatically deduce left-hand side’s data type, templated type aliases as a shortcut for nested typedefs of templated classes, and initializer lists mainly for a more elegant and shorter initialization of the standard library vector type\(^8\).

In the code excerpts and figures a capital \(T\) and a capital \(U\) stand for template parameters when not defined otherwise. For convenience for the complete chapter the namespaces std and dco are assumed to be included by

```cpp
using namespace std;
using namespace dco;
```

All mathematical notations used in the following are introduced in Chapter 2.

3.2 First- and Higher-Order Modes

The features of dco/c++ introduced in this section are the basis for the more complex applications later on. It will cover first- and second-order models and dco/c++’s interface for invoking them correctly. In addition, some remarks with respect to complexity and performance are being made. We start with a brief general description of how the interface of dco/c++ is designed. Afterwards, the respective models are explained in more detail including examples.

Basically the interface of dco/c++ consists of modes which are implemented as C++ structs. The modes correspond to derivative models, e.g. the tangent model, adjoint model, or higher-order models. Each mode has nested types and static functions which provide the features for the actual derivative computation. The most important data type that exists in each mode is called type. This type is the custom data type used for overloading the operations and function evaluations. In addition to the modes, global functions can be used to access or modify components of type (or vector<type>), namely value and derivative component. Those components do commonly exist in the type of all modes. For tangent modes the derivative component is the tangent of a variable and for adjoint modes the derivative component is the adjoint of a variable.

As in Chapter 3, the model definitions are based on the multi-variate vector function \(F : \mathbb{R}^n \to \mathbb{R}^m\), \(y = F(x)\) with \(x \in \mathbb{R}^n\) and \(y \in \mathbb{R}^m\). We assume a default declaration in C++ that looks like this:

\(^3\)Good descriptions of the features mentioned in the following can be found on http://en.cppreference.com/w/cpp.
\(^4\)https://gcc.gnu.org/
\(^5\)http://clang.llvm.org/
\(^6\)https://software.intel.com/en-us/c-compilers
\(^7\)https://www.visualstudio.com/
\(^8\)See for example http://en.cppreference.com/w/ for a short description of those features.
This function has inputs \( \mathbf{x} \equiv \mathbf{x} \), the return value as outputs \( \mathbf{y} \), and no side effects. The example that is used throughout this section is the so-called lighthouse example from [GW08, p.16]. Griewank and Walther use this example also to analyze non-differentiability in the context of AD. This is not considered here, i.e. the evaluation point is chosen such that differentiability is ensured. \( F \) is defined as

\[
\mathbf{y} = f(\mathbf{x}) = \begin{bmatrix}
[x_1] \cdot \tan([x_3] \cdot [x_4]) \\
[x_2] - \tan([x_3] \cdot [x_4]) 
\end{bmatrix},
\]

\[
\mathbf{y} \equiv \begin{bmatrix}
[x_2] \cdot [x_1] \cdot \tan([x_3] \cdot [x_4]) \\
[x_2] - \tan([x_3] \cdot [x_4]) 
\end{bmatrix},
\]

Equation (3.1)

\( F : \mathbb{R}^4 \rightarrow \mathbb{R}^2 \), and implemented as shown in Listing 3.1.

3.2.1 Tangent Mode

The dco/c++ type \texttt{gts1\textless T\textgreater} implements tangent 1st-order scalar mode, where all internal arithmetic is performed on data of generic type \( T \). The first-order tangent version results from the instantiation of \( F< \texttt{gts1\textless T\textgreater}::\texttt{type} > \) and computes with a proper driver

\[
y := F(\mathbf{x}) \quad \text{and} \quad y^{(1)} := \frac{\partial F}{\partial \mathbf{x}} \cdot \mathbf{x}^{(1)}
\]

with \( \mathbf{x}, x^{(1)} \in \mathbb{R}^n \) and \( \mathbf{y}, y^{(1)} \in \mathbb{R}^m \). An example driver computing function value and the column sum of the Jacobian

\[
y = F(\mathbf{x}) \approx \begin{bmatrix}
-0.0665 \\
-0.133
\end{bmatrix} \quad \text{and} \quad y^{(1)} = \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}) \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 1.74 \\ 3.42 \end{bmatrix}
\]

at \( \mathbf{x} = (1, 2, 1, 3)^T \) is shown in Listing 3.2.
3.2 First- and Higher-Order Modes

```cpp
9    cout << "y = " << y << endl;
10    cout << "y^(1) = " << derivative(y) << endl;
11    return 0;
12}
```

Listing 3.2: main_gt1s.cpp. Code example for a tangent projection with dco/c++. The tangent component \( x^{(1)} \) is assigned in line 3, the tangent projection is carried out implicitly in line 8 and the output tangents \( y^{(1)} \) are extracted in line 11.

Compiling main_gt1s.cpp and running the executable produces the following output.

<table>
<thead>
<tr>
<th>Output of main_gt1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = [ 1 \ 2 \ 1 \ 3 ] )</td>
</tr>
<tr>
<td>( x^{(1)} = [ 1 \ 1 \ 1 \ 1 ] )</td>
</tr>
<tr>
<td>( y = [ -0.0665314 \ -0.133063 ] )</td>
</tr>
<tr>
<td>( y^{(1)} = [ 1.74266 \ 3.41879 ] )</td>
</tr>
</tbody>
</table>

The cost for evaluating the tangent model once is

\[
\text{Cost}(F<\text{gt1s<\text{double>::type }>) = O(1) \cdot \text{Cost}(F<\text{double}>)
\]

with an approximate constant \( 1.2 \sim 2.5 \) (no strong bounds). The actual value depends on the character of the original program, for example the complexity of function evaluations and the reusability of their results for partial derivatives, see [GW08, Chapter 4.5: Complexity of Tangent Propagation] for more details. In addition, non-negligible cache effects can occur when doubling the memory segment per variable due to the tangent component.

The tangent mode is usually used either for computing the Jacobian entries explicitly or for solving a linear system with the Jacobian with matrix-free methods (e.g. the conjugate gradient method [HS52]). For computing dense Jacobians, \( n \) evaluations of the tangent model are required, choosing \( x^{(1)} \) to be the Cartesian basis vectors in \( IRR^n \). For sparse Jacobians on the other hand, the sparsity can and should be exploited not only for more efficient storing but also for more efficient computation, see [CPR74; GMP05]. Concerning matrix-free methods, implicit Jacobian-vector products are required. This is exactly what the tangent mode computes.

**Vector Mode** The dco/c++ type gt1v<\( t, p \) implements tangent 1st-order vector mode, where all internal arithmetic is performed on data of generic type \( t \). The vector length is defined by the positive integer template parameter \( p \). The first-order tangent vector version results from the instantiation of \( F<\text{gt1v<\( t, p \)::type }>) \) and computes with a proper driver

\[
y := F(x) \quad \text{and} \quad y^{(1)}_k := \frac{\partial F}{\partial x}(x) \cdot x^{(1)}_k \quad \text{for} \quad k \in \{1,\ldots,p\}
\]

with \( x \in IRR^n \), \( y \in IRR^m \), \( p \in IN^+ \) the vector length, and \( p \) tangent vectors \( x^{(1)}_k \in IRR^n \) and \( y^{(1)}_k \in IRR^m \). An example driver computing function value and the full Jacobian

\[
y = F(x) \quad \text{and} \quad (y^{(1)}_k)_{k=1,\ldots,4} = \frac{\partial F}{\partial x}(x) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]
at $x = (1, 2, 1, 3)^T$ is shown in Listing 3.3.

```cpp
int main() {
    vector<gtlv<double, 4>::type> x{1, 2, 1, 3};
    for (int i = 0; i < 4; ++i) derivative(x[i])[i] = 1.0;
    cout << "x = " << x << endl;
    cout << "x^(1) = " << derivative(x) << endl;
    auto y = F(x);
    cout << "y = " << y << endl;
    cout << "y^(1) = " << derivative(y) << endl;
    return 0;
}
```

Listing 3.3: `main_gtlv.cpp`. Code example for a tangent vector projection with dco/c++. The vector size $p$ is set to 4. The identity matrix from Equation (3.6) is assigned to $x_k^{(1)} \forall k$ in line 4, the tangent vector projection is carried out implicitly in line 9, and the output tangents $y_k^{(1)} \forall k$ are extracted in line 12.

Compiling `main_gtlv.cpp` and running the executable produces the following output. Vector-valued derivative components are printed in curved brackets.

```
Output of main_gtlv

x = [ 1 2 1 3 ]
x^(1) = [ (1;0;0;0) (0;1;0;0) (0;0;1;0) (0;0;0;1) ]
y = [ -0.0665314 -0.133063 ]
y^(1) = [ (-0.0665314;0.0310525;1.3336;0.444535)
        (-0.133063;-0.00442642;2.66721;0.889069) ]
```

The cost for evaluating the tangent vector model once is

$$\text{Cost}(\mathcal{F}<\text{gtlv}<\text{double}, p>::\text{type}>) = O(p) \cdot \text{Cost}(\mathcal{F}<\text{double}>)$$

(3.7)

with an approximate constant $0.5 \sim 2$ (no strong bounds). Consequently, the computation of 10 tangent directions at once in tangent vector mode can have a low cost compared to 10 tangent directions each by each in usual tangent mode. If $p$ is chosen too large on the other hand, it is even more likely that what previously fitted into the cache gets too big now and results in a much worse performance.

The tangent vector mode is only used when multiple projections are required at once, e.g. for computing a dense Jacobian explicitly, $\lceil n/p \rceil$ evaluations of the tangent vector mode are required.

### 3.2.2 Adjoint Mode

The dco/c++ type `ga1s<T>` implements adjoint 1st-order scalar mode, where all internal arithmetic is performed on data of generic type $T$. The adjoint modes of dco/c++ have considerably more features than the tangent modes. As already mentioned earlier (see Section 2.1), adjoint projections with AD require a data flow reversal consisting of a forward and a reverse section.
3.2 First- and Higher-Order Modes

Figure 3.1: Linearized DAG without and with assignment-level preaccumulation applied. Each vertex has a unique index (called tape index in dco/c++) and a corresponding adjoint value for the evaluation point \( x=1 \), \( z=1 \), and \( y^{(1)}=1 \). The edge labels are the local partial derivatives. The vertex sets \( \hat{X} \), \( \hat{Y} \), and \( \tilde{X} \), \( \tilde{Y} \) are the in- and outputs of the graphs \( \hat{G} \) and \( \tilde{G} \), respectively. Parallel edges in the preaccumulated graph exist, since the transformation is done by the compiler and the two occurrences of the variable \( x \) are not identifiable as being the same in the C++ programming language at compile time. A global renumbering took place as mentioned in Section 2.2.

\[ \hat{G} \]

\[ \tilde{G} \]

\[ X = \hat{X} \]

\[ Y = \tilde{Y} \]

\textbf{Remark.} Assignment-level preaccumulation is called statement-level reverse in Griewank’s book [GW08, Chapter 10.2].

In addition to assignment-level preaccumulation, dco/c++ features varied analysis by default. This is a well-known technique in AD [HNP05] and has the effect of only storing parts of the linearized DAG that do depend on the inputs somehow, i.e. for \( i \in X \), only add \( j \) to \( V \) if there exists a path from \( i \) to \( j \).

The first-order adjoint version results from the instantiation of \( F<\text{gals<T>}:\text{type}> \) and com-
putes with a proper driver

\[ y := F(x) \quad \text{and} \quad x_{(1)} := x_{(1)} + \left( \frac{\partial F}{\partial x} \right)^T \cdot y_{(1)} \tag{3.8} \]

with \( x, x_{(1)} \in \mathbb{R}^n \) and \( y, y_{(1)} \in \mathbb{R}^m \). An example driver computing function value and the row sum of the Jacobian

\[ y = F(x) \approx \begin{pmatrix} -0.0665 \\ -0.133 \end{pmatrix} \quad \text{and} \quad x_{(1)} = \left( \frac{\partial F}{\partial x} (x) \right)^T \cdot (1) \approx \begin{pmatrix} -0.200 \\ 0.0266 \\ 4.00 \\ 1.33 \end{pmatrix}, \tag{3.9} \]

at \( x = (1, 2, 1, 3)^T \) is shown in Listing 3.4.

```cpp
int main() {
  gals<double>::global_tape = gals<double>::tape_t::create();
  vector<gals<double>::type> x{1, 2, 1, 3};
  gals<double>::global_tape->register_variable(x);
  cout << "x = " << x << endl;

  auto y = F(x);
  gals<double>::global_tape->register_output_variable(y);
  derivative(y) = {1, 1};
  cout << "y_(1) = " << derivative(y) << endl;
  cout << "y = " << y << endl;

  gals<double>::global_tape->interpret_adjoint();
  cout << "x_(1) = " << derivative(x) << endl;

  gals<double>::tape_t::remove(gals<double>::global_tape);
  return 0;
}
```

Listing 3.4: main_gals.cpp. Code example for an adjoint projection with dco/c++.

After creating the tape in line 2, the forward section reaches from line 4 to line 10 consisting of activating the inputs in line 4, recording the tape in line 8 and marking the outputs in line 10. Afterwards the output adjoints \( y_{(1)} \) are assigned in line 12, the reverse section is triggered by the interpretation call in line 16, and the input adjoints \( x_{(1)} \) are extracted in line 18. At the very end, the tape is removed and its memory is released in line 20.

Compiling main_gals.cpp and running the executable produces the following output.

<table>
<thead>
<tr>
<th>Output of main_gals</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = [ 1 2 1 3 ]</td>
</tr>
<tr>
<td>y_{(1)} = [ 1 1 ]</td>
</tr>
<tr>
<td>y = [ -0.0665314 -0.133063 ]</td>
</tr>
<tr>
<td>x_{(1)} = [ -0.199594 0.026626 4.00081 1.3336 ]</td>
</tr>
</tbody>
</table>
The cost for evaluating the adjoint model once is

\[ \text{Cost}(\text{ga1s\textless{}\textgreater{}}) = O(1) \cdot \text{Cost}(\text{double}) \]  

(3.10)

with an approximate constant \(2 \sim 20\) (no strong bounds). The constant varies much more compared to tangent modes due to the huge amount of memory that needs to be accessed (the tape). For a more detailed theoretical analysis, see [GW08, Chapter 4.6: Complexity of Gradient Propagation]. The characteristics of a program may completely change due to the application of adjoint mode. E.g. a potentially originally computation bound program gets transformed into a memory bound one. Wisely, no hard bounds should be claimed therefore. Performance benchmarks for specific examples are shown in Section 6.2.1.

Though the adjoint mode seems excessively memory consuming due to the tape recording, it has also the immense advantage of being able to compute transposed Jacobian-vector products. Especially in the context of optimization problems or sensitivity analysis of a global scalar target functional (i.e. \(m = 1\)) a potentially extremely large gradient is required. The transposed Jacobian-vector product computes all the gradient entries with one adjoint projection. Taking into account the convenient way of applying an overloading tool in C++, the robust efficiency of dco/c++ shown later in Section 6.2.1, and the hybridization possibilities presented in this thesis, adjoint mode by AD is possibly quite gainful for a lot of applications, a few of which are described in Section 6.1. In addition to this most widespread application of the adjoint mode, it can also be used for computing dense or sparse Jacobians. As we will see later, usually the recording takes much more time than the interpretation. It may well be, that one interpretation is faster than a single tangent mode evaluation. Consequently, if \(n \approx m\), reusing the tape for \(m\) adjoint projections choosing \(y^{(1)}\) to be the Cartesian basis vectors in \(\mathbb{R}^m\) can be faster than doing the same with tangent projections. Sparsity can and should also be exploited in this case [CPR74; GMP05].

A helpful feature of the tape data structure is its capability of exporting its own data into a dot file that represents the recorded graph. With the Graphviz package\(^9\) [Ell+01] this graph can be visualized and printed to a pdf file. The tape recorded in the code Listing 3.4 generated the graph shown in Figure 3.2 (additional annotation were made).

**Vector Mode** The dco/c++ type \(\text{ga1v\textless{}\textgreater{}}\) implements adjoint 1st-order vector mode, where all internal arithmetic is performed on data of generic type \(\tau\). The vector length is defined by the positive integer template parameter \(p\). The overall logic corresponds exactly to the scalar adjoint mode \(\text{ga1s\textless{}\textgreater{}}\). The first-order adjoint vector version results from the instantiation of \(\text{F\textless{}\textgreater{}}\) and computes with a proper driver

\[ y := F(x) \quad \text{and} \quad x^{k(1)} := \left(\frac{\partial F}{\partial x}\right)^T \cdot y^{k(1)} \quad \text{for} \quad k \in \{1, ..., p\} \]  

(3.11)

with \(x \in \mathbb{R}^n\), \(y \in \mathbb{R}^m\), \(p \in \mathbb{N}^+\) the vector length, and adjoint vectors \(x^{k(1)} \in \mathbb{R}^n\), and \(y^{k(1)} \in \mathbb{R}^m\). An example driver computing function value and the full Jacobian

\[ y = F(x) \quad \text{and} \quad \left(x^{k(1)}\right)_{k=1,2} = \left(\frac{\partial F}{\partial x}(x)\right)^T \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \]  

(3.12)

at \(x = (1,2,1,3)^T\) is shown in Listing 3.5.

\(^9\)http://www.graphviz.org/
Figure 3.2: Graphical representation of the information stored in the dco/c++ tape for the example function Equation (3.1), see Listing 3.4 – to be pointed out that this visualization does not correspond to the actual memory layout of the tape. Additional annotations are added for a better understanding. Each variable registration and each assignment creates a new vertex in the graph. Red vertices correspond to inputs $x[0]$ to $x[3]$, while blue vertices correspond to outputs $y[0]$ and $y[1]$. Green vertices correspond to intermediate left-hand sides. Output vertices are marked by explicitly duplicating intermediate vertices. This is required to ensure different tape indices for each output. Each vertex consists of a unique tape index and an adjoint value (e.g. $y[1]$ has tape index 10 and adjoint value $y(1)_{10} = 1$). The edge weights correspond to the partial derivatives of successors with respect to predecessors.

```cpp
1 int main() {
2     galv< double, 2>:: global_tape = galv< double, 2>:: tape_t:: create();
3     vector<galv< double, 2>:: type> x(1, 2, 1, 3);
4     galv< double, 2>:: global_tape-> register_variable(x);
5     cout << "x = " << x << endl;
6     auto y = F(x);
7     galv< double, 2>:: global_tape-> register_output_variable(y);
8     for (int i = 0; i < 2; ++i) derivative(y[i])[i] = 1;
9     cout << "y(1) = " << derivative(y) << endl;
10    cout << "y = " << y << endl;
11 }
```
3.2 First- and Higher-Order Modes

Listing 3.5: main_galv.cpp. Code example for an adjoint vector projection with dco/c++. The vector size \( p \) is set to 2. Please see description of Listing 3.4. The only difference is the assignment of the output adjoint \( y_1 \) to the identity matrix from Equation (3.12) in line 12.

Compiling main_galv.cpp and running the executable produces the following output. Vector-valued derivative components are printed in curved brackets.

<table>
<thead>
<tr>
<th>Output of main_galv</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = [ 1 \ 2 \ 1 \ 3 ] )</td>
</tr>
<tr>
<td>( y_1 = [ (1;0) \ (0;1) ] )</td>
</tr>
<tr>
<td>( y = [ -0.06665314 \ -0.133063 ] )</td>
</tr>
<tr>
<td>( x_1 = [ (-0.06665314;\ -0.133063) \ (0.0310525;\ -0.00442642) \ (1.3336;\ 2.66721) \ (0.444535;\ 0.889069) ] )</td>
</tr>
</tbody>
</table>

The cost for evaluating the adjoint vector mode once is

\[
\text{Cost}(F<\text{galv}\langle\text{double}, \ p\rangle::\text{type}>) = O(p) \cdot \text{Cost}(F<\text{double}>) . \quad (3.13)
\]

A combination of the statements from adjoint scalar mode as well as tangent vector mode apply. The approximate constant is \( 0.5 \sim 20 \) (no strong bounds) and the vector length \( p \) needs to be carefully adjusted to the cache hierarchy of the architecture that is used.

The vector mode is always useful for \( m > 1 \) and in the presence of the need for multiple projections. This is primarily the case when computing Jacobians.

3.2.3 Higher-Order Modes

In the previous section the first-order data types of dco/c++ were introduced. It has been mentioned that the internal arithmetic is performed on a templated data type \( \tau \). This feature gets very convenient and powerful when computing higher-order derivatives. Higher-order modes are generated by nesting the first-order tangent or adjoint data types. This way the user can explicitly compose exactly the mode and corresponding type that is required. Explicitly nested types up to second order are introduced in this section. Though even higher modes are used later in this thesis, second-order modes fully illustrate the way how higher derivatives can be computed using dco/c++. Apart from the explicit nesting, also the compiler can automatically compose the required types where derivative nesting arises naturally or implicitly. This happens through instantiations of codes with a dco/c++ data type where the internal algorithm already uses dco/c++ to compute derivatives. A prime example is a sensitivity analysis of a Newton solver. The Newton solver uses first-order derivatives for computing the next Newton iterate. Consequently, running the Newton algorithm in tangent mode for example requires second derivatives implicitly. Multiple applications of such an implicit nesting will be shown later in this thesis in more detail in Section 4.4 and Section 6.1.4.

For being able to make more specific and less complex statements for what the second-order models actually compute, we use the multi-variate scalar function \( f : \mathbb{R}^n \rightarrow \mathbb{R}, y = f(x) \) with \( x \in \mathbb{R}^n \) and \( y \in \mathbb{R} \). We slightly change the default declaration from the previous section to


\[ y = f(x) : \mathbb{R}^4 \rightarrow \mathbb{R}, \quad y = \frac{|x_1| \cdot \tan(|x_3| \cdot |x_4|)}{|x_2| - \tan(|x_3| \cdot |x_4|)} + \frac{|x_2| \cdot |x_1| \cdot \tan(|x_3| \cdot |x_4|)}{|x_2| - \tan(|x_3| \cdot |x_4|)}, \quad (3.14) \]

where the scalar output \( y \) is the sum of the formerly defined outputs \([y]_1\) and \([y]_2\).

With \( x, y \in [t, a] \) and the type aliases

\[
\text{using TYPE_FO = gX1s<double>::type; // first-order data type}
\]

\[
\text{using TYPE_SO = gY1s<TYPE_FO>::type; // second-order data type}
\]

four second-order modes are possible by combining tangent and adjoint data types and instantiating \( F<\text{TYPE_SO}>\). Complexity-wise, the properties of the second-order models can be derived by a combination of the respective properties of the underlying first-order models. As mentioned in Section 2.1 and already shown in [GW08] and [Nau12], the three second-order adjoint models (which are those where at least one adjoint model is involved) are mathematically equivalent in terms of what they compute. In combination with \( \text{dco/c++} \) specific implementation details, they differ in terms of how they compute it and the runtime for computing full Hessian matrices are not equivalent at all. In Section 6.2.2 is therefore a benchmark for the different second-order models.

Tangent-over-Tangent

This model is generated for \( x = t \) and \( y = t \) and computes with a proper driver

\[
y := f(x), \quad (3.15a)
\]

\[
y^{(2)} := \frac{\partial f}{\partial x} \cdot x^{(2)}, \quad (3.15b)
\]

\[
y^{(1)} := \frac{\partial f}{\partial x} \cdot x^{(1)}, \quad (3.15c)
\]

\[
y^{(1,2)} := \frac{\partial^2 f}{\partial x_i \partial x_j} \cdot [x^{(1)}]_i \cdot [x^{(2)}]_j + \frac{\partial f}{\partial x} \cdot x^{(1,2)}, \quad (3.15d)
\]

with \( x, x^{(1)}, x^{(2)}, x^{(1,2)} \in \mathbb{R}^n \) and \( y, y^{(1)}, y^{(2)}, y^{(1,2)} \in \mathbb{R} \). Apart from the projection sums in those equations, usual application of the tangent-over-tangent mode include the computation of one Hessian entry \( \frac{\partial^2 f}{\partial x_i \partial x_j} \) for any \( i, j \) with \( x^{(1)} \) and \( x^{(2)} \) the \( i \)-th and \( j \)-th Cartesian basis vector in \( \mathbb{R}^n \), respectively, and \( x^{(1,2)} = 0 \). An example driver computing the projections for \( x^{(1)} = x^{(2)} = x^{(1,2)} = 1 \) and \( x = (1, 2, 1, 3)^T \) is shown in Listing 3.6.
3.2 First- and Higher-Order Modes

Listing 3.6: main_gt2s_gt1s.cpp. Code example for a second-order tangent projection with dco/c++ in tangent-over-tangent mode. The tangent components \( x^{(1)} \), \( x^{(2)} \), and \( x^{(1,2)} \) are assigned in lines 3 to 5 in the given order, the second-order tangent projection is carried out implicitly in line 12, and the output tangents \( y^{(1)} \), \( y^{(2)} \), and \( y^{(1,2)} \) are extracted in lines 15 to 17 in the given order.

Compiling main_gt2s_gt1s.cpp and running the executable produces the following output.

<table>
<thead>
<tr>
<th>Output of main_gt2s_gt1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \begin{bmatrix} 1 &amp; 2 &amp; 1 &amp; 3 \end{bmatrix} )</td>
</tr>
<tr>
<td>( x^{(1)} = \begin{bmatrix} 1 &amp; 1 &amp; 1 &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td>( x^{(2)} = \begin{bmatrix} 1 &amp; 1 &amp; 1 &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td>( x^{(1,2)} = \begin{bmatrix} 1 &amp; 1 &amp; 1 &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td>( y = -0.199594 )</td>
</tr>
<tr>
<td>( y^{(1)} = 5.16145 )</td>
</tr>
<tr>
<td>( y^{(2)} = 5.16145 )</td>
</tr>
<tr>
<td>( y^{(1,2)} = 31.6971 )</td>
</tr>
</tbody>
</table>

Tangent-over-Adjoint

This model is generated for \( x = \tau \) and \( y = a \) and computes with a proper driver

\[
\begin{align*}
y := f(x) & \quad (3.16a) \\
y^{(2)} := \left( \frac{\partial f}{\partial x} \right)^T x^{(2)} & \quad (3.16b) \\
x^{(1)} := \left( \frac{\partial f}{\partial x} \right)^T y^{(1)} & \quad (3.16c) \\
x^{(2)} := \left( \frac{\partial f}{\partial x} \right)^T y^{(2)} + y^{(1)} \cdot \frac{\partial^2 f}{\partial x^2} x^{(2)} & \quad (3.16d)
\end{align*}
\]

with \( x, x^{(1)}, x^{(2)} \in \mathbb{R}^n \) and \( y, y^{(1)}, y^{(2)} \in \mathbb{R} \). Apart from the projection sums in those equations, usual applications of the tangent-over-adjoint mode include the computation of one row of the Hessian (or a column – due to symmetry). In addition, matrix-free linear solvers with the Hessian as system matrix can use this model as an implicit Hessian-vector product in Equation (3.16d) with \( y^{(1)} = 1 \) and \( y^{(2)} = 0 \). As an adjoint mode is involved, the implementation has a forward and a reverse section. The notion behind this mode is that we take the first-order
adjoint mode and perform each operation on a data type of first-order tangent mode. Everything that was a value formerly now gets a tangent data type. That is especially important for assigning and extracting the respective tangent and adjoint components. An example driver computing the projections for $x^{(2)} = 1$, $y^{(1)} = 1$ and $x = (1, 2, 1, 3)^T$ is shown in Listing 3.7.

```cpp
int main() {
    using adjoint_mode = gals< gtls<double>::type >;
    adjoint_mode::global_tape = adjoint_mode::tape_t::create();
    auto x = vector< adjoint_mode::type >{1, 2, 3, 1, 3};
    adjoint_mode::global_tape->register_variable(x);
    derivative(value(x)) = {1, 1, 1, 1};
    cout << "x = " << x << endl;
    cout << "x^{(2)} = " << derivative(value(x)) << endl;
    auto y = f(x);
    adjoint_mode::global_tape->register_output_variable(y);
    cout << "y = " << y << endl;
    cout << "y^{(2)} = " << derivative(value(y)) << endl;
    value( derivative(y)) = 1;
    derivative(derivative(y)) = 1;
    adjoint_mode::global_tape->interpret_adjoint();
    cout << "y_{(1)} = " << value( derivative(y)) << endl;
    cout << "y_{(1)}^{(2)} = " << derivative(derivative(y)) << endl;
    x_{(1)} = (1, 2, 1, 3)^T
    x_{(1)}^{(2)} = (5.16145 -0.199594 4.00081 1.3336)
    adjoint_mode::tape_t::remove(adjoint_mode::global_tape);
    return 0;
}
```

Listing 3.7: main_gt2s_ga1s.cpp. Code example for a second-order adjoint projection with dco/c++ in tangent-over-adjoint mode. For convenience a type alias is introduced for the higher-level adjoint type in line 2. The overall structure is exactly the same as in the first-order adjoint mode Listing 3.4. The only difference is the assignments of the tangents $x^{(2)}$ in line 7 and $y^{(2)}$ in line 20.

Compiling main_gt2s_ga1s.cpp and running the executable produces the following output.

```
x = [ 1 2 1 3 ]
x^{(2)} = [ 1 1 1 ]
y = -0.199594
ey^{(2)} = 5.16145
y_{(1)} = 1
y_{(1)}^{(2)} = 1
x_{(1)}^{(1)} = [ -0.199594 0.026626 4.00081 1.3336 ]
x_{(1)}^{(2)} = [ 5.16145 -0.505766 19.6143 7.42716 ]
```

An interesting detail concerning this mode is that the second-order tangent direction $x^{(2)}$ needs to be set before recording the tape (cf. line 7 and 12 in Listing 3.7). This is required since $x^{(2)}$...
3.2 First- and Higher-Order Modes

has an influence on the second-order partials that are stored in the tape during recording. This in turn means that the tape must be re-recorded for different tangent directions $x^{(2)}$. Opposed to the adjoint-over-adjoint mode, this may be a drawback when computing full Hessians as shown in the benchmark in Section 6.2.2.

**Adjoint-over-Tangent**

This model is generated for $x = a$ and $y = \tau$ and computes with a proper driver

$$
y := f(x) \tag{3.17a}
$$

$$
y^{(1)} := \frac{\partial f}{\partial x} \cdot x^{(1)} \tag{3.17b}
$$

$$
x^{(2)} := \left( \frac{\partial f}{\partial x} \right)^T \cdot y^{(2)} + y^{(1)} \cdot \frac{\partial^2 f}{\partial x^2} \cdot x^{(1)} \tag{3.17c}
$$

$$
x^{(1)}_2 := \left( \frac{\partial f}{\partial x} \right)^T \cdot y^{(1)} \tag{3.17d}
$$

with $x, x^{(1)}, x^{(2)} \in \mathbb{R}^n$ and $y, y^{(1)}, y^{(2)} \in \mathbb{R}$. The adjoint-over-tangent mode has no advantages over the tangent-over-adjoint mode described previously. Mathematically as well as implementation-wise in dco/c++ both modes are very similar. Nonetheless, the driver looks differently compared to the tangent-over-adjoint. It is shown for $x^{(1)} = 1$, $y^{(2)} = y^{(1)} = 1$ and $x = (1, 2, 1, 3)^T$ in Listing 3.8.

```cpp
int main() {
    using gals_type=gals<double>::type;
    gals<double>::global_tape = gals<double>::tape_t::create();
    vector<gtls<gals_type>::type> x{1, 2, 1, 3};
    value(derivative(x)) = {1, 1, 1, 1};
    cout << "x = " << x << endl;
    cout << "x^{(1)} = " << value(derivative(x)) << endl;
    gals<double>::global_tape->register_variable(value(x));
    gals<double>::global_tape->register_variable(derivative(x));
    auto y = f(x);
    cout << "y = " << y << endl;
    cout << "y^{(1)} = " << value(derivative(y)) << endl;
    gals<double>::global_tape->register_output_variable(value(y));
    gals<double>::global_tape->register_output_variable(derivative(y));
    derivative(value(y)) = 1;
    derivative(derivative(y)) = 1;
    cout << "y^{(2)} = " << derivative(derivative(y)) << endl;
    cout << "y^{(1)}_{(2)} = " << derivative(derivative(y)) << endl;
    gals<double>::global_tape->interpret_adjoint();
}
```
Listing 3.8: main_ga2s_gt1s.cpp. Code example for a second-order adjoint projection with dco/c++ in adjoint-over-tangent mode. For convenience a type alias is introduced for the first-order adjoint type in line 2. The overall structure of the first-order adjoint mode Listing 3.4 is applied to the first-order tangent mode. Registering inputs and outputs needs therefore to additionally take place for the respective tangent components in lines 11-12 and 19-20. The tangent $y^{(1)}$ is already computed by the forward section, thus can be printed before the interpretation, cf. lines 17 and 28.

Compiling main_ga2s_gt1s.cpp and running the executable produces the following output.

---

Output of main_ga2s_gt1s

\[
\begin{align*}
x & = [1 \ 2 \ 1 \ 3] \\
x^{(1)} & = [1 \ 1 \ 1 \ 1] \\
y & = -0.199594 \\
y^{(1)} & = 5.16145 \\
y^{(1)}_{(1)} & = 1 \\
x^{(1)}_{(1)} & = [5.16145 \ -0.505766 \ 19.6143 \ 7.42716] \\
x^{(1)}_{(1)} & = [-0.199594 \ 0.026626 \ 4.00081 \ 1.33336]
\end{align*}
\]

---

Adjoint-over-Adjoint

This model is generated for $x = a$ and $\gamma = a$ and computes with a proper driver

\[
y := f(x)
\]

\[
x^{(1)} := \left(\frac{\partial f}{\partial x}\right)^T \cdot y^{(1)}
\]

\[
y^{(1)}_{(1)} := \frac{\partial f}{\partial x} \cdot x^{(1)}
\]

\[
x^{(2)} := \left(\frac{\partial f}{\partial x}\right)^T \cdot y^{(2)} + y^{(1)} \cdot \frac{\partial^2 f}{\partial x^2} \cdot x^{(1)}
\]

with $x, x^{(1)}, x^{(2)}, x^{(1)} \in \mathbb{R}^n$ and $y, y^{(1)}, y^{(2)}, y^{(1)} \in \mathbb{R}$. As already mentioned, mathematically this mode is equivalent to the previous two second-order adjoint modes. Apart from that, it is commonly propagated that the adjoint-over-adjoint mode is less efficient compared to one of the other second-order adjoint modes since it involves the data flow reversal twice. In conjunction with implementation details of dco/c++ (and probably other overloading tools which are based on preaccumulation techniques) this statement is not unconditionally true as shown in Section 6.2.2. Unfortunately, the usage of this mode is quite complicated because the two reversals introduce two different tapes in dco/c++.

Instead of a driver in form of source code only the principle of the data flow is shown and described in Figure 3.3. This way the complicated syntax is circumvented partially. The most interesting observation in Figure 3.3 is the fact that the ga2s_mode::global_tape including its interpretation is independent of the projection direction $x^{(1)}$ in Equation (3.18d). It only depends
3.2 First- and Higher-Order Modes

Figure 3.3: The principle data flow for a second-order adjoint projection with dco/c++ in adjoint-over-adjoint mode. \texttt{ga1s\_mode} corresponds to \texttt{ga1s\<double\>::type}. The part of the code which originates from the first-order adjoint (cf. Listing 3.4) including its tape is colorized in red and orange. The part which originates from the second application of adjoint mode including its tape is colorized in blue. The boxes stand for the tapes and the attached expressions with subscript (1) – including their mathematical equivalent – indicate positions of the adjoints of corresponding variables. Both modes have a forward and a reverse section. The forward section of \texttt{ga1s\_mode} consists of the recording of \( f \) with \texttt{ga2s\_mode} (in red) and the reverse section is the interpretation of the corresponding tape \texttt{ga2s\_mode::global\_tape} (in orange). Both parts (red and orange) are contained in the forward section of \texttt{ga1s\_mode::global\_tape} (blue tape).

on \( y_{(1)} \) which is fixed and equal to 1. Therefore, the recording of \texttt{ga1s\_mode::global\_tape} does not depend on \( x_{(1,2)} \) either. \( x_{(1,2)} \) does not enter the computation until assigned in \texttt{ga1s\_mode::global\_tape} as adjoints. Consequently, no re-recording of any tape needs to take place for projecting the Hessian into different directions. This proves to be beneficial in the benchmark Section 6.2.2.

Vector Modes

All second-order modes can potentially also be instantiated with the vector modes from Section 3.2.1 or Section 3.2.2. No crucial insight is expected to be gained from the corresponding mathematical models and they are therefore omitted here. Nonetheless, for the benchmark for computing full Hessians, the vector modes turn out to perform slightly better. The reasonable
combinations are therefore introduced in Section 6.2.2 when used.

### 3.3 High-Level Intrinsics

This section is closely related to Section 2.2, since it describes the interface of \texttt{dco/c++} for hybridization. Though not explicitly part of \texttt{dco/c++}, the examples implicitly also suggest coding guidelines for how to use this interface.

From now on, based on [Nau+15], the following overall problem structure

\[ z = p(F(P(\lambda))) \] (3.19)

is assumed to be given with global parameters \( \lambda \in \mathbb{R}^q \), the output \( z \in \mathbb{R}^w \), a preprocessor \( P : \mathbb{R}^q \rightarrow \mathbb{R}^n \), a postprocessor \( p : \mathbb{R}^m \rightarrow \mathbb{R}^w \), and the embedded function \( F : \mathbb{R}^n \rightarrow \mathbb{R}^m \). Algorithmically this problem is realized as

\[ x := P(\lambda), \quad y := F(x), \quad \text{and} \quad z := p(y) . \] (3.20)

The important property is that we capture all inputs of \( F \) in \( x \) and all outputs of \( F \) in \( y \), i.e. the function \( F \) has algorithmically no side effects. The part of the linearized DAG that corresponds to the evaluation of \( F \) is therefore valid for hybridization in the sense of Definition 9. According to Section 2.2, the input set \( \hat{X} \) corresponds exactly to the variables \( x \) and the output set \( \hat{Y} \) to the variables \( y \). For simplifying the notation, the \( \hat{\ } \) is omitted for the variables \( x \) and \( y \) in comparison to the notation in Section 2.2.

We assume that pre- and postprocessor are differentiated using \texttt{dco/c++} and a seamless integration of \textit{high-level intrinsics} for tangents and adjoints of the embedded function \( F \) is desired. A descriptive definition of that term is given in the following.

**Definition 12.** Given a component of a program whose respective linearized DAG is valid for hybridization in the sense of Definition 9, a high-level intrinsic of \texttt{dco/c++} enhances this component with special implementations regarding its derivative computation. The enhancement is usually related to computational efficiency and memory requirements, and potentially exploits knowledge about mathematical properties. From the user’s perspective a seamless integration is desired, i.e., making use of this enhancement requires as little changes as possible in the surrounding code components.

**Example 5.** The main function for computing an adjoint projection of the problem structure Equation (3.20) with \texttt{dco/c++} could look like the following listing. The data type \texttt{gals<
double>::type} is used. From the user’s perspective, the way of calling \( F \) is independent of the underlying derivative implementation: either by plain operator and function overloading or by making use of a high-level intrinsic. Of course, plain overloading is only possible if the source code is available.

```cpp
template <typename T> vector<T> P(const vector<T> &lambda);
template <typename T> vector<T> F(const vector<T> &x);
template <typename T> vector<T> p(const vector<T> &y);

int main() {
    vector< gals<double>::type > lambda{1, 2, 3, 4};
    gals<double>::global_tape = gals<double>::tape_t::create();
```
3.3 High-Level Intrinsics

gals<double>::global_tape->register_variable(lambda);

auto x = P(lambda);
auto y = F(x);
auto z = p(y);

for (auto &zi : z) derivative(zi) = 1.0;

gals<double>::global_tape->interpret_adjoint();

cout << "x'((l)) = " << derivative(x) << endl;
return 0;
}

Listing 3.9: Application of first-order adjoint mode of dco/c++ to the main function. Please compare Listing 3.4 for a description.

dco/c++ provides two different approaches to realize this task. The first approach is called user-defined tangents and adjoints. Logically, we decouple the computation of derivative projections of the embedded function completely from dco/c++ via template specialization and callback functions. This approach very generically allows for intercepting the usual derivative computation of dco/c++ and can be used for symbolic adjoint approaches (see Chapter 4), a coupling with compiler generated code (see Section 3.3.3), for checkpointing (cf. Chapter 5), or for even more general tasks like gathering information about derivatives or interfacing to other AD libraries like, e.g., adjoint MPI [Sch+10]. The other approach is called local gradients and only possible for adjoint mode. A (optimized) graph transformation (see Section 2.2) is performed manually and the resulting graph is added to the tape. This approach is less generic, but very useful for small program components that are frequently called.

In this section template specialization techniques are used for the tangent and adjoint data type of dco/c++. For making this more convenient two templated type aliases are introduced:

```cpp
template<typename T> using gtls_type = gtls<T>::type;
template<typename T> using gals_type = gals<T>::type;
```

Most of the functionality we use template specialization for could also be achieved by overloading. Nonetheless, the specialization approach is much more generic, which can get very important especially for higher-order modes and implicit nesting of dco/c++ data types.

3.3.1 User-Defined Tangents and Adjoints

**Tangent Mode** User-defined tangents can be added to dco/c++ by providing a specialized implementation for the embedded function $F$. This specialized function needs to be called whenever the template parameter $\tau$ in Listing 3.9 is deduced to $\text{tls}_{\text{type}}<U>$ for the original function $F$. $U$ is another template typename parameter and is required to enable this specialization for any base type $U$—also for higher-order nested dco/c++ types. With the signature of $y = F(x)$ given in Listing 3.9, the specialized declaration is given as

```cpp
template<typename U>
vector<gtls_type<U>> F(const vector<gtls_type<U>> &x);
```
Figure 3.4: Graphical representation of the data flow for a user-defined tangent projection with dco/c++. The input $x$ gets split into its value and tangent component $x$ and $x^{(1)}$. The external implementation of the tangent projection $t_1_F$ is called computing output value and tangent component $y$ and $y^{(1)}$. Both components are then joined again into the output variable $y$.

Inside this function the input value and tangent components are split ($x$ and $x^{(1)}$), the mapping

$$y := F(x) \quad \text{and} \quad y^{(1)} := \frac{\partial F}{\partial x} \cdot x^{(1)}$$

is implemented externally, and the output value and tangent components ($y$ and $y^{(1)}$) are then joined again. This corresponds to the splitting into three parts of the tangent mode in Equation (2.27). Under the assumption that the external tangent projection is implemented in a function

```cpp
template <typename T>
void t1_F(const vector<T> &x, const vector<T> &t1_x, vector<T> &y, vector<T> &t1_y);
```

the corresponding data flow is shown in Figure 3.4. The conforming code for implementing the data flow from the figure is given in Listing 3.10.

```cpp
1 template <typename U>
2 vector<gtls_type<U>> F(const vector<gtls_type<U>> &x) {
3 vector<gtls_type<U>> y (2);
4 vector<U> py (2), t1_py (2);
5 vector<U> px = value(x);
6 vector<U> t1_px = derivative(x);
7 t1_F(px, t1_px, py, t1_py);
```
3.3 High-Level Intrinsics

```cpp
value(y) = py;
derivative(y) = t1_py;
return y;
}
```

Listing 3.10: gtis_hybrid_f.hpp: User-defined tangent projections with dco/c++. Besides the original return value \( y \), the passive values \( py \) and \( t1_py \) of type \( U \) (the base type) are allocated. After splitting input value and tangent components in lines 6-7, the external tangent projection \( t1_F \) is called in line 9 and the output value and tangent components are joined again in lines 11-12.

**Adjoint Mode** User-defined adjoints can be added to dco/c++ in a similar way, though a more complex interaction is required due to the data flow reversal. Special implementations need to be provided not only for the forward but also for the reverse section, i.e., for the tape recording and the tape interpretation step. During recording only function values

\[
y := F(x)
\]  

need to be computed. In addition, a callback function is supposed to perform the adjoint projection

\[
x(1) := x(1) + \left( \frac{\partial F}{\partial x} \right)^T \cdot y(1)
\]  

during tape interpretation. Here the incremental adjoint mode is crucial, see Equation (2.28), and therefore embedded into the interface itself. The former step is again – likewise the tangent case – realized with template specialization, while the latter makes use of a callback mechanism. The template specialization looks similar to the one in the previous section and is given as

```cpp
template <typename U>
vector< gals_type<U> > F(const vector< gals_type<U> > &x);
```

With an object called `external_adjoint_object_t`, data required for the adjoint computation can be transferred from the forward section to the reverse section. Filled with data, this object is passed to the callback function

```cpp
template <typename U>
void callback_F(external_adjoint_object_t< gals_type<U> > &D);
```

which gets triggered by the tape interpreter, when the data flow reversal reaches the point where the adjoint projection of \( F \) needs to be performed. This corresponds to the splitting into three parts of the incremental adjoint mode in Equation (2.28). Under the assumption that an external adjoint projection is implemented in a function

```cpp
template <typename T>
void a1_F(const vector<T> &x, vector<T> &a1_x, vector<T> &a1_y);
```

the corresponding data flow (forward and reverse section) is shown in Figure 3.5. For understanding the figure and the following implementations, a few further features of dco/c++ need to be introduced here. The most relevant one is the external adjoint object which only exists for adjoint modes. It is the central element of the interface for user-defined adjoints. This type
combines the information about inputs and outputs with data to be stored and transferred to the callback function. Memory for the external adjoint object is allocated and managed by the tape. The tape therefore acts as a factory for the `external_adjoints_object_t`. Very convenient is the capability of storing and restoring arbitrary data in and from the data object via templated member functions `write_data` and `read_data`, respectively. Though in the example only one object is written (a `std::vector<U>`), the general logic is consistent with a `queue`, i.e. a first-in-first-out data structure, see Listings 3.11 and 3.12 in the following.

The conforming implementation of the specialized function for the forward section is given in Listing 3.11 and the implementation of the callback function for the reverse section in Listing 3.12.

```cpp
1 template <typename U>
2 vector< gals_type<U> > F(const vector< gals_type<U> > &x) {
3     auto *D = dco::tape(x)->template
4         create_callback_object< external_adjoints_object_t< gals_type<U> > >();
5
6     vector<U> px = D->register_input(x);
7     D->write_data(px);
8
9     vector<U> py = F(px);
10    dco::tape(x)->insert_callback(callback_F<U>, D);
11
12    return D->register_output(py);
13 }
```

Listing 3.11: `gals_hybrid_.hpp`: Forward section of the user-defined adjoint projection with dco/c++. First, in lines 3 and 4 an external adjoint object is created by the tape `x` was registered in. Inputs are registered to the object in line 6. This function call returns a `vector<U>` with `px=x`. `px` is written to the external adjoint object in line 7 because it is required for `a1_f` in the callback function later. The original function gets invoked in line 9 followed by inserting the callback into the tape in line 10 and returning the registered output of type `vector< gals_type<U> >` in line 12. Compare left-hand side of Figure 3.5 for a graphical representation of the data flow.

```cpp
1 template <typename U>
2 void callback_F(external_adjoints_object_t< gals_type<U> > *D) {
3     const vector<U>& px = D->template read_data< vector<U> >();
4
5     vector<U> a1_py(2), a1_px(px.size());
6     D->get_output_adjoints(a1_py);
7
8     a1_F(px, a1_px, a1_py);
9
10    D->increment_input_adjoints(a1_px);
11 }
```

Listing 3.12: `gals_hybrid_.hpp`: Reverse section of the user-defined adjoint projection with dco/c++. The callback function gets the external adjoint object `D` as an input. First, `D` is used in line 3 to read the data (here `x`) as a `const vector<U>&` which circumvents a copy. If required, a copy can be made by omitting the `&`. The adjoint values `y(1)` of the output are fetched from the tape in line 6 followed by calling the external adjoint projection `a1_f` in line 8. At the end, the computed input adjoints `x(1)` are incremented in the tape. Compare right-hand side of Figure 3.5 for a graphical representation of the data flow.

For being save in terms of memory management, all types need to have copy constructors for deep copying and a corresponding destructor.
When using the tape's feature of exporting its data to a graph, one can see the actual derivative information which is stored. In Figure 3.6 a graphical representation is shown. Here the pre- and postprocessor are chosen to be identities, therefore the graph can directly be compared to the graph without the user-defined adjoint. As mentioned in the introduction of this section, the tape does not know how to compute partial derivatives of the embedded function since the projection is completely decoupled through the callback mechanism. When comparing both tapes from Figure 3.6, already for the small example we can observe that less nodes and edges
need to be saved — although for a decent memory comparison the size of the external adjoint object including the stored data would also need to be taken into account.

This kind of treatment for higher-level intrinsics is very well suited for large functions, where the computational complexity dominates over the complexity of the amount of data to be copied and shifted around. A positive example is the symbolic treatment of a nonlinear solver shown in more detail in Section 4.2. When differentiating a nonlinear solver symbolically, the solution vector of the nonlinear system is the only data that has to be stored for the adjoint projection. During the adjoint computation, the solution is then used to build up a Jacobian matrix which in turn is the system matrix of a linear system to solve. It already gets clear, that the computation outweighs the amount of memory by far. In addition, a huge amount of tape memory is avoided, since the algorithm to solve the nonlinear system can be executed without taping. On the other hand, when looking at quite small functions, where computational complexity and the amount of memory are roughly in the same order, preaccumulated local gradients are the more efficient way of treating high-level intrinsics in dco/c++; this is presented in the next section.

3.3.2 Local Gradients

This feature is only available for adjoint modes. The interface opens up the internal data structure of dco/c++ to add arbitrarily transformed subgraphs to the resulting dco/c++ tape. As explained in Section 2.2, only valid transformations should be performed such that the incremental adjoint projection is still correct. Using this interface for a special treatment of
3.3 High-Level Intrinsics

\( F \) avoids the callback mechanism required for user-defined adjoints and the overhead of bookkeeping external adjoint data objects or function pointers.

Under the assumption that we retrieved explicit expressions for computing the edge labels for the transformed subgraph \( \tilde{G} \) from somewhere – either manually with pen and paper or by a computer algebra system – we can add vertices and edges including their labels to the graph via the type \( \text{local\_gradient\_t} \). Memory for the local gradient object is allocated by the tape – consistent with the memory management of the external adjoint data object presented in the previous section. In the following the interface is explained and demonstrated for the multi-variate scalar function

\[
y = g(x) = \sum_{i=1}^{n-1} |x|_i \cdot |x|_{i+1} \quad \text{for } g : \mathbb{R}^n \to \mathbb{R}, \ x \in \mathbb{R}^n, \text{ and } y \in \mathbb{R} \quad (3.24)
\]

with partials

\[
\frac{\partial y}{\partial |x|_i} = |x|_2, \quad \frac{\partial y}{\partial |x|_{i-1}} = |x|_{i-1} + |x|_{i+1} \quad \text{for } 1 < i < n, \text{ and } \quad \frac{\partial y}{\partial |x|_n} = |x|_{n-1}. \quad (3.25)
\]

The original function implementation is given in Listing 3.13.

```cpp
template<typename T>
T g(const vector<T>& x) {
    T y = 0;
    for (size_t i = 0; i < x.size()-1; ++i)
        y += x[i]*x[i+1];
    return y;
}
```

Listing 3.13: The original implementation for the function \( g \) in Equation (3.24). The sum is implemented as a loop incrementing the local variable \( y \) that gets returned at the end.

Usual taping in \texttt{dco/c++} appends one vertex including partials attached to corresponding edges for each instance of the program variable \( y \), i.e. \( n-2 \) vertices with 3 edges each: \( y \) depends on \( x[i] \), \( x[i+1] \), and itself. Having the direct expression for the partials Equation (3.25) at hand, the dependency could be stored in a more compact form. In fact, only one vertex with \( n \) edges and corresponding partials is required. A comparison of the resulting tapes for \( n = 4 \) is shown in Figure 3.7. For this example, handling this function as intrinsic saves 50% of the edges (8 vs. 4) and two-thirds of the vertices (3 vs. 1). If such a function is called a lot of times in the overall program, a huge benefit is expected – not only in terms of memory consumption, but also for the runtime of the interpretation step due to the smaller tape.

For the implementation, \( g \) gets specialized for adjoint data types \texttt{gals\_type<U>} like in the previous sections and in addition to the function value computation, the local gradient interface is used. The specialized function is shown and described in Listing 3.14.

```cpp
1 template<typename U>
2 gals\_type<U> g(const vector<gals\_type<U>> &x) {
3     gals\_type<U> y;
4     size_t n = x.size();
5     auto grad = dco::tape(x)->template
6     12See assignment-level preaccumulation in Section 3.2.2
```
Figure 3.7: Graphical representation of the tape for the example function Equation (3.24) for \( n = 4 \). On the left-hand side the tape with the subgraph \( \hat{G} \) is shown for the original implementation Listing 3.13. In comparison on the right-hand side the resulting tape with the transformed subgraph \( \tilde{G} \) is shown when using the local gradient interface Listing 3.14. The intermediate vertices \( \hat{Z} \) (in green, overwritten instances of \( y \)) are removed from the graph completely. The red vertices are inputs and blue vertices are outputs. A global renumbering took place as mentioned in Section 2.2.

Listing 3.14: High-level intrinsic implementation using the local gradient interface of dco/c++. This implementation computes the function value and adds the derivative information to the tape explicitly using an object of type \( \text{local\_gradient\_t} \). The interface for constructing this object in lines 6 and 7 takes the output variable \( y \) and an upper bound on the number of edges that are expected to be added later. The function value is computed on the value components of the variables in lines 9 and 13. Gradient entries are added by calling the member function put of the object \( \text{grad} \). Apart from an index shift, lines 10, 14, and 16 implement exactly Equation (3.25).

In contrary to the great flexibility of the user-defined adjoints (Section 3.3.1), the option of manipulating the tape directly pursues only one goal. Whenever the programmer knows where and how to transform the linearized DAG locally such that it gets smaller or such that edge
labels are computed more cheaply, the interface for local gradients should be used.

### 3.3.3 Coupling with an AD Source Transformation Tool

It is widely accepted and also observed in many cases that C/C++ source-to-source AD compilers generate more efficient tangent and adjoint code than overloading tools do\(^\text{13}\). On the other hand, the major drawback of current AD compilers is that they do not cover the full set of C features nor nearly the full set of C++ features. However, one can luckily observe that many numerical kernels are written only in a small subset of those languages. Therefore it promises to be beneficial to couple overlapping and compiler capabilities and combining great flexibility with good efficiency. In the following the coupling of dco/c++ with the compiler dcc is discussed.

dcc is just an example here, a prominent alternative would be e.g. Tapenade [HP13]. dcc [Han+10; Nau12; Sch+12; Foe14] is a source-to-source compiler that generates first- and higher-order tangent and adjoint code for numerical programs written in a subset of C/C++. It is also developed at the institute for Software and Tools for Computational Engineering (STCE) at the RWTH Aachen.

The coupling of dcc and dco/c++ is very straightforward with the interface described for user-defined tangents and adjoints Section 3.3.1. Especially the interface to the externally implemented tangent and adjoint functions used in the other section is very close to what dcc generates. We therefore just briefly look at the signatures dcc produces. A function 

\[ y = F(x), F : \mathbb{R}^n \to \mathbb{R}^m, x \in \mathbb{R}^n, \text{ and } y \in \mathbb{R}^m \]

is assumed to be implemented with the signature given as

```c
void F(int n, int m, double* x, double* y)
#pragma ad indep x
#pragma ad dep y
```

where \( x \approx x \) and \( y \approx y \). The signature changed compared to the example in the previous sections, since dcc only handles raw pointers without any further qualifiers. Directly after the declaration, dcc expects pragma-based information about independents (inputs \( x \)) and dependents (outputs \( y \)). This information is used for activity / useful / to-be-recorded analysis [HNP05].

For getting a code that computes the tangent projection

\[ y := F(x) \quad \text{and} \quad y^{(1)} := \frac{\partial F}{\partial x} \cdot x^{(1)}, \quad (3.26) \]

we apply dcc in tangent mode and get the respective code with the signature

```c
// ++++++++++++++++++++++++++++++++++
// + Code generated by dcc 1.0 +
// ++++++++++++++++++++++++++++++++++
void t1_F(int n, int m, double* x, double* t1_x, double* y, double* t1_y)
#pragma ad indep x t1_x
#pragma ad dep y t1_y
```

where \( t1_x \approx x^{(1)} \) and \( t1_y \approx y^{(1)} \). dcc automatically adds the correct dependency pragmas for the newly introduced variables to ensure a correct reapplicability to its own output. The input tangents are additional inputs and the output tangents additional outputs, since the data flow

---

\(^{13}\)With further development of the C++ standard and coming along more compile time programming capabilities (e.g. template meta programming or constexpr syntax), overloading tools may close the efficiency gap in the future.
is unchanged in tangent mode. With only a few adjustments, this function could be used in Listing 3.10, Section 3.3.1.

For getting a code that computes the adjoint projection

\[ x(1) := x(1) + \left( \frac{\partial F}{\partial x} \right)^T \cdot y(1) \]  

(3.27)

we apply dcc in adjoint mode and get the respective code with the signature

```
// ++++++++++++++++++++++++++++++++++
// + Code generated by dcc 1.0 +
// ++++++++++++++++++++++++++++++++++
void a1_F(int bmode_1, int n, int m, double* x, double* a1_x,
          double* y, double* a1_y)
#pragma ad indep x a1_x a1_y
#pragma ad dep y a1_x
```

where a1_x=\(x(1)\) an additional in- and output and a1_y=\(y(1)\) an additional input. In adjoint mode, dcc introduces a new integer parameter called bmode_1 that controls if only function values are computed (set to 2) or if the adjoint is computed in joint mode (set to 1), i.e. the forward and the reverse section is executed. The joint mode is explained in more detail in Section 5. In the recording part of the user-defined adjoint we can now either call the original function or the dcc-generated one with bmode_1=2. The callback function needs to call a1_F with bmode_1=1 to actually compute adjoints. With those changes in mind, a1_F can be used in Listing 3.11 and 3.12, Section 3.3.1.

As can be seen, the user-defined tangent and adjoint interface is well-suited for this coupling. As a side remark, the approach shown here is also very closely related to the general checkpointing approach not looked at in more detail here in the context of dco/c++ (see [Nau12, Chapter 2.3] for a practical view on this).

### 3.4 AD Test Suite

For evaluating the performance of dco/c++ and other AD overloading tools in C++, the AD test suite has been developed by the author with constant support of S. Gezgin (student researcher). The goal of this development was twofold. On the one hand, an internal benchmarking is required: An enhancement implemented in dco/c++ needs to be diagnosed as such. Since the runtime ratios especially for the adjoint mode vary so much from application to application, a benchmarking with a larger test problem collection seems the reasonable way to go. On the other hand, a relative benchmarking with respect to other AD overloading tools for C++ is informative.

Emphasis is put on benchmarking a plain application of first-order adjoint mode for computing the gradient of a scalar objective function. The test suite is able to automatically gather runtime measurements for five different AD overloading tools and a set of test problems that can be executed per tool and test for a range of problem sizes. Key is a single common interface adapter for all tools that is used in the implementations of the test problems. The adapter takes into account the differences between the respective tool interfaces and combines the logic of the tools altogether. The generically written test problems can then be instantiated and executed with the data types the different tools provide. In the following an overview is given about
3.4 AD Test Suite

Figure 3.8: Overall test suite structure. In the upper left corner is the main python driver. It takes as user input the tool/test-combinations to be benchmarked. If --build is set, the build system is triggered to generate all required executables taking per tool configuration and per test configuration files into account that may define special include paths, special libraries to link, or compilers/compiler flags to use (box on the lower left). After that, the benchmark is executed for the selected combinations (upper right) and the SQL database is filled with measurement data. This data can later be read and used by the visualizer (lower right).

how the test suite is designed in general and how the common interface adapter is designed in particular. A description of the tools and test problems supported by the test suite as well as results are shown in Section 6.2.1.

The test suite is written in python and C++, but also includes a nontrivial build system which makes use of SCons\textsuperscript{14}. The main python driver triggers the required builds and runs the various executables. Each executable built by SCons represents the benchmark for one test/tool-combination. The tool selection is done by a \texttt{typedef} and the test selection is done via the include search path as explained in further detail later on. The measured data – consisting of runtime and memory consumption – is collected in an underlying SQL\textsuperscript{15} database. For visualization, a python script is used in combination with PyQt\textsuperscript{16} and matplotlib\textsuperscript{17} for being able to interactively choose data set and visualization type. The overall test suite structure is shown in Figure 3.8. The rationale we pursued was a modular software design to have on the one hand as less duplicated code as possible but featuring on the other hand many tools and tests in a generic and flexible way. We achieved this by using inheritance and template programming in C++ together with making use of the C preprocessor. The source code is organized as nested git\textsuperscript{18} repositories to split build system, main python driver and most importantly the test problem collection from each other. This way additional tests can be implemented, added, and committed without requiring to use the complete test suite machinery.

Each tool/test-combination gets its own executable. This is required to keep the executables small and to circumvent any weird influence among the various tools and their libraries we have

\textsuperscript{14}http://www.scons.org/
\textsuperscript{15}http://en.wikipedia.org/wiki/SQL, currently sqlite is used.
\textsuperscript{16}http://sourceforge.net/projects/pyqt/
\textsuperscript{17}http://matplotlib.org/
\textsuperscript{18}http://git-scm.com/
Figure 3.9: Structure of the source files and linking procedure for generating an executable for one tool/test-combination; description in text.

observed otherwise. All executables are compiled from two source files: main.cpp (same for all tools and tests) and run_[tool].cpp. The structure of those files and the linking procedure is shown in Figure 3.9. For the tool [tool] and the test [test problem], the executable is build in the folder [tool]/[test problem]/ and called main.exe. The two object files and potential libraries that are required by the tool (e.g. libadolc.a for ADOL-C) are linked together. The first compilation unit is independent of the selected tool or test: main.cpp initializes the Data Collector for time and memory measurements, it establishes the connection to the SQL database, and calls void run(int n). The definition of this function is linked later and comes from the compilation of run_[tool].cpp. This second compilation unit depends on both, [tool] and [test problem]. Each [tool] has its own run_[tool].cpp where the tool selection takes place via the correct includes (e.g. #include "dco.hpp"; typedef ga1s<double>::type ad_type; ) which is used in the generically written and included run_generic.cpp. This source file is again independent of the [tool] and in turn includes test_problem.hpp which is provided by the test problem and selected by the build system by setting the correct include search paths. It instantiates test_problem with the typedef slipped in from run_[tool].cpp followed by executing the nested run function. This structure makes it very straightforward to add new tests or tools to the suite. In the following we want to show the pattern for writing a test problem and the tool adapter in more detail.

As already mentioned, we combined all interfaces of the different tools in a single adapter. The interface naming was chosen similar to the one of dco/c++. Emphasis is put on one first-order adjoint evaluation only, which is also reflected in the features the interface provide. In particular, all advanced hybridization features are not covered. The generic interface is called ADTypeWrapper and is a templated struct. Each tool has its own specialization thereof. The wrapper includes the following tasks (with corresponding dco/c++ interface):

- initialization, n is the problem size
3.4 AD Test Suite

- ADTypeWrapper<T>::initialize(n)
- \texttt{ga1s< double>::global_tape}\ = \texttt{ga1s< double>::tape_t::create();}

- **mark input variable** \(x\)
- ADTypeWrapper<T>::register_variable(x)
- \texttt{ga1s< double>::global_tape->register_variable(x);}

- **start the recording**
  - ADTypeWrapper<T>::start_recording()
  - empty

- **set the output adjoint to** \(v\)
  - ADTypeWrapper<T>::seed_adjoint(v)
  - derivative(y) = \(v\);

- **run the interpretation**
  - ADTypeWrapper<T>::interpret_adjoint()
  - \texttt{ga1s< double>::global_tape->interpret_adjoint();}

- **get value component of** \(x\)
  - ADTypeWrapper<T>::get_value()
  - \texttt{return value(x);}\

- **get adjoints of** \(i\)-th previously marked input
  - ADTypeWrapper<T>::get_adjoint(i)
  - \texttt{return derivative(x[i]);}

For verification purposes a reference value can be passed to the \texttt{ADTypeWrapper} to check for correct results later. The function for starting the recording is empty for \texttt{dco/c++}. But for writing tests that are compatible also with the other tools, this interface is required. The default call order of those functions in a test problem is shown in Figure 3.10.

In the author’s opinion, the common interface is surprisingly small. This is of course owed to the fact that only one first-order adjoint projection is possible. Nonetheless, the core of all overloading tools is the floating point data type overloading the standardized operators and functions. Therefore the most important part of the interface to the tools is fixed anyway. Investing more effort into a common interface should maybe be considered.

Unfortunately, the source code and documentation of the AD test suite is not yet publicly available. Please contact the author for more information.
Figure 3.10: Default call order of a test problem implementation using the generic interface adapter called ADTypeWrapper. The items describe what the code in between the interface calls needs to do.

```c++
void test_problem<T>::run(size_t n) {
    ADTypeWrapper<T>::initialize(number_of_parameters);
    allocate required memory
    compute values for active inputs
    ADTypeWrapper<T>::register_variable(x[i], v[i]);
    ADTypeWrapper<T>::start_recording();
    compute output y from registered variables x
    ADTypeWrapper<T>::seed_adjoint(y, 1.0);
    ADTypeWrapper<T>::interpret_adjoint();
    retrieve function value from ADTypeWrapper<T>::get_value(y)
    retrieve adjoints from ADTypeWrapper<T>::get_adjoint(i)
    compute reference_value from function value and adjoints
    ADTypeWrapper<T>::set_reference_value(reference_value);
    deallocate memory
    clean up
}
```
4 Hybrid Tangents and Adjoints

This chapter concretizes the statements from the introduction concerning efficiency improvements of an overall AD solution with the help of hybrid tangents and adjoints. Though the main focus is on adjoints, also tangents are considered in most of the cases. We cover derivative projections of $d$-th-order linear and first-order nonlinear solvers in detail. The former derivations are based on [Gil08], where first-order tangents and adjoints of a variety of matrix operations are collected. For nonlinear solvers, [Gil92] and [GW08, Chapter 15: Implicit and Iterative Differentiation] give a broader base, where first-order tangents and adjoints of nonlinear solvers are considered, especially in the context of iterative methods. The latter citation is strongly recommended for the interested reader for further details. From now on the term *symbolic* is used whenever mathematical properties are exploited for computing derivatives of algorithms, as opposed to the term *algorithmic* whenever plain AD is applied.

Symbolic $d$-th-order derivative projections are derived for direct linear solvers in Section 4.1 and symbolic first-order derivative projections for nonlinear solvers in Section 4.2. In the context of the latter, also optimization problems are considered briefly, increasing the differentiation level by one. In addition, formulations for an efficient computation of tangents and adjoints of linear solvers in special contexts (within Newton’s method or in matrix-free solvers) are derived in Section 4.3. In all cases conclusions are drawn about resulting computational complexities. In Section 4.4, a combination and application of the hybridization options is demonstrated in an extensive case study. Here also a link to continuous adjoints of a time-dependent partial differential equation is established.

Though the nonlinear solvers used in real-world codes usually do not compute exact solutions to the nonlinear systems, the implicit functions theorem\(^1\) is applied for the symbolic approaches in the following. Therefore, full convergence of the solution algorithms is assumed for all derivations. As soon as iterative, non-exact methods are taken into account, errors are introduced for the solution and the respective derivatives. An analysis thereof is not discussed more profoundly in this thesis; further details can be found in [GW08, p. 367]. Descriptively, potential discrepancies in the results computed by the algorithmic and the symbolic solvers depend on the given problem as well as on the accuracy of the approximation of the solution. The algorithmic approach, however, has the advantage of computing exact\(^2\) derivatives of the computations that are actually performed by the program. Nonetheless, the symbolic approaches are mostly better in terms of runtime and memory consumption (see complexity results in the following sections), and therefore desirable to be used. Since for real-world applications the computation of hard bounds for the errors introduced by symbolic derivations is difficult, a pragmatic and reasonable approach is therefore to have multiple differentiation techniques available for comparison – the algorithmic *and* the symbolic. On the other hand, using *the* symbolic approach is usually not uniquely defined in a bigger program, since mathematical properties can be exploited on many levels. We therefore refine Figure 1.1 (from the introduction) on symbolic and algorithmic dif-

---

\(^1\)see an interesting historical view on this theorem in [SR02]

\(^2\)at least up to machine accuracy
differentiation paths for a special, but – in the author’s opinion – representative problem. This problem is defined and described mathematically and implementation-wise in the case study later, but already shown here for illustrative reasons. We focus on adjoints in the following. Basically, a parameterized one-dimensional time-dependent nonlinear partial differential equation (PDE) is solved for given initial and boundary values. For timestepping, an implicit scheme is used and Newton’s method is applied to the arising nonlinear system. In each Newton step, a linear system with the Jacobian matrix is to be solved. This structure is shown on the left-hand side of Figure 4.1. The upper left corner stands for the mathematical formulation of the PDE, approaching step-by-step the solution algorithm on the lower left. Differentiation can now be applied on each of those algorithmization-levels to switch to the right-hand side. The right-hand side corresponds to the adjoint formulation. To get an algorithm for the adjoint solution on the lower right, every path can be taken, but of course each yields a different result in general due to discretization and/or convergence errors. The paths are labeled with the corresponding sections where further details can be found; see also description of the figure.

Parts of this chapter have already been published in [Nau+15] and based on that, [SLN15] considered second-order tangent projections for nonlinear solvers.

### 4.1 Direct Linear Solver

In this chapter, the computation of $d$-th-order tangents and adjoints for direct solvers of systems of linear equations

$$A \cdot x = b$$  \hspace{1cm} (4.1)

are considered, where $A = A(z) \in \mathbb{R}^{n \times n}$, $b = b(z) \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, and $z \in \mathbb{R}^q$. The symbolic approach to the computation of first-order derivatives of the solution $x$ with respect to the right-hand side $b$ as well as to the system matrix $A$ has been proposed previously (see for...
4.1 Direct Linear Solver

In this chapter a generic description of the symbolic approach for the differentiation of order $d$ is developed.

The linear solver is assumed to be embedded into an overall structure given as

$$
\begin{pmatrix}
A \\
b
\end{pmatrix} := P(z), \quad x := S(A, b), \quad \text{and} \quad y := p(x)
$$

throughout this chapter. The preprocessor $P(z)$ computes the matrix $A$ and right-hand side $b$ and precedes the solution of the linear system $S(A, b)$. The postprocessor $p(x)$ succeeds $S$ and computes a scalar, overall output $y \in \mathbb{R}^n$, i.e., $P : \mathbb{R}^q \rightarrow \mathbb{R}^{n \times n + n}$ and $p : \mathbb{R}^n \rightarrow \mathbb{R}$. The overall structure perfectly fits into the discussion of hybridization in AD, Section 2.2, and the introduction to the interface for high-level intrinsics in dco/c++, Section 3.3. As shown later, the computation of higher-order tangents and adjoints require a linear system with multiple right-hand sides to be solved. To do so, either the decomposition can be reused if solving with direct methods, or efficient algorithms can be applied to solve iteratively with multiple right-hand sides [SG95]. In the following, we assume an application of a direct method. We follow the notation from Section 2.1 for higher-order models. For a given index set $D$, whenever the index set $K$ occurs, it is meant for all $K \subseteq D$. For example, if $d = 2$, $D = \{1, \ldots, d\} = \{1, 2\}$, and $x \in \mathbb{R}^n$, the variable $x^{(K)}$ stands for the $2^d = 4$ elements $x, x^{(1)}, x^{(2)}$, and $x^{(1,2)}$, each in $\mathbb{R}^n$. In addition, whenever superscripts occur on symbols with $\hat{a}$, the superscripts are meant to slip down and be appended to each element individually. For example if $\hat{A} = (A^{(1)}, A^{(2,3)})$ then $\hat{A}^{(4)} = (A^{(1,4)}, A^{(2,3,4)})$.

Measurements and results are discussed in Section 6.3.

**Tangent Mode** The chain rule applied to Equations (4.2) requires the $d$-th-order tangent version

$$
x^{(K)} := S^{(D)}(A^{(K)}, b^{(K)})
$$

with $D = \{1, \ldots, d\}$ and $K$ as described in the introduction, cf. Definition 5. $d$-th-order tangent versions of preprocessor $P$ and postprocessor $p$ are assumed to be available. For example, they can be obtained through application of dco/c++ in $d$-th-order tangent mode to the given implementations of $P$ and $p$. The focus is on the efficient evaluation of Equation (4.3a).

**First-order** As derived in [DM48] and – besides other matrix derivative results – nicely summarized in [Gil08], the first-order tangent model of the solution of the linear system

$$
A \cdot x = b
$$

is given by the solution of

$$
A \cdot x^{(1)} = b^{(1)} - A^{(1)} \cdot x,
$$

with the input tangents of $A$ and $b$, $A^{(1)}$ and $b^{(1)}$, respectively. Rewriting the two given systems of linear equations into one bigger system yields

$$
\begin{pmatrix}
A & 0 \\
A^{(1)} & A
\end{pmatrix}
\begin{pmatrix}
x \\
x^{(1)}
\end{pmatrix}
=
\begin{pmatrix}
b \\
b^{(1)}
\end{pmatrix}.
$$

This can be solved in a block forward substitution algorithm, yielding the solution of Equation (4.4a) and Equation (4.4b). Since the diagonal elements are both the original matrix $A$, we can decompose $A$ once and use it for solving both rows in the system Equation (4.5).
**4 Hybrid Tangents and Adjoints**

**d-th-order** Based on the extended system of linear equations (4.5), the substitution

\[
\hat{A}_1 = \begin{pmatrix} A & 0 \\ A^{(1)} & A \end{pmatrix}, \quad \hat{x}_1 = \begin{pmatrix} x \\ x^{(1)} \end{pmatrix}, \text{ and } \hat{b}_1 = \begin{pmatrix} b \\ b^{(1)} \end{pmatrix}
\]

(4.6)
yields

\[
\hat{A}_1 \cdot \hat{x}_1 = \hat{b}_1.
\]

(4.7)

We observe, that \( \hat{x}_1 \equiv x^{(K)} \) for \( d = 1 \) and similarly for \( \hat{b}_1 \). With a recursive application of Equation (4.4), this yields the tangents \( \hat{x}_1^{(2)} \) of \( \hat{x}_1 \) (superscript \( (2) \) since we in fact have second-order tangents now) as the solution of

\[
\begin{pmatrix} \hat{A}_1 & 0 \\ A^{(2)} & \hat{A}_1 \end{pmatrix} \begin{pmatrix} \hat{x}_1 \\ \hat{x}_1^{(2)} \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_1^{(2)} \end{pmatrix}.
\]

(4.8)

By resubstituting \( \hat{A}_1, \hat{b}_1 \) and \( \hat{x}_1 \), we get the system for computing second-order tangents by letting the superscript slip down yielding

\[
\begin{pmatrix} A & 0 & 0 & 0 \\ A^{(1)} & A & 0 & 0 \\ A^{(2)} & 0 & A & 0 \\ A^{(1,2)} & A^{(2)} & A^{(1)} & A \end{pmatrix} \begin{pmatrix} x \\ x^{(1)} \\ x^{(2)} \\ x^{(1,2)} \end{pmatrix} = \begin{pmatrix} b \\ b^{(1)} \\ b^{(2)} \\ b^{(1,2)} \end{pmatrix},
\]

(4.9)

where \( \hat{x}_2 \equiv x^{(K)} \) for \( d = 2 \) and similarly for \( \hat{b}_2 \). A \( d \)-times recursive application yields the formula

\[
\hat{A}_d \cdot \hat{x}_d = \hat{b}_d
\]

(4.10)

with

\[
\hat{A}_d = \begin{pmatrix} \hat{A}_{d-1} & 0 \\ A^{(d)} & \hat{A}_{d-1} \end{pmatrix}, \quad \hat{x}_d = \begin{pmatrix} \hat{x}_{d-1} \\ \hat{x}_{d-1}^{(d)} \end{pmatrix}, \text{ and } \hat{b}_d = \begin{pmatrix} \hat{b}_{d-1} \\ \hat{b}_{d-1}^{(d)} \end{pmatrix},
\]

(4.11)

where \( \hat{A}_d \) remains a lower block triangular matrix, with the original system matrix \( A \) as diagonal elements. To compute all required tangents \( \hat{x}_d \equiv x^{(K)} \) from Equation (4.3a) multiple systems with the original matrix \( A \) need to be solved with different right-hand sides. The complexity for computing tangents of \( d \)-th order is therefore given by the decomposition of the system matrix \( A \), the solution of linear systems of dimension \( n \) (i.e. forward and backward substitution), and the computation of the respective right-hand sides. Those in turn consist of a number of matrix-vector and vector-vector operations. The complexities are defined in a recursive way and can be expressed in direct formulas shown in Lemma 13. For comparison, a full algorithmic approach has the complexity

\[
c^d_t \cdot (O(n^3) + O(n^2))
\]

(4.12)

for order \( d \), where \( c_t \) is the constant for applying the tangent mode once, and the two terms in the brackets are the complexities for the decomposition and the solution of the linear system.
Proposition 13. For $d$-th-order tangent modes, the number of matrix-vector multiplications $N_t^M(d)$ and the number of vector-vector operations $N_t^v(d)$ are equal and given by
\[ N_t^v(d) = N_t^M(d) = 3^d - 2^d. \] (4.13)
The number of required linear system solves $N_t^S(d)$ using the decomposition of matrix $A$ is given by
\[ N_t^S(d) = 2^d. \] (4.14)

In the dense case, since the complexity for matrix-vector operations is $O(n^2)$, for vector-vector operations $O(n)$, and for one solution of a linear system $O(n^2)$, the overall complexity for computing $d$-th-order tangents amounts to
\[ 3^d \cdot O(n^2) + (3^d - 2^d) \cdot O(n). \] (4.15)

Proof. The proof is by induction. For 0-th order, no matrix-vector or vector-vector operations are performed. Based on the recursive application of Equation (4.5) we get a recursive formula for $N_t^M(d)$ as follows.
\[ N_t^M(0) = 0 \] (4.16)
\[ N_t^M(d) = 3 \cdot N_t^M(d - 1) + 2^{d-1} \quad \text{for } d > 0. \] (4.17)

We assume Equation (4.13) holds. For $d = 1$ (induction basis) we get same values via Equation (4.17)
\[ N_t^M(1) = 3^1 - 2^1 = 1 \] (4.18)
and via Equation (4.13)
\[ N_t^M(1) = 3 \cdot N_t^M(0) + 2^0 = 1. \] (4.19)
The inductive step is as follows.
\[ N_t^M(d) = 3^d - 2^d \
= 3 \cdot 3^{d-1} - 2 \cdot 2^{d-1} \
= 3 \cdot (3^{d-1} - 2^{d-1}) + 2^{d-1} \
= 3 \cdot N_t^M(d - 1) + 2^{d-1}. \] (4.20)
The direct formula Equation (4.14) immediately follows from the recursion
\[ N_t^v(0) = 1, \quad N_t^v(d) = 2 \cdot N_t^v(d - 1). \] (4.21)

\[ \square \]

Adjoint Mode For the adjoint mode, we are interested in an efficient evaluation of the $d$-th-order adjoints and tangents
\[
\begin{pmatrix}
  x^{(K)}
  \\
  A^{(K)}
  \\
  b^{(K)}
\end{pmatrix}
:=
S^{(D)}(A^{(K)}, b^{(K)}, x^{(K)}). 
\] (4.22)
for $D = \{2, \ldots, d\}$, cf. Definition 6. This function is assumed to be embedded into $d$-th-order adjoints of $P$ and $p$ by using the high-level intrinsic interface of dco/c++, Section 3.3.
4 Hybrid Tangents and Adjoints

First-order From [Gil08] follows

\[ A \cdot x = b \]  
\[ A^T \cdot b^{(1)} = x^{(1)} \]  
\[ A^{(1)} = -b^{(1)} \cdot x^T \]  

which corresponds to the two-step procedure of first solving the extended linear system

\[
\begin{pmatrix}
A & 0 \\
0 & A^T
\end{pmatrix}
\begin{pmatrix}
x \\
b^{(1)}
\end{pmatrix} =
\begin{pmatrix}
b \\
x^{(1)}
\end{pmatrix}
\]

(4.24)

followed by the explicit computation of Equation (4.23c). It should be noted, that the rows in Equation (4.24) are decoupled and can be solved independently. This is very useful since the solution of the first row (i.e. the solution \( x \)) is required during the forward section and the solution of the second row (i.e. the adjoints \( b^{(1)} \)) during the reverse section.

d-th-Order The \( d \)-th-order adjoint is derived by the application of the tangent mode to the adjoint Equation (4.23) \((d - 1)\) times, see Section 2.1. The substitution

\[ \hat{A}_1 = \begin{pmatrix} A & 0 \\ 0 & A^T \end{pmatrix}, \quad \hat{x}_1 = \begin{pmatrix} x \\ b^{(1)} \end{pmatrix}, \quad \text{and} \quad \hat{b}_1 = \begin{pmatrix} b \\ x^{(1)} \end{pmatrix} \]

(4.25)

yields

\[ \hat{A}_1 \cdot \hat{x}_1 = \hat{b}_1, \]

(4.26)

which is a linear system that can be treated the same way as Equation (4.5) to get the tangents

\[
\begin{pmatrix}
\hat{A}_2 & 0 & 0 \\
0 & A^{(2)} & 0 \\
A^{(2)} & 0 & A \\
0 & A^{(2)T} & 0
\end{pmatrix}
\begin{pmatrix}
\hat{x}_2 \\
\hat{x}_1^{(2)} \\
b^{(1)} \\
b^{(2)}
\end{pmatrix} =
\begin{pmatrix}
b \\
x^{(1)} \\
b^{(1)} \\
x^{(2)}
\end{pmatrix}.
\]

(4.27)

Resubstitution yields the second-order adjoint model

\[
\begin{pmatrix}
A & 0 & 0 & 0 \\
0 & A^T & 0 & 0 \\
A^{(2)} & 0 & A & 0 \\
0 & A^{(2)T} & 0 & A^T
\end{pmatrix}
\begin{pmatrix}
x \\
b^{(1)} \\
x^{(2)} \\
b^{(2)}
\end{pmatrix} =
\begin{pmatrix}
b \\
x^{(1)} \\
b^{(2)} \\
x^{(2)}
\end{pmatrix}.
\]

(4.28)

A \((d - 1)\) times recursive application yields the formula

\[ \hat{A}_d \cdot \hat{x}_d = \hat{b}_d \]

(4.29)

with

\[ \hat{A}_d = \begin{pmatrix} \hat{A}_{d-1} & 0 \\ \hat{A}^{(d)}_{d-1} & \hat{A}_{d-1} \end{pmatrix}, \quad \hat{x}_d = \begin{pmatrix} \hat{x}_{d-1} \\ \hat{x}_{d-1}^{(d)} \end{pmatrix}, \quad \text{and} \quad \hat{b}_d = \begin{pmatrix} \hat{b}_{d-1} \\ \hat{b}_{d-1}^{(d)} \end{pmatrix}, \]

(4.30)

where – similar to the tangent case – \( \hat{A}_d \) is a lower block triangular matrix. \( \hat{A}_d \) has now the original system matrix and its transpose, \( A \) and \( A^T \), on all diagonal elements. The solution
for all required tangents and adjoints therefore only require solutions of linear systems with $A$ and $A^T$ with different right-hand sides. The computation of the right-hand side includes matrix-vector and vector-vector operations.

In addition to the formulas for higher-order derivatives of $x$ and $b$, the formula for the higher-order adjoints of $A$ can be derived by a straight-forward differentiation of the explicit formula Equation (4.23c), which yields, e.g., for second-order adjoints

$$A^{(2)}_{(1)} = -b^{(2)}_{(1)} \cdot x^T - b_{(1)} \cdot x^{(2)T} \tag{4.31a}$$

and for third-order adjoints

$$A^{(2,3)}_{(1)} = -b^{(2,3)}_{(1)} \cdot x^T - b^{(2)}_{(1)} \cdot x^{(3)T} - b_{(1)} \cdot x^{(2)T} - b^{(3)}_{(1)} \cdot x^{(2,3)T}. \tag{4.31b}$$

The complexities are defined in a recursive way and can be expressed in direct formulas shown in Lemma 14. For comparison, a full algorithmic approach has the complexity

$$c_a \cdot c_t \cdot c_d^d - c_t \cdot (O(n^3) + O(n^2)). \tag{4.32}$$

for order $d$, where $c_t$ and $c_a$ are the constants for applying the tangent and adjoint mode once, respectively, and the two terms in the brackets are the complexities for the decomposition and the solution of the linear system.

**Proposition 14.** For $d$-th-order adjoint modes, the number of matrix-vector multiplications $N_{M}^a(d)$ and the number of vector-vector operations $N_{v}^a(d)$ are equal and given by

$$N_{v}^a(d) = N_{M}^a(d) = 2 \cdot (3^{d-1} - 2^{d-1}). \tag{4.33}$$

The number of required (transposed) linear system solves $N_{S}^a(d)$ using the decomposition of matrix $A$ is given by

$$N_{S}^a(d) = 2^d. \tag{4.34}$$

The number of outer products like in Equation (4.23c) is

$$N_{p}^a(d) = 2 \cdot 2^{d-1} - 1, \tag{4.35}$$

and the number of matrix-matrix additions is

$$N_{MM}^a = 3^{d-1} - 2^{d-1}. \tag{4.36}$$

For the dense case, since the complexity for matrix-vector operations is $O(n^2)$, for vector-vector operations $O(n)$, for outer products and matrix-matrix additions $O(n^2)$, and for one solution of a linear system $O(n^3)$, the overall complexity for computing $d$-th-order adjoints amounts to

$$(3^d + 2^{d-1} - 1) \cdot O(n^2) + 2 \cdot (3^{d-1} - 2^{d-1}) \cdot O(n). \tag{4.37}$$
4 Hybrid Tangents and Adjoints

Proof. The proof is by induction. From Equation (4.30) follows a recursive formula for \( N_{M}^a(d) \) given as

\[
N_{M}^a(1) = 0 \quad (4.38)
\]

\[
N_{M}^a(d) = 3 \cdot N_{M}^a(d - 1) + 2^{d-1} \quad \text{for} \ d > 1 . \quad (4.39)
\]

We assume Equation (4.33). For \( d = 2 \) (induction basis) we get same values via Equation (4.39)

\[
N_{M}^a(2) = 3 \cdot N_{M}^a(1) + 2^1 = 2 \quad (4.40)
\]

and via Equation (4.33)

\[
N_{M}^a(2) = 2 \cdot (3^1 - 2^1) = 2 . \quad (4.41)
\]

The inductive step follows.

\[
N_{M}^a(d) = 2 \cdot \left(3^{d-1} - 2^{d-1}\right) \quad (4.42)
\]

\[
= 2 \cdot \left(3 \cdot 3^{d-2} - 2 \cdot d^{d-2}\right) \quad (4.43)
\]

\[
= 2 \cdot \left(3 \cdot \left(3^{d-2} - 2^{d-2}\right) + 2^{d-2}\right) \quad (4.44)
\]

\[
= 3 \cdot \left(2 \cdot \left(3^{d-2} - 2^{d-2}\right)\right) + 2^{d-1} \quad (4.45)
\]

\[
= 3 \cdot N_{M}^a(d - 1) + 2^{d-1} . \quad (4.46)
\]

Equation (4.34) follows immediately from the recursion

\[
N_{S}^a(0) = 1 \quad \text{and} \quad N_{S}^a(d) = 2 \cdot N_{S}^a(d - 1) \quad \text{for} \ d > 0 . \quad (4.47)
\]

From Equation (4.31) follows the recursion

\[
N_{p}^a(0) = 0 \quad \text{and} \quad N_{p}^a(d) = N_{p}^a(d - 1) + 2^{d-1} \quad \text{for} \ d > 0 . \quad (4.48)
\]

We assume Equation (4.35). For \( d = 1 \) (induction basis) we get same values via Equation (4.48)

\[
N_{p}^a(1) = 0 + 2^0 = 1 \quad (4.49)
\]

and via Equation (4.35)

\[
N_{p}^a(1) = 2 \cdot 2^0 - 1 = 1 . \quad (4.50)
\]

The inductive step follows.

\[
N_{p}^a(d) = 2 \cdot 2^{d-1} - 1 \quad (4.51)
\]

\[
= 2 \cdot 2^{d-2} - 1 + 2^{d-1} \quad (4.52)
\]

\[
= N_{p}^a(d - 1) + 2^{d-1} . \quad (4.53)
\]

Also from Equation (4.31) follows the recursion

\[
N_{MM}^a(1) = 0 \quad \text{and} \quad N_{MM}^a(d) = N_{MM}^a(d - 1) + 2^{d-1} - 1 \quad \text{for} \ d > 1 . \quad (4.54)
\]
4.2 Nonlinear Solver

We assume Equation (4.36). For \( d = 2 \) (induction basis) we get the same values via Equation (4.54)

\[
N_{MM}^a(2) = 0 + 2^1 - 1 = 1
\]

and via Equation (4.36)

\[
N_{MM}^a(2) = 2 \cdot 2^1 - 3 = 1.
\]

The inductive step follows.

\[
N_{MM}^a(d) = 2 \cdot 2^{d-1} - (d + 1) = 2 \cdot 2^{d-2} + 2^{d-1} - ((d - 1) + 1) - 1 = N_{MM}^a(d-1) + 2^{d-1} - 1.
\]

Proposition 15. The \( d \)-th-order adjoint of \( A \)

\[
\tilde{A} := A^{(2, \ldots, d)}_1 \text{ satisfies } \text{rank}(\tilde{A}) \leq 2^{d-1}.
\]

Proof. Follows immediately from Equation (4.48), since \( \tilde{A} \) is the result from \( 2^{d-1} \) outer products each of rank 1.

4.2 Nonlinear Solver

In this chapter, the computation of first-order tangents and adjoints for solvers of systems of parameterized nonlinear equations

\[
F(x, \lambda) \overset{1}{=} 0
\]

is considered, where \( F : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^n \) continuously differentiable and \( \lambda \in \mathbb{R}^m \). With parameters \( \lambda \), the exact solution \( x = x^* \) is sought such that Equation (4.61) is fulfilled.

The mathematical derivations shown in this section are partially already published in [Nau+15]. Similar to the previous section, the implementation \( S \) of the nonlinear solver is embedded into a preprocessor \( P \) and a postprocessor \( p \) as follows

\[
y = p(S(x_0, \lambda)) = p(S(x_0, P(z)))
\]

where \( z \in \mathbb{R}^q \) and \( P : \mathbb{R}^q \rightarrow \mathbb{R}^m \) denotes the part of the computation that precedes the nonlinear solver \( S : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n \) and the postprocessor \( p : \mathbb{R}^n \rightarrow \mathbb{R} \) maps the approximate result

\[
\tilde{x} = S(x_0, \lambda) \iff F(\tilde{x}, \lambda) \approx 0
\]

onto the scalar objective \( y \). \( x_0 \in \mathbb{R}^n \) is the starting value for \( x \) used by the nonlinear solver \( S \). The discussion in this chapter is based on the following algorithmic description of Equation (4.62):

\[
\lambda := P(z) \text{, } \tilde{x} := S(x_0, \lambda) \text{, and } y := p(\tilde{x})
\]

We assume tangent and adjoint projections of \( p \) and \( P \) to be available, e.g. by application of dco/c++. Complexities for algorithmic and symbolic approaches are shown in Table 4.1. Measurements and results are discussed in Section 6.3.
4 Hybrid Tangents and Adjoints

<table>
<thead>
<tr>
<th></th>
<th>algorithmic NLS</th>
<th>symbolic NLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tangent mode</td>
<td>$\nu \cdot O(n^3)$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>adjoint mode</td>
<td>$\nu \cdot O(n^3)$</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

Table 4.1: Computational complexities of algorithmic and symbolic tangent and adjoint modes of differentiation for $\nu$ nonlinear iterations applied to systems of $n$ nonlinear equations (NLS).

...in an Optimization Context In addition, first-order tangents and adjoints are considered for $S$ solving a well-posed parameterized unconstrained minimization problem

$$\min_{x \in \mathbb{R}^n} \varphi(x, \lambda), \quad \varphi : \mathbb{R}^{n+m} \to \mathbb{R}, \quad x \in \mathbb{R}^n \quad (4.65)$$

with parameters $\lambda \in \mathbb{R}^m$ and $\varphi$ at least two times continuously differentiable near all points of interest. In this case, the above introduced nonlinear system is given as

$$F(x, \lambda) = \left( \frac{\partial \varphi(x, \lambda)}{\partial x} \right)^T = 0 \quad (4.66)$$

and originates from the first-order optimality condition of Equation (4.65).

**Tangent Mode** The chain rule applied to Equations (4.64) requires the tangent version

$$\begin{pmatrix} \bar{x} \\ \bar{x}^{(1)} \end{pmatrix} := S^{(1)}(x^0, \lambda, \lambda^{(1)}) = \left( \frac{\partial S(x^0, \lambda)}{\partial \lambda} \right) . \quad (4.67a)$$

This function is assumed to be embedded into tangent versions of $P$ and $p$, $P^{(1)}$ and $p^{(1)}$, respectively. Tangents of $x^0$ are not considered since the starting value is assumed to be a constant. In the following the efficient computation of $S^{(1)}$ is discussed.

Directional derivatives of the solution are computed by a tangent version of the solver under the assumption that the exact primal solution $x^*$ has been reached, i.e. $x = x^*$. $F(x, \lambda) = 0$ can be differentiated symbolically in this case. Consequently, the computation of the directional derivative amounts to the solution of a linear system based on the Jacobian of $F$ with respect to $x$. Differentiation of $F = F(x, \lambda) = 0$ at the solution $x = x^*$ with respect to $\lambda$ yields

$$\frac{dF}{d\lambda} = \frac{\partial F}{\partial \lambda} + \frac{\partial F}{\partial x} \cdot \frac{\partial x}{\partial \lambda} = 0 \quad (4.68)$$

and hence

$$\frac{\partial x}{\partial \lambda} = -\frac{\partial F^{-1}}{\partial x} \cdot \frac{\partial F}{\partial \lambda} . \quad (4.69)$$
The computation of the directional derivative
\[ \tilde{x}^{(1)} = \frac{\partial \tilde{x}}{\partial \lambda} \cdot \lambda^{(1)} = -\frac{\partial F^{-1}}{\partial \tilde{x}} \cdot \frac{\partial F}{\partial \lambda} \cdot \lambda^{(1)} \] (4.70)
amounts to the solution of the linear system
\[ \frac{\partial F}{\partial \tilde{x}} \cdot \tilde{x}^{(1)} = -\frac{\partial F}{\partial \lambda} \cdot \lambda^{(1)} \] (4.71)
the right-hand side of which can be obtained by a single evaluation of the tangent routine of \( F \) for given \( \tilde{x}, \lambda, \) and \( \lambda^{(1)} \). The direct solution of Equation (4.71) requires the \( n \times n \) Jacobian \( \frac{\partial F}{\partial \tilde{x}} \), which is preferably accumulated using tangent mode AD while exploiting potential sparsity. Refer to [GF02] for an analysis of the consistency with the exact tangent projection \( x^{*(1)} \) of the exact solution \( x^* \).

**...in an Optimization Context** Differentiation of \( \frac{\partial \varphi}{\partial \tilde{x}} = 0 \) for \( \varphi = \varphi(x, \lambda) \) at the solution \( x = x^* \) with analogous derivation, yields a symbolic tangent solver which needs to solve the linear system
\[ \frac{\partial^2 \varphi}{\partial \tilde{x}^2} \cdot \tilde{x}^{(1)} = -\left( \frac{\partial^2 \varphi}{\partial \tilde{x} \partial \lambda} \right)_i \cdot \left[ \lambda^{(1)} \right]_i \] (4.72)
the right-hand side of which can be obtained by a single evaluation of the second-order adjoint routine of \( \varphi \) for given \( x, \lambda, \) and \( \lambda^{(1)} \). The direct solution of Equation (4.72) requires the \( n \times n \) Hessian \( \frac{\partial^2 \varphi}{\partial \tilde{x}^2} \).

**Adjoint Mode** The chain rule applied to Equations (4.64) requires the adjoint version
\[ \left( \begin{array}{c} \tilde{x} \\ \lambda^{(1)} \end{array} \right) := S_{(1)}(x^0, \lambda, \tilde{x}^{(1)}) = \left( \begin{array}{c} S(x^0, \lambda) \\ (\frac{\partial S}{\partial x})^T \cdot \tilde{x}^{(1)} \end{array} \right). \] (4.73a)
This function is assumed to be embedded into adjoint versions of \( P \) and \( p \) by using the high-level intrinsic interface of \texttt{dco/c++}, Section 3.3. No adjoints of \( x^0 \) are considered since the solution is independent of the starting value when fully converged. In the following the efficient computation of \( S_{(1)} \) is discussed.

Adjoint of the solution are computed by an adjoint version of the solver also under the assumption that the exact primal solution \( x^* \) has been reached. The transpose of Equation (4.69) results in
\[ \left( \begin{array}{c} \partial x \\ \partial \lambda \end{array} \right)^T = -\left( \frac{\partial F}{\partial \tilde{x}} \right)^T \cdot \left( \frac{\partial F}{\partial \lambda} \right)^{-T} \] (4.74)
and hence
\[ \lambda^{(1)} = \left( \begin{array}{c} \partial x \\ \partial \lambda \end{array} \right)^T \cdot \tilde{x}^{(1)} = -\left( \frac{\partial F}{\partial \tilde{x}} \right)^T \cdot \left( \frac{\partial F}{\partial \lambda} \right)^{-T} \cdot \tilde{x}^{(1)}. \] (4.75)
Consequently, the symbolic adjoint solver needs to solve the linear system
\[ \left( \begin{array}{c} \frac{\partial F}{\partial \tilde{x}} \end{array} \right)^T \cdot z = -\tilde{x}^{(1)} \] (4.76)
followed by a single call of the adjoint model of $F$ to obtain

$$\lambda_{(1)} = \left(\frac{\partial F}{\partial \lambda}\right)^T \cdot z$$

(4.77)

for given $x = x^*, \lambda$. The direct solution of Equation (4.76) requires the transpose of the $n \times n$ Jacobian $\frac{\partial F}{\partial x}$ which is preferably accumulated using tangent mode AD while exploiting sparsity. Consistency with the exact adjoint projection is also shown in [GF02] similar to the tangent case in the previous section.

...in an Optimization Context Differentiation of $\frac{\partial \varphi}{\partial x} = 0$ for $\varphi = \varphi(x, \lambda)$ at the solution $x = x^*$ with analogous derivation, yields a symbolic adjoint solver which needs to solve the symmetric linear system

$$\frac{\partial^2 \varphi}{\partial x^2} \cdot z = -\tilde{x}_{(1)}$$

(4.78)

followed by a single call of the second-order adjoint model of $\varphi$ to obtain

$$\lambda_{(1)} = \left(\frac{\partial^2 \varphi}{\partial [x] \partial \lambda}\right)^T \cdot [z]_i$$

(4.79)

for given $x = x^*$ and $\lambda$. The solution of Equation (4.78) requires the $n \times n$ Hessian $\frac{\partial^2 \varphi}{\partial x^2}$.

4.3 Linear Solvers in Special Contexts

Depending on the context a linear solver gets called in, further mathematical properties can be exploited for computing derivative projections even cheaper. In this section, two possible contexts are shown. This section is surely not meant as an exhaustive discussion on that but rather as demonstrator how the context can and should be exploited.

In Section 4.3.1, the linear solver is embedded into a Newton-type nonlinear solver. The consideration of the context when using the symbolic approach for the linear solver reduces the number of required tangent and adjoint model evaluations from $n$ to one for computing tangents and adjoints of the Newton step with respect to the parameters, respectively. Matrix-free linear solvers are the subject in Section 4.3.2. The main focus there is not on efficiency but on feasibility. Since the system matrix is not explicitly in memory, the goal is to treat the linear solver symbolically with the constraint of having only a matrix-vector product implementation available. Nonetheless, a formulation is derived, where tangent and adjoint projections of the implicit matrix-vector product are sufficient to compute overall tangents and adjoints.

Both special contexts appear in the case study Section 4.4.

4.3.1 ... in a Nonlinear Solver Context

This section is closely related to Section 4.2. For Newton-type solvers for the nonlinear system

$$F(x, \lambda) = 0$$

(4.80)

already described in detail in Section 4.2, a combination of the algorithmic and symbolic modes can be used, where a symbolic approach is taken for the differentiation of the linear Newton
system as part of an otherwise algorithmic approach to the differentiation of the nonlinear solver. Use of a direct linear solver makes this approach mathematically equivalent to the fully algorithmic method as both the Newton system and its derivative projections are solved with machine accuracy. In accordance with Section 4.1, the computational complexity of the evaluation of tangents and adjoints of the Newton step with respect to the parameters can be reduced from cubic to quadratic through the reuse of the factorization of the system matrix. In addition, exploitation of the context reduces the number of required tangent and adjoint projections of $F$ from $n$ to one.

The quality of the result of a symbolic nonlinear solver as described in Section 4.2 depends on the accuracy of the primal solution. Moreover, in the Newton case, the computational effort is dominated by the solution of the linear system in each iteration. Symbolic differentiation of the linear solver as part of an algorithmic differentiation approach to the enclosing nonlinear solver aims for a reduction of the computational cost of the adjoint while preserving the accuracy of the algorithmic tangent or adjoint solver if an accurate solution of the linear Newton system is available. Exact sensitivities of what is computed (that is, an approximation of the solution of the primal nonlinear system) are obtained if the primal Newton steps are computed exactly.

In Newton’s method, a linear system with the Jacobian matrix of $F$ needs to be solved in each iteration. The right-hand side is given as the negative residual. The solution $s \in \mathbb{R}^n$ of the Newton system

$$A := \frac{\partial F}{\partial x}(x, \lambda), \quad b := -F(x, \lambda), \quad \text{and} \quad A \cdot s = b \quad (\Rightarrow s),$$

is the search direction (Newton step).

**Tangent Mode** Plain application of tangent mode to the computation of $A$ and $b$ in combination with the symbolic approach to the linear solver (see Section 4.1) yields for one Newton system solve

$$A := \frac{\partial F}{\partial x},$$

$$A^{(1)} := \frac{\partial}{\partial (x, \lambda)} \left( \frac{\partial F}{\partial x} \right), \quad \left( \frac{x^{(1)}}{\lambda^{(1)}} \right) \quad (4.82b)$$

$$b := -F,$$

$$b^{(1)} := -\frac{\partial F}{\partial (x, \lambda)} \cdot \left( \frac{x^{(1)}}{\lambda^{(1)}} \right),$$

$$A \cdot s = b \quad (\Rightarrow s) \quad (4.82e)$$

$$A \cdot s^{(1)} = b^{(1)} - A^{(1)} \cdot s \quad (\Rightarrow s^{(1)}).$$

Please note, that Equation (4.82b) needs to perform $n$ second-order tangent projections implicitly to compute $A^{(1)}$. Since the dependence of $A$ and $b$ from the parameters $\lambda$ and $x$ is explicitly known in the context of a Newton system, we can omit Equations (4.82b) and (4.82d)
by inserting them directly into Equation (4.82f) yielding the following equation for \( s^{(1)} \)

\[
A \cdot s^{(1)} = - \frac{\partial F}{\partial (x, \lambda)} \cdot \left( x^{(1)} \lambda^{(1)} \right) - \frac{\partial}{\partial \left( \frac{\partial F}{\partial x} \right)} \cdot \left[ \frac{\partial F}{\partial x} \lambda^{(1)} \right]_i \cdot s
\]

(4.83a)

\[
= - \frac{\partial F}{\partial (x, \lambda)} \cdot \left( x^{(1)} \lambda^{(1)} \right) - \frac{\partial^2 F}{\partial x \partial \lambda} \cdot \left[ s \right]_j \cdot \left( x^{(1)} \lambda^{(1)} \right)_i .
\]

(4.83b)

For the computation of the right-hand side, this formulation only requires one evaluation of the second-order tangent model instead of \( n \) in Equation (4.82b).

Consistency with the algorithmic approach is given naturally, as we compute the algorithmic tangent projection with a direct linear solver.

**Adjoint Mode** The algorithmic adjoint of the enclosing nonlinear solver consists of a forward and a reverse section. In the following, we cut out the forward and the reverse section of the linear solver for one Newton step.

Plain application of adjoint mode to the computation of \( A \) and \( b \) in combination with the symbolic approach to the linear solver (see Section 4.1) yields for one Newton system solve in the forward section

\[
A := \frac{\partial F}{\partial x} (x, \lambda) , \quad b := -F(x, \lambda) , \quad A \cdot s = b \quad (\Rightarrow s) ,
\]

(4.84a)

and in the reverse section

\[
A^T \cdot b^{(1)} = s^{(1)} \quad (\Rightarrow b^{(1)})
\]

(4.84b)

\[
A^{(1)} := -b^{(1)} \cdot s^T
\]

(4.84c)

\[
\left( x^{(1)} \lambda^{(1)} \right) := - \left( \frac{\partial F}{\partial (x, \lambda)} \right)^T \cdot b^{(1)} + \left( \frac{\partial}{\partial \left( \frac{\partial F}{\partial x} \right)} \right)^T \cdot [A^{(1)}]_{ij} .
\]

(4.84d)

Please note, that Equation (4.84d) performs an adjoint projection of the Jacobian in the second term. This corresponds to \( n \) evaluations of the second-order adjoint model and should be avoided. To achieve that, since the dependence of \( A \) and \( b \) from the parameters \( \lambda \) and \( x \) is explicitly known in the context of a Newton system, we can omit Equation (4.84c) by inserting it directly into Equation (4.84d) yielding the following equation for the second term in Equation (4.84d)

\[
\left( \frac{\partial}{\partial (x, \lambda)} \right)^T \cdot [A^{(1)}]_{ij} = \left( \frac{\partial^2 F}{\partial x \partial \lambda} \right)^T \cdot [b^{(1)}]_i \cdot [s]_j
\]

(4.85a)

\[
= - \left( \frac{\partial^2 F}{\partial x \partial \lambda} \right)^T \cdot [s]_j
\]

(4.85b)

\[
= - [b^{(1)}]_i \cdot \left( \frac{\partial^2 F}{\partial x \partial \lambda} \right)^T \cdot [s]_j .
\]

(4.85c)

The formulation Equation (4.85c) only requires one evaluation of the second-order adjoint model of \( F \) instead of an adjoint model of \( \frac{\partial F}{\partial x} \) in the original formulation in Equation (4.84d).
4.3 Linear Solvers in Special Contexts

4.3.2 Iterative Matrix-Free Methods

An iterative, matrix-free method is a method for solving linear systems without explicitly holding the coefficients of the matrix in memory. The matrix only exists implicitly as an implementation of a matrix-vector product \( A \cdot v \) with \( A \in \mathbb{R}^{n \times n} \), \( v \in \mathbb{R}^n \). In the following we assume \( A = A(\lambda) \) with parameters \( \lambda \in \mathbb{R}^m \). Matrix-free methods are often used, when the matrix is either too big to be saved in memory, or if only an implicit implementation for the matrix-vector product exists. The latter is the case if using for example a tangent model of \( F(x) \) to solve the system with \( A = \frac{\partial F}{\partial x} \). Wide-spread iterative matrix-free methods for solving a well-posed linear system

\[
A \cdot x = b
\]

are either the conjugate gradient (CG) method [HS52] if symmetric, or the generalized minimal residual (gmres) method [SS86] if non-symmetric. Both methods only require products of the matrix \( A \) and an arbitrary vector \( v \). The implementation of the solution algorithm itself usually only has access to an interface for the matrix-vector product, let’s assume similar to

```cpp
template <typename T, typename MATRIX>
void mult(const std::vector<T> &v,
          const std::vector<T> &lambda,
          std::vector<T> &y);
```

for computing \( y = A(\lambda) \cdot v \). In our example, the matrix \( A \) is passed as the template parameter MATRIX.

We can now apply the results from Section 4.1 to compute tangents and adjoints symbolically. Though the computational complexity is not necessarily improved by a symbolic treatment in the context of iterative methods, as we will see, the memory consumption in the adjoint mode can be substantially reduced.

While the right-hand side \( b \) is an explicit input, the matrix \( A \) is only implicitly entering the solver. Nonetheless, symbolic tangent and adjoint projections of the linear solver are possible without changing the original interface of \( \text{mult} \), i.e., a formulation is derived, where only tangent and adjoint projections of the implicit matrix-vector product are required.

**Tangent mode** From

\[
A \cdot x^{(1)} = b^{(1)} - A^{(1)} \cdot x
\]

follows

\[
A \cdot x^{(1)} = b^{(1)} - \frac{\partial (A \cdot x)}{\partial(x, \lambda)} \cdot \begin{pmatrix} 0 \\ \lambda^{(1)} \end{pmatrix},
\]

where the projection on the right-hand side requires the tangent model of \( \text{mult} \). This can be generated, e.g., by instantiating above given function \( \text{mult} \) with the tangent data type of dco/c++. With the new right-hand side, the original matrix-vector product can be used since the system matrix has not changed. Under the assumption that the original system Equation (4.86) and the tangent system Equation (4.88) converge similarly, the symbolic approach has no advantage over the algorithmic approach.
4 Hybrid Tangents and Adjoints

**Adjoint mode** For computing adjoints of the parameters $\lambda$, we need to solve

$$A_{(1)} = -b_{(1)} \cdot x^T \quad \text{with} \quad A^T \cdot b_{(1)} = x_{(1)} \quad (4.89)$$

followed by the projection

$$\lambda_{(1)} = \left( \frac{\partial [A(\lambda)]_{ij}}{\partial \lambda} \right)^T \cdot [A_{(1)}]_{ij} \cdot x_{(1)} \quad (4.90)$$

For solving the transposed system Equation (4.89) matrix-free, products $A^T \cdot v$ are required and can be computed by the adjoint model of `mult` given as

$$\left( x_{(1)} \right) = \left( \frac{\partial (A \cdot x)}{\partial \lambda} \right)^T \cdot y_{(1)} \quad (4.91)$$

Consequently, for $y_{(1)} = v$ we get $x_{(1)} = A^T \cdot v$. From the projection Equation (4.90) follows

$$\lambda_{(1)} = - \left( \frac{\partial (A \cdot x)}{\partial \lambda} \right)^T \cdot b_{(1)} \quad (4.92)$$

as shown in Lemma 16. Hence, for Equation (4.92) also an adjoint model of `mult` is required. This can be generated, e.g., by instantiating above given function `mult` with the adjoint data type of dco/c++. The symbolic approach has the huge advantage of not taping the iteration itself by making use of the high-level intrinsic interface of dco/c++, Section 3.3.

**Proposition 16.** For $A = A(\lambda) \in \mathbb{R}^{n \times n}$ and a matrix-vector product $y = A(\lambda) \cdot x$, where $\lambda \in \mathbb{R}^n$ and $x \in \mathbb{R}^n$ independent inputs, follows from Equation (4.89)

$$\left( \frac{\partial [A(\lambda)]_{ij}}{\partial \lambda} \right)^T \cdot [A_{(1)}]_{ij} = - \left( \frac{\partial (A \cdot x)}{\partial \lambda} \right)^T \cdot b_{(1)} \cdot x^T \
\quad (4.93)$$

Proof.

$$\left( \frac{\partial [A(\lambda)]_{ij}}{\partial \lambda} \right)^T \cdot [A_{(1)}]_{ij} = - \left( \frac{\partial [A(\lambda)]_{ij}}{\partial \lambda} \right)^T \cdot [b_{(1)} \cdot x^T]_{ij} \quad (4.94)$$

$$= - \left( \frac{\partial [A(\lambda)]_{ij}}{\partial \lambda} \right)^T \cdot [b_{(1)}]_i \cdot [x]_j \quad (4.95)$$

$$= - \left( \frac{\partial (A \cdot x)}{\partial \lambda} \right)^T \cdot b_{(1)} \cdot x^T \quad (4.96)$$

\[\square\]

### 4.4 An Extensive Case Study

Based on the hybridization on linearized DAGs in Section 2.2 and its implementation in dco/c++ as high-level intrinsics, see Section 3.3, this section materializes the different hybridization approaches introduced earlier in this chapter in form of a case study. The main point in this section is to demonstrate the interplay between mathematics and implementation for computing hybrid
4.4 An Extensive Case Study

adjoints with the help of \texttt{dco/c++}. The corresponding complete source code is attached to this thesis and variable/function names therein are as close as possible to their mathematical pendant. Apart from being involved in second-order projections, no tangent modes are discussed in the following.

After stating the one-dimensional time-dependent nonlinear partial differential equation (PDE), its starting and boundary conditions, and a corresponding data fitting problem (Section 4.4.1), the used discretization schemes and numerical methods are shown (Section 4.4.2). This also includes the description of the various derivatives required by the methods, e.g., the data fitting makes use of gradient and Hessian of a cost functional. Afterwards, the continuous adjoint formulation for the PDE is derived in Section 4.4.3. The implementation thereof is used later to demonstrate that even continuous adjoints can be coupled with \texttt{dco/c++}. After showing the implementation of the case study with fully algorithmic derivatives using \texttt{dco/c++} (Section 4.4.4), the various hybridization levels are presented in more detail in Section 4.4.5.

To not blur the main point of this section, the mathematical equations are stated in a way that the notion is clear, though not mathematically waterproof. All stated problems are assumed to behave nicely in a numerical sense and satisfy the conditions on continuity and differentiability that are required. Conceptual figures of C++ implementations are shown inlined to clarify the derivative information required by stating the respective \texttt{dco/c++} data types that functions get instantiated with. The source code and documentation of the case study can be found in attached\_code/case\_study.

### 4.4.1 One-Dimensional Time-Dependent Convection-Diffusion Equation

For parameter and state
\[ \lambda = \lambda(x) \in \mathbb{R}^+ \quad \text{and} \quad u = u(x, \lambda, t) \in \mathbb{R}^+, \] (4.97a)
respectively, we define the residual right-hand side function
\[ \frac{\partial u}{\partial t} = R(\lambda, u, t) = \lambda \frac{\partial^2 u}{\partial x^2} - u \frac{\partial u}{\partial x} \quad \text{on} \quad \Omega_x = [0, 1] \quad \text{and} \quad \Omega_t = [0, \text{te}]. \] (4.97b)
with time \( t \). \( \lambda(x) \) is the diffusion coefficient. Apart from the neglected term \( \frac{\partial \lambda}{\partial x} \frac{\partial u}{\partial x} \), this is the standard convection-diffusion equation. Boundary conditions are chosen to be
\[ u(0, \lambda, t) = 1 \quad \text{and} \quad u(1, \lambda, t) = 0 \quad \forall \ t \in \Omega_t. \] (4.98)

With \( u^0 = u^0(x) \) the initial value for \( u \) at \( t = 0 \), \( u^m = u^m(x) \) measurements of \( u \) at \( t = \text{te} \), and \( \lambda_g = \lambda_g(x) \) a guess for the diffusion coefficient, the solution \( u \) of Equation (4.97) enters the cost functional
\[ \varphi(\lambda, \lambda_g, u^0, u^m) = \int_{\Omega_x} \left( (u|_{t=\text{te}} - u^m)^2 + (\lambda - \lambda_g)^2 \right), \quad \varphi : \mathbb{R}^4 \to \mathbb{R}, \] (4.99a)
which corresponds to a usual least-squares formulation with Tikhonov regularization term [Tik43]. The minimizer \( \lambda^* \) is defined as
\[ \lambda^* = \lambda^*(\lambda_g, u^0, u^m) = \arg \min_{\lambda} \varphi(\lambda, \lambda_g, u^0, u^m). \] (4.99b)
4 Hybrid Tangents and Adjoint

In addition, the sensitivities of the minimizer with respect to measured values

$$\frac{\partial \lambda^*}{\partial (\lambda_g, u^0, u^m)}$$ (4.100)

are assumed to be of interest, e.g. for uncertainty quantification [Maı10].

4.4.2 Discretization and Numerical Methods

In space and time, all variables and functions are discretized on an equidistant two-dimensional mesh with

$$x_i = i \cdot \Delta x \quad \text{and} \quad t^j = j \cdot \Delta t,$$ (4.101)

for $$i = 1, \ldots, n$$, $$j = 1, \ldots, M$$, $$\Delta x = \frac{1}{n-1}$$ and $$\Delta t = \frac{t_e}{M-1}$$. The discretized variables and functions described by one value at each grid point are written in bold, i.e.

$$\lambda, \lambda_g, \lambda^*, u_j^i \in \mathbb{R}^n, \quad \text{and} \quad u^0, u^m \in \mathbb{R}^n,$$ (4.102)

where $$u^i_j$$ denotes the discretized state at timestep $$j$$. For Equation (4.97) the diffusion (second-order) term is discretized with central finite differences, whereas for the convection (first-order) term an upwind scheme is used. The discretized residual for time step $$j$$ is denoted by $$r_j^i \in \mathbb{R}^n$$ for $$j = 1, \ldots, M$$ and given as

$$[r_j^i]_i = [\lambda]_i \cdot \frac{[u^i_j]_{i-1} - 2 \cdot [u^i_j]_i + [u^i_j]_{i+1}}{\Delta x^2} - [u^i_j]_i \cdot \frac{[u^i_j]_i - [u^i_j]_{i-1}}{\Delta x},$$ (4.103)

where the upwind scheme has a fixed sign because $$u^i_j$$ is positive for all $$j$$ as defined in Equation (4.97a).

For temporal discretization an implicit (backward) Euler scheme is used yielding

$$\frac{u^j_{i+1} - u^j_i}{\Delta t} = r^j_i,$$ (4.104)

which requires a nonlinear system solve in each time iteration. This nonlinear system is solved with Newton’s method and yields the iteration

$$u^j_{\text{new}} = u^j_{\text{old}} + \Delta u \quad \text{with} \quad \left( \frac{\partial r^j_{i+1}}{\partial u^j_{i+1}} - I_n \right) \Delta u = -u^j_{\text{old}} + \Delta t \cdot r^j_{i+1}$$ (4.105)

for getting $$u^j_{i+1} = u^j_{\text{new}}$$ when converged; $$I_n$$ is the identity matrix in $$\mathbb{R}^{n \times n}$$. The required Jacobian matrix in Equation (4.105) is very sparse due to the compact discretization stencil of the residual from Equation (4.103). Sparsity should be exploited during the computation of the Jacobian and during the solution of the linear systems.

The minimization of Equation (4.99) is carried out by a matrix-free inexact Newton/CG algorithm [DES82] with a simple line search using bisection. With the line search parameter $$\alpha \in \mathbb{R}$$ and the discretized cost functional

$$\Phi = \Phi(\lambda, \lambda_g, u^0, u^m), \quad \Phi : \mathbb{R}^{4 \times n} \rightarrow \mathbb{R}$$ (4.106)
4.4 An Extensive Case Study

Figure 4.2: Derivative tensors required by the numerical methods from Section 4.4.2. \( \partial \) and \( \partial^2 \) denote first- and second-order derivatives, respectively, with respect to corresponding input variables.

we get the iteration

\[
\lambda^{k+1} = \lambda^k + \alpha \Delta \lambda \quad \text{with} \quad \frac{\partial^2 \Phi}{\partial \lambda^2} (\lambda^k) \cdot \Delta \lambda = - \left( \frac{\partial \Phi}{\partial \lambda} (\lambda^k) \right)^T ,
\]

(4.107)

and an approximation for the optimizer

\[
\tilde{\lambda} = \lambda^K \quad \text{with} \ K \ \text{the index of the last Newton iteration}. \quad (4.108)
\]

The Hessian in Equation (4.107) is potentially dense. The system is therefore solved with a matrix-free iterative solver. Equation (4.100) corresponds to the discrete pendant

\[
\frac{\partial \tilde{\lambda}}{\partial (\lambda_g, u^0, u^m)} \in \mathbb{R}^{n \times 3n} . \quad (4.109)
\]

The computation of this derivative tensor with above described numerical methods require a variety of derivatives. The nested structure of the problem leads to implicit embedding of derivatives into derivatives of higher levels as shown in Figure 4.2 and described in the following sections.

4.4.3 Continuous Adjoint

In addition to the symbolic nonlinear and linear solvers introduced in Sections 4.1 and 4.2, we derive a continuous adjoint, i.e. symbolically on the PDE formulation. The solution algorithm for this is also coupled with dco/c++ later. General remarks concerning continuous and discrete adjoint formulations can be found in the introduction Section 1.

With the adjoint state variable

\[
\bar{u} = \bar{u}(x, \lambda, t) \in \mathbb{R} \quad (4.110)
\]

the continuous adjoint formulation for the PDE (4.97) is given as

\[
- \frac{\partial \bar{u}}{\partial t} = \bar{R}(\lambda, u, \bar{u}, t) = \frac{\partial^2 \left( \lambda \bar{u} \right)}{\partial x^2} + u \frac{\partial \bar{u}}{\partial x} \quad \text{on} \quad \Omega_x = [0, 1] \ \text{and} \ \Omega_t = [0, t^e], \quad (4.111)
\]
with given parameters \( \lambda \) and solution \( u \). The initial conditions of the adjoint variable at \( t = t^e \) are
\[
\bar{u}^e = \bar{u}^e(x), \tag{4.112}
\]
and the boundary conditions
\[
\bar{u}(0, \lambda, t) = \bar{u}(1, \lambda, t) = 0 \quad \forall \ t \in \Omega_t. \tag{4.113}
\]
In addition, the projection of the adjoint state \( \bar{u} \) onto the adjoint of the parameters is given as
\[
\bar{\lambda} = \int_{\Omega_t} \int_{\Omega_x} \bar{u} \cdot \frac{\partial^2 u}{\partial x^2}. \tag{4.114}
\]
The derivation is done via partial integration of the Lagrangian and not shown in detail here – see, e.g., [Oth08] for a clear and well-structured presentation of a continuous adjoint derivation.

Discretization of the adjoint equation is chosen to be similar to the corresponding discrete adjoint stencil, i.e., applying the rules for generating adjoint code by AD on the primal discretization followed by an index shift. This is done to get equal results for both approaches, the discrete and continuous adjoint. We use the primal mesh from Equation (4.101) with the discretized adjoint residual \( \bar{r}^j = \bar{r}^j(\lambda, u, \bar{u}) \in \mathbb{R}^n \) at time step \( j \)
\[
[\bar{r}^j]_i = \frac{[\lambda]_{i-1} \cdot [\bar{u}^j]_{i-1} - 2 \cdot [\lambda]_i \cdot [\bar{u}^j]_i + [\lambda]_{i+1} \cdot [\bar{u}^j]_{i+1}}{\Delta x^2} + \frac{[u^{j+1}]_{i+1} \cdot [\bar{u}^j]_{i+1} - ([u^{j+1}]_{i-1} - 2 \cdot [u^{j+1}]_i) \cdot [\bar{u}^j]_i}{\Delta x}. \tag{4.115}
\]
Since the diffusion term is a standard central finite difference scheme, the convection term follows no standard discretization rule.

As in the primal, an implicit Euler method is used yielding
\[
\frac{\bar{u}^{j-1} - \bar{u}^j}{\Delta t} = \bar{r}^{j-1} = A \cdot \bar{u}^{j-1}, \tag{4.116}
\]
where the matrix \( A \in \mathbb{R}^{n \times n} \) is the Jacobian matrix of the linear (in \( \bar{u}^{j-1} \)) residual \( \bar{r}^{j-1} \). In each time iteration, we have to solve the linear equation
\[
(\Delta t A - I_n) \cdot \bar{u}^{j-1} = -\bar{u}^j, \tag{4.117}
\]
where \( I_n \) is again the identity matrix in \( \mathbb{R}^{n \times n} \).

4.4.4 Implementation using dco/c++

This section describes how the derivatives required by the algorithms from Section 4.4.2 and visualized in Figure 4.2 are computed by making use of plain dco/c++, i.e., no hybridization included. Based on that, the next section demonstrates the application of hybridization on different levels. There, the affinity to the code nicely gives a feeling for the actual costs of evaluating adjoints by showing the required function instantiations.
4.4 An Extensive Case Study

Figure 4.3: The overall structure of the case study. $U$ and $T$ are floating point template types – either double or a dco/c++ data type. On the left-hand side is the structure of the implementation of the fitting problem, which requires first- and second-order derivative models of the cost functional. Those models are generated by instantiation with first- and second-order dco/c++ data types, respectively. On the right-hand side is the structure of the implementation of the cost functional, which requires the primal residual and its Jacobian. The Jacobian is computed using the first-order tangent vector data type of dco/c++. The code is fully templated as suggested already in Chapter 3, i.e. the floating point data type used is always a C++ template argument. This is especially useful for the C++ compiler-based automatic generation of higher-order derivatives shown below. For better readability, throughout this section, the dco/c++ modes and their respective types are used interchangeably, i.e. the instantiation $f\langle gals\langle double \rangle \rangle$ corresponds to $f\langle gals\langle double \rangle::type \rangle$. The following templated type alias is used throughout the code

```cpp
template<typename T> using vec_t = typename std::vector<T>
```

for vector valued variables. A corresponding matrix type

```cpp
template<typename T> struct mat_t : public vec_t<vec_t<T>>
```

is derived from the standard vector extending it with further constructors and helper functions. Parts of the code use the Eigen library, a C++ header library [G+10] for linear algebra.

The code is described bottom-up, starting with the implementation of the residual Equation (4.103) and going step-by-step to the implementation of the sensitivity problem Equation (4.109). The overall calling structure of the fitting problem and the cost functional is shown and described in Figure 4.3. The upcoming descriptions of further details are excerpts with emphasis on the derivative computations.

The Residual The primal problem – Equation (4.103) – is implemented as the struct primal and with mainly one important templated member functions for evaluating the residual, i.e.

```cpp
template<typename T, typename U>
static vec_t<T> residual(const vec_t<T> &u,
```
The Implicit Time Stepping

The implicit timestepper is implemented as a Newton algorithm exploiting the sparsity of the Jacobian matrix during computation as well as during the solution of the linear system. The algorithm itself is written by the author, while the sparse linear system solver is taken from Eigen. Full templatization makes Eigen predestinated for operator overloading. In the Jacobian computation the residual function implementation is instantiated with dco/c++ data types. Since the sparsity of the residual is known (compact finite difference stencil, see Section 4.4.2), Jacobian compression [CPR74; GMP05] is employed only requiring three tangent projections. Those projections are computed using the tangent vector mode (gt1v<T,3>, see Section 3.2.1). The driver for the Jacobian computation is shown in Listing 4.1 and in turn templated with template parameter T. This is again required for instantiation with dco/c++ data types for the next level of derivatives.

```cpp
1 template <typename T> void matrix(eigen_sparse_matrix<T> &mat,
2 const vec_t<T> &lambda,
3 const vec_t<T> &u,
4 const vec_t<T> &u_previous)
5 {
6 using TRIPLET = Eigen::Triplet<T>; size_t n = u.size();
7 vec_t<typename dco::gt1v<T,3>::type> tu(n), tlambda(n);
8 for (size_t i = 0; i < n; ++i) {tu[i] = u[i]; tlambda[i] = lambda[i];}
9 for (int i = 0; i < 3; i++)
10 for (size_t j = i; j < n; j+=3)
11 derivative(tu[j])[i] = 1.0;
12 auto res = primal::residual(tu, tlambda);
13 for (size_t i = 0; i < res.size(); ++i) res[i] += u_previous[i]-tu[i];
14 vec_t<TRIPLET> coefffs; coefffs.reserve(3*n);
15 for(int i = 0; i < 3; i++)
16 for (size_t j = i; j < n; j+=3)
17 for (size_t k = std::max(0ul, j - (3-1)/2);
18 k <= std::min(n-1, j + (3-1)/2); k++)
19 coefffs.push_back(TRIPLET(k, j, derivative(res[k])[i]));
20 mat.setFromTriplets(coefffs.begin(), coefffs.end());
21 }
```

Listing 4.1: The Jacobian computation required for the Newton method used in the implementation of the implicit timestepper. The tangent vector mode of dco/c++ is used to compute the Jacobian exploiting its sparsity. The matrix format is a sparse data structure of the Eigen library, which is similar to a coordinate format: row-index, column-index and value, see lines 5, 23, and 24. Lines 15 and 16 correspond to the residual of the nonlinear system, i.e. the right-hand side of Equation (4.105). The Jacobian compression takes place in lines 11-13 and the decompression in lines 19-23.

The Fitting and its Sensitivities

The function template

\[
\text{const vec_t<U> &lambda);}
\]

which returns for a given state \( u^j = u \) and \( \lambda^j = \lambda \) the residual \( r^j \). The templatization is required to instantiate the member functions of primal with dco/c++ data types when tangent or adjoint models of the residual are needed.
solves the PDE and computes the value of the cost functional $\Phi$. The fitting algorithm instantiates $\phi$ with a first- and second-order adjoint data type from dco/c++, i.e. $\phi<\text{ga1s}<T>$ and $\phi<\text{ga1s}<\text{gt1s}<T>>$, respectively. The former is used for computing the gradient, which is the negative right-hand side of Equation (4.107), and the latter is used in each CG iteration whenever Hessian-vector products of type $\frac{\partial^2 \Phi}{\partial \lambda^2} \cdot v$ are needed. No external libraries are used for the fitting and the line search algorithms. The fitting is implemented in the function template

```cpp
template <typename T>
T phi(const size_t m, const size_t n,
      const vec_t<T> &lambda,
      const vec_t<T> &lambda_g,
      const vec_t<T> &u0,
      vec_t<T> &ue,
      const vec_t<T> &um);
```

4.4.5 Hybridization with dco/c++

The combination of symbolic and algorithmic adjoints is possible on different levels while solving the PDE but also on different levels while solving the fitting problem. Both applications are discussed in the following.

**Adjoint of the Partial Differential Equation** In addition to the fully algorithmic differentiation of the solution algorithm of the PDE, three different levels of symbolic differentiation paths are shown in Figure 4.4:

- The PDE level corresponds to a continuous adjoint formulation shown in Section 4.4.3 (marked in red).
The nonlinear system level assumes convergence of the nonlinear solver and solves a symbolically derived linear system for the adjoint as described in Section 4.2 (marked in green, called *symbolic NLS* in the following). This path is usually called the *discrete* adjoint.

The linear system level exploits the exactness of the direct linear solver and solves a symbolically derived linear system for the adjoint as described in Section 4.1 and more specifically in Section 4.3.1, since we have the linear solver in the special context of a nonlinear system (called *symbolic LS* in the following).

In the following, we first look at the required instantiations for the fully algorithmic approach. Afterwards, we compare the structure for the first two paths of above given list: the continuous adjoint (red) and the symbolic NLS adjoint (green).

The fully algorithmic approach for the computation of \( \frac{\partial \Phi}{\partial \lambda} \) Equation (4.106) can be derived from Figure 4.3. The cost functional \( \phi \) is instantiated with the adjoint data type `ga1s< double >`. As can be seen further down in the call stack, this requires second derivatives of the residual in adjoint-over-tangent mode corresponding to the data type `gt1v< ga1s< double >, 3>`. All computations are taped during the forward section and the reverse section is a straightforward interpretation of the tape without any callbacks. Using this approach yields exact derivatives of what is actually computed. This is achieved for the cost of taping the complete solution process including, e.g., the computation of the Jacobian matrix. Both other approaches, that we look at in more detail here, circumvent this enormous taping as seen mathematically in Section 4.4.3 and Section 4.2. Figure 4.5 shows the coupling with the continuous adjoint and Figure 4.6 the symbolic NLS mode.

**Adjoints of the Fitting Problem** In addition to the fully algorithmic differentiation of the implementation of the fitting problem, two levels of symbolic differentiation are possible, see Figure 4.7.

- The nonlinear system level assumes convergence of the nonlinear solver for the optimization problem and solves a symbolically derived linear system for the adjoint as described in...
4.4 An Extensive Case Study

Figure 4.5: Continuous adjoint for the computation of $\frac{\partial}{\partial \lambda}$ Equation (4.109). During the forward section only the main function is taped, and the differentiation order is reduced in $\phi$ yielding the timestepper to be called with data type `double`. The solution of the PDE is transferred via the callback interface of `dco/c++` to the reverse section (dotted arrow). The adjoint of $\phi$ is computed by solving the dual (adjoint) equation using the same timestepping mechanism as the primal solution. Since the continuous adjoint (or dual) equation is linear, the implicit timestepper only requires one iteration. In each timestep, the adjoint state variable is projected to the gradient with `dual::project`, see Equation (4.114).

Section 4.2 (called `symbolic FIT` in the following).

- The linear system level assumes convergence of the matrix-free iterative linear solver and also solves a symbolically derived linear system for the adjoint as described in Section 4.1 and more specifically in Section 4.3.2.

In Section 4.1 iterative solvers are not examined. In the implementation an inexact Newton method is used, i.e. convergence of the linear solver can in fact not be assumed. Nonetheless, the symbolic approach is implemented and analyzed in Section 6.3 runtime-wise and memory-wise. The fully algorithmic approach can again be derived from Figure 4.3. Instantiation of `fitting` with the adjoint data type `gals<double>` requires instantiations of `phi` with `gals<gtls<gals<double>>>`. This yields the already mentioned fourth-order instantiation of `primal::residual` with `gtlv<gals<gtls<gals<double>>>>>`. 3>

The calling structure of the symbolic FIT mode is shown in Figure 4.8. Taping the full fitting is circumvented by the symbolic adjoint. Nonetheless, when instantiating `fitting` with `double`, `phi` is still called with adjoint `dco/c++` data types. The computation of the adjoint of the PDE can still be done algorithmically or symbolically. The mode with the smallest memory footprint is therefore, when enabling not only symbolic FIT but also, e.g., symbolic NLS. Those combinations are analyzed in Section 6.3.
4 Hybrid Tangents and Adjoints

Figure 4.6: Symbolic NLS mode for the computation of Equation (4.109). During the forward section, the main function, \( \phi \), and the timestep loop in \( \text{timestepper} \) are taped. Whenever the nonlinear solver \( \text{implicit\_timestep} \) is met, the differentiation order is reduced and the nonlinear system is solved with data type \textit{double}. The solution vector of each nonlinear system is transferred to the corresponding part of the reverse section via the callback interface of \texttt{dco/c++} (dotted arrow). While computing the adjoints of the previously taped parts (main, \( \phi \), and \( \text{timestepper} \) — including the loop), the adjoint of the nonlinear system is computed in callback functions by solving the transposed Jacobian system and performing one adjoint projection through the primal residual. Similarities to the data flow of the continuous approach are obvious.

Figure 4.7: Algorithmic and symbolic differentiation paths for the solution algorithm of the fitting problem in Equation (4.99). The algorithmic path corresponds to a differentiation on the level of the solution algorithm. The other paths include symbolically derived adjoints. Since the solution algorithm for the fitting problem contains the computation of derivatives of the solution of the PDE, Figure 4.4 needs to be appended to the lower left and right corner of this figure.
4.4 An Extensive Case Study

Figure 4.8: Symbolic FIT for the computation of \( (\frac{\partial \tilde{\lambda}}{\partial \lambda} g, u_0, u_m) \) Equation (4.118). During the forward section, the differentiation order is reduced in fitting and the fitting algorithm is run with data type \texttt{double}. The solution of the fitting is then transferred to the reverse section via the callback interface of \texttt{dco/c++} (dotted arrow). The adjoint of the nonlinear system is computed by solving the required linear system followed by a second-order adjoint projection through \( \phi \).
5 Call Tree Reversal

The Call Tree Reversal (CTR) is a special case of the more general checkpointing approach, which is a widely used technique to resolve the problem of limited available memory when computing sensitivities of numerical simulation programs with the adjoint mode of Algorithmic Differentiation (AD). In CTR, this is done via argument checkpointing of function calls and an out-of-context recomputation using the previously stored data. Such a checkpointing technique results in a trade-off between memory requirement and additional runtime. Though CTR is not directly linked with the hybridization techniques discussed in this thesis, it fits quite well into the general context of resolving the memory consumption problem when applying adjoint AD. For implementing checkpointing schemes figured out by CTR, the high-level intrinsics interface of dco/c++ (see Section 3.3) can be used. Still, the subject of this chapter is in an early stage and not practically applicable in a productive environment. Nonetheless, a first proof of concept with a nontrivial simulation code has been done and is presented in Section 6.4.

While the concept of CTR is already described not only in AD-related text books [GW08; Nau12] but also in many papers [NU05; Utk+08; HP13], it lacks a general mathematical description of constraints and cost functional. A very famous optimal checkpointing solution for a special (but wide-spread) program structure, namely a time loop, exists and is implemented in an algorithm called revolve [Gri92; GW00]. The prerequisites assumed for the program structure in revolve is currently not representable with CTR as it is defined in this chapter and in [Nau12]. Nonetheless, further extensions are possible and planned for the future to include the solutions that are feasible with revolve also into the search space of CTR. For solving CTR, heuristics have been proposed in [Nau12, Chapter 2.3]. But so far there exists no literature where a successful application of those heuristics have been described. Often, a fixed solution is taken (fully joint mode) which is known to be feasible in most cases, but far away from an optimal solution.

This chapter is organized as follows. First, the formal definition of CTR is given (Section 5.1), followed by a description of the heuristics proposed in [Nau12] (Section 5.2). Afterwards, in Section 5.3, emphasis is put on the mixed integer programming formulation for CTR. This makes up the major part of this chapter and is followed by illustrating possibilities for reasonable penalization terms for the cost functional and approaches for a search space reduction in Section 5.4. As results we demonstrate in Section 6.4 a successful application of this method to the two-dimensional Bratu equation implemented with the software library PETSc\(^1\). Additionally, comparisons to the proposed heuristics are presented and the impact of the different penalization terms on the computed solution is shown.

Despite the fact that checkpointing can be used in the context of AD with source transformation tools and overloading tools, the terminology used here is taken from AD by overloading, for example recording or interpretation of the tape. The same terminology can be found already in the previous chapters of this thesis. In addition, the terms interpretation and reversal are used interchangeable.

5 Call Tree Reversal

To indicate the practical software framework in which CTR is embedded, the following figure shows the general procedure how to get from an original program \( P \) to the optimally checkpointed adjoint program \( \tilde{P} \) with the help of a dynamic call tree\(^2\) \( T \).

\[ P \rightarrow \tilde{P} \rightarrow T \rightarrow y \rightarrow \tilde{P} \]

The original program \( P \) is transformed to \( \tilde{P} \) with a source-to-source compiler (implementation details for this step are beyond the scope of the main part of this thesis, but can be found in Appendix A.2). In addition to the computations in \( P \), \( \tilde{P} \) gathers profiling information during runtime required for CTR, i.e. the annotated call tree \( T \), see Definition 18 later. Once \( T \) is generated and stored, either the heuristics or a mixed integer optimizer are used to solve for a (optimal) checkpointing scheme \( y \). This checkpointing scheme defines the adjoint call tree \( T \) which then needs to be implemented in the adjoint program \( \tilde{P} \) manually. \( \tilde{P} \) implements a feasible adjoint of \( P \) efficiently.

The source code and documentation of the optimization framework for CTR implemented by the author can be found in attached_code/call_tree_reversal – see also Appendix A.2.

5.1 Call Tree Reversal Problem

CTR is a special case of the more general DAG\(^3\) reversal problem (also data flow reversal problem), which is defined and proven to be NP-complete in [Nau09]. On this basis, CTR suggests a focus on subroutine arguments as potential checkpoints. Still, also CTR is NP-complete [Nau08a] as a special case of DAG reversal.

**Definition 17.** A call tree \( T = (S, r, \chi) \) with vertices \( S \) is a rooted tree with root \( r \in S \) induced by an execution of a program. Each vertex \( s \in S \) represents an instance of a subroutine and each edge connects caller and callee. The ordered set \( \chi[s] \subset S \) contains the callees of \( s \). The call order is from left to right and depth-first.

**Definition 18.** An annotated call tree is obtained by attaching values \( M^c[s] \in IR \) and \( m[s] \in IR^{n[s]+1} \) to each vertex \( s \in S \), where \( n[s] = |\chi[s]| \). \( M^c[s] \) is the size of the subroutine argument checkpoint and \( m[s] \) contains the tape sizes corresponding to the part of \( s \) executed before \( (m[s])_0 \), in-between \( (m[s])_i \), \( i \in \{1, \ldots, n[s] - 1\} \) and after \( (m[s])_{n[s]} \) its subroutine calls. Those values can graphically be attached to the edges. The required tape size for one subroutine is denoted by \( m[s] = \sum_{i=0}^{n[s]} m[s]_i \in IR \).

**Example 6.** The information of an annotated call tree can be attached to a graphical representation of \( T \). In this example, root \( r = f \), \( n[f] = 1 \), \( n[g] = 2 \), \( n[h] = n[i] = 0 \), \( m[f]_{[0,1]} = \{15, 5\} \), \( m[g]_{[0,1,2]} = \{10, 10, 10\} \), \( m[h]_0 = 100 \) and \( m[i]_0 = 50 \); \( M^c[g] = 5 \), \( M^c[h] = 10 \) and \( M^c[i] = 5 \).

\(^2\)The dynamic call tree is induced by one specific program run as opposed to the static call tree already known at compiler time.

\(^3\)directed acyclic graph
5.1 Call Tree Reversal Problem

Figure 5.1: Possible run modes for each subroutine. Left from top to bottom: execution without any additional action; execution plus recording the corresponding tape; interpretation of the tape including the release of memory. Right from top to bottom: storing the subroutine arguments required for an out-of-context evaluation of the subroutine later; restoring arguments. Combinations are possible, e.g., restoring arguments and run the subroutine with taping.

The edges are drawn as directed edges though not required for unambiguousness.

We recall a few basics from Section 2.1 (foundations) and Section 3.2.2 (adjoint mode with \texttt{dco/c++}) in the following. The plain adjoint mode, i.e., without checkpointing, consists of two phases: the forward section and the reverse section. During the forward section the tape is recorded, whereas during the reverse section it is interpreted and adjoints are propagated from the outputs to the inputs. The tape requires an amount of memory roughly linearly dependent on the number of floating point operations performed by the program, which quickly becomes infeasible. Subroutine checkpointing therefore follows the goal of trading memory for runtime by interleaving forward and reverse sections of different subroutines. For illustrating subroutine checkpointing, we follow the graphical representation from [Nau08a; Nau12; NU05].

**Definition 19.** Each subroutine has five possible run modes shown and described in Figure 5.1.

**Definition 20.** Each subroutine can be called either in \textit{split} or in \textit{joint} mode defined by a binary variable \(y_\sigma \in \{0,1\}\) bound to the callee \(\sigma \in S\): \(y_\sigma = 0\) corresponds to split mode, \(y_\sigma = 1\) corresponds to joint mode. With the run modes from Figure 5.1, the rules in Figure 5.2 are defined for caller \(s \in S\) and callee \(\sigma \in \chi[s]\).

**Remark.** The terms \textit{split} and \textit{joint} are equivalent to the terms \textit{strong} and \textit{weak}, respectively, from [GW08, Chapter 12.2: Reversal of Call Trees].

**Definition 21.** Given a call tree \(T\), the corresponding adjoint call tree \(\overline{T}\) is the result of applying a given reversal scheme \(y \in \{0,1\}^{|S|} = (y[s])_{s \in S}\) to \(T\) with the rules from Definition 20. In addition, the root \(r\) is always run in recording mode first, directly followed by its interpretation.

**Lemma 22.** Execution of \(\overline{T}\) with a call order from left to right and depth-first computes correct adjoints of the original program.
5 Call Tree Reversal

Figure 5.2: Rules for the run modes of the callee depending on the run mode of its caller and the call mode (split or joint). In split mode ($y[\sigma] = 0$, left) the callee behaves the same way as the caller. In joint mode ($y[\sigma] = 1$, right) on the other hand, the callee’s run modes are different than those of its callee. Since the callee is only checkpointing its arguments as part of its caller’s tape recording, the callee’s tape recording needs to take place during interpretation of the caller’s tape after restoring the arguments.

\[
\begin{array}{ccc}
\text{run mode of } s & \text{run mode of } \sigma \\
\hline
\begin{array}{c}y[\sigma] = 0 \\
\begin{array}{c}
\begin{array}{c}
\text{Split mode (left)} \\
\text{Joint mode (right)}
\end{array}
\end{array}
\end{array}
\end{array}
\]

Figure 5.3: (a) shows a simple call tree: $f$ calls $g$. (b) shows the split mode for $g$, i.e. $y[g] = 0$. The subroutine $g$ is called in record mode, when its caller is also recording. After having finished recording of the root $f$, its interpretation begins. When the reversal meets the point where adjoints of $g$ are required, the tape for $g$ is interpreted followed by the remainder of $f$’s tape. (c) shows joint mode for $g$, i.e. $y[g] = 1$. The execution starts with recording $f$ interrupted by storing the arguments of $g$ and running it. The execution gets handed back to recording the remainder of $f$. During interpretation of $f$’s tape, at the point where we need adjoints of $g$, the arguments of $g$ are restored, followed by recording and interpreting the tape for $g$. Afterwards, the remainder of $f$ is interpreted.

Proof. Descriptively done in [GW08, Chapter 12.2].

Example 7. In Figure 5.3, graphical representations of a simple example call tree and respective adjoint call trees for split and joint modes are shown.

With the given notation, CTR from [Nau12, Chapter 2.3] gets the following form.

Definition 23. CTR is to find a reversal scheme $y = y[s]_{s \in S}$ for the call tree $T(S, r, \chi)$ for a given upper bound $M$ on the available persistent memory such that the corresponding reversal is feasible and optimal, i.e., the additional memory consumption for the reversal is lower than $M$, and there is no other feasible reversal scheme $y'$ with a cost $c(y') < c(y)$.

With a corresponding cost $\bar{c}[s] \in \mathbb{R}$ for running a subroutine $s \in S$ once (including its own subroutines), the natural cost function also described in [Nau12] is chosen to be the additional cost introduced by the reversal scheme due to the reevaluation of each checkpointed subroutine and thus given as

\[
c^\delta(y, \bar{c}) = \sum_{s \in S} y[s] \cdot \bar{c}[s], \quad c^\delta : \{0, 1\}^{|S|} \times \mathbb{R}^{|S|} \to \mathbb{R} \quad \text{with} \quad \bar{c} = (\bar{c}[s])_{s \in S}.
\]
5.2 Heuristic Approaches

In that case $c(y) = c_s(y, \bar{c})$ is to be minimized in CTR. We may want to add different penalization terms later. This is done in Section 5.4.

Both heuristics presented in the following are unmodified taken from [Nau12], but presented in a more detailed shape. For Algorithm 1 and Algorithm 2, the set of subroutines $S$ is defined to be a totally ordered set with the order $S_i < S_j : m[S_i] \leq m[S_j]$. Additionally, we assume an implementation to be present, that performs the test if a given reversal scheme $y$ is feasible or not (explicitly shown later in Section 5.3). In addition, both heuristics make two assumptions. First, they assume that the global joint mode, i.e., $y \equiv 1$, is a feasible solution. And second, they assume that the cost function gets improved the more subroutines are handled in split mode. Since the checkpoint size of a subroutine is in the very most cases smaller than its tape size, also the latter assumption is reasonable.

The Largest-Memory-Increase-First (LMI) heuristic (Algorithm 1) begins with a fully joint tree, i.e. $y \equiv 1$, and switches the modes of the subroutines one by one, starting with the subroutine having the largest recording size. As soon as the scheme is not feasible anymore, the algorithm stops.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Data:} the totally ordered set $S$
\State \textbf{Result:} approximation to the optimal and feasible reversal scheme $y$
\State $y[S_j] = 1 \quad \forall j \in \{1, ..., |S|\}$
\State $i = |S|$;
\While {$y$ a feasible scheme and $i > 1$ do}
\State $y[S_i] = 0$;
\State $i = i - 1$;
\EndWhile
\State $y[S_i] = 1$;
\end{algorithmic}
\end{algorithm}

Algorithm 1: Largest-Memory-Increase-First (LMI) heuristic.

The Smallest-Memory-Increase-First (SMI) heuristic (Algorithm 2) also begins with a fully joint tree, i.e. $y \equiv 1$, but switches the modes of the subroutines one by one, starting with the subroutine having the smallest recording size. As soon as the scheme is not feasible anymore, the algorithm stops.

Remark. SMI is identical to the Largest-Memory-Decrease-First (LMD) heuristic starting with a fully split reversal and approaching the feasible solution from the other side. This incarnation of the same heuristic is expected to reach the approximate solution faster. It is shown in Algorithm 3 and runtimes are compared in Section 6.4.

In Section 6.4 those heuristics are compared to the mixed integer programming formulation not only runtime-wise, but also in terms of solution quality and flexibility.

5.3 Mixed Integer Programming Formulation

In this section, an equation system for the (maximum) memory consumption of the adjoint call tree is derived. To do so, the following additional real-valued variables and corresponding
5 Call Tree Reversal

Data: the totally ordered set $S$
Result: approximation to the optimal and feasible reversal scheme $y$

1. $y[S_j] = 1 \quad \forall j \in \{1, ..., |S|\};$
2. $i = 2;$
3. while $y$ a feasible scheme and $i \leq |S|$ do
   4. $y[S_i] = 0;$
   5. $i = i + 1;$
4. end
5. $y[S_i] = 1;$

**Algorithm 2:** Smallest-Memory-Increase-First (SMI) heuristic.

Data: the totally ordered set $S$
Result: approximation to the optimal and feasible reversal scheme $y$

1. $y[S_j] = 0 \quad \forall j \in \{1, ..., |S|\};$
2. $i = |S|;$
3. while $y$ an infeasible scheme and $i > 1$ do
   4. $y[S_i] = 1;$
   5. $i = i - 1;$
4. end

**Algorithm 3:** Largest-Memory-Decrease-First (LMD) heuristic.

Equations are introduced. We formulate the equations directly in terms of the reversal scheme, i.e. the binary vector $y$, and therefore end up with a valid integer programming problem. The recording part on the one hand is distinguished from the descending and ascending part of the reversal on the other hand.

• The recording (forward) part: $M^f[s]$ is defined for each subroutine $s \in S$. It contains the memory consumption required for recording $s$ including all its subroutines. This is done under consideration of the corresponding call modes (split or joint) of all subroutines of $s$, i.e. for all $\sigma \in \chi[s]$. The derivation for the equation is shown in Figure 5.4 and yields the recursive formula

$$M^f[s] = m[s] + \sum_{\sigma \in \chi[s]} (1 - y[\sigma]) \cdot M^f[\sigma] + y[\sigma] \cdot M^c[\sigma]. \quad (5.2)$$
5.3 Mixed Integer Programming Formulation

\[ y[\sigma] = 0: \text{When recording } s, \text{ also } \sigma \text{ gets recorded, i.e. } M_f[s] = m[s] + M_f[\sigma] + \cdots \]

\[ y[\sigma] = 1: \text{When recording } s, \text{ } \sigma \text{ gets only checkpointed, i.e. } M_f[s] = m[s] + M_c[\sigma] + \cdots \]

Figure 5.4: The memory consumption for recording subroutine \( s \) amounts to its own memory \( m[s] \) and the sum over the memory for recording or checkpointing all its callees \( \sigma \) according to their respective mode \( y[\sigma] \) yielding Equation (5.2).

- The reversal part: \( M^r[s]_i \) is defined for each subroutine \( s \in S \) and \( i \in \{0, \ldots, n[s]\} \). It contains the memory consumption state before starting the reversal of the \( i \)-th part of \( s \). We need to distinguish between the state before reversing the last part of \( s \) (\( M^r[s]_{n[s]} \)) and an intermediate state. The last part’s state (descending part) retrieves information from the caller’s memory state (which is reversed before), whereas each intermediate position’s state (ascending part) retrieves information from the callee’s memory consumption state.

The descending part: \( M^r[s]_{n[s]} \) contains the memory consumption before starting to reverse the last part of \( s \). We get different equations for the root vertex (top-level routine, caller only, see derivation in Figure 5.5) and all the other vertices (callee, maybe caller, see derivation in Figure 5.6). This is caused by the requirement, that the top-level routine is always reversed directly after recording (see Definition 21). The derivation from Figures 5.5 and 5.6 yields

\[ M^r[r]_{n[r]} = M_f[r] \]  \hspace{1cm} (5.3)

\[ M^r[s]_{n[s]} = M^r[\kappa]_i - m[\kappa]_i + y[s] \left( M_f[s] - M^r[s] \right) \]  \hspace{1cm} (5.4)

for all \( s \in S \setminus \{r\} \) and \( \kappa, i : s = \chi[\kappa]_i \).

Figure 5.5: The root vertex \( r \) is always reversed directly after recording; the memory consumption before reversing its last part is therefore identical to the memory consumption of its recording. This yields Equation (5.3).
5 Call Tree Reversal

(a) $y[s] = 0$: On the left and the right the recording and interpretation part of $\kappa$, respectively. The memory consumption before reversing the last part of $s$ is the memory consumption after having finished reversing its successor callee (in the original call order) minus the memory that gets freed by reversing the intermediate part of the caller $\kappa$ in between the calls to its successor and $s$: $M^r[s]_{n[s]} = M^r[\kappa]_i - m[\kappa]_i$ with $i : s = \chi[\kappa]_i$.

(b) $y[s] = 1$: On the left and the right the recording and interpretation part of $\kappa$, respectively. The memory consumption is similar as in Figure 5.6a; additional terms are introduced due to restoring the arguments of $s$ ($M^C[s]$ gets freed) and recording $s$ ($M^I[s]$ needs to be considered) before being able to reverse it: $M^r[s]_{n[s]} = M^r[\kappa]_i - m[\kappa]_i + M^I[s] - M^C[s]$.

Figure 5.6: The descending part: Combining the equations from above taking $y[s]$ into account yields Equation (5.4).

The ascending part: $M^r[s]_i$ is defined for each subroutine $s \in S$ and $i \in \{0, \ldots, n[s] - 1\}$. It contains the current memory consumption before starting to reverse the $i$-th part of $s$. There is no difference between the root vertex and the other vertices. The derivation is shown in Figure 5.7 and yields

$$M^r[s]_i = M^r[\sigma]_0 - m[\sigma]_0,$$

(5.5)

for all $s \in S$ and $i \in \{0, \ldots, n[s] - 1\}$ with $\sigma = \chi[s]_{i+1}$. 
5.3 Mixed Integer Programming Formulation

With the variables $M^f[s] \in R$ and $M^r[s] \in R^{n[s]}$ from above and the already introduced reversal scheme $y$, we get the full equation system to retrieve the course of the memory consumption while traversing the adjoint call tree. The forward part is given by

$$M^f[s] = m[s] + \sum_{\sigma \in \chi[s]} (1 - y[\sigma]) \cdot M^f[\sigma] + y[\sigma] \cdot M^r[\sigma] \quad (5.6)$$

for all $s \in S$. The descending part for the root $r$ is defined as

$$M^r[r]_{n[r]} = M^f[r] \quad (5.7)$$

and for all $s \in S \setminus \{r\}$ as

$$M^r[s]_{n[s]} = -m[k]_{1} + M^r[k]_{1} + y[s] \left( M^f[s] - M^c[s] \right) \quad (5.8)$$

with $k, i : s = \chi[k]_{i}$. The ascending part for all $s \in S$ is given by

$$M^r[s]_{i} = M^r[\sigma]_{0} - m[\sigma]_{0} \quad \forall i \in \{0, \ldots, n[s] - 1\}, \quad (5.9)$$
with σ = χ[s]_{i+1}. In addition to those equations the constraint

\[ M^r[s]_{n[s]} \leq M \quad \forall s \in S. \]  

(5.10)

ensures feasibility, because a maximal memory consumption can only occur before reversing the last part of a subroutine. CTR is fully stated as an integer programming problem with the cost functional Equation (5.1).

### 5.4 Penalization and Search Space Reduction

Apart from the obvious choice for the cost functional already defined in Equation (5.1), we may add additional penalization terms. Two additional terms are proposed in the following. Pragmatically speaking, implementation of joint mode for a subroutine may result in nontrivial effort, since the argument checkpoint possibly contains non-obvious data like global or static variables, memory locations hidden behind pointers, etcetera. Nevertheless, this effort usually needs to be invested only once per subroutine, as opposed to each subroutine call. Highly desirable is therefore, that the number of different subroutines that get checkpointed is kept small. We therefore introduce the set \( \hat{S} \subseteq S \), where each subroutine is only represented once, though there may exist multiple instances in \( S \). Correspondingly, we introduce an additional binary variable \( \hat{y} = (\hat{y}[s])_{s \in \hat{S}} \), where \( \hat{y}[s] = 1 \) if at least one instance of \( s \) is checkpointed. Penalization is done with a constant factor \( \hat{c} \in \mathbb{R} \) representing the implementation effort for any \( s \in \hat{S} \). This yields an additional term for the cost functional

\[ c^i(\hat{y}) = \hat{c} \cdot \sum_{s \in \hat{S}} \hat{y}[s]. \]  

(5.11)

Penalization can also be applied per checkpoint to keep the overall number of checkpoints small with a penalization constant \( \tilde{c} \in \mathbb{R} \) yielding the term

\[ c^p(y) = \tilde{c} \cdot \sum_{s \in \hat{S}} y[s]. \]  

(5.12)

In combination with the original cost function we end up with

\[ c(y) = c^s(y) + c^i(\hat{y}) + c^p(y), \]  

(5.13)

where the influence of the various penalization terms can be controlled by the corresponding constants \( \hat{c} \) and \( \tilde{c} \). The impact on the solution will be shown in Section 6.4.

Search space reduction on the other hand is required, since the number of called subroutines in an instance of a program easily exceeds a few hundreds of thousand (see Section 6.4). This directly transforms to the search space dimension of CTR. But the requirement for reducing this problem size is twofold. The first drawback of taking too many subroutines into account is the impact on the performance while collecting the profiling information for generating the call tree. The other problem is of course the problem dimension itself when trying to solve CTR. We follow two strategies for a problem size and search space reduction.

Following the first strategy, we directly ignore specific subroutines during the generation of the call tree. This is done mainly via three different categories:
5.4 Penalization and Search Space Reduction

- a subroutine is defined in a file, which we explicitly exclude due to some reason (e.g. a file containing negligible helper routines), or
- a subroutine is part of a namespace we do not want to change / checkpoint anyway (e.g. the std namespace), or
- a subroutine has a specific signature (e.g. is defined as inline).

The second strategy takes place as a preprocessing step to the solver. For a given memory threshold value subroutines are automatically inlined if they have a tape size lower than the threshold. In this case, the subroutine’s tape size gets added to the caller’s tape size. This may at least be very useful for subroutines generating no tape at all. Both strategies should be applied carefully to not discard characteristic subroutines of the problem. Nonetheless, both have also proven to be helpful in the case study and yielded reasonable runtimes for solving CTR.
6 Results and Measurements

This chapter summarizes selected results of the author’s work by revisiting the three principal points discussed in this thesis: 
dco/c++, hybridization, and Call Tree Reversal.

- First, further developments of 
dco/c++ in combination with scientific collaborations has led to a number of projects, some of which are briefly recapitulated in Section 6.1. Remarks concerning the use of hybridization are already included there. The reader may get an impression what kind of projects can successfully be approached with 
dco/c++. Closely related is the plain performance of 
dco/c++ which is discussed in Section 6.2. This also includes the comparison of different second-order modes for computing Hessian matrices. A general comparison between different overloading tools is given based on the AD test suite shown in Section 3.4.

- Second, in Section 6.3, the hybridization approach is put into focus referencing already published results and showing runtime and memory consumption enhancements demonstrated with the case study Section 4.4. General remarks concerning complexities of the different approaches are underpinned with measurements and in addition the reader may get a feeling of the possibilities of hybridization. In the author’s opinion, at the end, 
dco/c++ is the helping hand taking care of keeping the chain rule in place and differentiating only the very lowest model definition by plain AD.

- Third, in Section 6.4, the Call Tree Reversal problem is addressed, starting with a description of the solution approach for the mixed integer programming formulation. Based on a case study, comparisons are drawn of resulting reversal schemes of the heuristics and the integer solvers. Additionally, the impact of the different cost functionals are tested.

6.1 Reference Projects

dco/c++ has been used in many projects since the development of its core. The more features were introduced, the more applications were possible. In this section, selected applications are presented. Those range from atmospheric remote sensing over computational fluid dynamics to a biotechnological application. Section-wise, after a general introduction into the discussed software, results concerning 
dco/c++ are shown with remarks on hybridization. At this point I want to clearly state, that the achieved results presented in this section were only possible thanks to very productive and enthusiastic project partners. Those partners are acknowledged by inlining corresponding citations at the end of each section.

6.1.1 JURASSIC

The Institute of Energy and Climate Research – Stratosphere at Research Center Jülich has been developing the Juelich Rapid Spectral Simulation Code Version 2 (JURASSIC2) for deriving
atmospheric constituents from measurements taken remotely from aircrafts or satellites. This ill-posed inverse problem is highly sensitive with respect to errors in the measurements. It is commonly solved by gradient based methods [Rod00] applied to an appropriately regularized problem formulation. For computing the gradients, an efficient adjoint version of the forward simulation model is required. For the adjoint, the exploitation of the problem structure turns out to be essential. The results shown in this section are already published in the articles cited below.

Figure 6.1: Limb sounding configuration. $y_m$ are the measurements, $x^*$ the unknown atmospheric state and the line-of-sights illustrate the ray tracing approach.

JURASSIC2 is a retrieval processor used in the field of atmospheric remote sensing. It aims to derive atmospheric variables such as temperature or trace gas volume mixing ratios from emitted or scattered radiation. These can then be used to improve long-term climate models and short-term weather-forecast. JURASSIC2 has been optimized to evaluate measurements in the infrared part of the spectrum made by airborne or satellite-borne limb-sounders, which receive radiation tangential to the surface of the earth. The derivation of such quantities from infrared measurements is an inverse problem. Initially, only a forward simulation model is available that maps a state of atmospheric quantities onto simulated measurements. The retrieval process numerically inverts this process and – depending on the method – requires first and/or second derivatives of the forward simulation model. The time required for the retrieval is often dominated by the evaluation of these derivatives. Derivative values were previously gained by the application of finite difference methods. The focus of our work was on addressing the accuracy as well as runtime drawbacks of finite differences by using Algorithmic Differentiation. Therefore, we applied ADOL-C to the forward model and analyzed the impact on runtime and convergence rate of the Newton-like iteration methods. In JURASSIC, the forward model is represented by $y = F(x)$, $F : \mathbb{R}^n \to \mathbb{R}^m$, where $x \in \mathbb{R}^n$ define the state of the atmosphere and $y \in \mathbb{R}^m$ the measurements – see Figure 6.1. The inverse problem can be formulated as a minimization of the cost function

$$J(x) = \frac{(x - x_\alpha)^T S_\alpha^{-1} (x - x_\alpha)}{i} + \frac{(F(x) - y_m)^T S_e^{-1} (F(x) - y_m)}{ii}, \quad (6.1)$$

consisting of a regularization term (I) including an a priori state $x_\alpha \in \mathbb{R}^n$ and a weighted least-squares term (II) including the measurements $y_m \in \mathbb{R}^m$. $S_\alpha$ and $S_e$ are covariance matrices of

90
Table 6.1: Memory consumption of the local reversal and runtime comparison – in MByte and seconds for finite differences (FD) and using dco/c++.

<table>
<thead>
<tr>
<th>Case</th>
<th>n</th>
<th>m</th>
<th>Memory</th>
<th>Runtime FD</th>
<th>Runtime dco/c++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>6</td>
<td>3.0</td>
<td>0.03</td>
<td>0.028</td>
</tr>
<tr>
<td>2</td>
<td>175</td>
<td>150</td>
<td>4.2</td>
<td>2.54</td>
<td>0.659</td>
</tr>
<tr>
<td>3</td>
<td>46080</td>
<td>9191</td>
<td>145.5</td>
<td>32095</td>
<td>5165</td>
</tr>
</tbody>
</table>

the error distribution of \(x_\alpha\) and \(y_m\), respectively. As we only have one output, adjoint mode of AD is the method of choice. Due to the huge memory requirements for taping the entire \(J(x)\), a global reversal is impractical for real world measurement data. Therefore, the inner structure of \(F(x)\) is exploited to apply a local reversal, which is related to an hierarchical approach [BH96] or interface contraction/narrowing [GW08, Chapter 10.2]. Thereby, we consider \(F(x) = H(G(x))\) and obtain

\[
J'(x) = F'(x)^T S_e^{-1}(F(x) - y_m) + S_\alpha^{-1}(x - x_\alpha)
\]

(6.2)

\[
G'(x)^T H'(G(x))^T S_e^{-1}(F(x) - y_m) + S_\alpha^{-1}(x - x_\alpha),
\]

(6.3)

with nonlinear mapping \(G(x) = (g_i(x))_{i=1,...,p}\), \(g_i : \mathbb{R}^n \rightarrow \mathbb{R}\) and the linear mapping \(H\). Having this formulation adjoint mode is applicable to \(G(x)\) yielding the required product \(G'(x)^T \cdot \bar{v}\). Moreover, the elements \(g_i(x)\) are mutually independent, which enables us to apply the adjoint mode to every \(g_i\) locally as well as to exploit the parallelism. Hence, the memory allocated for the tape of \(g_i\) can be reused during the reversal of \(g_{i+1}\), which decreases the memory consumption by orders of magnitude. Table 6.1 shows the memory consumption of the local adjoint mode solution and a runtime comparison of a single gradient computation of \(J(x)\) on three test cases. For numerical reasons the main focus in JURASSIC is not on the computation of \(J'(x)\) but rather on the Jacobian \(F'(x)\) as it is assumed to remain unchanged over several iteration steps. Fortunately, it is possible to exploit the same structure described above while computing \(F'(x)\). This reduces the cost of computing \(F'(x)\) in plain adjoint mode from \(O(m) \cdot \text{Cost}(F)\) to \(O(1) \cdot \text{Cost}(F)\).

References

6 Results and Measurements

Figure 6.2: Two example computations of adjoint OpenFOAM. Sensitivities are computed on the body of a VW polo car of approximately two million cells and a motorbike. The color represents the sensitivities of the total drag force on the bodies with respect to porosity. Courtesy of Arindam Sen and the Aboutflow project [Pro17] for the pictures.

6.1.2 OpenFOAM

OpenFOAM$^1$ is a Computational Fluid Dynamics (CFD) software package written in C++. OpenFOAM features a wide range of applications and a growing user base in both academia and industry. Adjoint methods are also in CFD crucial ingredients of state-of-the-art gradient-based solvers for high dimensional optimization problems.

In the following, one possible optimization problem is sketched which is used for research at the institute for Software and Tools for Computational Engineering. For a given discretization of the computational domain (into a potentially very large number of $n$ individual cells) topology optimization [BP03; Oth08] penalizes cells which are undesirable for the flow pattern. This penalization is realized by adding a source term $-\alpha \mathbf{v}$ to the $k$D ($k \in \{2, 3\}$) Navier-Stokes equations, thus blocking cells with high values of impermeability $\alpha$ for the flow:

$$\mathbf{v} \cdot \nabla = \nu \nabla^2 \mathbf{v} - \frac{1}{\rho} \nabla p - \alpha \mathbf{v} \quad (6.4)$$

$$\nabla \cdot \mathbf{v} = 0 \quad (6.5)$$

where $\mathbf{v} \in \mathbb{R}^k$ and $\nu, \rho, p, \alpha \in \mathbb{R}$. The discretized [FP99] equations can then be solved for the discretized velocity field $U \in \mathbb{R}^{k \times n}$ and pressure field $p \in \mathbb{R}^n$ by a suitable method such as the SIMPLE-Algorithm [PS72]. Gradient information $\partial J / \partial \alpha \in \mathbb{R}^n$ becomes necessary to determine an $\alpha \in \mathbb{R}^n$ which minimizes a given cost functional $J \in \mathbb{R}$. Being able to compute the gradient, one can iteratively refine $\alpha$ from an initial guess, for example, by applying steepest descent

$$\alpha^{n+1} = \alpha^n - \lambda \cdot \frac{\partial J}{\partial \alpha}$$

with local line search parameter $\lambda$.

dco/c++ was successfully applied to OpenFOAM 2.3.1 for above described problem formulation. This is documented and published in the articles listed below. Since many real-world simulations have runtimes of hours or even days, for those cases, plain application of dco/c++

$^1$http://www.openfoam.com/
is impossible due to huge memory requirements. It is therefore crucial to apply checkpointing techniques and hybrid methods to be able to solve reasonable sized problems. This is heavily pursued for OpenFOAM. Symbolic adjoints of linear and nonlinear solvers (see Chapter 4) as well as reverse accumulation techniques [Chr94] have been successfully coupled with dco/c++. A results of that can be seen in Figure 6.2. The goal of getting a flexible discrete adjoint of OpenFOAM via dco/c++ and hybridization for very large problem sizes is still ongoing research.

References


6.1.3 PETSc

PETSc\(^2\) stands for Portable, Extensible Toolkit for Scientific Computation and is a library for scientific applications modeled by partial differential equations. It is written in C. The results shown here are already published in [Lot+13]. The work presented in the paper is meant as a proof of concept that fully discrete adjoints of PETSc v3.3 are possible with the available AD tools. We proved that the combination of dco/c++ and Adjoint MPI is robust enough to compute fully algorithmic discrete adjoints of PETSc. Four distinct steps were necessary. First, dco/c++ had to be applied to PETSc by overloading all operations performed on variables of type PetscReal and PetscScalar – compilation was then done by a C++ compiler. Second, BLAS had to be adjoined by writing adjoint BLAS (Basic Linear Algebra Subprograms) functions by hand. Third, the LAPACK [And+99] routines were adjoined using again dco/c++. Finally, the adjoint MPI library was used to adjoin all the MPI communication. For the adjoint BLAS library as well as the adjoint MPI library, the high-level intrinsic interface of dco/c++ was used.

However, a conclusion on the performance of the discrete adjoint especially with respect to speedups through parallelization was not possible at this time. The benchmarks in Table 6.2 show why. As case study, we took ex5 from the PETSc examples, which solves the Bratu equation from the MINPACK-2 test suite [ACM91] and we computed the gradient of an artificial cost functional (norm of the solution) with respect to the boundary values. All tests were conducted on the MPI nodes of the Bull HPC-Cluster at RWTH Aachen. They are 2-socket systems equipped with Intel Westmere EP processors with 24GB memory each. With a computational grid of only 128 × 128 a memory usage of 16GB is reached in adjoint mode. The corresponding runtime is 3.8s. With such a runtime we are far from any real-world application. Increasing the

\(^2\)http://www.mcs.anl.gov/petsc

93
### Results and Measurements

<table>
<thead>
<tr>
<th>Memory Usage</th>
<th>Original RT (s)</th>
<th>Adjoint Mode RT (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Process</td>
<td>16GB</td>
<td>0.22</td>
</tr>
<tr>
<td>2 Processes</td>
<td>8GB</td>
<td>0.14</td>
</tr>
<tr>
<td>4 Processes</td>
<td>4GB</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 6.2: Memory footprint and runtime for PETSc test cases. Scaling of the runtime of the original function (Original RT) and the adjoint thereof (Adjoint Mode RT). The memory consumption (Memory Usage) is mainly due to the taping procedure of dco/c++ and Adjoint MPI.

The number of processes for the $128 \times 128$ to more than 4 with the goal to also increase the available memory is not reasonable either, since the overhead due to communication takes its toll.

First non-published but promising steps were made towards hybrid adjoints of PETSc with dco/c++ by adjoining the nonlinear solver symbolically. Nonetheless, the successful application of dco/c++ and the adjoint MPI library described above is an important basis hybridization can build upon.

#### References


#### 6.1.4 Nixe-2.0

The Institute of Process Systems Engineering at RWTH Aachen is developing the C++ ODE (ordinary differential equation) solver library NIXE 2.0 as the successor of NIXE 1.0, which was first presented in [Han+10]. Results shown in this section are already published in [Lot+15]. Emphasis in this work was on the interface to the users code for NIXE when coupled with dco/c++. This is a very successful application of hybrid methods to a real-world ODE solver, as we also coupled dco/c++ and a previously published modified discrete adjoint method [Han+10].

Technical or physical systems are often modeled by means of parametric ODEs. In the following, we consider parameter estimation with ODEs embedded. The structure of the objective function $\Phi$ is shown and explained in Figure 6.3. If the minimization is carried out by a second-order derivative-based method, the numerical optimizer requires first and second derivatives of the objective with respect to the free parameters. According to the chain rule of differential calculus, derivatives of the objective naturally include derivatives of the embedded ODE solutions. The approach is modularized and flexible in that adjoint subroutines can be generated automatically or optionally be coded manually.

We consider parametric initial value problems (IVPs) of type

$$
\dot{x} = f(t, x, p), \quad t \in [t_0, t_f], \quad x(t_0) = x_0(p),
$$

(6.6)
6.1 Reference Projects

![Diagram](image)

Figure 6.3: Illustration of the structure of the objective function $\Phi$. The parameters $p \in \mathbb{R}^{n_p}$ enter together with state $x_{i-1} \approx x(t_{i-1}) \in \mathbb{R}^{n_x}$ the $i$-th embedded ODE solver call $S_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \rightarrow \mathbb{R}^{n_x}$, $i = 1, \ldots, N$. The solver is called $N$ times in a loop, where for each intermediate state a contribution $\varphi_i$ is computed. Those contributions are then used to form the objective $\Phi$.

where $x(t) \in \mathbb{R}^{n_x}$ is the state vector, $\dot{x}(t)$ its time derivative, $t$ is the independent time variable, $p \in \mathbb{R}^{n_p}$ is a time-invariant parameter vector, $t_0$ and $t_f$ are the initial and final times, respectively. The functions $f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \rightarrow \mathbb{R}^{n_x}$ and $x_0 : \mathbb{R}^{n_p} \rightarrow \mathbb{R}^{n_x}$ provide the right-hand side of the ODE and the initial values, respectively. They are assumed to be $\kappa + J$ and $\kappa$ times continuously differentiable, respectively, with $\kappa \geq 2$. The number $J \geq 1$ will denote the order of the integration method for the numerical solution of the IVP. Under weak assumptions on $f$ and $x_0$, the solution $x(t, p)$ is $\kappa$ times continuously differentiable with respect to $p$, which we assume in the following. For details on the existence theorems and mathematical properties of ODE systems we refer to [HNW93]. In addition, we assume the overall objective function to be $\kappa$ times continuously differentiable.

NIXE implements the solution of (6.6), but also a modified discrete adjoint method. The genericity of C++ facilitates the seamless coupling of dco/c++ with NIXE. This combination enables arbitrary-order adjoint evaluation of the solver.

Apart from NIXE itself and dco/c++, the key ingredient of AD-enabled NIXE is C++ templates. Two important prerequisites are needed. First, all floating point operations within NIXE are performed with a generic data type $\mathbb{T}$, which enables execution of NIXE with double as well as dco/c++ data types. Second, the IVP definition is generic with respect to the floating point data type $\mathbb{T}$. The function EvalF needs to be implemented by the user to compute the right-hand side $f$ of the ODE (6.6), see the following listing.

```cpp
template<typename T>
struct model {
    static void EvalF(...) { ... }
    ...
};
```

This model definition is passed to NIXE as a class template (rather than an already instantiated template class) to enable its instantiation with arbitrary data types. The main NIXE routine is declared as follows.

```cpp
template<typename T>
class MODEL, typename T> void nixe(...);
```
6 Results and Measurements

Figure 6.4: Instantiation graph for \( \text{nixe} \) and \( \text{adjointNixe} \), the top-level subroutines of \( \text{NIXE} \) for simulation and modified discrete adjoint sensitivity analysis, respectively. \( M \) is the model class template.

It is generic with respect to arithmetic type \( T \) and the generic IVP class definition. The arguments to the function include the parameters \( p \) and the previous state \( x_{i-1} \) as inputs and the current state \( x_i \) as output (see Figure 6.3).

Integration of the ODE in (6.6) requires the evaluation of \( f \) and of its Jacobian. Furthermore, the modified hand-written adjoint integrator \( \text{adjointNixe} \) mentioned earlier, requires adjoint projections of \( f \). The instantiation graph with the corresponding dco/c++ data types is shown in Figure 6.4. All required derivative models are generated automatically.

An adjoint of \( \text{NIXE} \) can now be computed either by calling the hand-written adjoint \( \text{adjointNixe} \) with the data type \( \text{double} \) or by calling the original function \( \text{nixe} \) with a dco/c++ adjoint data type. Both versions should yield similar but not identical results, because \( \text{adjointNixe} \) is based on modified discrete adjoints that merely approximate the exact discrete adjoints. This difference can also be observed in the overall convergence behavior of the nonlinear solver for the parameter estimation problem further down. For second-order optimization algorithms the Hessian of the objective including \( \text{NIXE} \) is required. It can be computed by instantiating the functions and classes in Figure 6.4 with a tangent data type instead of \( \text{double} \). Nested templates in C++ allows us to add another order of derivative information to the model with minimal programming effort. Derivatives of arbitrary order can be implemented very elegantly.

Both ways of adjoining \( \text{NIXE} \) are seamlessly integrated into an overall AD version of the objective function with dco/c++. Thus, AD can be applied to arbitrary objective functions with embedded calls of the ODE integrator. This is a prime example for hybrid adjoints using dco/c++.

References


6.1.5 Smaller Projects

In the following two smaller projects are presented. Both were carried out in form of Bachelor’s theses. References are also inlined below.
Ceres

The Ceres solver [A+] is an open source C++ library for modeling and solving nonlinear least-squares problems. It is developed by Google Inc. first internally and since 2012 as open source. For the solution of the nonlinear systems, Ceres solver comes with AD capabilities including only a tangent mode implementation. The interface to AD is designed seamlessly and elegantly using template techniques in C++. Ceres is designed to solve problems in the form

$$\min_x \frac{1}{2} \sum_{i=0}^{k} \| [F]_i(x) \|^2 \quad \text{where} \quad x \in \mathbb{R}^n \quad \text{and} \quad [F]_i : \mathbb{R}^n \to \mathbb{R}^m \quad (6.7)$$

usually representing fitting problems. As described in the manual, Ceres was mainly written to reconstruct 3D geometry from an unorganized set of photographs. This was shown to be possible with pictures taken by tourists of the city of Rome [Aga+11]. The models Ceres was designed for have the important property that all $[F]_i$ from Equation (6.7) are mutually independent. Exploiting this structure while computing Jacobian matrices is promising when using the adjoint mode of AD [LNU12] (see also Section 6.1.1).

We employed dco/c++ adjoint mode on Ceres including the exploitation of above mentioned structure. This approach was able to speed up test problems with large dimension $n$ and $m = 1$ impressively, while real applications with small $n \approx 12$ and $m = 2$ showed only a slight effect. This work was done under the author’s guidance in a Bachelor’s thesis.

References


TinyFlow

TinyFlow is a non-validated three-dimensional unsteady incompressible DNS solver with Boussinesq approximation [Etl08; Bou03] for modeling temperature dependency on vertical forces. The code was originally written by the author while working at the Institute of Meteorology and Climatology (MuK), University of Hanover in 2010. The implementation was first done in Fortran and later translated into C++. The Fortran implementation was based on the numerics used in the simulation software PALM [RS01] also developed at the MuK. Over the last five years, dco/c++ and dco/fortran were employed on TinyFlow for research as well as teaching activities.

In contrast to a symbolic treatment of the linear solver using direct methods as shown in Section 4.1, TinyFlow was used to analyze the effect of solving both the original as well as the transposed linear system with matrix-free iterative linear solvers like the GMRES method [SS86] as shown in Section 4.3.2. For matrix-free methods, implicit products of the system matrix with an arbitrary vector are required. In contrast to Section 4.3.2, here we know that the implicit matrix-vector product already is a tangent model of the multi-variate vector function $F$. Applying the approach from Section 4.3.2 would required second-order derivatives implicitly. This can be circumvented as shown in the following.

3http://aboutflow.sens.qmul.ac.uk/events/nice2014/training/
6 Results and Measurements

Using the backward Euler scheme for the time stepping yields a nonlinear system \( F(u) = 0 \), \( F : \mathbb{R}^n \to \mathbb{R}^n \) per iteration with state \( u \in \mathbb{R}^n \) at next time step. \( n \) is the total number of unknowns in the state variable \( u \) representing the three-dimensional velocity field as well as the temperature distribution. Applying Newton’s algorithm, linear systems of type

\[
u^{i+1} = u^i + x^i \quad \text{with} \quad \frac{\partial F}{\partial u}(u^i) \cdot x^i = -F(u^i)
\]

need to be solved. In TinyFlow this is done iteratively and matrix-free with GMRES using the tangent model of AD to compute required Jacobian-vector products

\[
\frac{\partial F}{\partial u}(u^i) \cdot v \quad \text{for arbitrary } v.
\]

Symbolic treatment of the linear solver as shown in Section 4.1 for an overall adjoint requires the solution of linear systems of type

\[
\left( \frac{\partial F}{\partial u}(u^i) \right)^T \cdot b = x_{(1)}
\]

with incoming adjoints \( x_{(1)} \). Obviously, this system can be solved again matrix-free and iteratively with GMRES using the \textit{adjoint} model of \( F \) to compute transposed Jacobian-vector products

\[
\left( \frac{\partial F}{\partial u}(u^i) \right)^T \cdot v \quad \text{for arbitrary } v.
\]

Effects of different convergence bounds have been investigated under the author’s guidance in a Bachelor’s thesis.

References

○ T. Lajewski. \textit{Analysing the Coupling of Discrete and Continuous Adjoint for an Iterative Linear Solver}. Bachelor’s Thesis, RWTH Aachen. 2012

6.2 \texttt{dco/c++}

In this section, results concerning the plain application of \texttt{dco/c++} are shown. First, measurements from the AD test suite (see Section 3.4) are presented including a comparison of five different AD C++ overloading tools. And second, the efficiency of computing a full Hessian with higher-order modes of \texttt{dco/c++} is shown including also nested vector modes.

6.2.1 AD Test Suite Results

As described in Section 3.4, the AD test suite has been written to have a solid basis for statements about performance. This is required, since meaningful analysis of runtime benefits of applying hybrid approaches over plain AD relies on a good AD tool underneath. To get a reasonable picture of the performance of \texttt{dco/c++}, we compared its runtimes with those of four different AD tools. Before a brief description of the tools, we start with a disclaimer.
Disclaimer. No specialists of the other AD tools were involved in optimizing runtimes of their respective tool. Nonetheless, we, the authors of the AD test suite, tried to use the tools in the most efficient way to the best of our knowledge. The runtimes of dco/c++ turn out to be the best under this circumstance, but we are convinced that also the other tool developers are capable of optimizing there tools to get similar results. In addition, the results presented here are only a snapshot in time and are likely to change in the future.

The following five tools are included when running the test suite.

- **dco/c++**

  ADOL-C\(^4\) is certainly one of the first (if not the first) AD overloading tool for C++. It was initially developed by A. Griewank [GJU96] and later taken over by A. Walther [WG12]. It is rich in features including convenient drivers for computing derivative tensors in tangent and adjoint mode while exploiting sparsity [GPW08], optimal checkpointing of time integration [GW00], or computing piecewise linear functions [Gri13]. ADOL-C is open source under the Eclipse Public License or the GNU General Public License.

- **Sacado\(^5\)** is developed at the Sandia National Laboratories mainly by E. Phibbs [PP12]. It features tangent and adjoint modes also in higher-order and exploits the expression template mechanism in C++. Sacado is distributed via the Trilinos project (see website in footnote).

- **Adept\(^6\)** is developed by R. J. Hogan [Hog14]. It only features first-order derivatives, but is built upon very efficient data structures and also exploits the expression template mechanism in C++. The library is released under the GNU General Public License.

- **CppAD\(^7\)** is developed by B. M. Bell [Bel12] and features arbitrary-order derivatives. It is distributed under Eclipse Public License or the GNU General Public License.

The following set of test problems is currently included. Each test problem has a set of parameters \( \lambda \in \mathbb{R}^n \) with problem size \( n \) and a cost function \( f = f(\lambda) \) with \( f : \mathbb{R}^n \rightarrow \mathbb{R} \). For the measurements, one adjoint projection is computed yielding \( \frac{\partial f}{\partial \lambda}(\lambda) \).

- **1d-convection-diffusion equation (conv-diff)**
  The parameterized one-dimensional convection-diffusion equation
  \[
  \frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2} - u \frac{\partial u}{\partial x}
  \]  
  described in further detail in Section 4.4. Here, an explicit Euler method is used for temporal discretization. The cost functional \( f \) is defined in Equation (4.99a).

- **direct linear solver (gauss)**
  Core is the solution of a dense parameterized linear system with Gaussian elimination. The cost functional is chosen to be
  \[
  f(\lambda) = \|A(\lambda)^{-1}\lambda\|_2
  \]
with $\lambda \in \mathbb{R}^n$ and $A(\lambda) \in \mathbb{R}^{n \times n}$.

- **3D cross-frame field** (crossframe)
  This is a code coming from a method for constructing a 3D cross-frame field, a 3D extension of the 2D cross-frame field as applied to surfaces in applications such as quadrangulation and texture synthesis [Hua+11]. The code itself is a straight-line code with approximately 80 lines. It is generated by Maple\textsuperscript{TM} [Mon+05] with common subexpression optimization switched on. The code was kindly provided by D. Bommes from a test case of the software package CoMISo [BZ14; BZK12].

- **Lax-Wendroff** (LW) and Toon (Toon)
  Those two test problems are described in further detail in [Hog14], where they are already used as performance tests for AD overloading tools by R. Hogan, the author of Adept. Both solve the one-dimensional advection equation
  \[
  \frac{\partial u}{\partial t} = -u \frac{\partial u}{\partial x}
  \]  
  with state $u$, time $t$, spacial coordinate $x$, and the initial state $x_0$ as parameter. The equation is solved on an equidistant grid with either Lax-Wendroff [LW60] (linear scheme) or Toon [Too+88] (nonlinear scheme). The discretized cost functional is given as
  \[
  f(x_0) = \|x_e(x_0)\| \tag{6.15}
  \]
  with end state $x_e$.

Measurement were performed for above given test problems and the previously described AD tools. The figures show the runtime ratio $R$, which is the runtime of one gradient computation divided by the runtime of one passive function evaluation with data type double, i.e.
  \[
  R = \frac{\text{runtime for one adjoint projection}}{\text{runtime of function evaluation}}. \tag{6.16}
  \]
  The respective runtimes are the minimum out of 50 measurements performed on an Intel(R) Core(TM) i5-3337U with 4Gb of RAM. The runtime for computing the gradient is split into tape recording and tape interpretation times. All tests fit nicely into memory for avoiding swapping effects.

In Figure 6.5 timings are shown for all tools. The runtime ratios range from 3 (Adept and dco/c++ for Toon) to around 120 (ADOL-C for crossframe). The huge differences are probably due to those reasons:

- The efficiency of the respective tool.

- The tests have very different memory access patterns, the internal data structure of the tools on the other hand have fixed memory layouts. If the original program is very cache-efficient (i.e. data locality is good), the runtime ratio is worse than for a very inefficient program.

- In addition to that, the complexity of function evaluations required during the forward run and the reusability of their results for partial derivatives have an impact on the runtime ratio. $\sqrt{x}$ for example is expensive but its result is reusable for the partial derivative $0.5/\sqrt{x}$, i.e., if many square root operations are performed, the runtime ratio gets better.
Figure 6.5: Test suite results: Runtime ratio $R$ for all test problems and all tools.
6 Results and Measurements

The last point can be seen when comparing LW and Toon. Both are very cache-friendly algorithms as for each timestep they just loop twice over two big chunks of memory.

- LW
  \[
  \text{for } (\ldots) \quad f[i] = 0.5 \cdot c \cdot (q[i] + q[i+1] + c \cdot (q[i] - q[i+1]));
  \text{for } (\ldots) \quad q[i] += f[i-1] - f[i];
  \]

- Toon
  \[
  \text{for } (\ldots) \quad f[i] = (\exp(c \cdot \log(q[i]/q[i+1])) - 1.0) \\
  \quad \times q[i] \cdot q[i+1] / (q[i] - q[i+1]);
  \text{for } (\ldots) \quad q[i] += f[i-1] - f[i];
  \]

Nonetheless, the factors for both differ a lot, which gets clear when recognizing the \(\exp\) and \(\log\) functions in Toon. For \(n = 500\), e.g., the function evaluation runtimes are 0.017s and 0.2s for LW and Toon, respectively. The adjoint runtimes for dco/c++ on the other hand differ less with 0.13s and 0.41s. Those numbers directly translate to the factors shown in the figures. The hardest test seems to be crossframe. It has the specialty of being a generated code, which has already been optimized for as few operations as possible by Maple\textsuperscript{TM}. In general, Adept, CppAD and dco/c++ are the best performing tools for all tests. In Figure 6.6 only those tools are shown. In all tests, Adept and dco/c++ are by far the fastest tools and for two of the tests, dco/c++ is remarkably faster than Adept (ca. 45%). From those measurements one can conclude, that dco/c++ seems to be well-suited to approach efficient hybrid differentiation modes.

6.2.2 Performance of Hessian Computation

As shown in Section 3.2.3, four different second-order models exist in Algorithmic Differentiation and so they do in dco/c++. Each second-order model can be used to compute the Hessian \(H\)
of a twice continuously differentiable function

\[ f = f(\lambda) : \mathbb{R}^n \to \mathbb{R}. \quad (6.17) \]

The Hessian \( H = \frac{\partial^2 f}{\partial \lambda^2} \) is assumed to be dense in this section. Most of the results shown here are also valid for sparse Hessians when using compression techniques [GMP05]. When compressing the Hessian, the runtimes for a complete Hessian computation shown here scale with the respective compression ratio. The function \( f \) used for measurements in this section is again the implementation of the one-dimensional time-dependent convection-diffusion equation from Section 4.4. \( f \) is evaluated for \( n = 200, M = 5, \Delta t = 0.001, \lambda_i = 0.005 \forall i, \lambda g_i = 2.0 \forall i \) and \( u_i^m = 0 \forall i \).

In addition to the scalar second-order modes, vector modes are used if a performance gain is expected. This applies for second-order tangent mode (tangent-over-tangent) and yields a stacked vector mode \( \text{gtlv< gtlv< double, p:::type, p> \text{. But this also applies for the second-order adjoint modes forward-over-reverse and reverse-over-reverse. Those two yield the adjoint vector modes} \}

\[ \text{gals< gtlv< double, p:::type > and gals< galv< double, p:::type >,} \]

respectively. Vectorization of the first application of the adjoint mode is useless, since \( f \) is a scalar function.

We revisit the impact of the vector modes for a Hessian computation briefly. For a vector length \( p \in \mathbb{N}^+ \) and corresponding tangent vectors \( \lambda^{(1)}_k, \lambda^{(2)}_l \in \mathbb{R}^n \), the vectorized forward-over-forward tangent model is able to compute

\[ y^{(1,2)}_{k,l} = \frac{\partial^2 f}{\partial \lambda_i \partial \lambda_j} \cdot \lambda^{(1)}_k \cdot \lambda^{(2)}_l \quad \text{for} \quad k, l \in \{1, \ldots, p\} \quad (6.18) \]

with one overloaded function evaluation. The values \( y^{(1,2)}_{k,j} \) correspond to a \( p \times p \)-block of the Hessian. For tangent vectors \( \lambda^{(2)}_k \in \mathbb{R}^n \), the vectorized forward-over-reverse adjoint model is able to compute

\[ \lambda^{(2)}_{(1)k} = y^{(1)} \cdot \frac{\partial^2 f}{\partial \lambda^2} \cdot \lambda^{(2)}_k \quad \text{for} \quad k \in \{1, \ldots, p\} \quad (6.19) \]

with one overloaded function evaluation. The values \( \lambda^{(2)}_{(1)k} \) correspond to a \( p \times n \)-block of the Hessian. For adjoint vectors \( \lambda^{(1,2)}_k \in \mathbb{R}^n \), the vectorized reverse-over-reverse adjoint model is able to compute

\[ \lambda^{(2)}_k = y^{(1)} \cdot \frac{\partial^2 f}{\partial \lambda^2} \cdot \lambda^{(1,2)}_k \quad \text{for} \quad k \in \{1, \ldots, p\} \quad (6.20) \]

with one overloaded function evaluation. Similar to the other adjoint model, the values \( \lambda^{(2)}_k \) correspond to a \( p \times n \)-block of the Hessian. Both adjoint modes (forward-over-reverse and reverse-over-forward) compute exactly the same values, since the Hessian is symmetric.

Runtime (ratio) and memory (ratio) measurements for the different modes are shown and described in Table 6.3. The optimal vector sizes for the vector modes depend on the system architecture, the specific function implementation, the compiler (flags), and much more. For the setup we had, the optimal length for tangent-over-tangent mode was 5, whereas for the adjoint
6 Results and Measurements

<table>
<thead>
<tr>
<th>Mode</th>
<th>H · v</th>
<th>H</th>
<th>rss</th>
<th>rss ratio</th>
<th># eval f</th>
<th># interpr.</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtls-over-gtls</td>
<td>2.6</td>
<td>53k</td>
<td>6k</td>
<td>1</td>
<td>20k</td>
<td>—</td>
<td>496s</td>
</tr>
<tr>
<td>gt1v-over-gt1v</td>
<td>1.6</td>
<td>32k</td>
<td>13k</td>
<td>2</td>
<td>800</td>
<td>—</td>
<td>294s</td>
</tr>
<tr>
<td>gtls-over-gals</td>
<td>4.2</td>
<td>836</td>
<td>61k</td>
<td>9</td>
<td>200</td>
<td>200</td>
<td>7.8s</td>
</tr>
<tr>
<td>gt1v-over-gals</td>
<td>2.2</td>
<td>433</td>
<td>267k</td>
<td>40</td>
<td>20</td>
<td>20</td>
<td>4.0s</td>
</tr>
<tr>
<td>gals-over-gtls</td>
<td>10.8</td>
<td>2170</td>
<td>123k</td>
<td>19</td>
<td>200</td>
<td>200</td>
<td>20s</td>
</tr>
<tr>
<td>gals-over-gals</td>
<td>1.5</td>
<td>290</td>
<td>139k</td>
<td>21</td>
<td>1</td>
<td>200</td>
<td>2.7s</td>
</tr>
<tr>
<td>galv-over-gals</td>
<td>0.9</td>
<td>187</td>
<td>256k</td>
<td>39</td>
<td>1</td>
<td>20</td>
<td>1.7s</td>
</tr>
</tbody>
</table>

Table 6.3: Timings for Hessian computation using different second-order data types of dco/c++. The first column defines the used mode, the following two columns are runtime ratios to the runtime of a passive function evaluation for one Hessian-vector product (H · v) and the full Hessian computation (H), respectively. rss is the maximal resident set size (memory consumption) of the program and rss ratio the ratio to rss of the passive function evaluation. The number of calls to the function implementation is given in column # eval f, whereas the number of tape interpretations is given in column # interpr. The overall runtime for a full Hessian computation is listed in the last column. Measurements were performed on an Intel(R) Core(TM) i5-3337U.

modes it was 10. Chosen mode and vector size determine the number of function evaluations and number of tape interpretations required to compute the full Hessian.

Both second-order tangent modes are slow compared to the adjoint modes. Nevertheless, least memory consumption (rss) is required there. The most impressive insight can be gained for the adjoint-over-adjoint mode. Both (non-vectorized and vectorized) are faster than all the other modes with the drawback that memory consumption is high. The performance gain can be explained as follows. It can be seen in Figure 6.6 that for dco/c++ the interpretation step is always much quicker than the recording step. Since the reverse-over-reverse mode copes with the Hessian computation using only one recording step (# eval f) and many interpretation steps (# interpr.), see Section 3.2, this yields a better performance. The code for those measurements is available in results/src/hessian_computation.

6.3 Hybridization with dco/c++

This section first briefly summarizes the results from an article already published concerning first-order symbolic linear and nonlinear solvers [Nau+15]. Those results correspond to Sections 4.1 and 4.2. Afterwards, runtime and memory measurements for the case study Section 4.4 are shown, demonstrating the various benefits possible with the help of hybridization.

6.3.1 Symbolic Linear and Nonlinear Solvers

In this section the most important results from [Nau+15] are revisited briefly. The benchmarks in the article were performed for a code which solves a steady nonlinear one-dimensional parameterized partial differential equation. After discretization, the core and most complex part is the solution of one nonlinear system. The nonlinear system is solved with Newton’s method,
where the arising linear system is solved with a dense Gaussian elimination algorithm. In the article, we looked at three differentiation modes, namely the algorithmic, the symbolic LS, and the symbolic NLS mode. The terms correspond to the terms used in Section 4.4. The algorithmic mode is a plain application of \texttt{dco/c++} to the solution algorithm of the nonlinear system, the symbolic LS mode treats the linear system solver symbolically, and the symbolic NLS mode treats the nonlinear solver symbolically. In the article, the complexities for for runtime overhead and memory overhead are analyzed and verified with the measurements shown in Figure 6.7. The order of complexities for the solution of the nonlinear system is dominated by the dense linear system solve, i.e. $O(n^3)$. The overhead for computing adjoints fully algorithmically should be in the same order, but is in fact $O(n^{2.4})$. This is expected to converge to cubic behavior for larger problem sizes. The symbolic LS overhead is $O(n^{2.0})$ as expected for additional forward and backward substitutions. The symbolic NLS overhead is $O(n^{2.8})$ also as expected for one additional linear system solve $\sim O(n^3)$. Nonetheless, the absolute value of the symbolic NLS overhead has a much smaller complexity constant for both, runtime and memory, as can be seen in Figure 6.7.

6.3.2 Case Study

In the following, the performance for the various differentiation modes described in Section 4.4 is benchmarked. The first benchmark measures runtime and memory consumption for the computation of the gradient $\frac{\partial \Phi}{\partial \lambda}$, where $\Phi$ depends on parameters $\lambda, \lambda_g, u^0, u^m$, and the state variable $u$, which is the solution of the discretized one-dimensional time-dependent convection-diffusion equation (the PDE). To compute the adjoint of $\Phi$ (i.e. its gradient), the adjoint of the solution of the PDE is required. This is where the different symbolic approaches come into play: symbolic LS mode, symbolic NLS mode, and continuous mode. Symbolic LS mode treats the sparse direct linear system solve symbolically, the symbolic NLS mode treats the nonlinear system solve symbolically and the continuous mode solves the continuous adjoint equations. The
second benchmark measures runtime and memory consumption for one adjoint projection

\[\left( \frac{\partial \lambda^*}{\partial (\lambda_g, u^0, u^m)} \right)^T \cdot 1 \quad \text{where} \quad \lambda^* = \arg \min_{\lambda} \Phi(\lambda, \lambda_g, u^0, u^m).\]  

(6.21)

The minimization algorithm is using Newton’s method with a matrix-free conjugate gradient method for solving the arising linear system. Two different symbolic approaches are analyzed: symbolic CG mode and symbolic FIT mode. The former treats the linear system solve symbolically, while the latter differentiates the first-order optimality condition symbolically. The same definition as the one in the previous section is used for the runtime ratio

\[R = \frac{\text{runtime for one adjoint projection}}{\text{runtime of function evaluation}}.\]  

(6.22)

The runtime ratio measurements and the corresponding memory requirements for the first benchmark are shown and described in Figure 6.8. In comparison to the results from Section 6.3.1 and [Nau+15], the runtime of the symbolic LS mode is worse than the fully algorithmic one. This is due to the different characteristics of the linear solvers and their system matrices. In the code from [Nau+15], a dense linear solver is used to solve a system with a densely computed Jacobian. In this case, on the other hand, a sparse linear solver is used to solve with a sparsely computed Jacobian as explained in more detail in Section 4.4.4. The complexity of computing the Jacobian is therefore linear for the former case \(O(n)\cdot\text{Cost}(r)\) with the residual \(r\) as opposed to constant for the latter \(O(3)\cdot\text{Cost}(r)\), since only 3 tangent projections are required). Therefore, the overhead for copying data and managing the external adjoint function properly outweighs the expected runtime benefits for symbolic LS. But it seems that the runtimes for symbolic LS and algorithmic mode get closer for larger problem sizes. Nonetheless, avoiding the taping of the Jacobian computation, its decomposition and solving the system saves a lot of memory.
6.3 Hybridization with dco/c++

### Problem Size

<table>
<thead>
<tr>
<th>No diff. + Symbolic NLS</th>
<th>10</th>
<th>30</th>
<th>100</th>
<th>1100</th>
<th>8000</th>
<th>80000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic</td>
<td>4s</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symbolic NLS</td>
<td>2.3s</td>
<td>5.8s</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symbolic CG</td>
<td>3.3s</td>
<td>9.6s</td>
<td>30s</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>+ Symbolic NLS</td>
<td>2.3s</td>
<td>4.0s</td>
<td>9.6s</td>
<td>90s</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symbolic FIT</td>
<td>0.4s</td>
<td>1.2s</td>
<td>4.2s</td>
<td>46s</td>
<td>306s</td>
<td>—</td>
</tr>
<tr>
<td>+ Symbolic NLS</td>
<td>0.4s</td>
<td>0.6s</td>
<td>1.1s</td>
<td>9.0s</td>
<td>79s</td>
<td>712s</td>
</tr>
</tbody>
</table>

### Memory Consumption

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>10</th>
<th>30</th>
<th>100</th>
<th>1100</th>
<th>8000</th>
<th>80000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No diff. + Symbolic NLS</td>
<td>9MB</td>
<td>9MB</td>
<td>9MB</td>
<td>11MB</td>
<td>43MB</td>
<td>396MB</td>
</tr>
<tr>
<td>Algorithmic</td>
<td>5GB</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>+ Symbolic NLS &amp; LS</td>
<td>1.8GB</td>
<td>5.7GB</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symbolic CG</td>
<td>0.7GB</td>
<td>1.8GB</td>
<td>5.5GB</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>+ Symbolic NLS</td>
<td>0.3GB</td>
<td>0.4GB</td>
<td>0.7GB</td>
<td>5.3GB</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symbolic FIT</td>
<td>12MB</td>
<td>18MB</td>
<td>47MB</td>
<td>0.6GB</td>
<td>4.3GB</td>
<td>—</td>
</tr>
<tr>
<td>+ Symbolic NLS</td>
<td>12MB</td>
<td>12MB</td>
<td>12MB</td>
<td>15MB</td>
<td>49MB</td>
<td>450MB</td>
</tr>
</tbody>
</table>

Table 6.4: Measurements for the runtime and memory consumption over different problem sizes for computing the adjoint projection Equation (6.21). Memory consumption is measured as maximal resident set size. The base memory footprint of the binary is 12MB.

The symbolic NLS mode and the continuous mode clearly have the same complexities in terms of runtime as well as memory consumption. The symbolic NLS mode has a small taping overhead for the reversal of the timestepper (compare Figure 4.5 and Figure 4.6). The base memory footprint of the binary is 5.5MB, which is why the curves for symbolic NLS and continuous modes do not increase up to a certain problem size.

With respect to runtime and memory consumption, symbolic NLS or continuous modes are the way to go. But as already discussed earlier, both modes do not compute exact derivatives in terms of what the algorithm actually computes. A lot of research is done in this field concerning the discrete and the continuous adjoint approach, see e.g. [NJ01; PZG05; OD09]. It is therefore very valuable for verification reasons to have multiple differentiation modes available. The symbolic LS mode seems to best fit this purpose especially concerning the memory consumption.

Runtime and memory consumption measurements for the second benchmark are shown in Table 6.4. Here it is possible to combine the distinct differentiation modes of the optimization algorithm with the various differentiation modes of the PDE solution algorithm. We have therefore not only the algorithmic, the symbolic CG, and the symbolic FIT modes, but also combinations of those with the PDE-level modes. In the table only the combinations with algorithmic and symbolic NLS modes are shown because they already illustrate the possible performance gain. In each row, step by step, the memory consumption decreases, the more symbolic information is added to the adjoint computation. Simultaneously also the runtime
6 Results and Measurements

Figure 6.9: Annotated call tree (see Definition 18) for an artificial and simple example. Recording and checkpoint sizes are given in the figure; \( f \) calls \( g \) twice and \( h \) once. The corresponding costs are \( c[f] = 200 \), \( c[h] = 120 \), \( c[i] = 30 \), and \( c[g] = 40 \). The cost \( c[h] \) is chosen to be much higher than its corresponding memory requirement to mimic some time consuming operation in \( h \) like e.g. file I/O, which has no effect on the recording size.

<table>
<thead>
<tr>
<th>( y[...]=1 )</th>
<th>MIP, ( \hat{c} = \check{c} = 0 )</th>
<th>MIP, ( \hat{c} = 50 ), ( \check{c} = 0 )</th>
<th>MIP, ( \hat{c} = 0 ), ( \check{c} = 50 )</th>
<th>LMI</th>
<th>LMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>95</td>
<td>125</td>
<td>90</td>
<td>85</td>
<td>125</td>
</tr>
<tr>
<td>Overhead</td>
<td>35%</td>
<td>40%</td>
<td>60%</td>
<td>60%</td>
<td>40%</td>
</tr>
<tr>
<td>( c(y) )</td>
<td>70</td>
<td>130</td>
<td>170</td>
<td>120</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 6.5: Solution for the call tree from Figure 6.9 for different penalization parameters \( \hat{c} \) and \( \check{c} \) for the MIP solver (first three columns) as well as for the heuristics (last two columns).

decreases. As a base case, memory and runtimes are given for the undifferentiated fitting algorithm using symbolic NLS for computing gradient and Hessian projections. Compared to that base case, runtime and memory overhead of the symbolic FIT + symbolic NLS is fairly small. When using all symbolic modes available, \( \text{dco/c++} \) is mainly responsible for 1. the correct routing of adjoint values and 2. derivative computations of the residual function. The more manual effort is put into specific routines, the better the performance of the adjoint gets – without loosing the flexibility of an AD solution.

The code for those measurements is available in \( \text{results/src/case\_study} \).

6.4 Call Tree Reversal

In this section we want to present results for the Call Tree Reversal problem (CTR) from Chapter 5 on two very different examples. The first one is an artificial, very simple call tree, which we use to demonstrate the behavior of the algorithms and the influence of the various cost functions. The other call tree originates from a bigger case study based on the scientific computing library PETSc\(^8\).

The first example is shown in Figure 6.9. For an upper bound on the available memory of \( M = 125 \), we get the results shown in Table 6.5. First, we can observe that with the simple cost function \( c_s(\mathbf{y}) \) (second column), the optimal solution of this example \( (\mathbf{y}[i] = \mathbf{y}[g_0] = 1) \) is found by the MIP algorithm. The additional memory requirement is 95, the runtime overhead 35\%.

\(^8\)http://www.mcs.anl.gov/petsc
and the value of the cost function amounts to 70 \(= \tilde{c}[i] + \tilde{c}[g_0]\). This solution is better than the solutions found by the heuristics (LMI and LMD). The other observation is, that the choice for the parameters \(\hat{c}\) and \(\hat{c}\) yields the exact influence we wanted to achieve. When penalizing each checkpoint of a routine not already checkpointed before \((\hat{c} = 50)\), we observe that the MIP solver picks \(y[g_0] = y[g_1] = 1\) to avoid the penalization for \(y[i] = 1\). This yields a gap to the optimal runtime overhead of 5%. When penalizing every additional checkpoint \((\tilde{c} = 50)\), the MIP solver picks \(y[h] = 1\), which is a better solution because only one routine is checkpointed. On the other hand, this yields an additional runtime overhead of 25%. It is clear, that the heuristics shown in Section 5.2 and proposed by [Nau12] (LMI and LMD) can not take the more complex cost functional into consideration. This small example already shows the higher flexibility of the MIP formulation.

To demonstrate applicability to more realistic codes, we build upon the work done in [Lot+13]. Therefore, the second call tree we want to look at is generated by the solution of the Bratu equation from the MINPACK-2 test suite [ACM91] implemented in PETSc.

Let’s first look again onto the overall framework already described in Chapter 5:

\[
\begin{array}{cccc}
\mathcal{P} & (I) & \mathcal{P} & (II) \\
& (III) & \mathcal{Y} & (IV) \\
& & \mathcal{P} & (V) \\
\end{array}
\]

To retrieve the call tree, we first needed to apply the source-to-source compiler (step (I)) to the complete code base of PETSc. This is required for the instrumentation of the source code, see also Appendix A.2 for implementation details of the source-to-source compiler. A few statistics are shown in Table 6.6. PETSc v3.3 consists of 1362 source files (.c) and 147 header files (.h). The compiler needed to parse a total number of 557 files – the part of PETSc that was used by the example. Changes were applied to 373 .c files and one .h file. This can be explained by the fact, that most header files only contain declarations and therefore don’t need any instrumentation. We can observe in total a number of 2717 instrumented functions. To keep this number as small as possible (see Section 5.4), we excluded, e.g., all functions from namespace std.

Table 6.7 summarizes the information about the call trees of the second benchmark for varying problem sizes (from 10 up to 250). It is important to keep in mind that the call tree we are looking at is the dynamic call tree (generated at runtime) opposed to the static one (compile time). The number of routine calls is therefore growing for increasing problem sizes. Similarly, the required tape size for the adjoint is of course growing: from 3.29MB for a problem size of 10 to 94GB for a problem size of 250. The runtime on the other hand is increasing less quickly especially for the biggest case. This seems a bit odd, but the test was repeated a couple of times with same results. When looking at the remaining routines after applying a memory
6 Results and Measurements

<table>
<thead>
<tr>
<th>problem size</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>routine calls</td>
<td>7700</td>
<td>10k</td>
<td>31k</td>
<td>98k</td>
<td>443k</td>
</tr>
<tr>
<td>tape size</td>
<td>3.29M</td>
<td>26.19M</td>
<td>500M</td>
<td>5G</td>
<td>94G</td>
</tr>
<tr>
<td>runtime</td>
<td>0.1</td>
<td>0.14</td>
<td>2.1</td>
<td>20.1</td>
<td>40.0</td>
</tr>
<tr>
<td>memory threshold</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>remaining routines</td>
<td>299</td>
<td>11</td>
<td>464</td>
<td>901</td>
<td>2103</td>
</tr>
</tbody>
</table>

Table 6.7: Statistics for the second benchmark: Routine calls is the number of nodes in the call tree, tape size the recording size in byte for \( y \equiv 0 \) (i.e. no checkpointing), the memory threshold is the cut-off value in Mbyte described in Section 5.4, runtime is the wallclock time in seconds required to compute the solution of SFI, and remaining routines is the remaining number of nodes in the call tree.

<table>
<thead>
<tr>
<th>Memory Bound</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>200</th>
<th>250</th>
<th>1000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMI</td>
<td>1.3</td>
<td>2.0</td>
<td>2.4</td>
<td>2.4</td>
<td>5.2</td>
<td>10.5</td>
<td>17.5</td>
<td>17.9</td>
</tr>
<tr>
<td>LMD</td>
<td>1.1</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
<td>12.9</td>
<td>7.4</td>
<td>0.22</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Table 6.8: Runtime comparison for the equivalent heuristics Smallest-Memory-Increase-First (SMI) and Largest-Memory-Decrease-First (LMD) for problem sizes \( n = 50 \) and \( n = 100 \). Values in seconds.

threshold, we can conclude that a huge amount of called routines (ca. 95%) are irrelevant for the tape’s memory consumption. From now on, we therefore apply the memory threshold for all measurements.

In the following, we compare the different heuristics and different Mixed-Integer Programming (MIP) solvers for the second benchmark. As described in Section 5.2, the Smallest-Memory-Increase-First (SMI) heuristic finds the same solution as the Largest-Memory-Decrease-First (LMD) heuristic. The claim was that LMD usually performs better in terms of runtime. A comparison of the two heuristics for the second benchmark is shown in Table 6.8, and in fact, LMD is almost always the better choice. Solely for the bigger case \( (n = 100) \) and a very low memory bound \( (200) \) SMI finds the solution faster. In Table 6.9 we look at the runtime of the commercial MIP solvers GUROBI and CPLEX. Tests were performed for two different relative tolerance on the gap between the optimal objective and the objective of the current best integer solution. For many cases both solvers perform very similar. The exceptional cases are marked underlined blue in the table. In all cases, where a notable difference can be observed, CPLEX performs better. This is runtime-wise for memory bounds of 50 and 100 and in terms of solution quality especially for a memory bound of 200. We therefore use only CPLEX for further measurements.

The following three tables show the quality of the solutions found by the two heuristics and the MIP solver. The largest case with \( n = 250 \) is chosen for measurements as it is the most complex call tree. Performance and flexibility of the MIP formulation is demonstrated for the three different cost functions introduced in Section 5.4. The simplest cost function only takes the runtime overhead into account (no penalization terms for required checkpoints). The solutions found by LMI, LMD, and CPLEX for varying memory bounds is shown in Figure 6.10. For all cases the solution found by CPLEX is either much better or equal to the LMD solution.
### 6.4 Call Tree Reversal

<table>
<thead>
<tr>
<th>Solver \ Memory Bound</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUROBI tol = 0.0</td>
<td>20.5s</td>
<td>7.7s</td>
<td>3.1s</td>
<td>3.0s</td>
</tr>
<tr>
<td></td>
<td>55.1%</td>
<td>36.8%</td>
<td>20.2%</td>
<td>5.3%</td>
</tr>
<tr>
<td>GUROBI tol = 0.5</td>
<td>2.7s</td>
<td>2.8s</td>
<td>2.9s</td>
<td>2.8s</td>
</tr>
<tr>
<td></td>
<td>67.8%</td>
<td>49.5%</td>
<td>31.0%</td>
<td>5.3%</td>
</tr>
<tr>
<td>CPLEX tol = 0.0</td>
<td>12.2s</td>
<td>4.1s</td>
<td>3.0s</td>
<td>2.8s</td>
</tr>
<tr>
<td></td>
<td>55.1%</td>
<td>36.8%</td>
<td>20.2%</td>
<td>5.3%</td>
</tr>
<tr>
<td>CPLEX tol = 0.5</td>
<td>2.7s</td>
<td>2.8s</td>
<td>2.7s</td>
<td>2.7s</td>
</tr>
<tr>
<td></td>
<td>67.3%</td>
<td>52.0%</td>
<td>25.0%</td>
<td>5.3%</td>
</tr>
</tbody>
</table>

Table 6.9: Runtime comparison for MIP solvers: GUROBI vs. CPLEX for problem size $n = 50$. Particular variations are marked. $tol$ is the relative tolerance on the gap between the optimal objective and the objective of the current best integer solution. 0.5 means 50%.

LMI is far from being competitive with runtime overheads more than three times worse. The total number of checkpointed routines is shown in Figure 6.11 for the solutions of LMI, LMD, CPLEX with the simple cost functional, and CPLEX with the cost function that takes the penalization for each additionally checkpointed routine into consideration (CPLEXp), i.e. $\hat{c} > 0$ in Equation (5.12). Interestingly, LMD requires far less checkpoints than CPLEX for a memory bound of $\geq 4000$. This is of course only by chance, since this information is not influencing LMD. Nonetheless, CPLEXp always finds the feasible solution with the fewest checkpoints required. This is especially true for the memory bounds 4000 (489 vs. 210) and 64000 (8 vs. 1). The next comparison is done for the cost function that also penalizes each checkpoint of a subroutine not already checkpointed at least once, i.e. $\hat{c} > 0$ in Equation (5.11). In addition to the LMI, LMD, and CPLEX, the MIP formulation was used to take the penalization for newly checkpointed routines into account (CPLEXi). As one can see in Figure 6.12, CPLEXi always finds a solution with less distinct routines checkpointed.

One can conclude that the solution quality of the Largest-Memory-Decrease-First (or Smallest-Memory-Increase-First) heuristic is for many cases quite good for this example. The number of checkpointed routines as well as the runtime overhead of the solution found by LMD is getting worse for very tough memory bounds. LMD is still by far better in all cases than its sibling...
6 Results and Measurements

Figure 6.11: The total number of checkpointed subroutines, i.e. $|y|$, for the different solution methods: Largest-Memory-Decrease-First (LMD), Largest-Memory-Increase-First (LMI) and via the MIP formulation solved with CPLEX. The MIP formulation is solved for two different cost functions: the standard one (CPLEX) and the one penalizing each checkpoint (CPLEXp). Please note the logarithmic scale.

Figure 6.12: The number of checkpointed distinct subroutines, i.e. $\hat{|y}|$, for the different solution methods: Largest-Memory-Decrease-First (LMD), Largest-Memory-Increase-First (LMI) and via the MIP formulation solved with CPLEX. The MIP formulation is solved for two different cost functions: the standard one (CPLEX) and the one penalizing each checkpoint of a subroutine not checkpointed at least once before (CPLEXi).

Largest-Memory-Increase-First heuristic. Nonetheless, the flexibility is much greater when using the MIP formulation in conjunction with CPLEX. Solely the availability of the optimal solutions computed by CPLEX makes a well-founded assessment of the heuristics achievable. Besides further tests concerning the runtime of the heuristics and the MIP solvers, future work should now focus on options for implementing checkpointing schemes as automatically as possible.
7 Conclusion and Outlook

The software development process for adjoint codes is a challenging task. The intention of this thesis was to ease this task by contributing the software dco/c++ including the methodology of combining algorithmic and symbolic approaches to differentiation.

Conclusion Since Chapter 2 gathers already known theory about Algorithmic Differentiation, the implementation details Chapter 3 including the underlying software dco/c++ are mostly novel contribution. Especially the high-level intrinsic interface presented in Section 3.3 as the implementation of the hybridization introduced in Section 2.2 turned out to be a crucial basis for the later results. In addition, the higher-order capabilities shown in Section 3.2.3 have led to interesting results in Section 6.2.2, in particular the second-order adjoint-over-adjoint mode proved to be reasonable for specific cases.

Chapter 4 presents already known as well as novel results from the viewpoint of hybridization based on Algorithmic Differentiation. The arbitrary-order projections of the linear solver including respective complexity estimations is presented for the first time. Indeed are the derivations for the nonlinear solver on the other hand established knowledge. Nonetheless, the mixed differentiation modes for the linear solvers in special contexts Section 4.3 revealed new possible differentiation routes and can be seen as examples how to approach a nested structure of numerical methods when differentiating symbolically. Apart from that, especially the presentation of the extensive case study including the various differentiation paths and the respective implementation with dco/c++ is possibly a quite valuable result.

The derivation of the formulation of the Call Tree Reversal Problem in Chapter 5 as an integer programming problem is a fully novel contribution based on the problem definition stated previously. The integer problem formulation is a much more general approach than the originally proposed heuristics and it turned out to be more flexible and still applicable to reasonable sized real-world problems as shown in Section 6.4.

Outlook Continued work could definitely be carried out on the one hand concerning the tool development and on the other hand concerning the derivation of efficient tangent and adjoint projections for further common numerical algorithms.

The overloading technique seems to be quite promising, especially when taking the capabilities of high-level intrinsics into account. The growing set of features the new C++ standard brings up is likely to promote overloading tools even further. For example the new type deduction mechanisms (auto keyword) possibly enable an automatic extension of the assignment-level preaccumulation to basic blocks. Indeed, this could shift the AD by overloading performance more towards that of AD by source transformation. In addition, tighter interlacing of AD by overloading and AD by source transformation is to be examined. This could be a more automatic coupling of different tools. Another interesting approach that tends into the same direction is an adjoint source generation realized by an overloading tool, i.e. a generic dump of the tape built up during the overloaded execution. With an automatic compilation and relinking procedure, this
could by very beneficial, especially for codes running on more complex architectures like GPUs. First steps in a Master’s thesis [Sch15b] already show reasonable performance enhancements. Another promising approach is to make use of plugin capabilities of C++ compilers (like Clang or gcc) to perform preprocessing steps. This could include an enhanced instrumentation for being able to better analyze the code and pinpoint performance problems related to (in our case) dco/c++.

Apart from further tool developments, additional common numerical algorithms such as integration or interpolation methods should be analyzed from the viewpoint of efficient differentiation. Similar to the article “Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation” by M. Giles [Gil08], an extended set of tangent and adjoint algorithms should be developed that potentially could be part of a collection *adjoint numerical recipes*; referring to “Numerical Recipes” by W. Press [Pre07]. To be more specifically, the next steps should be the development of higher-order tangent and adjoint projections of the nonlinear solver. Results in this direction are partially already done and even published [SLN15].

With respect to the integer programming formulation of the Call Tree Reversal Problem, extensions are possible on the practical as well as on the theoretical side. Practical aspects are the gathering of the call tree information as well as the actual implementation of the resulting checkpointing scheme. Both steps could be enhanced by further developing code instrumentation capabilities. In particular the programming effort for implementing the resulting checkpointing scheme should be automated as much as possible. On the theoretical side, the integer formulation could be extended to also include possible output checkpoints already proposed by [Nau12, Chapter 2.3]. In addition to that, the Call Tree Reversal Problem itself should possibly be extended to also include checkpointing schemes like revolve [GW00] suggests.
A Appendix

A.1 Attached Code

Various source codes are attached to this thesis. The corresponding package can be retrieved from the author by email. Please extract archive and see the README files in the respective subfolders. All codes are only tested under Linux.

Included:

- **attached_code/dco_cpp**: empty folder
  Put dco.hpp here. Please write an email to the author to retrieve the header.

- **attached_code/dco_cpp_examples**: examples for dco/c++
  All examples from the introduction to dco/c++ in Section 3.2 and 3.3.

- **attached_code/call_tree_reversal**:
  Main solver framework (see Section 5) as well as example data for the results shown in Section 6.4. Either Cbc\(^1\) can be used, or a license for CPLEX\(^2\) is required. For more information, please contact the author.

- **attached_code/case_study**:
  Source code of the PDE and the dco/c++ hybridization from Section 4.4.

- **attached_code/hessian_computation**:
  Source code for the second-order data types performance test from Section 6.2.2.

---

\(^1\)https://projects.coin-or.org/Cbc

A Appendix

A.2 Call Tree Reversal Implementation Details

In this section, a few further details concerning the implementation of the Call Tree Reversal solver framework as well as the source-to-source compiler for automatic code instrumentation are given. This is closely related to Chapter 5.

A.2.1 Mixed Integer Solver Framework

We decided to define the problem using CMPL\(^3\). CMPL is a mathematical programming language and an open-source alternative to commercial ones, e.g. GAMS\(^4\), but nevertheless provides an intuitive language to define the problem. Additionally, it is possible to solve the resulting equations with different solvers, e.g. CPLEX\(^5\) or GUROBI\(^6\). For our problem, we observed best runtimes with CPLEX, see Section 6.4. Thanks to the IBM Academic Initiative\(^7\) and the Academic Licenses of GUROBI\(^8\), we could use both solvers at no charge. CMPL comes with a python interface fitting quite well into our pre- and post-processing done in python anyway. The global setup is shown and described in Figure A.1. While the implementation of the python driver is straightforward, we want to focus on the problem definition in CMPL. In Listings A.1, A.2, A.3, and A.4 we show examples for the definition of data filled from the python driver, the definition of the binary variable \(y\), the definition of the objective function and an example for a constraint. Those listings are mostly self-explanatory. For making it easier for the reader to compare those listings with the mathematical equations from Section 5.3, we rewrite two of

---

\(^3\)Coinp|Coin| Mathematical Programming Language – https://projects.coin-or.org/Cmpl
\(^4\)General Algebraic Modeling System: http://www.gams.com/
\(^6\)Gurobi Optimizer: http://www.gurobi.com/
\(^7\)http://www-304.ibm.com/ibm/university/academic/pub/page/teaching_topics
\(^8\)http://www.gurobi.com/products/licensing-and-pricing/academic-licensing
those equations here:

\[
c(y) = \sum_{s \in S} y[s] \cdot c[s] + \hat{c} \cdot \sum_{s \in \hat{S}} \hat{y}[s] + \hat{c} \cdot \sum_{s \in \hat{S}} y[s] ,
\]
\[
M^f[s] = m[s] + \sum_{\sigma \in \chi[s]} (1 - y[\sigma]) \cdot M^f[\sigma] + y[\sigma] \cdot M^c[\sigma] .
\]

Listing A.1: Parameter declaration in CMPL. MemoryBound is the upper bound on the available memory \( M \), IPenalize is the penalization constant \( \hat{c} \), and CPPenalize is the penalization constant \( \tilde{c} \). Subroutines set is the declaration for the set of all subroutines \( S \).

Listing A.2: Definition of a variable in CMPL. \( y \) is a binary variable defined over the set of all subroutines Subroutines and our main variable to solve for.

Listing A.3: Definition of the objective function in CMPL. This corresponds to Equation (A.1). CMPL supports vector operations like the scalar product \( y[]^T \ast \text{CostWeight}[] \).

Listing A.4: Definition of a constraint in CMPL. This corresponds to Equation (A.2). Edges is a set of two-tuples defining the arcs in the call tree and therefore corresponds to \( \chi \). CMPL supports \( \forall \) constructs \( (s \in \text{Subroutines}) \) as well as the \( \sum \) operation.

A.2.2 Retrieving Call Tree Information: Clang LibTooling and dco/c++

For retrieving the call tree information for a specific instance of a program, we follow a two step procedure. First, we apply a source-to-source transformation to the original program \( P \). This is done by making use of the so-called LibTooling capabilities of Clang. The resulting program \( \tilde{P} \) requires an additional C++ header-only library (which is part of dco/c++) and can be compiled the same way as \( P \) before, i.e. with the same compilers, compiler flags and build system. This will generate an executable that computes not only the original function values, but also gathers call tree information and dumps it to a csv (comma separated values) file. The slowdown in runtime highly depends on the strategy of reducing the problem size (see Section

\[9\]http://clang.llvm.org/docs/LibTooling.html
A Appendix

5.4. The tooling support in Clang is very straightforward. The tool itself uses the Clang library and is written in C++. The core is shown and described in Listing A.5. The description of the instrumentation library in dco/c++ is beyond the scope of this thesis.

Listing A.5: LibTooling in Clang. The core functionality lives in the instance of RefactoringTool, which is initialized with the compilation database and the file to work on. Among others, the former contains the information about include paths and compilation flags. The ast_matchers::finder is Clang’s setup to pinpoint the code fragments, where the user’s callback (here: FcnCallback) takes over control. The most interesting part is the so-called matcher, defined in lines 14-22. It is self-explanatory – we get a callback, whenever there is a function declaration found in the AST (abstract syntax tree), which is – in C++ terms – a definition, not implicit, not a template instantiation (but of course explicit template specializations), not extern C and not already instrumented before.

An example how the tool works is given in Listing A.6 and Listing A.7. For each file and each function definition that meets the constraints given in the description of Listing A.5, it now introduces, first, the dco/c++ header required for instrumentation and, second, in each function implementation at the very beginning an instance of the instrumentation object dco::instrumentation_data. This instance gets created whenever entering this function, and gets destroyed when leaving, thus capable of collecting information about this specific function.

Listing A.6: The original program can contain any C++ conform language element Clang supports; for examples function templates, a template function specialization, and a class member function.

```cpp
#include "dco.hpp"
template <typename T> void f(T *x) {
  do sth.
}
template <>
void f(double *x) { do sth.
   }
class C {
  void f(double k) { do sth.
   };
}

118
A.2 Call Tree Reversal Implementation Details

dco::instrument::instrumentation_data
dco_instrumentation_data("void f( T *, )");
// do sth.
}
template <> void f<double>(double *x) {
  dco::instrument::instrumentation_data
  dco_instrumentation_data("void f( double *, )");
  // do sth.
}
class C { void f(double k) {
  dco::instrument::instrumentation_data
  dco_instrumentation_data("void f( double, )");
  // do sth
  }
};

Listing A.7: The transformed program.
Bibliography


Bibliography


[Gil08] M. Giles. “Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation”. In: [Bis+08], 2008, pp. 35–44.


[HNP05] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. ““To be recorded” Analysis in Reverse-Mode Automatic Differentiation”. In: Future Generation Computer Systems 21.8 (2005), pp. 1401–1417.
Bibliography


Bibliography


