Discrete Adjoints: Theoretical Analysis, Efficient Computation, and Applications

HABILITATION

zur Erlangung des akademischen Grades

Doctor rerum naturalium habilitatis
(Dr. rer. nat. habil.)

vorgelegt

der Fakultät Mathematik und Naturwissenschaften
der Technischen Universität Dresden

von

Jun.Prof. Dr. Andrea Walther

geboren am 9. September 1970 in Bremerhaven

Gutachter:	Prof. Dr. habil. Hubert Schwetlick, TU Dresden
	Prof. Christian H. Bischof, Ph.D., RWTH Aachen
	Prof. Lorenz T. Biegler, Ph.D., Carnegie Mellon University

Eingereicht am: 16. Mai 2007

My attention was drawn to this research area by Prof. Andreas Griewank. I would like to thank him for his inspiration as well as for many fruitful and lively discussions. Furthermore, I thank Prof. Hubert Schwetlick for his support during the preparation of this thesis. Special thanks are due to my husband for his patience, understanding and encouragement during times of doubt. Furthermore I am grateful to my parents for the lifelong support and help they gave me. Moreover I wish to thank all my colleagues at the Institute of Scientific Computing, TU Dresden, for the pleasant working atmosphere.

Contents

Chapter 1

Discrete Adjoints: Analysis, Computation, Applications

1.1 Introduction

The computation of derivatives forms an important ingredient for a wide range of applications. For example, essentially all calculus-based optimization algorithms employ at least derivative information of first order. Even second order derivatives are often approximated or computed exactly to accelerate and/or stabilize the optimization process. However, there are numerous additional areas, where the usage of derivatives is required. This includes the solution of nonlinear equations, the integration of ordinary differential equations (ODEs) using implicit methods, real-time optimization and sensitivity analysis. Derivatives of very high order are required for the integration of ODEs and differential-algebraic equations (DAEs) when applying Taylor methods. For the calculation of derivative information, there are two important aspects that have to be taken into account:

- The required accuracy of the derivatives.
- The computational complexity in terms of runtime and memory required for the derivative computation.

To calculate the needed derivatives, one may employ either analytical derivatives, finite differences, or automatic differentiation. These three approaches will be described briefly and analyzed with respect to the two aspects mentioned above in the following paragraphs.

Analytical Derivatives If the function the derivative of which has to be computed is given in a closed form, i.e., as an explicit formula, the basic rules of differentiation, as for example the product rule and the quotient rule, can be used to derive an explicit formula also for the corresponding first order derivatives. This can be done either by hand or one may apply computer algebra tools as for example MAPLE to derive an analytic representation of the derivatives. The first alternative results for complicated and/or large expressions in an error-prone and time-consuming process, whereas the second approach yields for smaller applications quite reasonable results.

Analytic derivatives are not only available for functions given as a closed expression but also for applications based on ordinary or partial differential equations. To illustrate this fact for ODEs, we introduce the following optimal control problem

$$\begin{aligned} \text{Minimize} \quad & J(y) = \varphi(y(t_f)) \\ \text{s.t.} \quad & \frac{dy}{dt}(t) = f(y(t),u(t)) \quad t \in [0,t_f], \qquad y(0) = y^0, \end{aligned} \tag{1.1}$$

where $y(t) \in \mathbb{R}^n$ denotes the state and $u(t) \in \mathbb{R}^m$ denotes the control. The dynamics are given by the right-hand side function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$. The initial state is described by the vector $y^0 \in \mathbb{R}^n$. To compute the value of the objective function, $\varphi : \mathbb{R}^n \to \mathbb{R}$ evaluates the state at the final time t_f, but other objective functions, for example of tracking type, are possible here. For simplicity, we assume that all functions are sufficiently smooth such that the existence of a solution is ensured and all necessary differentiations can be performed. As can be seen, the state y is determined by the control u. Hence, let $y = \psi(u)$ denote the solution of the state equation (1.1) for a given control u. Then, one can reformulate the objective function such that it depends only on u, i.e., we have $\tilde{J}(u) = J(y(u))$. There exists two distinct ways for calculating derivative information of the objective function $J(u)$. The first one determines for a given control u the sensitivity $s(t) \in \mathbb{R}^n$ in the direction of $d(t) \in \mathbb{R}^m$ using the sensitivity equation

$$\frac{ds}{dt}(t) = f_y(\psi(u)(t),u(t))\, s(t) + f_u(\psi(u)(t),u(t))\, d(t), \qquad s(0) = 0.$$

One obtains immediately the *forward* or *sensitivity* representation of the derivative of $\tilde{J}$ in direction d that is given by

$$D\tilde{J}(u)d = \nabla\varphi(\psi(u)(t_f))\, s(t_f)\,.$$

Alternatively, one may employ the adjoint differential equation

$$-\frac{d\lambda}{dt}(t) = f_y(\psi(u)(t),u(t))^T\, \lambda(t), \qquad \lambda(t_f) = \nabla\varphi(\psi(u)(t_f))^T.$$

The solution $\lambda(t) \in \mathbb{R}^n$ of this linear ODE is called the adjoint variable and yields the gradient's *backward* or *adjoint* representation [16, Section 2.4]

$$(D\tilde{J}(u)(t))^T = f_u(\psi(u)(t),u(t))^T\, \lambda(t) \in \mathbb{R}^{m\times 1}.$$

It follows that one has to integrate the adjoint equation in *backward* direction from t_f to t for calculating the gradient at time t. Furthermore, one has to note that the complete state y has to be known for this gradient calculation as soon as the right-hand side function f is nonlinear in y.

Similar approaches can be used also if the state y is not determined by ODEs but partial differential equations (PDEs). That is also for optimal control problems covered by PDEs the derivative calculation may by based on sensitivity or adjoint partial differential equations. Comprehensive introductions to this topics can be found in the books [81, 139].

A main advantage of analytical derivatives is the exactness of the derivative information. Furthermore, for first order derivatives, the computational complexity is quite often well when differentiating closed form expressions of moderate size. If the application is based on differential equations, the complexity of the derivative calculation depends on the problem dimensions and whether the sensitivity or adjoint equation is employed. The sensitivity equations are especially suitable for the optimization of a few parameters, whereas the adjoint approach provides a very efficient way in terms of runtime to compute gradients for distributed control problems. For time-dependent problems the memory requirement of the adjoint approach may be high due to the forward solution that is required for the integration of the adjoint equation in the case of nonlinear PDEs.

Applying the analytical derivative calculation recursively allows also the computation of higher order derivatives. For example, MAPLE can be used to derive explicit expressions of higher order derivatives for a given formula. However, these expressions are frequently very expensive to evaluate, see, e.g., [125] for an experimental study. The repeated differentiation is also applicable for differential equations. For example, the second order adjoint equation is used in [120] to compute second order derivatives. However, the differential equations become rather complicate and difficult to handle.

To conclude this paragraph, one can say that the analytical differentiation allows the computation of exact derivative information. The efficiency of this approach depends on the specific problem at hand. A main drawback of this approach is that it is not applicable if one does not have a closed form representation of the function to be differentiated either as explicit formula or based on differential equations. Here, prominent examples are function evaluations given as so-called legacy code. One important open question is the relation between the continuous derivative formulation using either sensitivity differential equations or adjoint differential equations and the discrete derivative information obtained for the discretized problem. A more detailed analysis of this topic forms one main part of the thesis at hand. Pointers to existing literature on this subject are given in Sec 1.4 together with a description of the own contributions.

Finite Differences If one assumes that the function to be differentiated is sufficiently smooth, then there exists a corresponding Taylor expansion. Considering for simplicity a function $f : \mathbb{R} \mapsto \mathbb{R}$, one obtains for a given point x and a step size h the formula

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots\ .$$

Ignoring the terms of order two and higher, one derives the following approximation of the first derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{1.2}$$

that requires only two function evaluations. Alternative approximation formulas based on the same idea can be found in [124]. Due to this very simple approach to approximate derivatives the usage of finite differences is very wide spread. Furthermore, it is the only possible approach to provide derivative information if the function evaluation is given only as black box and no further information is available.

However, there are several drawbacks of this approach. The first one is the appropriate choice of the step size h. If one chooses h too large, the truncation error due to the cancellation of the Taylor expansion may cause a large error in the approximation of the derivative. On the other hand, if h is too small, representation errors on the computer due to the limited accuracy of computations may lead to significant errors in the approximation of the derivative. Obviously, this problem becomes even worse if the function does not map from $\mathbb{R}$ to $\mathbb{R}$ but from $\mathbb{R}^n$ to $\mathbb{R}^m$. For that reason, some sophisticated step size determination algorithms have been developed. They are used for example in NPSOL [62] if the user does not provide the derivative calculation. Besides the inaccuracy of the derivatives, the computational complexity of the derivative calculation is the second main draw back. Using the cheapest approximation given by (1.2), one needs $n+1$ function evaluations to approximate the gradient of an objective function f with $f : \mathbb{R}^n \mapsto \mathbb{R}$. Hence, the finite difference approach is often not feasible for optimization problems where the value of n is large and the function evaluation is computationally expensive.

Automatic Differentiation Automatic Differentiation (AD) is a technique that provides exact derivative information of a smooth vector-valued function

$$F : \mathbb{R}^n \to \mathbb{R}^m, \qquad x \mapsto y = F(x),$$

evaluated by a computer program. The book by Griewank [70] gives a comprehensive introduction to AD.

The key idea in automatic differentiation is the systematic application of the chain rule. For this purpose, the computation of F is decomposed into a typically very long sequence of simple evaluations, e.g. additions, multiplications, and calls to elementary operations such as sin() or exp(). The derivatives of the simple operations with respect to their arguments can be easily calculated. Applying the chain rule to the overall decomposition then yields derivative information of the whole sequence of operations with respect to the input variables. The two basic methods of AD, namely the *forward* mode and the *reverse* mode, are described in Sec. 1.2 including the corresponding complexity estimates for the derivative calculation. It is important to note that AD yields derivative information that is exact within working accuracy. This means that no truncation errors occur. One main difference to the analytic computation of derivative is that AD does not derive a closed form expression for the derivative. Instead, exact derivative information given as numerical values is propagate forward or backward through the function evaluation procedure. Throughout this thesis, AD will be used to provide first- and second-order derivatives. However, it should be mentioned that also derivatives of arbitrary order can be evaluated exactly with AD techniques when propagating Taylor polynomials through the function evaluation procedure. This technique is used for example in [126] for the computation of Lie derivatives and Lie brackets and in [78, 116] for the integration of high-order DAEs using Taylor methods.

In the context of optimization problems, the forward mode of AD can be seen as a discrete version of the sensitivity approach. Conversely, the reverse mode of AD involves discrete adjoints somehow related to the continuous adjoint equation. Despite the fact that these parallels have already been hinted at in [68], a detailed theoretical analysis of the relations between the exact discrete derivatives provided by AD and the corresponding continuous derivative formulation is still the subject of ongoing research. Additionally, the computation of discrete adjoints face the same problem with respect to memory requirement as the continuous adjoint: As soon as nonlinear dependencies occur in the function evaluation intermediate results of the function evaluation are needed for discrete adjoints that are calculated once the function evaluation is finished.

During the last few decades the theoretical analysis and implementations of AD are mainly dedicated to the efficient evaluation of discrete adjoints. Since the computation of discrete adjoint is now feasible also for larger codes, the research activities recently were extended also to the development of new mathematical algorithms or the improvement of existing mathematical methods based on discrete adjoint information and related results.

The topics mentioned in the last two paragraphs form the motivation for the present thesis focused on the calculation of discrete adjoint information and its application in mathematical algorithms. The research results can be split into three aspects: First, the efficient computation of adjoint information including the reduction of the corresponding memory requirement and the exploitation of the problem structure. Second, the relation to the continuous adjoint approach and the consequences for example for optimization algorithms. Finally, the development of new algorithms involving discrete adjoint information or the improvement of existing mathematical methods. The remaining sections of this chapter explain the technique of automatic differentiation in more detail (Sec. 1.2). Furthermore, an overview about the current research and the own contributions with respect to the three aspects mentioned above are given: Section 1.3 is dedicated to the illustration of recent results with respect to the efficient calculation of discrete adjoints. Current results with respect to the relations of the discretize-then-optimize approach and the optimize-then-discretize approach are sketched in Sec. 1.4. This section includes also a discussion of discrete adjoints versus continuous adjoints. Finally, the application of discrete adjoints and related algorithms is the subject of Sec. 1.5.

1.2 Two Basic Modes of Automatic Differentiation

First results on automatic differentiation were published in the 1960s, see, e.g., [156, 157, 159]. Since then, extensive research activities have lead to a thorough understanding and analysis of the basic modes of AD including theoretical complexity results with respect to runtime and memory requirement. Besides the theoretical foundation, several AD tools have matured over the past years to a state that they are able to produce discrete sensitivity and adjoint information of large and unstructured codes. A very incomplete list comprises for example the tools ADIFOR [13], ADOL-C [154], CppAD [39], OpenAD [142], TAF [59] and Tapenade [85]. Information about these and other tools as well as pointers to literature on AD can be found on the web-page of the AD-community www.autodiff.org. In the remainder of this section, the main complexity results and the two basic implementation strategies will be described in more detail.

1.2.1 Function Evaluations

To derive complexity estimates for the runtime and memory requirement needed by automatic differentiation, it is assumed that the function $F : \mathbb{R}^n \mapsto \mathbb{R}^m, y = F(x)$, to be differentiation is evaluated by a computer program. For a given argument vector x the evaluation procedure can be decomposed into a sequence of elementary operations $\varphi_i, 1 \le i \le l$, the derivatives of which can be computed exactly. Here, we assume for simplicity that all elementary operations map into the real numbers. The set of elementary operations that may occur during the computation of y varies in dependence of the programming language that is applied. Obviously, the contents of the set can be very different considering for example function evaluations coded in C++, Fortran 90 or even Matlab.

One possible formalization of the decomposition into elementary operations is shown in Figure 1.1 and called *evaluation procedure* [70]. Here, the *precedence relation* $j \prec i$ denotes that the intermediate value v_i depends directly on the preceding intermediate value v_j. As can be seen, the evaluation procedure

```
for $i = 1, \ldots, n$
    $v_{i-n} = x_i$
for $i = 1, \ldots, l$
    $v_i = \varphi_i(v_j)_{j \prec i}$
for $i = 1, \ldots, m$
    $y_i = v_{l-i+1}$
```

Figure 1.1: Evaluation procedure

consists of an initialization part using the independent variables x in the first for-loop, the actual function evaluation in the second for-loop, and the extraction of the dependent variables in the last for-loop. This representation of a function evaluation will be employed in the next subsections to describe the forward and the reverse mode, respectively. We will use the coordinate transformation from Cartesian coordinates to spherical coordinates, i.e., the equations

$$y_1 = \sqrt{{x_1}^2 + {x_2}^2 + {x_3}^2} \qquad y_2 = \arccos\left(\frac{x_1}{\sqrt{{x_1}^2 + {x_2}^2}}\right)$$
$$y_3 = \operatorname{arccot}\left(\frac{x_3}{\sqrt{{x_1}^2 + {x_2}^2}}\right) \tag{1.3}$$

$$\begin{array}{lcl}
v_{-2} & = & x_1 \\
v_{-1} & = & x_2 \\
v_0 & = & x_3 \\
v_1 & = & v_{-2}^2 \\
v_2 & = & v_{-1}^2 \\
v_3 & = & v_0^2 \\
v_4 & = & v_1 + v_2 \\
v_5 & = & v_4 + v_3 \\
v_6 & = & \sqrt{v_4} \\
v_7 & = & v_{-2} * v_6 \\
v_8 & = & v_0 * v_6 \\
v_9 & = & \sqrt{v_5} \\
v_{10} & = & \arccos(v_7) \\
v_{11} & = & \operatorname{arccot}(v_8) \\
y_1 & = & v_9 \\
y_2 & = & v_{10} \\
y_3 & = & v_{11}
\end{array}$$

Figure 1.2: Evaluation procedure (coordinate transformation)

as example to illustrate the concepts presented in this section. That is, we consider the function $F : D \mapsto \mathbb{R}^3$, $y = F(x)$, that is defined by Equation (1.3) on the domain $D = \{(x_1, x_2, x_3) \in \mathbb{R}^3 \,|\, x_i > 0, i = 1, 2, 3\}$. A possible evaluation procedure for the coordinate transformation example is shown in Fig. 1.2. The common subexpression ${x_1}^2 + {x_2}^2$ is computed only once and reused afterwards during the function evaluation. In the following subsections, we will see that the derivative calculation benefits directly from the reusage of the computed value. This forms one major difference to the analytic derivative calculation using computer algebra tools where the exploitation of common subexpressions is still a critical point, see, e.g., [125].

As alternative approach, one may employ the computational graph defined by the elementary operations. This formalization is especially well suited if structural information about the function evaluation, e.g., sparsity of the derivative matrices, has to be exploited. This will be explained in more detail in Sec. 1.3. For the coordinate transformation example (1.3) the corresponding computational graph is shown in Fig. 1.3.

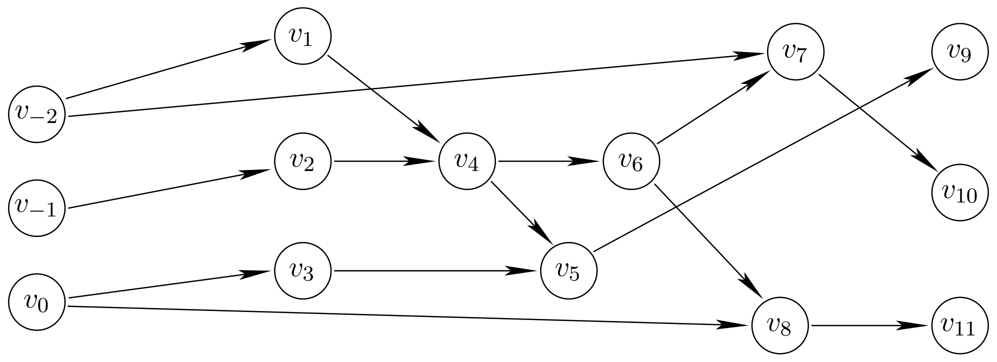

Figure 1.3: Computational graph for coordinate transformation

It is assumed for simplicity in this introduction to AD that there are no overwrites, i.e., the values of all intermediate values v_i are available during the whole function evaluation and therefore also during the derivative calculation. The consequences of overwrites for the evaluation of the derivatives using the reverse mode are discussed in Subsec. 1.3.1.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
v_{-2} & = & x_1 & \dot{v}_{-2} & = & \dot{x}_1 \\
v_{-1} & = & x_2 & \dot{v}_{-1} & = & \dot{x}_2 \\
v_0 & = & x_3 & \dot{v}_0 & = & \dot{x}_3 \\
v_1 & = & v_{-2}^2 & \dot{v}_1 & = & 2v_{-2}\dot{v}_{-2} \\
v_2 & = & v_{-1}^2 & \dot{v}_2 & = & 2v_{-1}\dot{v}_{-1} \\
v_3 & = & v_0^2 & \dot{v}_3 & = & 2v_0\dot{v}_0 \\
v_4 & = & v_1 + v_2 & \dot{v}_4 & = & \dot{v}_1 + \dot{v}_2 \\
v_5 & = & v_4 + v_3 & \dot{v}_5 & = & \dot{v}_4 + \dot{v}_3 \\
v_6 & = & \sqrt{v_4} & \dot{v}_6 & = & -\dot{v}_4/(2v_6) \\
v_7 & = & v_{-2}v_6 & \dot{v}_7 & = & \dot{v}_{-2}v_6 + v_{-2}\dot{v}_6 \\
v_8 & = & v_0 v_6 & \dot{v}_8 & = & \dot{v}_0 v_6 + v_0\dot{v}_6 \\
v_9 & = & \sqrt{v_5} & \dot{v}_9 & = & -\dot{v}_5/(2v_9) \\
v_{10} & = & \arccos(v_7) & \dot{v}_{10} & = & -\frac{1}{\sqrt{1-v_7^2}}\dot{v}_7 \\
v_{11} & = & \operatorname{arccot}(v_8) & \dot{v}_{11} & = & -\frac{1}{1+v_8^2}\dot{v}_8 \\
y_1 & = & v_9 & \dot{y}_1 & = & \dot{v}_9 \\
y_2 & = & v_{10} & \dot{y}_2 & = & \dot{v}_{10} \\
y_3 & = & v_{11} & \dot{y}_3 & = & \dot{v}_{11}
\end{array}
$$

Figure 1.4: Forward mode differentiation (coordinate transformation)

1.2.2 The Forward Mode of AD

In Subsec. 1.2.1, we assumed that each elementary operation can be easily and exactly differentiated with respect to its input values. This assumption provides a very simple way to propagate derivative information through an evaluation procedure. One only has to extend each elementary statement by the corresponding statement for its total derivative using the chain rule and the other basic differentiation rules. This approach is illustrated Fig. 1.4 using the coordinate transformation example. Here, $\dot{x} = (\dot{x}_1, \dot{x}_2, \dot{x}_3)$ serves as input value for the derivative computation.

In the case of a general evaluation procedure, the total derivative of an elementary operation $v_i = \varphi_i(v_j)_{j \prec i}$ is given by

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial \varphi_i}{\partial v_j}(v_j)_{j \prec i}\dot{v}_j \ .$$

Hence, for a general smooth function $y = F(x)$ evaluated by an evaluation procedure one obtains the derivative calculation shown in Fig. 1.5. Analyzing the mathematical meaning of the output values $\dot{y} = (\dot{y}_1, \ldots, \dot{y}_m)$, it follows that the forward mode differentiation of AD yields the directional derivative

$$\dot{y} = F'(x)\dot{x},$$

see, e.g., [70]. Note that the derivative calculation for each elementary operation is exact and that the derivatives of the several elementary operations are combined using the chain rule. Hence, the overall derivative $\dot{y} = F'(x)\dot{x}$ is exact for given values of x and $\dot{x}$ within the used working accuracy.

```
for $i = 1, \ldots, n$
    $v_{i-n} = x_i, \qquad \dot{v}_{i-n} = \dot{x}_i$
for $i = 1, \ldots, l$
    $[\, v_i,\, \dot{v}_i \,] = \left[\varphi_i(v_j)_{j \prec i}, \sum_{j \prec i} \frac{\partial \varphi_i}{\partial v_j}(v_j)_{j \prec i} \dot{v}_j\right]$
for $i = 1, \ldots, m$
    $y_{m-i+1} = v_{l-i+1}, \quad \dot{y}_i = \dot{v}_{l-i+1}$
```

Figure 1.5: Forward mode differentiation

As mentioned already at the beginning of this introduction, the second important criterion for the derivative calculation is the computational effort required to compute the derivative information, i.e., the runtime behavior and the memory requirement. A very flexible complexity measure for the runtime behavior was introduced in [70] taking different cost for an addition, a multiplication, a general nonlinear operation, and a memory access into account. Under quite general assumption on the relations between these four components of the complexity measure, e.g. that a multiplication is at least as costly as an addition, one can derive the following complexity estimate

$$\mathrm{TIME}(F'(x)\dot{x}) \le c\,\mathrm{TIME}(F(x)) \qquad \text{with} \qquad c \in [2,\, 2.5]$$

for a general smooth function $y = F(x)$ and arbitrary input values x and $\dot{x}$ [70]. Hence, the runtime needed by the forward mode of AD to compute the exact directional derivative $F'(x)\dot{x}$ is comparable or only slightly higher than the runtime needed by finite differences to approximate the same derivative information.

In Subsec. 1.2.1 we assumed that no overwrites occur during the evaluation procedure. However, for the forward mode of AD well-defined overwrites of intermediate values v_i cause no real problem. That means, as long as the evaluation procedure is well defined, i.e., computes the value $y = F(x)$ correctly, the same overwrites can also be used for the derivative values $\dot{v}_i$. In [70, Section 2.4], this property of *forward compatibility* is defined in more detail. Then, one only has to ensure that the evaluation of the elementary operation and the corresponding derivative calculation are performed in the correct order. For example, let us assume that the statement $v_i = v_j * v_k$ overwrites the value of v_j. Then the derivative calculation given by $\dot{v}_i = \dot{v}_j * v_k + v_j * \dot{v}_k$ has to be evaluated before the execution of the elementary operation to ensure that the value of v_j is still available. Hence, in the general forward mode differentiation as stated in Fig. 1.5, the square brackets stand for a combined calculation of v_i and $\dot{v}_i$ leaving the order of the corresponding calculations open. If this is taken into account for the implementation of AD, the memory requirement for the forward mode of AD is given by

$$\mathrm{RAM}(F'(x)\dot{x}) = 2\,\mathrm{RAM}(F(x)),$$

where RAM stands for **r**andom **a**ccess **m**emory.

To conclude this subsection on the forward mode of AD, it should be mentioned that the scalar forward mode presented here can be extended to a vector version to compute multiple directional derivatives $F'(x)\dot{X}$ for $\dot{X} \in \mathbb{R}^{n\times p}$. This generalization allows a considerable reduction in terms of runtime in comparison to the evaluation of p single directional derivatives $F'(x)\dot{x}$ yielding the same derivative information. Furthermore, the forward mode as presented in this subsection can be extend to the propagation of higher order derivative using Taylor polynomials instead of only first order information. Finally, there are several results with respect to the differentiation of iterative processes, as for example fixpoint iterations. Detailed information on these topics and additional pointers can be found in [70].

1.2.3 The Reverse Mode of AD

The forward mode of AD can be easily motivated using a straightforward propagation of derivatives in combination with the calculation of the function value using the evaluation procedure. So far, there does not exist a similar obvious derivation for the reverse mode. The reverse mode and similar approaches were introduced in the literature for example in the context of the estimation of round off errors [108], nonlinear sensitivity analysis [158], and the computation of partial derivatives [134]. To motivate the reverse mode of AD, one may use an extended system of equations, the implicit function theorem or a representation of the Jacobian matrix as a product of l matrices [70]. In this thesis, we will use the reversal of the matrix-products to illustrate the reverse mode of AD and the corresponding computation of discrete adjoints.

Once more, we assume that the evaluation procedure does not contain overwrites. Then, all intermediates derivatives $\dot{v}_i$ are simultaneously defined. Hence, they can be combined in a large vector $\dot{v} = (\dot{v}_{1-n}, \ldots, \dot{v}_{-1}, \dot{v}_0, \ldots, \dot{v}_l)^T \in \mathbb{R}^{n+l}$. Analyzing the general forward mode of AD given by Fig. 1.5, the propagation of the derivative information in the second for-loop is given by the recursion

$$\dot{v} = A_i \dot{v} \quad \text{for} \quad 1 \le i \le l,$$

where A_i denotes the matrix

$$A_i \equiv I_{n+l} + e_{n+i}\left[\left(\frac{\partial \varphi_i}{\partial v_{1-n}}(v_j)_{j\prec i}, \ldots, \frac{\partial \varphi_i}{\partial v_{i-1}}(v_j)_{j\prec i}\right) - e_{n+i}\right]^T,$$

I_{n+l} the identify matrix in $\mathbb{R}^{(n+l)\times(n+l)}$, and e_j the jth unit vector in $\mathbb{R}^{n+l}$. Additionally, we introduce the projection matrices

$$P_n = [I_n, 0, \ldots, 0] \in \mathbb{R}^{n\times(n+l)} \qquad \text{and} \qquad Q_m = [0, \ldots, 0, I_m] \in \mathbb{R}^{m\times(n+l)}$$

onto the first n and last m components of a $(n + l)$ vector, respectively. This yields the representation of the general forward mode of AD shown in Fig. 1.5 as a chain of matrix products:

$$\dot{y} = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \dot{x}.$$

Comparing this equation with the identity $\dot{y} = F'(x)\dot{x}$, one obtains the product representation

$$F'(x) = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T$$

Transposing the chain of matrix-products yields for the matrix-vector product $\bar{x} = (F'(x))^T \bar{y}$ the equation

$$\bar{x} = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T Q_m^T \bar{y} \qquad \text{with}$$
$$A_i^T \equiv I_{l+n} + \left[\left(\frac{\partial \varphi_i}{\partial v_{1-n}}(v_j)_{j \prec i}, \ldots, \frac{\partial \varphi_i}{\partial v_{i-1}}(v_j)_{j \prec i} \right) - e_{n+i} \right] e_{n+i}^T.$$

Hence, for a given vector $\bar{v} = (\bar{v}_{1-n}, \ldots, \bar{v}_{-1}, \bar{v}_0, \ldots, \bar{v}_l)^T \in \mathbb{R}^{n+l}$ the multiplication with A_i^T from the left corresponds to the following operations on the components $\bar{v}_j$:

- All $\bar{v}_j$ with $i \neq j \not\prec i$ are left unchanged.
- All $\bar{v}_j$ with $j \prec i$ are incremented by $\frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} \bar{v}_i$.
- Subsequently, $\bar{v}_i$ is set to zero.

Since the computation of partial derivatives is based on the intermediate values v_i, the evaluation procedure has to be evaluated before the computation of the derivative values $\bar{v}$ can be performed. Hence, we obtain for a general smooth function $y = F(x)$ the derivative calculation shown in Fig. 1.6. Here, the first

```
**for** $i = 1, \ldots, n$
    $v_{i-n} = x_i, \quad \bar{v}_{i-n} = 0$
**for** $i = 1, \ldots, l$
    $v_i = \varphi_i(v_j)_{j \prec i}, \quad \bar{v}_i = 0$
**for** $i = 1, \ldots, m$
    $y_i = v_{l-i+1}$
**for** $i = 1, \ldots, m$
    $\bar{v}_{l-i+1} = \bar{y}_i$
**for** $i = l, \ldots, 1$
    $\bar{v}_j \mathrel{+}= \bar{v}_i \frac{\partial \varphi_i}{\partial v_j}(v_j)_{j \prec i} \quad \forall j \quad \text{with} \quad j \prec i$
    $\bar{v}_i = 0$
**for** $i = 1, \ldots, n$
    $\bar{x}_i = \bar{v}_{i-n}$
```

Figure 1.6: Reverse mode differentiation without overwrites

three for-loops, also called the forward sweep, correspond to the evaluation of the function $y = F(x)$. Subsequently, in the so-called adjoint sweep consisting of the remaining three for-loops, the derivative values are computed. It is important to note that the reverse mode of AD yields the value of the product

$$\bar{x}^T = \bar{y}^T F'(x)$$

$$
\begin{aligned}
& v_{-2} = x_1,\ \bar{v}_{-2} = 0,\ v_{-1} = x_2,\ \bar{v}_{-1} = 0,\ v_0 = x_3,\ \bar{v}_0 = 0 \\
& \quad v_1 = v_{-2}^2,\ \bar{v}_1 = 0 \\
& \quad\quad v_2 = v_{-1}^2,\ \bar{v}_2 = 0 \\
& \quad\quad\quad v_3 = v_0^2,\ \bar{v}_3 = 0 \\
& \quad\quad\quad\quad v_4 = v_1 + v_2,\ \bar{v}_4 = 0 \\
& \quad\quad\quad\quad\quad v_5 = v_4 + v_3,\ \bar{v}_5 = 0 \\
& \quad\quad\quad\quad\quad\quad v_6 = \sqrt{v_4},\ \bar{v}_6 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad v_7 = v_{-2} * v_6,\ \bar{v}_7 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad v_8 = v_0 * v_6,\ \bar{v}_8 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad v_9 = \sqrt{v_5},\ \bar{v}_9 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad v_{10} = \arccos(v_7),\ \bar{v}_{10} = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad v_{11} = \operatorname{arccot}(v_8),\ \bar{v}_{11} = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad y_1 = v_9,\ y_2 = v_{10},\ y_3 = v_{11} \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_9 \mathrel{+}= \bar{y}_1,\ \bar{v}_{10} = \bar{y}_2,\ \bar{v}_{11} = \bar{y}_3 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_8 \mathrel{+}= \bar{v}_{11} - \tfrac{1}{1+v_8^2},\ \bar{v}_{11} = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_7 \mathrel{+}= \bar{v}_{10} - \tfrac{1}{\sqrt{1-v_7^2}},\ \bar{v}_{10} = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_5 \mathrel{-}= \bar{v}_9/(2v_9),\ \bar{v}_9 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_0 \mathrel{+}= \bar{v}_8 v_6,\ \bar{v}_6 \mathrel{+}= \bar{v}_8 v_0,\ \bar{v}_8 = 0 \\
& \quad\quad\quad\quad\quad\quad\quad \bar{v}_{-2} \mathrel{+}= \bar{v}_7 v_6,\ \bar{v}_6 \mathrel{+}= \bar{v}_7 v_{-2},\ \bar{v}_7 = 0 \\
& \quad\quad\quad\quad\quad\quad \bar{v}_4 \mathrel{-}= \bar{v}_6/(2v_6),\ \bar{v}_6 = 0 \\
& \quad\quad\quad\quad\quad \bar{v}_4 \mathrel{+}= \bar{v}_5,\ \bar{v}_4 \mathrel{+}= \bar{v}_5,\ \bar{v}_5 = 0 \\
& \quad\quad\quad\quad \bar{v}_1 \mathrel{+}= \bar{v}_4,\ \bar{v}_2 \mathrel{+}= \bar{v}_4,\ \bar{v}_4 = 0 \\
& \quad\quad\quad \bar{v}_0 \mathrel{+}= \bar{v}_3 2 v_0,\ \bar{v}_3 = 0 \\
& \quad\quad \bar{v}_{-1} \mathrel{+}= \bar{v}_2 2 v_{-1},\ \bar{v}_2 = 0 \\
& \quad \bar{v}_{-2} \mathrel{+}= \bar{v}_1 2 v_{-2},\ \bar{v}_1 = 0 \\
& \bar{x}_1 = \bar{v}_{-2},\ \bar{x}_2 = \bar{v}_{-1},\ \bar{x}_3 = \bar{v}_0
\end{aligned}
$$

Figure 1.7: Reverse mode differentiation (coordinate transformation)

and therefore a derivative information that can not be approximated easily with finite differences. For the coordinate transformation, the resulting derivative computations is shown in Fig. 1.7.

Using once more the complexity measure for the runtime behavior as introduced in [70], one can derive for the reverse mode of AD the runtime estimate

$$\mathrm{TIME}(\bar{y}^T F'(x)) \leq c\, \mathrm{TIME}(F(x)) \qquad \text{with} \qquad c \in [3,\, 4] \tag{1.4}$$

for a general smooth function $y = F(x)$ and arbitrary input values x and $\bar{y}$. It follows from the last inequality that the runtime required for computing the gradient of a scalar-valued function within working accuracy can be bounded above by a small multiple of the time needed for the function evaluation itself. Hence, the runtime needed for computing the gradient is completely independent of n, i.e., the number of its components. Here, we face an important difference in comparison to the approximation of gradient information based on finite differences, where the runtime increases for a gradient calculation linearly with n. Therefore, Eq. (1.4) is also known as *cheap gradient result.* Similar good complexity results in terms of runtime can be derived also for the analytical derivative computation based on adjoint differential equations.

In addition to the runtime behavior, one also has to analyze the memory requirement of the derivative computation. So far, it is implicitly assumed in the representation of the reverse mode that each intermediate value has an own memory pad that is not overwritten by another intermediate result. Obviously, this assumption usually does not hold for codes of reasonable size. However, the values of the intermediates v_j are needed to compute the partial derivatives $\frac{\partial\varphi_i}{\partial v_j}(v_j)_{j\prec i}$ during the adjoint sweep. A very simple remedy to this problem is the storage of the value v_i before it is overwritten by the value $\varphi_i(v_j)_{j\prec i}$ and to recover the value before the discrete adjoint of the statement $v_i = \varphi_i(v_j)_{j\prec i}$ is computed. This yields a memory requirement of sequentially accessed storage that is proportional to the number of elementary operations performed, i.e., one has for this basic approach

$$\mathrm{MEM}(\bar{y}^T F'(x)) \sim l,$$

where MEM stands for the required amount of sequentially accessed memory. The resulting runtime effects will be discussed in more detail in Subsec. 1.3.1.

With respect to the randomly accessed memory one can show that

$$\mathrm{RAM}(\bar{y}^T F'(x)) \leq 2\,\mathrm{RAM}(F(x)),$$

holds for reasonable assumptions on the evaluation procedure [70]. Once more, one finds related complexity results for the analytical derivative computation based on adjoint differential equations, where the full solution of the original differential equation has to be stored in the nonlinear case.

It should be mentioned that the scalar reverse mode presented here can be extended again to a vector version to compute multiple derivatives of the form $\bar{Y}^T F'(x)$ for $\bar{Y} \in \mathbb{R}^{m\times q}$. This generalization allows a considerable reduction in terms of runtime in comparison to the evaluation of q single directional derivatives $\bar{y}^T F'(x)$ yielding the same derivative information. Furthermore, there are several results with respect to the differentiation of iterative processes. Detailed information on these topics and further pointers to additional references can be found in [70].

Throughout we will consider the reverse mode of AD and therefore the application of an AD tool as our method of choice to provide discrete adjoint information. For this reason, a rather detailed description of the underlying process is given in this subsection. The term *discrete* reflects the fact that the computer program used to compute the function value is based on discrete information, i.e., specific numerical values. This is in contrast to the continuous adjoint information provided for example by the adjoint differential equations in continuous form. Obviously, as alternative to the application of an AD tool, discrete derivative information can be obtained by handcoding of a corresponding calculation in a time-consuming and very error-prone process.

1.2.4 Implementation Strategies

There are two major approaches for implementing AD: *source transformation* and *operator overloading*. Both approaches will be described briefly in the remainder of this subsection. However, a third alternative, namely the integration of AD into the compiler should be mentioned. Only recently, this strategy has been implemented as a prototype version for the NAG Fortran compiler [114]. The integration of AD into a compiler requires comprehensive knowledge of the compiler internals. Therefore, this alternative is difficult to realize. However, it is very attractive for users as well as for researchers working on AD: The user has not to deal with another software tool and the AD developer could exploit all the information that is already available inside the compiler.

Source Transformation

An AD tool based on source transformation generates for a given source code to evaluate a function a new source code for evaluating additionally the required derivative information. This technique is especially well suited for the automatic differentiation of Fortran 77 codes because this programming language allows an appropriate analysis of the code.

The main ingredient of this approach for the developer of an AD tool is a more or less complete compiler technology that has to be provided. This is due to the fact that one has to perform a lexical analysis and a syntax analysis of the given program. Subsequently, a semantic analysis is required to obtain for example a dependence analysis in order to determine the independent variables, the dependent variables, and all intermediate variables that have to be differentiated. This yields a static data flow analysis including a symbol table and error handler. Once, all this information is available, the forward mode or the reverse mode of AD can be applied to derive the source code for the computation of the required derivative information. Subsequently, an optimization of the generated code can be performed yielding a new source file for computing the desired derivative objects. Obviously, an AD tool based on source transformation is very challenging from a developers point of view, where the AD part itself comprises a comparable small part. Source transformation is used for example by the packages ADIFOR [13], TAF [59], and Tapenade [85].

The development of the AD tool OpenAD [142] that is also based on source transformation started quite recently. Here, the idea is to transform a given computer program into an XML format. Then, all AD technology is based on and applied to this internal representation. Subsequently, the XML representation extended by the derivative calculation is unparsed to obtain a computer program that can be compiled with standard compilers. The advantage of this approach is that a function evaluation given in any programming language can be differentiated as soon as a corresponding parser into an XML representation is available. Hence, the only remaining but nevertheless difficult part is the provision of appropriated parsers and unparsers for XML since the AD technology for the XML representation can be exploited.

Operator Overloading

For a program written in C or C++ it is considerably more difficult to obtain the required information as for example a dependency analysis for an efficient AD implementation based on source transformation. This is due to the concepts of dynamic memory allocation and pointers provided by C and C++. Therefore, currently there exists no AD tool based on source transformation implementing the reverse mode of AD, whereas there are some packages based on operator overloading for the automatic differentiation of programs written in C or C++. Here one can distinguish two different strategies.

Using operator overloading, an obvious strategy to implement AD is the definition of a new class that contains the derivative values in combination with the values of the original variables. The implementation of this approach is very simple for the forward mode of AD. Applying the reverse mode of AD, the derivative values are propagated backwards. That is, one starts computing the derivatives of the dependents with respect to the last intermediate values and traverses backwards through the evaluation process until the independents are reached. Therefore, it is not possible to compute the derivatives simultaneously with the values of the intermediates as proposed above for the forward mode. To overcome this difficulty one may use a new class to build a graph that represents the function to be differentiated during the function evaluation. This method is used for example by the AD tool FADBAD [8]. To this end, a new data structure is defined that allows the generation of appropriate backward references during the function evaluation and stores the local derivatives during the reverse sweep. It is important to note that the pointers in the resulting graph are oriented in the opposite direction compared to the control flow and the access to nodes happens largely at random. Furthermore, for bigger problems the fact that the complete computational graph is kept in “core” may cause problems because of the corresponding memory requirement.

Instead of keeping the derivative value in one composite structure with the value of the variable one may use the operator overloading facility to log for each operation during the function evaluation the corresponding operator and the variables that are involved to generate an internal representation of the function in addition to the function evaluation. This can be viewed as an representation of the computational graph associated with the evaluation of the function at a specific point x. Subsequently, the internal function representation is used to compute the desired derivative objects. The AD tools ADOL-C [154] and CppAD [39] are based on this technique.

During the work on this habilitation, the AD tool ADOL-C was extended continuously to provide in addition to the theoretical results an implementation that can be used for the efficient differentiation of C and C++ tools, see, e.g., [104, 150].

1.3 Efficient Computation of Discrete Adjoints

In Subsec. 1.2.3, the basic idea of computing discrete adjoint with the reverse mode of AD was illustrated and only a short comment was made on the memory required for the differentiation process. In this section, we describe the problem in more detail and present several alternatives for the handling of the potentially very large amount of memory that is needed to compute discrete adjoints.

Furthermore, we discuss methods for the efficient computation of discrete adjoints by exploiting specific properties of the function to be differentiated. These properties comprise sparsity and fixpoint iterations.

1.3.1 Recomputation versus Storage

To simplify the presentation of the forward mode and the reverse mode, we made in Subsec. 1.2.1 the assumption that the intermediate values v_i are not overwritten during the function evaluation. Obviously, this assumption does not hold for any reasonable program, since the reusage of storage forms one important ingredient of an efficient programming style.

As mentioned briefly in Subsec. 1.2.2, overwrites cause no problem for the forward mode of AD if the derivative computation is adapted appropriately. This situation changes completely for the reverse mode of AD. From Figs. 1.6 and 1.7 one can conclude that the intermediate values v_i are required in reverse order for the derivative calculation. Hence, if overwrites occur during the function evaluation, information required for the adjoint sweep may be lost, e.g., for the derivative calculation involving nonlinear operations as multiplications or calls to intrinsic functions such as sin(). There are two alternatives to provide the required intermediate values: First, one may recompute the required intermediate values. Second, as described already in Subsec. 1.2.3, one may store them during the function evaluation onto a strictly sequential accessed data structure. The former alternative is used as default option by the AD tool TAF. However, due to the required recomputation the computing time for the derivative information is then proportional to l^2 where l denotes as throughout the number of elementary operations evaluated during the computation of the function value [70]. Hence, this approach is often not feasible. The second alternative, namely the storage of all intermediate values onto an appropriate data structure is used for example by the AD tools ADOL-C and CppAD. The resulting memory requirement is proportional to l, i.e., the number of elementary operations performed during the function evaluation. For real-world problems this fact may lead to an unacceptable memory requirement. Additionally the computation of the derivatives may be slowed down considerably due the memory accesses. Here, one has to note that for the derivation of the runtime estimate (1.4), one assumes that all intermediate values required for the derivative calculation are accessibly at a negligible cost. As long as the corresponding sequential data structure can be kept in main memory, this assumption do hold. However, for large-scale function evaluations, as for example the simulation of the flow around an airfoil, the access to the sequential data structure is quite often the critical point for the computing time of the derivatives. We will use Speelpennings example given by

$$f : \mathbb{R}^n \mapsto \mathbb{R}, \qquad y = \prod_{i=1}^{n} x_i$$

for the illustration of this fact. The corresponding gradient evaluation using the reverse mode of AD is shown in Fig. 1.8. As can be seen, the only intermediate

```
**for** $i = 1, \ldots, n$
    $v_{i-n} = x_i$
$v_1 = 1$
**for** $i = 1, \ldots, n$
    store$(v_1)$
    $v_1 = v_1 * v_{i-n}$
$y_i = v_1, \quad \bar{v}_1 = 1$
**for** $i = l, \ldots, 1$
    read$(v_1)$
    $\bar{v}_{i-n} = \bar{v}_1 * v_1, \; \bar{v}_1 = \bar{v}_1 * v_{i-n}$
**for** $i = 1, \ldots, n$
    $\bar{x}_i = \bar{v}_{i-n}$
```

Figure 1.8: Reverse mode AD (Speelpenning)

value v_1 is overwritten by each elementary operation that is evaluated for the computation of the function f. Hence, the value of v_1 has to be stored onto the sequential data structure in each elementary operation and read from the data structure for each computation of an discrete adjoint $\bar{v}_i$.
These observation hold for all implementations of AD independent of a source transformation or an operator overloading approach. Hence, also the corresponding effects on the overall runtime that are caused by storing and retrieving the values can be observed for both implementation strategies. This fact is illustrated in Fig. 1.9 for the AD tools ADOL-C and Tapenade. As can be seen, the runtime for one gradient calculation grows

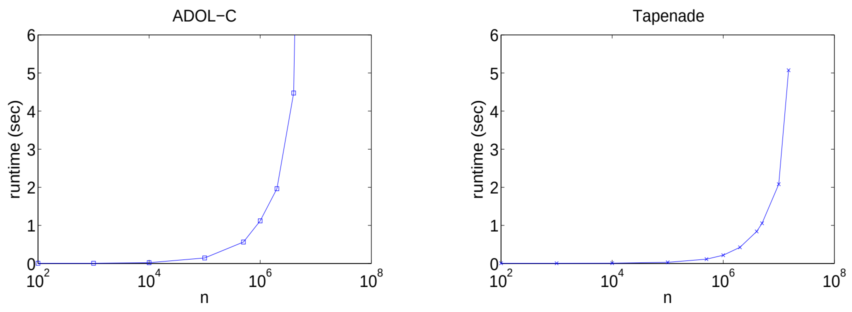

Figure 1.9: Execution time for gradient computation (Speelpennings example)

very slowly for $n \leq 100000$ and $n \leq 1000000$, respectively. For larger values of n, the runtime increases dramatically. This is due to the fact that for these values of n another memory medium, namely the disc, is used to store the intermediate values required for the adjoint sweep. One has to note that this example only serves to illustrate the influences of the memory access cost for different sizes of the function to be differentiated. Naturally, the specific development of the runtime depends on the function to be differentiated as well as the specific internal settings of the AD tools.

To overcome the problems that are caused by the memory requirement of the reverse mode of AD, several checkpointing approaches have been developed. If the considered function evaluation has no specific structure, one may allow the user of an AD tool to place checkpoints somewhere during the function evaluation to reduce the overall memory requirement. This simple approach is provided for example by the AD tool TAF [59]. As alternative, one may exploit the call graph structure of the function evaluation to place checkpoints at the entries of certain subroutines. This reversal strategy leads to a joint reversal for the subroutines, see, e.g. [70], and therefore to a reduction of the memory requirement. The subroutine-oriented checkpointing is used, e.g., by the AD tools Tapenade [85] and OpenAD [115].

As soon as one can exploit additional information about the structure of the function evaluation, an appropriate adapted checkpointing strategy can be used. This is in particular the case if a time-stepping procedure is contained in the function evaluation allowing the usage of a time-stepping oriented checkpointing. If the number of time steps N is known a-priori and if the computational costs of the time steps are almost constant, one can compute checkpointing schedules in advance. This approach is referred to as offline checkpointing. One very popular offline checkpointing strategy is to distribute the checkpoints equidistantly over the time interval, also known as windowing in the PDE context, see, e.g., [6, 9], or multi-level checkpointing in the AD context, see, e.g., [87]. However, it was shown in [153] that this approach is not optimal. A more advanced but still not optimal approach is the binary checkpointing used for example in [105]. However, optimal checkpointing schedules based on binomial coefficients can be computed for an a-priori known number of time steps in advance to achieve an optimal, i.e. minimal, runtime increase for a given number of checkpoints [69, 76]. Hence, the required recomputations are reduced to a minimum for a given number of checkpoints.

A detailed comparison of the windowing technique with the binomial checkpointing is contained for the first time in [153], which is also part of this thesis (see Chapter 2). The analysis presented in [153] yields that for a given number of checkpoints and a given upper bound on the number of repeated forward integrations, the binomial checkpointing approach allows the adjoint computation for a larger time horizon than the windowing technique. Furthermore, it is shown that the time needed for the adjoint computation coincides for both approaches if the windowing is not used recursively. Moreover, it is also proved in [153] that for a given number of checkpoints the time required for the adjoint computation with a recursive windowing exceeds the runtime needed for the adjoint computation with the binomial checkpointing. Therefore, windowing can be seen as not optimal, despite the fact that it is commonly used. The theoretical analysis contained in [153], i.e., the relation of the maximal number N of time steps the adjoint of which can be computed using either the windowing technique or binomial checkpointing as well as the analysis of the overall runtime required by the two approaches were derived by the author of this thesis.

So far, the binomial checkpointing algorithm was only available as separate software tool called revolve, see also [76]. Hence, the user was responsible for the integration and usage of the driver routine revolve into the application for the exploitation of the optimal checkpointing. Recently, the routine revolve was integrated in cooperation with the author of this thesis into the AD tool ADOL-C to allow an automated usage of the binomial checkpointing [104]. The runtime results presented in [104] show for the first time that checkpointing may lead even to a decrease in runtime despite the fact that a considerable amount of intermediate information has to be recomputed. The numerical example that serves to illustrate these runtime effects of the checkpointing procedure is an industrial robot as depicted in Fig. 1.10 that has to perform a fast turn-around maneuver. We denote by $q = (q_1, q_2, q_3)$ the angular coordinates of the robot's joints, where q_1 is referring to the angle between the base and the two-arm system. The robot is controlled via three control functions u_1, u_2, u_3 that denote the respective angular momentum applied to the joints (from bottom to top) by electrical motors. The control problem under consideration is to minimize the energy-related objective where the final time t_f is given yielding the target function

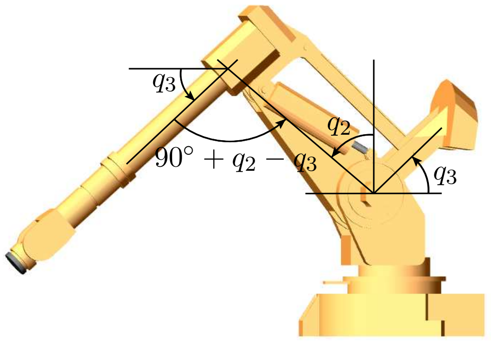

Figure 1.10: Robot ABB IRB 6400

$$J(q,u) = \int_0^{t_f} [u_1(t)^2 + u_2(t)^2 + u_3(t)^2]\, dt.$$

The complete equations of motion for this model can be found in [103]. The robot's task to perform a turn-around maneuver is expressed by means of initial and terminal conditions in combination with control constraints [66]. However, for illustrating the runtime effects of the checkpointing facility integrated in ADOL-C only the gradient computation of $J(q,u)$ with respect to u was considered.

To compute an approximation of the trajectory x, the standard Runge-Kutta method of order 4 was applied for time integration resulting in about 800 lines of code. The integration and derivative computations were computed using an AMD Athlon64 3200+ (512 kB L2-cache) and 1GB main memory. The resulting averaged runtimes in seconds for one gradient computation are shown in Fig. 1.11, where the runtime required for the derivative computation without checkpointing, i.e., the basic approach (BA), is illustrated by a dotted line. The runtime needed by the checkpointing approach (CP) using $c = 2, 4, 8, 16, 32, 64(, 128, 256)$ checkpoints is given by the solid line. To illustrate the corresponding savings in memory requirement, Table 1.1 shows the tape sizes for the basic approach as well as the tape and checkpoint sizes required by the checkpointing version. The tape size for the later varies since the number of independents is a multiple of N due to the distributed control u. One basic checkpointing assumption, i.e., the more checkpoints are used the less runtime the execution needs, is clearly depicted by case $N = 1000$ in Fig. 1.11. The smaller runtime for the basic approach completes the setting. However, the more interesting cases for this example are $N = 100$ and $N = 5000$, respectively. In these situations a smaller runtime was achieved even though checkpointing was used. These results are due to a well-known effect, namely that computing from a level of the memory hierarchy that offers cheaper access cost may result in a significant smaller runtime. For the robot example the computation could be redirected from main memory mainly into the L2-cache of the processor ($N = 100$) and from at least partially hard disk access completely into

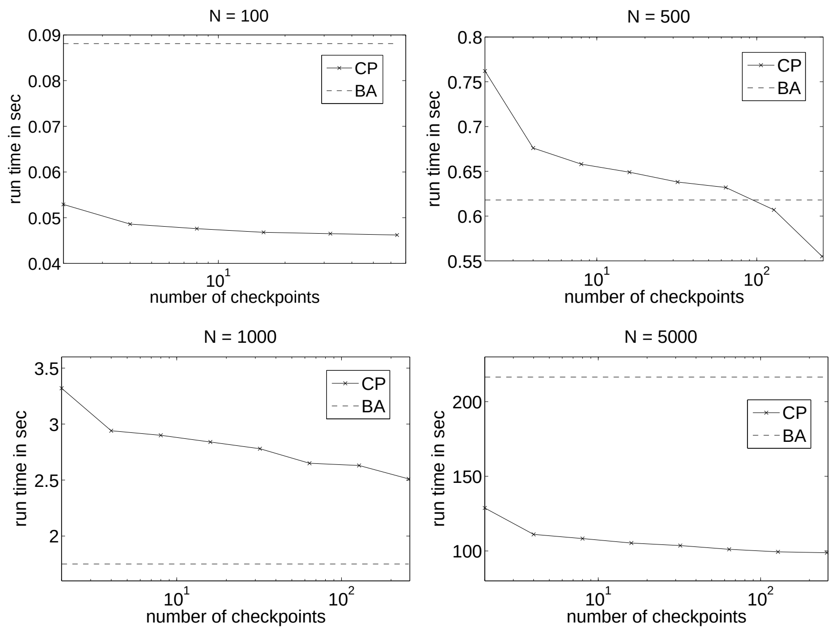

Figure 1.11: Comparison of runtimes for $N = 100, 500, 1000, 5000$

# time steps N	100	500	1000	5000
	without checkpointing			
tape size (KByte)	4.388,7	32.741,9	92.484,7	1.542.488,1
	with checkpointing			
tape size (KByte)	79,3	237,3	434,8	2.014,9
checkpoint size (KByte)	11,4	56,2	112,2	560,2

Table 1.1: Memory requirements for $N = 100, 500, 1000, 5000$

the main memory ($N = 5000$). The remaining case from Fig. 1.11 ($N = 500$) depicts a situation where only the tape and a small number of the most recent checkpoints can be kept within the L2-cache. Hence, a well chosen ratio between N and c causes in this case a significantly smaller recomputation rate and results in a decreased overall runtime, making the checkpointing attractive.

Obviously, the time stepping procedure used for the integration of the state equation may rely also on some adaptive steering of the size of the time steps. Then, the number of time steps performed is known only after the complete integration. Hence, an offline checkpointing is intractable. Instead, one may apply a straightforward checkpointing by placing a checkpoint each time a certain number of time steps has been executed. This approach transforms the uncertainty in the number of time steps to a uncertainty in the number of checkpoints needed and is used for example by CVODES [132]. However, when the amount of memory per checkpoint is very high one certainly wants to determine the number of checkpoints required a-priori. Having a fixed number of checkpoints to store intermediate states but an unknown number of time steps for which the adjoint has to be computed on the base of the forward trajectory, one has to decide on the fly, i.e., during the forward integration, where to place the checkpoints. Hence, without knowing how many time steps are left to perform, one has to analyze the current distribution of the checkpoints. Depending on the time steps performed so far, one may then discard the contents of one checkpoint to store the current available state. Obviously, one may think that this procedure could not be optimal since it may happen that one reaches the final time just after replacing a checkpoint, in which case another checkpoint distribution may be advantageous. A surprising efficient heuristic strategy to rearrange the checkpoints was developed and analyzed in the diploma thesis [135]. Here, a checkpoint distribution is judged by computing an approximation of the overall re-computation cost caused by the current distribution. This number is compared with an approximation of the re-computation cost if one resets a checkpoint to the currently available state. Despite the fact that significant simplifications are made for approximating the required re-computations, the resulting adjoint computations are amazingly cheap in comparison to an optimal checkpointing strategy. Here, one has to note that this optimal cost can be computed obviously only afterwards when the number of time steps is known. This heuristic for online checkpointing was implemented in the procedure arevolve [95]. However, a main drawback of arevolve is that it is so far not possible to prove for this algorithm an upper bound on the deviation from the optimal checkpointing schedule because a heuristic is used to judge the current checkpointing distributions. A new procedure for online checkpointing that yields a provable time-optimal adjoint computation for a given number of checkpoints was proposed in cooperation with the author of this thesis in [89] and is currently expended to more general checkpointing approaches.

1.3.2 Exploitation of Sparsity

Quite often, the required derivative matrices of first and second order are sparse. For example, the discretization of partial differential equations describing the considered problem may yield sparse Jacobians and Hessians. When an optimization is coupled with the simulation it is important to take the sparsity information into account for maintaining the efficiency of the overall process. Therefore, some of the tools for nonlinear optimization assume already that the user provides the Jacobians and Hessians already in a sparse format, see, e.g., [143, 146].

For the efficient computation of sparse Jacobian matrices, there are two different approaches: If the sparsity pattern of the sparse Jacobian is known, well-established coloring algorithms in combination with AD allow its economical evaluation by compression techniques. Alternatively, one may use the computational graph defined by the function evaluation and apply elimination techniques to generate efficient evaluation methods for the whole Jacobian. Naturally, the second alternative can be also applied for dense Jacobians. In the remainder of this section, the two approaches are illustrated in more detail.

The overall process for computing a sparse Jacobian using compression techniques is illustrated in Fig. 1.12. In step 1 of the process, the sparsity pattern of the Jacobian is computed. It may also be needed to set up data structures

$$\begin{tikzcd} f \arrow[r, "\textcircled{1}"] & P \arrow[r, "\textcircled{2}"] & S \arrow[r, "\textcircled{3}"] & A = JS \text{ or } A = S^T J \\ \text{function} & \text{sparsity pattern} & \text{seed matrix} & \text{compressed form} \end{tikzcd}$$

Figure 1.12: Computing sparse Jacobians with compression techniques

for an efficient storage and factorization of the sparse Jacobian. Compared to the evaluation of the full Jacobian $F'(x)$ in real arithmetic for a function $y = F(x) \in \mathbb{R}^m$ with $x \in \mathbb{R}^n$, computing the Boolean matrix $P \in \{0,1\}^{m \times n}$ representing its sparsity pattern in the obvious way requires considerably less runtime and memory. For that purpose, the AD tools ADOL-C and TAF provide the propagation of so-called Boolean patterns through the function evaluation. Here, one may simply use a combination of unit vectors. More sophisticated strategies were study for example in [74]. In step 2, the task of finding a suitable seed matrix $S \in \mathbb{R}^{n \times p}$ or $S \in \mathbb{R}^{m \times p}$ such that p attains a minimal value is closely related to graph coloring problems which are well known to be NP-hard. Fortunately, various algorithms based on heuristics have been developed that generally yield very good approximations to the optimal coloring. For a comprehensive overview of this topic the reader is referred to [56]. The entries of the Jacobian J can be recovered from the compressed matrix A *directly*, i.e., without requiring any further arithmetic, or via *substitution*, where an additional arithmetic work is required. In general, a seed matrix suitable for a substitution method has a smaller number of columns or rows p, respectively, compared to one suitable for a direct method, an advantage attained at the cost of a slightly higher computational effort involved in the matrix reconstruction step.

Ideally, step 1, i.e., the computation of the sparsity pattern P and step 2, i.e., the calculation of the so-called seed matrix S using graph coloring have to be performed only once. Subsequently, in step 3, the compressed representation of the sparse Jacobian can be computed using a vector version of AD. Finally, the matrix entries of the sparse Jacobian have to be calculated with a suitable recovery strategy.

As an alternative, the calculation of sparse Jacobians may also rely on elimination rules for the corresponding computational graph as defined in Subsec. 1.2.1. The aim of these elimination techniques is the derivation of an bipartite graph with the following properties: First, the independent variables and the dependent variables form two disjoint sets. Second, the labels of the edges connecting these sets represent the corresponding entries of the Jacobian. They should be computed with as less computation effort as possible. For the coordinate transformation example, the corresponding bipartite graph is shown by Fig. 1.13, where $c_{i,j}$ denotes the partial derivative $\frac{\partial v_i}{\partial v_j}$. There are several approaches to derive the bipartite graph from the original one, as for example forward edge elimination, backward edge elimination, forward vertex elimination, and backward vertex elimination [70]. These techniques still rely on the forward or backward propagation of information through the compu-

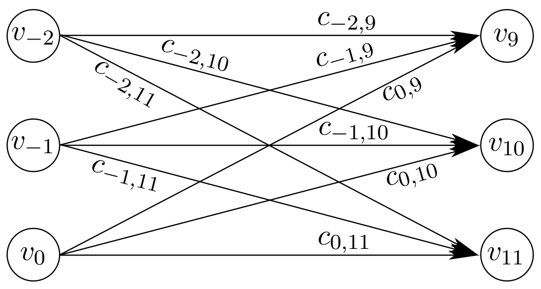

Figure 1.13: Bipartite computational graph for coordinate transformation

tational graph. Here, it can be shown that the back elimination corresponds to the forward mode and the forward elimination to the reverse mode as presented in Sec. 1.2, see [70]. Straightforward generalizations of these techniques are so-called cross country elimination procedures, where edges or vertices are eliminated in an order that does not necessarily correspond to the numbering of the vertices or edges. These elimination techniques have in common that they yield a bipartite graph representing the entries of the Jacobian. However, they may differ significantly in the number of operations required to compute the desired partial derivatives. It is proved that the problem of finding an elimination order that yields the Jacobian entries at minimal computational cost is NP-hard for a general function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ [112]. Therefore, there has been an extensive research on the derivation of efficient heuristics including for example greedy Markowitz approaches or genetic algorithms, see, e.g., [110, 111].

The exploitation of sparsity for the computation of Jacobians has been an area of active research over the last years. Several algorithms following the approaches described above have been developed and analyzed theoretically, see, e.g., [56, 57, 97]. Furthermore, there exist already some software packages to apply these approaches, see, e.g. [113, 154]. The situation changes completely if the sparsity of a Hessian matrix has to be exploited. So far, only AMPL can compute the structural information about the Hessian automatically based on the partial separability of the function to be differentiated [54]. Subsequently, this sparsity information is exploited for the efficient computation of sparse Hessians. As an alternative, the calculation of sparse Hessians may also rely on elimination rules for the computational graph of the Hessian. This approach was first considered in [43] and is still subject of current research.

As alternative a compression technique similar to the one shown in Fig. 1.12 can be used also to compute a compressed form of the sparse Hessian and to reconstruct the desired Hessian entries afterwards. For this purpose, an algorithm to compute the sparsity pattern of a Hessian based on the propagation of index domains and nonlinear interaction domains along with the function evaluation was proposed for the first time by the author in [150]. The results of the paper will be sketched here briefly and are also part of the thesis at hand (see Chapter 3). The index domains $\mathcal{X}_i$ are defined for each intermediate value

v_i computed during the function evaluation such that

$$\left\{0 \le j \le n \,:\, \frac{\partial v_i}{\partial x_j} \not\equiv 0\right\} \subseteq \mathcal{X}_i \tag{1.5}$$

holds [70]. If equality holds in (1.5), the index domain $\mathcal{X}_i$ contains for the intermediate value v_i the indices of all independent variables x_j which influence the value of v_i. A proper subset relation will occur in (1.5) if degeneracies arise in the function evaluation.

The nonlinear interaction domains $\mathcal{N}_i$ are defined for each input variable x_i, $1 \le i \le n$, such that

$$\left\{0 \le j \le n \,:\, \frac{\partial^2 y}{\partial x_i \partial x_j} \not\equiv 0\right\} \subseteq \mathcal{N}_i \tag{1.6}$$

is valid [150]. Hence, if equality holds in (1.6) the nonlinear interaction domains $\mathcal{N}_i$ contains for the independent variable x_i the indices of all independent variables x_j that influence the value of the dependent variable y together with the independent variable x_i in a nonlinear way. Here again, degeneracies may cause a proper subset relation in (1.6). Furthermore, equality may not hold if dead ends are contained in the function evaluation; that is, if there are intermediate variables v_i that were computed but not subsequently needed to calculate y. After the computation of the nonlinear interaction domains, the sparsity pattern of the Hessian can be easily constructed using the entries of the set $\mathcal{N}_i$.

Using an operator overloading approach for the implementation of AD, the index domains and the nonlinear interaction domains can be computed easily using the generated internal function representation. A detailed description of the updating procedure can be found in [150]. If a proper subset relation occurs for the index domains or the nonlinear interaction domains, the algorithm derived in [150] will yield an overestimate for the sparsity pattern, which results in an increase in runtime but not in incorrect derivative values. The theoretical analysis contained in [150] yields the following complexity result

Theorem 1.3.1 (Computing Sparsity Patterns of Hessians)**.** *Let $OPS(NID)$ denote the number of operations needed to generate all $\mathcal{N}_i$, $1 \le i \le n$ by Algorithm II as presented in Sec. 3.2.3. Then, the inequality*

$$OPS(NID) \le 6(1+\hat{n}) \sum_{i=1}^{l} \bar{n}_i$$

is valid, where l is the number of elemental functions evaluated to compute the function value, $\bar{n}_i = |\mathcal{X}_i|$, and $\hat{n} = \max_{1\le i\le n} |\mathcal{N}_i|$.

Similar to the compression technique for sparse Jacobians, the product of a derivative matrix and a seed matrix has to be evaluated to compute the compressed sparse Hessian. So far, only the scalar second order adjoint mode to compute Hessian$\times$vector products was available, see, e.g., [70]. In [150], the author presents for the first time also a vector version of the second order adjoint mode and a corresponding complexity estimate. It was shown that the upper bound on the runtime

$$\text{TIME}(H(x)S) \le \omega_{soad}\, p\, \text{TIME}(f(x)) \quad \text{with} \quad \omega_{soad} \in [7,\, 10].$$

using the scalar order adjoint mode to compute $H(x)S$ for a seed matrix $S \in \mathbb{R}^{n\times p}$ can be reduced to the upper bound

$$\text{TIME}(H(x)S) \le \omega_{soadp}\text{TIME}(f(x)) \quad \text{with} \quad \omega_{soadp} \in [4+3p, 4+6p].$$

This saving is comparable to the runtime reduction that can be achieved by switching from the scalar forward mode to the vector forward mode or from the scalar reverse mode to the vector reverse mode for computing first order derivatives.

The computation of the sparsity pattern as well as the vector version of the second order adjoint mode were incorporated by the author in the AD tool ADOL-C for numerical tests. Currently, the influence of the two basic compression techniques, namely the direct approach and an approach based on substitution, on the overall computing time for the computation of a sparse Hessian are studied and will be published in the near future [55]. The experimental results contained in that paper show that sparsity exploitation via compression techniques enables one to affordably compute Hessians of dimensions that could not have been computed otherwise. For sizes where dense Hessian computation is at least possible, the saving in runtime obtained by exploiting sparsity via coloring is drastic. Of the two sparsity-exploiting approaches, an acyclic coloring based substitution method is found to be faster, considering the overall process.

1.3.3 Differentiation of Fixpoint Iterations

The black-box differentiation of iterative processes may be a critical point with respect to the accuracy of the obtained derivatives but also with respect to the computational complexity in terms of runtime and memory requirement. One first remedy for time integration, i.e. the technique of checkpointing, was already presented in Subsec. 1.3.1. Additionally, there are also several results concerning the efficient and exact computation of derivative information for a second class of iterative processes, namely fixpoint iterations, see, e.g. [61]. In these cases, so far two different techniques are applied to overcome the problems caused by black-box differentiation. First, one may employ a so-called piggy-back approach for the derivative calculation as presented in [72]. Here, derivative information is propagated with the function evaluation also for the reverse mode differentiation to compute adjoint information. By this means, the storage of all intermediates computed in all fixpoint iterations is replaced by the storage of the intermediates of only one iteration at any time since the adjoint of each iteration is computed immediately with the iteration step. Similar methods are known as one-shot optimization and used for example in aerodynamic shape optimization, see, e.g., [86, 98].

As alternative, one may split the fixpoint iteration and the corresponding derivative computation. Hence, first the fixpoint iteration is performed up to a certain accuracy to obtain feasibility. Subsequently, the derivatives of this iterative process are computed up to the desired exactness to apply for example a gradient-based optimization. Using the forward mode of AD, this splitting can be modified such that the propagation of the derivative calculation is performed along with the fixpoint iteration only after a certain accuracy is reached in the undifferentiated process. Then, the fixpoint iteration is stopped as soon as the desired exactness of the function evaluation and the derivative information is reached. For an application of this strategy see, e.g., [60]. The strategy changes when computing discrete adjoints with the reverse mode of AD after the evaluation of a fixpoint iteration. Then a corresponding iterative process, the so-called reverse accumulation, for the calculation of the discrete adjoint and an appropriated steering of the derivative computation to ensure a desired accuracy as proposed [31] can be used. This algorithm is implemented in TAF for the efficient computation of discrete adjoints [60]. Only recently, a similar facility was incorporated in cooperation with the author in ADOL-C. It has been already applied successfully for the optimization of an airfoil in a cooperation with the DLR Braunschweig [53, 130].

1.4 Discrete Adjoints versus Continuous Adjoints

The reverse mode of AD is applied to function evaluations given as computer program to calculate the desired discrete adjoint information. Hence, AD is applied to compute the exact derivative of a discrete approximation of the considered function in its analytical or continuous form. Therefore, the reverse mode of AD does not yield necessarily an approximation to the exact derivatives of the continuous formulation of the function to be differentiated but consistent derivative information for its discrete representation as computer program. However, in numerous cases AD provides exactly what the user of AD requires. Nevertheless, the observation explained above yields immediately the question about the relation of the discrete derivative information provided by AD and the continuous derivatives that can obtained from the continuous problem formulation.

1.4.1 Two Different Optimization Approaches

In the literature, the resulting optimization approaches are frequently called *discretize-then-optimize* methods when using the consistent discrete derivatives provided for example by AD and *optimize-then-discretize* methods when the derivatives are based on the continuous formulation using for example the sensitivity or adjoint differential equation. So far, each of the two different approaches is applied for optimization purposes when adjoint information is required due to structure of the considered problem.

The optimize-then-discretize approach is used especially for PDE-constraint optimization problems, see, e.g., [5, 52]. Recently, the continuous approach has been extended for optimal control problems to avoid even the discretization of the control completely, see, e.g., [41].

One the other hand, frequently only a discrete description of the function as computer program is available for optimization purposes. For these cases, consistent discrete adjoints could be derived by hand in an error-prone and time-consuming process, since these adjoint discretization schemes may not coincide with the integration schemes used for the state equations. Corresponding results can be found, for example, in [18, 44, 84] for optimal control problems based on ODEs, and in [2, 34, 109, 137] for the optimization of a PDE-based model. To avoid the hand-coding of discrete adjoints, the reverse mode of automatic differentiation can be applied. Furthermore, the usage of AD ensures an automated handling of boundary values in a consistent way.

Since the optimization algorithm often differs only in the generation of the derivative information, the discussion about the two different approaches can be frequently reduced to the choice between discrete adjoint information for the discretize-then-optimize methods and continuous adjoint information for the optimize-then-discretize methods. The consequences of the two different approaches for an overall optimization process are sketched in Fig. 1.14. As

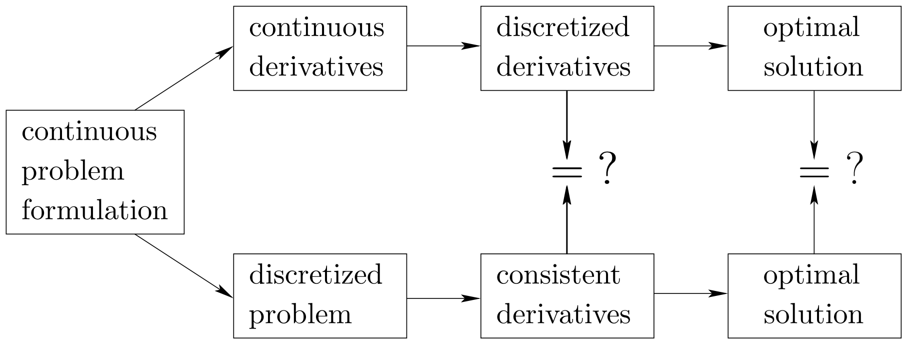

Figure 1.14: Optimize-then-discretize versus discretize-then-optimize

can be seen, the first question is whether the derivative information provided for the optimization process coincides and the second question is whether the same optimal solution can be achieved. Examples for discrepancies can be found frequently in the literature, see, e.g., [30] for the adjoint computation in a quite general setting and [2, 34] for a general PDE-based optimal control problem when using stabilized finite elements methods.

It was mentioned already in [68], that the reverse mode of AD yields a discrete adjoint somehow related to the continuous adjoint equation. However, so far only a limited theoretical analysis of the discrete adjoint information provided by AD is available. There are at least two questions to be answered. First, the relation between the discrete adjoints generated by AD and the continuous adjoints belonging to the discrete formulation of the function under consideration has to be studied. In [47], first theoretical aspects were studied for optimal control problems governed by ODEs. For optimal control problems based on ODEs, a further study of the subject is contained in [67]. This paper, which is also part of this thesis (see Chapter 4), compared for the first time the gradient information obtained by the optimize-then-discretize approach with the discrete adjoints provided by AD in an discretize-then-optimize approach. As shown in this paper, the difference of the obtained derivative information occurs if an interpolation of the control is required by the applied Runge-Kutta method. Obviously, these interpolations based on the values of the control influence the value of the objective function in the discrete function formulation. However, the interpolation can not be avoided for a recursive discretization used by the discretize-then-optimize method. The influence of the interpolation is completely neglected by the optimize-then-discretize approach. Besides this analysis of the different adjoint information obtained by the two approaches, the influence on the overall optimization process is also shown in [67]. The contribution of the author of this thesis to the paper [67] comprises all parts that cover the derivative information generated by AD. This includes 4.3.4, parts of 4.4, and the analysis of the different derivative information and the analysis of the reason for these differences contained in 4.5. Furthermore, the remaining parts of the Sects. 4.5 and 4.3 were derived and written in a strong cooperation with the second author of [67].

The paper [149] is a direct continuation of the work presented in [67] and also part of this thesis (see Chapter 5). For recursive discretizations, the reverse mode of AD provides the consistent objective gradient which depends on the forward integration scheme as discussed, for example, in [48]. That is, the reverse mode of AD yields the exact discrete gradient information for the chosen discretization of the state equation. However, one has to note that AD computes the exact derivative of an approximation of the objective and may not yield an approximation to the exact derivatives of the objective. The purpose of the paper [149] is to analyze this discrepancy and its impacts for the recursive discretization in more detail. That means, for a fixed control function the convergence rate of the discrete adjoint associated with the corresponding discretized control problem to the corresponding continuous adjoint is analyzed. For that purpose, the discretization schemes automatically derived by AD to integrate the sensitivity equation and the adjoint differential equation of the underlying optimal control problem are presented and analyzed with respect to their theoretical properties. In detail, it is shown in [149] that the discretization method obtained for the sensitivity equation inherits the convergence properties of the discretization scheme used for the state equation. A similar result is proved for the adjoint differential equation provided that some additional assumptions on the coefficients of the original discretization method are fulfilled. Moreover, the computation of the required gradient information using the approximations of the sensitivity equation and the adjoint differential equation, respectively, is discussed. Alternatively, the gradient information provided by AD can be used as the exact discrete derivative information, which need not coincide with the one based on the approximation of the sensitivity and adjoint differential equation, respectively.

Hence, we choose for a given fixed control the discretize-then-optimize approach and compare the approximation that we achieve with the continuous solutions of the optimize-then-discretize method. Therefore, the results presented in [149] are related to [44, 84], where the convergence rate of the solution of the discretized control problem to the solution of the continuous problem is studied for the full discretization approach.

For PDE-constrained optimization processes the relation of discrete adjoint information provided for example by hand-coding or the usage of appropriate discretization schemes for the adjoint differential equation were studied in several papers, see, e.g., [2, 6, 7, 34]. Currently, similar topics are studied in a project supervised by the author of this thesis to examine the effects of AD in this context. The aim of this project is a semi-continuous adjoint generation, for example if AD is applied only for the reference element of a finite element discretization and not for the full FEM code. Then, the differentiation of grid structures or even the generation of the finite element grid could be avoided making this approach very attractive with respect to memory requirement and runtime.

1.4.2 Recomputation versus Storage for Continuous Adjoints

As explained already in Section 1.3.1 for the discrete adjoint approach, the intermediate results have to be either stored or recomputed. A similar situation occurs when the continuous adjoint differential equation is used to derive gradient information as sketched briefly in the remainder of this subsection.

Let y denote the state variables describing a velocity field, $p \in \mathbb{R}$ the pressure and u the control variables. A instationary, incompressible, viscous flow in a time-space cylinder $(0,T) \times \Omega$ is modeled by means of the Navier-Stokes equations. Together with the target function, one obtains the problem description

$$\min J(y,u) := \frac{1}{2}\int_0^T \int_\Omega |y(x,t) - y_d(x,t)|^2 dxdt + \frac{\mu}{2}\int_0^T \int_\Omega |u(x,t)|^2 dxdt$$
$$\begin{aligned} -\nu\Delta y + y \cdot \nabla y + \nabla p &= u && \text{in } (0,T) \times \Omega, \\ \text{div } y &= 0 && \text{in } (0,T) \times \Omega, \\ y|_{t=0} &= y_0. \end{aligned}$$

We assume that the velocity field is subject to adequate boundary conditions and y_d is a desired state. Together with the adjoint pressure ξ the corresponding adjoint variables λ then satisfies the system

$$\begin{aligned} -\tfrac{\partial \lambda}{\partial t} - \nu\Delta\lambda - (y \cdot \nabla)\lambda + (\nabla y)^T\lambda + \nabla\xi &= y - y_d && \text{in } (0,T) \times \Omega, \\ -\text{div } \lambda &= 0 && \text{in } (0,T) \times \Omega, \\ \lambda(x,t) &= 0 && \text{on } \partial\Omega \times (0,T), \\ \lambda(x,T) &= 0 && \text{in } \Omega, \end{aligned}$$

with suitable boundary conditions, see, e.g., [139]. As can be seen, the partial differential equation for the adjoint variable has to be integrated backwards in time due to the terminal condition. Additionally for a given control u, the computation of λ requires knowledge of the state $y(u)$ on the whole time horizon. Therefore, the storage of $y(u)$ may form a serious bottleneck for large time horizons where y represents a 2- or 3-dimensional flow velocity field.

There are only few contributions to control of the instationary Navier-Stokes equations that tackle the storage problem. Frequently, a time-stepping procedure with constant step sizes is applied for the integration of the time-dependent state equation allowing an offline checkpointing approach. The technique of equidistant checkpointing and repeated forward integration, also known as windowing, is discussed by Berggren and coauthors in [9, 10] or by Becker and coauthors in [6]. A corresponding study applying the binomial offline checkpointing is contained in [96], which is also part of the thesis at hand (see Chapter 6). In this paper the offline checkpointing procedure revolve originally designed for AD purposes is applied for reducing the memory requirement of the continuous adjoint computation. Using the proposed checkpointing technique, a reduction of the storage requirement of two orders of magnitude is observed, whereas the resulting slow down factor of the adjoint calculation caused by repeated forward integrations of the Navier-Stokes equations lies only between 2 and 3. In addition to this extension of binomial checkpointing also for the optimize-then-discretize approach, a new proof of optimality is given, see Sec. 6.3. It uses the new concept of frequency numbers first introduced in [135]. The new technique simplifies the proof of optimality significantly and yields in addition so far unknown properties of the optimal checkpointing strategies as for example an explicit formula of the frequency numbers, see Theorem 6.3.4. The contribution of the author of the thesis to the paper [96] comprises the further development of the presented checkpointing theory and of the procedure revolve as well as the analysis and interpretation of the numerical results that were obtained in a strong cooperation of the author of this thesis with the first author of [96].

1.5 Some Recent Mathematical Algorithms Based on Adjoint Information

Motivated for example by the complexity result for finite differences as presented in Sec. 1.1, it was widely assumed that the product of a vector and a Jacobian, i.e., $\bar{y}^T F'(x)$, can not be computed or approximated with low computational cost. As a consequence, major work was dedicated to so-called transposed-free algorithms, e.g., for the iterative solution of linear or nonlinear equations. However, using the reverse mode of AD one can evaluate vector$\times$Jacobian products without even coming close to forming a matrix within working accuracy. Additionally, the continuous adjoint approach for optimization problems modeled by differential equations became more popular. These two developments result in a more intensive work on adjoint-based algorithms either by employing the adjoint information directly for the optimization, e.g. by using gradient based-optimization [94, 130], or by designing new algorithms based on adjoint information. Therefore, the next two subsections present some recent results for adjoint-based mathematical methods.

1.5.1 Adjoint-based Quasi-Newton Updates

In contrast with the theoretical and practical attractions of the BFGS formula for positive definite Hessians, secant updates for non-symmetric Jacobians have rarely met with success across a wide range of problems. One possible explanation is that least change updates like the good and the bad Broyden formula are strongly dependent on inner product norms and hence the scaling in the domain or range of the underlying vector function. In contrast the BFGS and all other updates of the Broyden class including the SR1 formula are known to be invariant with respect to linear transformations on the variable domain, provided the initial matrix for the approximation is adjusted accordingly.

Exploiting the fact that AD allows to evaluate Jacobian×vector products and vector×Jacobian products exactly one can derive an approximation of the Jacobian that combines for the first time heredity and a least change property as shown in [128] for the solution of nonlinear systems. For this purpose, a direct secant condition as well as an adjoint secant condition are exploited. It is even possible to analyze for a class of adjoint-based quasi-Newton updates the rate of convergence in more detail [75] and to show global convergence if an adapted line search procedure is applied as globalization [129].

For solving a system of nonlinear equations, there is some freedom in specifying the adjoint secant condition [131]. When approximating the Jacobian of an equality-constrained optimization problem, the situation changes and the adjoint secant condition employs information on the adjoint variables of the optimization problem. Furthermore, adjoint-based quasi-Newton updates were first proposed for the solution of constraint optimization as two-sided rank 1 (TR1) update in [77] and in a slightly different form in [83]. For these reasons, adjoint-based update formulas will be presented here in more detail in the context of optimization problems.

Assume that an optimization strategy based on Sequential Quadratic Programming (SQP) is applied to solve a nonlinear programming problem of the general form

$$\min_{x \in \mathbb{R}^n} f(x) \qquad \text{s.t.} \qquad c(x) = 0, \tag{1.7}$$

where the objective $f : \mathbb{R}^n \to \mathbb{R}$ and the constraints $c : \mathbb{R}^n \to \mathbb{R}^m$ with $n \geq m$ are given smooth functions. Note that the Jacobian of the constraints

$$A(x) = (\nabla c_1(x), \ldots, \nabla c_m(x))^T \in \mathbb{R}^{m \times n}$$

may be dense, i.e., we do not assume that $A(x)$ has any structure. In the absence of inequality constraints, the SQP method can be observed as Newton's method applied to the KKT conditions of (1.7). The Newton step at iteration k can be written as:

$$\begin{bmatrix} B(x_k, \lambda_k) & A(x_k)^T \\ A(x_k) & 0 \end{bmatrix} \begin{bmatrix} s_k \\ \sigma_k \end{bmatrix} = - \begin{bmatrix} g(x_k, \lambda_k) \\ c(x_k) \end{bmatrix} \tag{1.8}$$

where λ_k denotes the current Lagrange multipliers, $B(x, \lambda)$ is the Hessian of the Lagrange function

$$L(x, \lambda) = f(x) + \lambda^T c(x)$$

and $g(x, \lambda) \equiv \nabla_x L(x, \lambda)$.

It is widely assumed that the Jacobian of the constraints is available using either finite differences or analytical methods such as the sensitivity equations.

However, both approaches may result in very time-consuming computations, especially if the Jacobian of the constraints is dense or unstructured. Therefore, we may apply the TR1 update to generate an approximation A_k of the exact constraint Jacobian $A(x_k)$. The TR1 update is defined by

$$A_{k+1} = A_k + \delta_k r_k \rho_k^T \tag{1.9}$$

where

$$\begin{aligned} y_k &\equiv c(x_k + s_k) - c(x_k), & \mu_k^T &\equiv g^T(x_k + s_k, \lambda_k + \sigma_k) - g^T(x_k + s_k, \lambda_k) \\ r_k &\equiv y_k - A_k\, s_k, & \rho_k^T &\equiv \mu_k^T - \sigma_k^T A_k, \\ \sigma_k &\equiv \lambda_{k+1} - \lambda_k \in \mathbb{R}^m \end{aligned}$$

to fulfill both the direct secant condition

$$A_{k+1}\, s_k = y_k + (\delta_k \rho_k^T s_k - 1) r_k \approx y_k \approx \nabla c(x_k)\, s_k$$

as well as adjoint (i.e., transposed) secant condition

$$\sigma_k^T A_{k+1} = \mu_k^T + (\delta_k \sigma_k^T r_k - 1)\rho_k^T \approx \mu_k^T$$

approximately for any $\delta_k \in \mathbb{R}$. When $\rho_k^T s_k = \sigma_k^T r_k \neq 0$ the two secant conditions on A_{k+1} are consistent and one can check that (1.9) with $\delta = 1/\rho_k^T s_k = 1/\sigma_k^T r_k$ is the only rank-one update satisfying them. It is possible to reconstruct the exact Jacobian $A(x_k)$ with m TR1 updates for the fixed iterate x_k as shown in the following proposition:

Proposition 1.5.1. *Let the iterate x_k be fixed and A_k be given. Choose linearly independent vectors v_i and w_i for $i \in \{0, \ldots, m-1\}$ such that with $z_i \equiv A(x_k)v_i$, $\tau_i \equiv w_i^T A(x_k)$, $A_{k,0} = A_k$, and*

$$A_{k,i+1} \equiv A_{k,i} + \frac{(z_i - A_{k,i}v_i)(\tau_i^T - w_i^T A_{k,i})}{(\tau_i^T - w_i^T A_{k,i})v_i} \tag{1.10}$$

the inequality $(\tau_i^T - w_i^T A_{k,i})v_i \neq 0$ holds. Then $A_{k,m} = A(x_k)$.

Proof: Due to the assumption $(\tau_i^T - w_i^T \tilde{A}_i)v_i \neq 0$ the TR1 update (1.10) is always well defined. First, we show by induction that $w_j^T A_{k,l} = \tau_j$ for $j = 0, \ldots, l-1$. By definition, the TR1 update satisfies the secant condition. Therefore, we have $w_0^T A_{k,1} = \tau_0$. Now, we assume that $w_j^T A_{k,l} = \tau_j$ holds for some value $l \geq 1$ and show that it holds also for $l+1$. We have

$$\begin{aligned} w_j^T(z_l - A_{k,l}v_l) &= w_j^T z_l - w_j^T A_{k,l}v_l = w_j^T A(x_k)v_l - w_j^T A_{k,l}v_l \\ &= w_j^T A(x_k)v_l - w_j^T A(x_k)v_l = 0 \end{aligned}$$

for all $j < l$. Using (1.10), we obtain

$$w_j^T A_{k,l+1} = w_j^T A_{k,l} = \tau_j \qquad \text{for all} \quad j < l$$

and $w_j^T A_{k,l+1} = \tau_l^T$ due to the secant condition. If m TR1 updates are performed, it follows that

$$w_j^T A_{k,m} = w_j^T A(x_k)$$

holds for the linearly independent steps w_j, $j = 0, \ldots, m-1$. This yields $A_{k,m} = A(x_k)$. $\square$

To prevent excessive variations in the determinant of $A_k A_k^T$ the factor δ_k can be limited, which effectively dampens the Jacobian update, see [79]. We can maintain a factorized null space representation of the approximated derivative information during the whole optimization procedure. Hence, instead of updating A_k directly, we may update the corresponding matrices

$$L_k \in \mathbb{R}^{m\times m}, \; Y_k \in \mathbb{R}^{m\times n}, \; Z_k \in \mathbb{R}^{n\times d}$$

with $d = n - m$, Y_k and Z_k orthonormal and L_k lower triangular such that

$$A_k = (L_k \, Y_k), \quad A_k Z_k = 0, \quad Y_k Z_k = 0.$$

For details on the update of this factorized representation, see [79].

1.5.2 Trust-region Algorithms with Inexact Jacobian Matrices

For numerous equality-constraint optimization problems, the underlying application yields very large but also well-structured and sparse derivative matrices. For example, this is quite often the case for PDE-constrained optimization. The exploitation of these structural properties allows the development of very efficient optimization algorithms, see, e.g., [143, 146]. However, there is also a wide range of applications where the derivative matrices have somehow orthogonal characteristics, i.e., they are of rather small size but dense. Examples for such a setting are Periodic Adsorption Processes (PAPs). PAPs are typically operated in a cyclic manner. These cycle models consist of bed models, PDAEs in time and space, solved for each step. After a relatively brief start-up period, the adsorption beds run in a cyclic steady state, that is, the bed conditions at the beginning of each cycle match those at the end of the cycle. This fact yields dense constraint Jacobians, where the time required for the computation of the Jacobian dominates the overall optimization process, see e.g., [101], where a reduced Hessian SQP approach was applied for the optimization of a PAP yielding a decomposition of (1.8). The Jacobian $A(x_k)$ was calculated from the direct sensitivity approach applied to the DAE system and an approximation of the reduced Hessian related to $B(x_k)$ was created by quasi-Newton updates. The solution of the optimization problem indicates that significant improvements are possible with modest computational effort. However, the bottleneck to more efficient calculations is the evaluation of direct sensitivities to obtain $A(x_k)$, which is a dense matrix. Hence, the efficient handling of the constraint Jacobian is essential to evaluate a proposed design, to analyze a process for safety, for controllability and operability, debottlenecking and retrofitting existing units, and for optimization of new design or existing installations.

The author of this thesis analyzed a class of trust-region sequential quadratic programming algorithms for the solution of minimization problems with nonlinear equality constraints in the paper [148] which is also part of this thesis (see Chapter 7). The proposed optimization method does not require the exact evaluation of the constraint Jacobian in each optimization step but uses only an approximation of this first-order derivative information. Hence, the presented approach is especially well suited for equality constrained optimization problems where the Jacobian of the constraints is dense. Globalization of the resulting nonlinear programming algorithm is provided by a composite step trust region approach with the tangential and normal steps and calculated by the approach of Byrd and Omojukun, see [117]. For composite-step trust-region methods that employ exact information, a comprehensive treatment of the convergence properties can be found in [36]. Implementations of the Byrd-Omojokun trust-region method are used successfully to solve equality constrained NLPs [4, 106]. Related implementations using augmented Lagrangian merit functions are proposed and analyzed in [46]. Extensions of this approach to a more general class of trust-region methods can be found in [42]. Box trust-region methods are analyzed in [64]. More recently, trust-region methods without penalty functions have been developed [49, 50, 51, 141].

The effects of inexact problem information on the global convergence of inexact SQP methods can be found, for example, in [100, 107, 144]. In a line search setting, the effects of inexact information on the global convergence are studied in [23]. For an inexact composite step trust-region SQP method a first proof of global convergence is given in [88], where the analysis is focused on inexactness arising from iterative system solves. The analysis and assumptions on inexactness presented in [148], i.e., Chapter 7, differ from [88] in the following way: We do not consider a splitting of the variables into state and control variables. Hence, we allow general unstructured approximations of the Jacobian $A(x)$ and the corresponding null space representation as well as inexactness due to iterative solves. The accuracy requirements for the presented first-order global convergence result are based on the feasibility and the optimality of the iterates. The corresponding criteria can be verified easily during the optimization process to adjust the approximation quality of the constraint Jacobian.

Recently, the trust region algorithm with inexact Jacobians was implemented using the two-sided rank one (TR1) update as proposed in [77]. The required gradients and Hessian-vector products are evaluated exactly using automatic differentiation. The numerical tests for several problems out of the CUTEr collection verify good practical performance of the proposed algorithm [151]. Additionally, an optimization problem based on a small simulated moving bed (SMB) system is studied in this paper. The cyclic behavior of the SMB system yields a constraint Jacobian that is dense. Hence, this optimization problem is representative for a whole class of problems where the evaluation and factoring of the constraint Jacobian is the dominant computational cost. As shown in [151], the problem involving a small simulated moving bed can be solved easily and efficiently using the proposed trust region algorithm based on inexact Jacobian information.

Chapter 2

Advantages of Binomial Checkpointing for Memory-reduced Adjoint Calculations

Andrea Walther and Andreas Griewank[1]

In M. Feistauer, V. Dolejší, P. Knobloch, and K. Najzar, eds.,
Numerical Mathematics and Advanced Applications,
Proceedings of ENUMATH 2003, Prague, pp. 834 – 843, Springer (2004)

Abstract:
Checkpointing techniques become more and necessary for the computation of adjoints. This paper presents the more common multi-level checkpointing as well as the less known binomial checkpointing. The checkpointing approaches are compared with respect to the number of time steps the adjoint of which can be calculated, the run-time needed for the adjoint calculation and the memory requirement. Some examples illustrate the shown results

2.1 Introduction

For many time-dependent applications, the corresponding simulations can be performed using ordinary or partial differential equations. Furthermore, quite often there are quantities that influence the result of the simulation. Throughout, we assume that these quantities are control functions, for example heating in and/or at the boundary of a domain. To compute an approximation of the simulated process for a time interval $[0, T]$, one applies an appropriate integration scheme given by

$$y_0 \;=\; y^0\,, \quad y_i \;=\; F_i(y_{i-1}, u_{i-1}, t_{i-1}) \qquad i = 1, \ldots, N\,,$$

where $y_i \in \mathbf{R}^n$ denotes the state and $u_i \in \mathbf{R}^m$ the control at time t_i for a time grid $t_0, \ldots, t_N$ with $t_0 = 0$ and $t_N = T$. The operator $F_i : \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R} \mapsto \mathbf{R}^n$

[1]Institute of Mathematics, Humboldt University Berlin, Germany

defines the time step to compute the state at time t_i. Note that we do not assume a uniform grid. To optimize a specific criterion or to obtain a desired state, the cost functional

$$\hat{J}(u) \quad = \quad J(y(u), u)$$

measures the quality of $y(u)$ and $u = (u_1, \ldots, u_N)$. Here, $y(u) = (y_1(u), \ldots, y_N(u))$ describes the dependence of the state y on the control u. For applying a calculus-based optimization method, one may use an adjoint integration

$$\bar{y}_N \quad = \quad 0\,, \quad \bar{y}_{i-1} \;=\; \bar{F}_i(\bar{y}_i, y_{i-1}, u_{i-1}, t_{i-1}) \qquad i = N, \ldots, 1\,, \tag{2.1}$$

motivated by the adjoint differential equation that belongs to the differential equation describing the state. Subsequently or concurrently, the desired derivative information $\hat{J}_u(u)$ can be reconstructed from $\bar{y} = (\bar{y}_0, \ldots, \bar{y}_N)$. The specific choice of the adjoint steps $\bar{F}_i$ depends on the forward integration and whether one prefers the continuous adjoint or the discrete adjoint formulation, see, e.g., [65, 76, 92]. For the purpose of this paper, it is only important to note that the adjoint integration has to be performed backwards in time and that the complete forward trajectory $y = (y_0, \ldots, y_{N-1})$ is required. Hence, storing all states $(y_0, \ldots, y_{N-1})$ during the forward integration and reading them in reverse order during the adjoint integration forms one simple possibility to overcome this difficulty. Then the computing time for the adjoint calculation consists of the evaluation of N time steps F_i storing the state y_{i-1} and the evaluation of N adjoint steps $\bar{F}_i$.

The storage requirement of the basic approach to calculate adjoints is proportional to the number N of time steps. If we want to calculate the adjoint of a real-world problem with thousands of time steps this memory requirement of the basic approach may become a serious problem. For example, for computing 3D flows with unstructured grids one may need easily 10 to 100 MBytes to store only one state vector y_i [102]. Therefore, it is reasonable to assume that due to their size, only a very limited number of intermediate states can be kept in memory. They may serve as checkpoints, such that the required information for the backward integration is generated piecewise during the adjoint calculation. Sections 2.2 and 2.3 present two different checkpointing techniques. The resulting run-times and memory requirements are compared in Section 2.4. Finally, some conclusions and an outlook are given in Section 2.5.

2.2 Uniform Checkpoint Distribution

To distribute the checkpoints equidistantly over the given number of time steps forms one obvious solution to the storage requirement problem. Subsequently the adjoints are computed for each of the resulting groups of time steps separately. Denoting the number of checkpoints used by c, the corresponding calculation of the adjoint values can be performed using the following algorithm where the counter i is identified with the state y_i:

Two-level Checkpointing

Initialization: Reserve space for c_1 checkpoints, store the initial state y_0 in the first one and set

$$c_2 = \begin{cases} \lceil N/(c_1+1) \rceil & \text{if} \quad c_1 \lceil N/(c_1+1) \rceil < N \\ \lfloor N/(c_1+1) \rfloor & \text{else} \end{cases}$$

Advance: Starting from the initial state, advance to state $c_1 \cdot c_2$ by performing the time steps F_i, $1 \leq i \leq c_1 \cdot c_2$. While integrating forward, store the states $(j-1)\,c_2$ in the checkpoints j for $j = 2, \ldots, c_1$.

Reverse:

```
do $p = c_1$, 0, -1
    Evaluate the time steps $F_i$, $p \cdot c_2 < i < N$ storing the states $i$,
    $p \cdot c_2 \leq i < N - 1$,
    perform the adjoint steps $\bar{F}_i$, $N \geq i > p \cdot c_2$ to calculate the adjoints,
    set $N = p \cdot c_2$, if $p > 0$ read the contents of checkpoint $p$.
end do
```

Fig. 2.1 sketches the two-level checkpointing for $N = 16$ time steps and $c = c_1 + c_2 = 6$. Throughout, the time steps are plotted along the vertical axis and the computing time required for the adjoint calculation is represented by the horizontal axis. Each solid horizontal line including the horizontal axis itself represents a checkpoint. The time, when a state is stored in a checkpoint, is marked with a black circle for the first level and with a black square for the second level. The slanted black lines represent the evaluation of time steps. The adjoint steps are drawn as dashed slanted lines. Finally, black arrows depict the usage of a state y_i for an adjoint step F_{i+1} without performing the corresponding time step F_i. This adjoint calculation is possible due to the assumed structure (2.1) of the adjoint steps. Note, that it may be required to evaluate F_N once to initialize the adjoints. This evaluation can be introduced right after the evaluation of F_{N-1} for $p = c_1$. For illustration purposes, we suppose throughout that all time steps and all adjoint steps have the same temporal complexity normalized to 1. However, to apply the presented optimal checkpointing techniques, only the identical temporal complexity of all time steps is required. In this example, 24 time steps are performed. Hence, the number of additional time step evaluations caused by the two-level checkpointing compared to the basic approach equals 9. Furthermore, at most 6 states have to be kept in memory.

The two-level checkpointing has been proposed several times in the literature, e.g., [29, 105], and is easy to implement. Naturally, one can apply two-level checkpointing repeatedly for the groups of time steps that are separated by equidistant checkpoints. This approach is called *multi-level checkpointing* [87] and sketched by Fig. 2.2 for the three-level case. The multi-level checkpointing is defined by the number of levels r, the number of checkpoints c_i that are uniformly distributed at level i, $i = 1, \ldots, r-1$, and the number of states

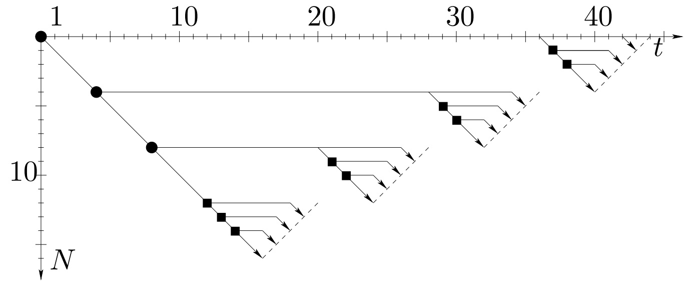

Figure 2.1: Two-level checkpointing for $N = 16$ time steps and $c = c_1 + c_2 = 6$ checkpoints

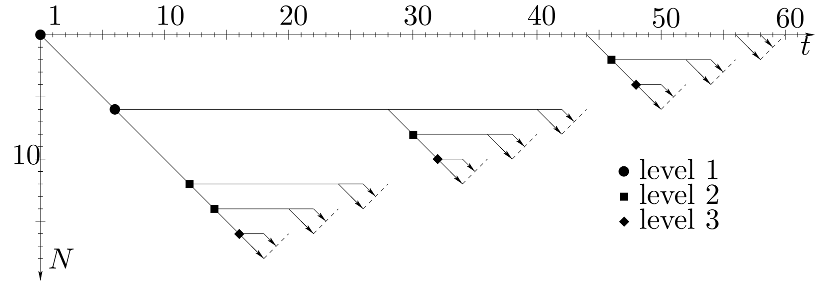

Figure 2.2: Three-level checkpointing for $N = 18$ time steps and $c = 5$ checkpoints

c_r that have to be stored at the highest level r. Hence, the parameters of the adjoint calculation shown in Fig. 2.2 are $c_1 = 2$, $c_2 = 2$, and $c_3 = 1$. For a given r-level checkpointing, one easily derives the following identities

$$N_r = \prod_{i=1}^{r}(c_i + 1), \quad M_r = \sum_{i=1}^{r} c_i, \quad T_r = \sum_{i=1}^{r} \frac{c_i\, N_r}{c_i + 1} = r\, N - \sum_{i=1}^{r} \prod_{\substack{j=1 \\ j \neq i}}^{r} (c_j + 1)\ ,$$

where N_r denotes the number of time steps for which the adjoint can be calculated using the specific r-level checkpointing. The corresponding memory requirement equals M_r. The number of time step evaluations required for the adjoint calculation is given by T_r, since at the first level $c_1 N_r/(c_1+1)$ time steps have to be evaluated to reach the second level. At the second level, one group of time steps is divided into $c_2 + 1$ groups. Hence, $c_2(N_r/c_1 + 1)/(c_2 + 1)$ time steps have to be evaluated in each group to reach the third level. Therefore, we obtain $(c_1 + 1)c_2(N_r/c_1 + 1)/(c_2 + 1) = c_2 N_r/(c_2 + 1)$ at the second level and so on. It follows that each time step F_i is evaluated at most r times. Hence, if we apply two-level checkpointing, each time step is evaluated no more than two times.

The two- as well as the multi-level checkpointing technique have the drawback that at each level the checkpoints are not reused. Each checkpoint stores at each level only one state and becomes idle as soon as the data that is stored in the checkpoint has been used for the adjoint calculation. A method that reuses the checkpoints as soon as possible is proposed in the next section.

2.3 Binomial Checkpoint Distribution

When one applies the checkpointing technique proposed in [76], the adjoint values are again generated piece by piece but only one state is employed for the adjoint calculation at any time. Therefore, the checkpointing procedure has to be adapted as follows:

```
Binomial Checkpointing

Initialization: Reserve space for $c$ checkpoints and store the initial state
$y_0$ in the first one.

do $p = N$, 1, -1

    Advance: Starting from the last checkpoint assigned, advance to
    state $p-1$.
    If checkpoints are free, set as many of them as possible to states $i$
    along the way.

    Reverse: Perform the adjoint step $\bar{F}_p$ to calculate the adjoint.
    If state $p-1$ is stored in a checkpoint, free the checkpoint up.

end do
```

The memory requirement of this checkpointing procedure equals $M_b = c$. Naturally, the question arises where one should place the checkpoints in the action "Advance" of the algorithm to minimize the number of time step evaluations. The application of the routine revolve ensures that the initiated checkpointing process is provably optimal with respect to the run-time increase for a given number of checkpoints [76]. More specifically, for the structure (2.1) of the adjoint steps considered here, the following complexity result holds:

Theorem 2.3.1. *Let N be the total number of time steps for which the adjoint has to be calculated. Suppose, up to c checkpoints are available at any time. Then the minimal number of time step evaluations needed for the adjoint calculation equals*

$$T_b \quad = \quad N\,r - \binom{c+r}{r-1},$$

where r the unique integer satisfying

$$\binom{c+r-1}{r-1} \; < \; N \; \leq \; \binom{c+r}{r}. \tag{2.2}$$

The proof of Theorem 2.3.1 (see [76]) constructs recursively checkpointing schedules that attain the minimal number T_b. For the optimal checkpointing procedures the positions of the checkpoints are given by binomial coefficients. This fact explains the name *binomial checkpointing.* Furthermore, the proof of Theorem 2.3.1 shows that each time step F_i is evaluated at most r times. Hence, r has the same meaning as in the previous section. It was proved earlier that

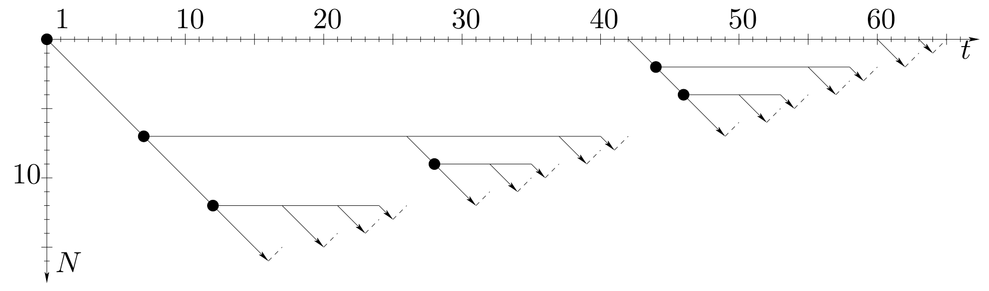

Figure 2.3: Binomial Checkpointing for $N = 16$ time steps and $c = 3$ checkpoints

a logarithmic growth of memory and run-time can be achieved using binomial checkpointing by providing an appropriate number of checkpoints [69].

The routine revolve implements the optimal binomial checkpointing and can be incorporated easily in an existing adjoint calculation [63, 96]. Moreover, one can build a heuristic based on revolve such that the adjoint calculation using binomial checkpointing becomes applicable also if the number of time steps is not known a-priori, e.g. due to adaptive time stepping, and/or if the temporal complexity of the time steps is not constant, e.g. due to implicit methods [95].

One optimal checkpointing schedule computed with revolve for $N = 16$ time steps and $c = 3$ checkpoints is shown in Fig. 2.3. Once more, it might be necessary to evaluate F_N once to initialize the adjoints. Since the situation is the same for the multi-level checkpointing and does not influence the results in the sequel, we ignore throughout the evaluation of F_N. For the example shown above, the number of time step evaluations equals $T_r = 33$. Compared to the two-level checkpointing, the computing time for the adjoint calculation increases by less than 50 %. Furthermore, only 3 states have to be kept in memory. Hence, the storage requirement is reduced by 50 %. The relation between the two checkpointing approaches will be discussed in more detail in the next section.

2.4 Comparison of Both Checkpoint Distributions

The integer r has the same meaning for both checkpointing approaches, namely the maximal number of times any particular time step F_i is evaluated during the adjoint calculation. Hence for comparing both approaches, assume at the beginning that r has the same value and that the same amount of memory is used, i.e. $M_r = M_b = c$.

Now, we examine the maximal number of time steps N^* for which an adjoint calculation can be performed using the two approaches. Assuming that r is a divisor of c and $M_r = c$, one obtains the identity

$$N_r^* \;=\; \left(\frac{c}{r}+1\right)^r \;=\; \left(\frac{c+r}{r}\right)^r \qquad \text{with} \qquad c_i = \frac{c}{r}, \quad i = 1, \ldots, r \; ,$$

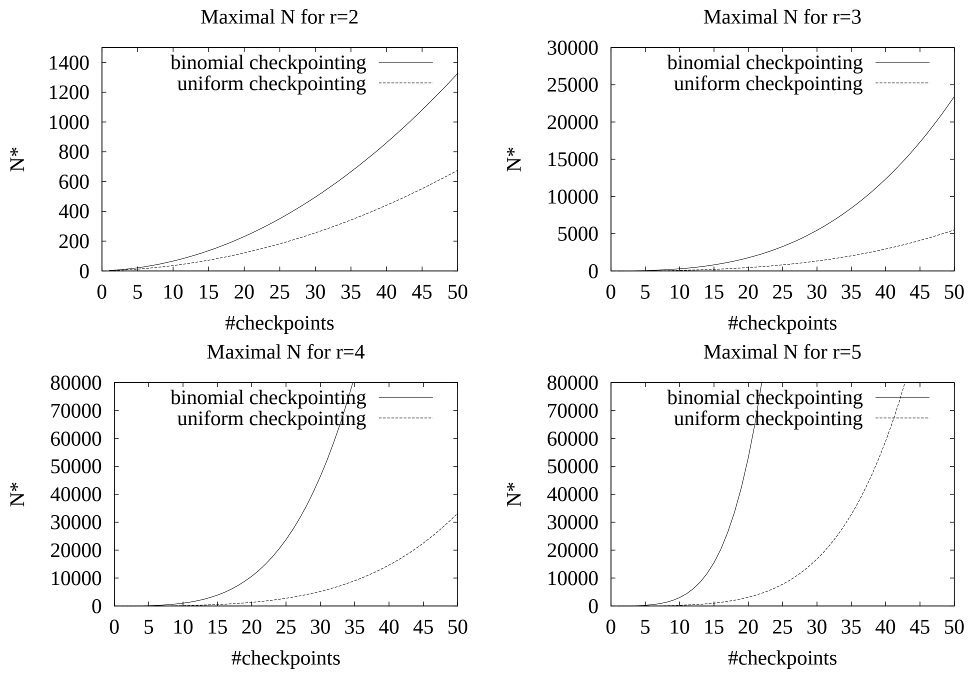

Figure 2.4: N_r^* and N_b^* for $r = 2, 3, 4, 5$

for the uniform checkpoint distribution because of the structure of N_r. Theorem 2.3.1 yields

$$N_b^* \;=\; \binom{c+r}{r} \;=\; \prod_{i=0}^{r-1}\left(\frac{c}{r-i}+1\right)$$

for the binomial checkpoint distribution. Obviously, one has

$$\frac{c}{r}+1 < \frac{c}{r-i}+1 \qquad \text{for } 0 < i \le r-1\ .$$

These inequalities yield $N_r^* < N_b^*$ if $r \ge 2$. Hence for all $r \ge 2$ and and a given c, binomial checkpointing allows the adjoint calculation for a larger number of time steps compared to uniform checkpointing. In more detail, using Stirling's formula we obtain

$$\frac{N_b^*}{N_r^*} \;\approx\; \binom{c+r}{r}\left(\frac{c}{r}+1\right)^{-r} = \frac{1}{\sqrt{2\pi r}}\left(\frac{c}{r}+1\right)^{c} \approx \frac{1}{\sqrt{2\pi r}}\exp(r)\ .$$

Hence, the ratio of N_b^* and N_r^* grows exponentially in r without any dependence on the number of available checkpoints. Fig. 2.4 shows N_r^* and N_b^* for the most important values $2 \le r \le 5$. Since r denotes the maximal number of times each time step is evaluated, we have the following upper bounds for the number of time steps evaluated during the adjoint calculation using r-level checkpointing and binomial checkpointing, respectively:

$$T_r \;=\; c\left(\frac{c}{r}+1\right)^{r-1} \;<\; r\,N_r^* \qquad \text{and} \qquad T_b \;=\; rN_b^* - \binom{c+r}{r-1} \;<\; r\,N_b^*\ .$$

For example, it is possible to compute the adjoint for $N = 23000$ time steps with only 50 checkpoints, less than $3N$ time step evaluations, and N adjoint steps using binary checkpointing instead of three-level checkpointing, where $N_3^* \leq 5515$. If we allow $4N$ time step evaluations then 35 checkpoints suffice to compute the adjoint for 80000 time steps using binomial checkpointing, where $N_4^* \leq 9040$. These numbers are only two possible combinations taken from Fig. 2.4 to illustrate the really drastic decrease in memory requirement that can be achieved if binomial checkpointing is applied.

However, usually the situation is the other way round, i.e. one knows N and/or c and wants to compute the adjoint as cheap as possible in terms of computing time. Here, the first observation is that r-level checkpointing introduces an upper bound on the number of time steps the adjoint of which can be computed, because the inequality $N \leq (c/r + 1)^r$ must hold. Furthermore, binomial checkpointing allows for numerous cases also a decrease in run-time compared to the uniform checkpointing. For a given r-level checkpointing and $M_r = c$, one has to compare T_r and T_b. Let r_b be the unique integer satisfying (2.2). Since at least one checkpoint has to be stored at each level, one obtains the bound $r \leq c$. I.e., one must have $c >= \log_2(N)$ to apply uniform checkpointing. Therefore, the following combinations of r and r_b are possible for the most important, moderate values of r:

$$r = 3 \;\Rightarrow\; r_b \in \{2, 3\}, \qquad r = 4 \;\Rightarrow\; r_b \in \{3, 4\}, \qquad r = 5 \;\Rightarrow\; r_b \in \{3, 4, 5\}\ .$$

For $3 \leq r \leq 5$, one easily checks that $T_r > T_b$ holds if $r_b < r$. For $r = r_b$, one can prove the following, more general result:

Theorem 2.4.1. *Suppose for a given N and a r-level checkpointing with $M_r = c$ that the corresponding r_b satisfying (2.2) coincide with r. Then, one has*

$$\begin{array}{lclcll} T_2 & = & 2N - c - 2 & = & T_b & \text{if } r = r_b = 2 \\ T_r & > & T_b & & & \text{if } r = r_b > 2. \end{array}$$

Proof: For $r_b = r = 2$ the identity $T_2 = T_b$ is clear. For $r = r_b > 2$, the inequality

$$\begin{aligned} \sum_{i=1}^{r} \prod_{\substack{j=1 \\ j \neq i}}^{r} (c_j + 1) &= \frac{(r-1)!}{(r-1)!} \left(\prod_{j=1}^{r-1} (c_j + 1) + (c_r + 1) \sum_{i=1}^{r-1} \prod_{\substack{j=1 \\ j \neq i}}^{r-1} (c_j + 1) \right) \\ &< \frac{1}{(r-1)!} \prod_{i=2}^{r} \left(\sum_{j=1}^{r} c_j + i \right) = \binom{c+r}{r-1} \end{aligned}$$

holds. Using the definitions of T_r and T_b, this relation yields immediately $T_r > T_b$. ■

Hence, except for the case $r = r_b = 2$, where T_r and T_b coincide, the run-time caused by binomial checkpointing is less than the one caused by multi-level checkpointing if $r = r_b$.

2.5 Conclusions

This article discusses several checkpointing techniques, i.e., multi-level checkpointing and binomial checkpointing. A detailed analysis of the number of time steps the adjoint of which can be calculated, the run-time needed for the adjoint calculation and the memory requirement is given.

One can conclude that binomial checkpointing allows adjoint calculations with a surprisingly small fraction of the memory needed by the basic approach. This storage reduction causes only a very moderate increase in run-time. On the other hand, we see that r-level checkpointing induces for a given number of checkpoints an upper bound on the number of time steps the adjoint of which can be computed. This upper bound can only be increased by introducing a next level of checkpointing. In addition it is shown that the run-time required for the adjoint calculation with r-level checkpointing exceeds the run-time needed for binomial checkpointing for the most important values of $r > 2$, whereas for $r = 2$ both methods yield the same run-time. However, for $r = 2$ and a given amount of memory, binomial checkpointing allows the adjoint computation for a larger number of time steps. Hence, even for $r = 2$ binomial checkpointing is preferable.

Moreover, it is quite often the case that the number N of time steps is not known a-priori, for example due to an adaptive time stepping method. Then, it becomes difficult to distribute the checkpoints for the two- or multi-level checkpointing such that the minimal run-time is attained. For binomial checkpointing the extension a-revolve deals with the unknown number of time steps by using a heuristic for the checkpoint placements. In addition, a-revolve can also handle time steps with varying temporal complexity. For time steps the cost of which do not change drastically, the heuristics implemented in a-revolve work well such that the corresponding adjoint calculation is only a few percentages slower than the one based on revolve [95]. Hence, binomial checkpointing provides memory-reduced adjoint calculation also in more general situations.

Chapter 3

Computing Sparse Hessians with Automatic Differentiation

Andrea Walther

To appear in ACM Transaction on Mathematical Software

Abstract:
A new approach for computing a sparsity pattern for a Hessian is presented: nonlinearity information is propagated through the function evaluation yielding the nonzero structure. A complexity analysis of the proposed algorithm is given. Once the sparsity pattern is available, coloring algorithms can be applied to compute a seed matrix. To evaluate the product of the Hessian and the seed matrix, a vector version for evaluating second order adjoints is analyzed. New drivers of ADOL-C are provided implementing the presented algorithms. Run-time analyses are given for some problems of the CUTE collection.

3.1 Introduction

Several solvers for nonlinearly constrained optimization problems allow or even require the provision of exact second order derivatives, e.g., [143, 146, 155]. Furthermore, exact second order derivatives are needed to compute parametric sensitivities, e.g., for the real-time control of dynamical systems, see [20]. Quite often, the corresponding Hessians are sparse, for example due to the discretization of a differential equation describing the considered problem. To maintain the efficiency of the algorithms, it is important to take this sparsity information into account. Therefore, some of the tools, e.g., [143, 146], assume that the user provides the Hessian in a sparse format.

As soon as a sparsity pattern for the Hessian is known, well-established coloring algorithms, see, e.g., [33, 56], in combination with Automatic Differentiation (AD) [70] allow the efficient computation of the required second order information. The overall process is illustrated in Figure 3.1. Ideally, the steps 1 and 2 of the process, i.e., the generation of the sparsity pattern P and the calculation of the so-called seed matrix S using graph coloring have to be performed only once. Subsequently, the entries of the sparse Hessian can be computed in a compressed form using the second order adjoint mode of AD. The knowledge of the sparsity pattern P is essential for the approach sketched in Figure 3.1.

$$\begin{tikzcd}
f \arrow[r, "\textcircled{1}"] & P \arrow[r, "\textcircled{2}"] & S \arrow[r, "\textcircled{3}"] & B = H\,S \\
\text{function} & \text{sparsity pattern} & \text{seed matrix} & \text{calculation of entries}
\end{tikzcd}$$

Figure 3.1: Computing sparse Hessians

So far, only AMPL [54] can compute structural information about the Hessian automatically. For that purpose, the partial separability of the differentiated function is exploited. In this paper, we propose and analyze a new algorithm for computing a sparsity pattern P. As an alternative to the present work, the calculation of sparse Hessians may also rely on elimination rules for the computational graph of the Hessian. This approach was first considered in [43] and is the subject of current research. Similar techniques are well-established for the computation of the complete Jacobian [110, 111].

We assume throughout that the function $f : \mathbb{R}^n \to \mathbb{R}, \quad x \mapsto y$, to be differentiated is at least twice continuously differentiable and given as a computer program in an imperative programming language. Then, the Hessian of f at a given point x defined by

$$H(x) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1}(x) & \cdots & \dfrac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}$$

is a symmetric matrix. Due to the second order derivatives, an entry

$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

in the Hessian can only be nonzero if the computation of $y = f(x)$ involves a term that depends nonlinearly on both x_i and x_j. In this paper, we propose a new algorithm that propagates appropriate nonlinearity information through the function computation. Subsequently, a sparsity pattern for the Hessian can be derived directly from the propagated index sets. For any AD-tool based on operator overloading, one can implement the proposed approach easily just as a new variant of the derivative calculation. In this paper, we present a new driver function of the AD-tool ADOL-C [154] to compute the required sparsity pattern. Then, we generate a seed matrix S to compute the entries of the sparse Hessian by applying a graph coloring algorithm first proposed in [33]. Subsequently, we present a vector mode for computing second order adjoint information. This vector version avoids the recomputation of intermediate results and reduces the cost to evaluate the Hessian-matrix product $H(x)S$ significantly. The proposed approach is implemented as a recent driver of ADOL-C that allows for the first time the computation of Hessian-matrix products instead of only Hessian-vector products.

This paper has the following structure. In Section 3.2, we introduce the function representation that is used to derive and analyze the proposed computation of a sparsity pattern. Subsequently, the propagation of nonlinearity is presented and a complexity analysis for the new algorithm is given. Section 3.3 sketches very briefly the graph coloring approach for generating the seed matrix S. Furthermore, it describes a new driver of ADOL-C implementing this algorithm. In Section 3.4, we present and analyze a vector version of the second order adjoint mode of AD. The corresponding implementation in ADOL-C is sketched. This includes also a new algorithm to compute the Hessian in a sparse format. Section 3.5 contains run-time analyzes to verify the complexity results. Finally, we draw some conclusions and give an outlook in Section 3.6.

3.2 Computing a Sparsity Pattern

3.2.1 Function Representation

Throughout, we assume that the calculation of $y = f(x)$ can be split into a presumably very long sequence of unary or binary operations. A formalization of the function evaluation similar to the one introduced in [70] is shown in Table 3.1. The first loop copies the current values of the independent variables $x_1, \ldots, x_n$ into the internal variables $v_{1-n}, \ldots, v_0$. The function evaluation itself consisting of l unary or binary operations is performed in the second loop. Finally the value of the dependent variable y is extracted from the corresponding internal variable v_l. As can be seen, each intermediate value v_i with $1 \leq i \leq l$ is computed by applying an *elemental function* φ_i. The function φ_i may have one or two arguments identified by the *precedence relation* $j \prec i$, where we have $\varphi_i(v_j)_{j\prec i} = \varphi_i(v_j)$ or $\varphi_i(v_j)_{j\prec i} = \varphi_i(v_j, v_{\hat{\jmath}})$ with $j < i$ and j, $\hat{\jmath} < i$, respectively. Hence, the precedence relation $j \prec i$ denotes that v_i depends directly on v_j.

Algorithm I: Function evaluation

```
for $i = 1, \ldots, n$
    $v_{i-n} = x_i$
for $i = 1, \ldots, l$
    $v_i = \varphi_i(v_j)_{j\prec i}$
$y = v_l$
```

Table 3.1: Formalization of evaluation

Since we assume that f is at least twice continuously differentiable, the set of elemental functions may comprise simple evaluations, e.g., additions, multiplications, and calls to intrinsic functions such as $\sin(x)$ or $\exp(x)$ provided by a high level computer language like Fortran or C such that they are two times differentiable. The approach presented below can be extended to piecewise-differentiable functions like $\max(v_j, v_{\hat{j}})$ or $\sqrt{v_j}$ as long as these elemental functions are evaluated on the differentiable parts.

3.2.2 Propagation of Nonlinear Interaction

Based on the decomposition into elemental functions, one can now define two different index sets to propagate nonlinearity information through the function evaluation. First, we will need *index domains*

$$\mathcal{X}_k \equiv \{j \le n \,:\, j - n \prec^* k\} \quad \text{for} \quad 1 - n \le k \le l$$

for all intermediate variables v_k as already defined in [70, Section 6.1]. Here, $\prec^*$ denotes the transitive closure of the precedence relation $\prec$. One can compute the index domains using the forward recurrence

$$\mathcal{X}_k = \bigcup_{j \prec k} \mathcal{X}_j \quad \text{from} \quad \mathcal{X}_{j-n} = \{j\} \quad \text{for} \quad 1 \le j \le n.$$

This approach yields the inclusion

$$\left\{ j \le n \,:\, \frac{\partial v_k}{\partial x_j} \not\equiv 0 \right\} \subseteq \mathcal{X}_k$$

and identity will hold as long as no degeneracy occurs. One example for a proper subset relation is given by the statement sequence

$$v_1 = \sin(v_0), \quad v_2 = \cos(v_0), \quad v_3 = v_1 * v_1, \quad v_4 = v_2 * v_2, \quad v_5 = v_3 + v_4$$

as mentioned already in [70]. Obviously, one has $\partial v_5/\partial v_0 = \partial v_5/\partial x_1 = 0$ but $\mathcal{X}_5 = \mathcal{X}_4 = \mathcal{X}_3 = \mathcal{X}_2 = \mathcal{X}_1 = \{1\}$. For the complexity analysis given in Sec. 3.2.3, we define $\bar{n}_k \equiv |\mathcal{X}_k|$ for all $1 - n \le k \le l$.

The index domains $\mathcal{X}_k$ belonging to the dependent variables can be used to exploit sparsity for the computation of Jacobian matrices as explained in [70, Chapter 7]. However, we want to go one step further in computing a sparsity pattern for the Hessian. Therefore, we need additional index sets $\mathcal{N}_i$, $1 \le i \le n$, called *nonlinear interaction domains* (NID) for all independent variables, such that

$$\left\{ j \le n \,:\, \frac{\partial^2 y}{\partial x_i \partial x_j} \not\equiv 0 \right\} \subseteq \mathcal{N}_i \,. \tag{3.1}$$

Once more, degeneracies may cause a proper subset relation in (3.1). In the case of second order derivatives considered here, degeneracy may, for example, arise through statement sequences such as $y = x(\sin^2(x) + \cos^2(x))$ given by

$$v_1 = \sin(v_0), \quad v_2 = \cos(v_0), \quad v_3 = v_1 * v_1, \quad v_4 = v_3 * v_0, \quad v_5 = v_2 * v_2,$$
$$v_6 = v_5 * v_0, \quad v_7 = v_6 + v_4.$$

Then, one has $\partial^2 v_7/\partial v_0^2 = \partial^2 v_7/\partial x_1^2 = 0$ but $\mathcal{N}_1 = \{1\}$.

After setting $\mathcal{N}_i = \emptyset$ at the beginning of the function evaluation, the NIDs have to be updated for each nonlinear operation that occurs during the function evaluation such that $\mathcal{N}_i$ contains the indices $1 \leq j \leq n$ of all independents that are combined in a nonlinear fashion with the independent x_i. This results in the algorithm shown by Table 3.2. As can be seen from Algorithm II, dead ends, i.e., intermediate variables v_k that were computed but not needed subsequently to calculate v_l, can be contained in the function evaluation. These dead ends may cause that identity does not hold in (3.1).

Algorithm II: Computation of nonlinear interaction domains

```
for $i = 1, \dots, n$
    $\mathcal{X}_{i-n} \leftarrow \{i\}, \quad \mathcal{N}_i \leftarrow \emptyset$
for $i = 1, \dots, l$
    $\mathcal{X}_i \leftarrow \bigcup_{j \prec i} \mathcal{X}_j$                                   (2)
    if $\varphi_i$ nonlinear then
        if $v_i = \varphi_i(v_j)$ then
            $\forall k \in \mathcal{X}_i : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$                  (3)
        if $v_i = \varphi_i(v_j, v_{\hat{\jmath}})$ then
            if $v_i$ linear in $v_j$ then
                $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_{\hat{\jmath}}$              (4)
            else
                $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$              (5)
            if $v_i$ linear in $v_{\hat{\jmath}}$ then
                $\forall k \in \mathcal{X}_{\hat{\jmath}} : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j$              (6)
            else
                $\forall k \in \mathcal{X}_{\hat{\jmath}} : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$              (7)
```

Table 3.2: Propagation of nonlinear interaction

Such dead ends have no consequences for the index domains $\mathcal{X}_k$, since they are defined for each intermediate value v_k. Hence, if a dead end is contained in the code, the corresponding index domains would be computed but they would have no influence on the index domain of the dependent variable y. This does not hold for the nonlinear interaction domains $\mathcal{N}_i$, since the $\mathcal{N}_i$ are defined only for the independent variables x_i. Hence, if a dead end occurs the $\mathcal{N}_i$ would be extended despite the fact that the computed values have no influence on the dependent variable y. This would result in an overestimate of the sparsity pattern.

To illustrate the algorithm, we will consider the function

$$f : \mathbb{R}^6 \to \mathbb{R}, \quad f(x) = \sin(x_1\, x_2) + \cos(x_3 + x_4) + 3(x_5 + x_6).$$

Table 3.3 shows the function evaluation and the development of the index sets $\mathcal{X}_i$ and $\mathcal{N}_i$ applying Algorithm II. As can be seen, the nonzero entries of the row i or column i of the Hessian $H(x)$ are given by the indices contained in the NIDs $\mathcal{N}_i$ for all $1 \leq i \leq 6$.

for $i = 1, \ldots, 6$			
$\mathcal{X}_{i-n} = \{i\}$,	$\mathcal{N}_i = \emptyset$		
$v_1 = v_{-5} * v_{-4}$,	$\mathcal{X}_1 = \{1, 2\}$,	$\mathcal{N}_1 = \{2\}$,	$\mathcal{N}_2 = \{1\}$
$v_2 = \sin(v_1)$,	$\mathcal{X}_2 = \{1, 2\}$,	$\mathcal{N}_1 = \{1, 2\}$,	$\mathcal{N}_2 = \{1, 2\}$
$v_3 = v_{-3} + v_{-2}$,	$\mathcal{X}_3 = \{3, 4\}$		
$v_4 = \cos(v_3)$,	$\mathcal{X}_4 = \{3, 4\}$,	$\mathcal{N}_3 = \{3, 4\}$,	$\mathcal{N}_4 = \{3, 4\}$
$v_5 = v_{-1} + v_0$,	$\mathcal{X}_5 = \{5, 6\}$		
$v_6 = 3 * v_5$,	$\mathcal{X}_6 = \{5, 6\}$		
$v_7 = v_2 + v_4$,	$\mathcal{X}_7 = \{1, 2, 3, 4\}$		
$v_8 = v_7 + v_6$,	$\mathcal{X}_8 = \{1, 2, 3, 4, 5, 6\}$		

Table 3.3: Function evaluation and execution of Algorithm II

3.2.3 Complexity Analysis

First, one has to note that an implementation of Algorithm II based on operator overloading does not require the coding of the if-statements. Here, the corresponding set operations can be coded together with or instead of the elemental function evaluation. Therefore, we ignore the if-statements in the following complexity analysis. To ensure that the proposed algorithm provides an efficient method to compute a sparsity pattern for the Hessian $H(x)$, we have to examine the set operations (2) - (7) of Algorithm II. These merging operations are performed in the following way, where flag denotes a logical array of length n that is set to true:

Algorithm III: Merging of two sets

1. For each i in the first set, put i in the new set and set flag(i) to false.
2. For each i in the second set, put i in the new set if flag(i) is true.
3. For each i in the first set, set flag(i) to true.

For the complexity result proved in the following theorem, we define the execution of one of the loop bodies as one operation MERGE. Hence, the operation count of the merging procedure is twice the length of the first list plus the length of the second list. It is possible to prove the following result:

Theorem 3.2.1 (Complexity result for Algorithm II). *Let OPS(NID) denote the number of operations MERGE needed by Algorithm II to generate all $\mathcal{N}_i$, $1 \leq i \leq n$. Then, the inequality*

$$OPS\big(NID\big) \leq 6(1 + \hat{n}) \sum_{i=1}^{l} \bar{n}_i \tag{3.10}$$

is valid, where l is the number of elemental functions evaluated to compute the function value of f, $\bar{n}_i = |\mathcal{X}_i|$, and $\hat{n} = \max_{1 \leq i \leq n} |\mathcal{N}_i|$.

Proof: The set operation (2) is either $\mathcal{X}_i \leftarrow \mathcal{X}_j$ if φ_i is unary or $\mathcal{X}_i \leftarrow \mathcal{X}_j \cup \mathcal{X}_{\hat{j}}$ if φ_i is binary. We obtain for the operation count of the set operation (2) that

$$\text{OPS}\Big(\mathcal{X}_i \leftarrow \bigcup_{j \prec i} \mathcal{X}_j\Big) \leq 3\bar{n}_i\,. \tag{3.11}$$

Furthermore, the number of elements in each NID $\mathcal{N}_i$, $1 \leq i \leq n$, can be bounded by

$$|\mathcal{N}_i| \leq \hat{n} \tag{3.12}$$

due to the definition of $\hat{n}$. Furthermore, we can conclude for the set operations (3), (5), and (7) that

$$|\mathcal{X}_i| = \bar{n}_i \leq \hat{n} \tag{3.13}$$

if v_i is the result of a nonlinear operation because of (3.12). Due to the same reason, we obtain for the set operations (4) and (6)

$$|\mathcal{X}_{\hat{\jmath}}| = \bar{n}_{\hat{\jmath}} \leq \hat{n} \quad \text{and} \quad |\mathcal{X}_j| = \bar{n}_j \leq \hat{n} \ , \tag{3.14}$$

respectively. Using (3.13) and (3.14), it follows for the set operations (3) - (7) that

$$\begin{aligned}
\text{OPS}\Big(\forall k \in \mathcal{X}_i : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i\Big) &\leq \ \bar{n}_i(2\hat{n} + \bar{n}_i) \leq \ \bar{n}_i(2\hat{n} + \hat{n}) = \ 3\hat{n}\bar{n}_i \\
\text{OPS}\Big(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_{\hat{\jmath}}\Big) &\leq \ \bar{n}_j(2\hat{n} + \bar{n}_{\hat{\jmath}}) \leq \ \bar{n}_i(2\hat{n} + \hat{n}) = \ 3\hat{n}\bar{n}_i \\
\text{OPS}\Big(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i\Big) &\leq \ \bar{n}_j(2\hat{n} + \bar{n}_i) \leq \ \bar{n}_i(2\hat{n} + \hat{n}) = \ 3\hat{n}\bar{n}_i \\
\text{OPS}\Big(\forall k \in \mathcal{X}_{\hat{\jmath}} : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j\Big) &\leq \ \bar{n}_{\hat{\jmath}}(2\hat{n} + \bar{n}_j) \leq \ \bar{n}_i(2\hat{n} + \hat{n}) = \ 3\hat{n}\bar{n}_i \\
\text{OPS}\Big(\forall k \in \mathcal{X}_{\hat{\jmath}} : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i\Big) &\leq \ \bar{n}_{\hat{\jmath}}(2\hat{n} + \bar{n}_i) \leq \ \bar{n}_i(2\hat{n} + \hat{n}) = \ 3\hat{n}\bar{n}_i \ .
\end{aligned}$$

The set operation (2) has to be executed exactly once for the calculation of the intermediate v_i, $1 \leq i \leq l$. Furthermore, at most two of the set operations (3) - (7) have to be executed. This yields the overall bound (3.10) due to the assumption of unary and binary operations. ■

3.2.4 Computing Sparsity Patterns

The AD-tool ADOL-C was augmented with the new driver

```
int hess_pat(short tag, int n, double* x, unsigned int** P, int option);
```

to compute a sparsity pattern for the Hessian for a given function according to Algorithm II. The first argument, i.e., tag, identifies the internal representation for which one wants to compute derivative information (see [154]). The next argument is used for a consistency check comparing this value to the one that is stored in the internal representation. The third argument, i.e., x, defines the point for which a sparsity pattern for the Hessian is computed. After the function call P contains a sparsity pattern for the Hessian, where P[j][0] contains the number of nonzero elements in the jth row. The components P[j][i], 0 <i ≤ P[j][0], store the indices of these entries. The usage of the routine hess_pat is described in more detail in [154] including information about allocation and deallocation schemes.

Obviously, the sparsity pattern P may vary as a function of the independent variable vector x for one of the following three reasons:

(A) Numerical values may be incidentally zero,

(B) fmin, fmax or other conditional assignments may flip to a different branch and

(C) the control flow may be completely changed.

According to the algorithm presented in this section, ADOL-C propagates generic dependencies and disregards incidental zeros that are due to cancellations or special values of the independent variables (case (A)). The treatment of case (B) is determined by the last argument option of the new driver. The default value is option $= 0$ resulting in a more conservative computation of a sparsity pattern for the Hessian. It accounts for all dependencies that might occur for any value of the independent variables. For example, the intermediate $v_i = \max(v_j, v_{\hat{j}})$ is always assumed to depend on all independent variables that v_j or $v_{\hat{j}}$ depend on and the index domain $\mathcal{X}_i$ is extended correspondingly. In contrast, the tight version option $= 1$ gives this result only in the unlikely event of an exact tie $v_j = v_{\hat{j}}$. Otherwise it sets the index domain $\mathcal{X}_i$ either to $\mathcal{X}_j$ or to $\mathcal{X}_{\hat{j}}$, depending on whether $v_i = v_j$ or $v_i = v_{\hat{j}}$ locally. Obviously, a sparsity pattern obtained with the tight option may contain more zeros than the one obtained with the safe option. On the other hand, it will only be valid at points belonging to an area where the function f is locally smooth and that contains the point at which the internal representation was generated. Case (C) results in a negative return value of the new driver indicating that the internal representation of the given function is not valid for the current argument x due to a change in the control flow. Then, before computing a sparsity pattern one has to generate a new internal representation by retaping the function evaluation at x. Details can be found in the ADOL-C documentation [154].

3.3 Computing the Seed Matrix

For computing sparse Jacobians, the application of compression techniques is now well-established. A comprehensive introduction to this approach can be found for example in [70]. Naturally, the same idea can also be exploited for computing sparse Hessians. Hence, the entries of the sparse Hessian are computed by evaluating the product

$$B = H(x)S \in \mathbb{R}^{n \times q}$$

for a so-called seed matrix $S \in \mathbb{R}^{n \times q}$. Here, as simplest option the columns of S are chosen as vectors the entries of which are either 0 or 1. After the computation of the matrix B, one has to reconstruct the entries of $H(x)$ from the available derivative information, see, e.g., [70]. Depending on the choice of the seed matrix, this may require solving a linear system whose matrix is either a permutation of the identity or a triangular matrix. In the first case, the entries of $H(x)$ can be directly extracted from B. The resulting evaluation scheme is therefore called *direct*. In the second case one has to solve simple equations. Hence, the resulting evaluation scheme is called *substitution-based*.

A first method to generate a seed matrix S was proposed by Powell and Toint [122]. Later, Coleman and Moré [33] observed that the task of finding a suitable S is equivalent to a graph coloring problem, where the symmetry of the derivative matrix can be exploited to reduce the required number q of columns in the seed matrix. The method proposed in [33] can be seen as a relaxed distance-2 and a restricted distance-1 coloring. Therefore, it is referred to as distance-$\frac{3}{2}$ coloring in [56]. This coloring method, recently studied also by Albertson et al. [3], is now used by ADOL-C to generate the seed matrix S, yielding a direct evaluation scheme. As alternative one may consider an acyclic coloring as proposed in [32] that gives a substitution-based evaluation scheme. This approach will be integrated into ADOL-C in the near future.

The new driver

```
int generate_seed_hess(int n, unsigned int** P, double*** S, int* q);
```

of ADOL-C has as input variables the number of independent variables n and a sparsity pattern P computed for example by the algorithm described in the last section or provided by the user. First, it performs a coloring of the adjacency graph defined by the sparsity pattern P. The number of colors needed for the coloring determines the number of columns q in the seed matrix. Subsequently, the function allocates the memory needed by S and initializes S according to the graph coloring. Additional information about the usage of generate_seed_hess including details about the specific memory management can be found in [154].

3.4 Evaluating Hessian-Matrix Products

For a scalar valued function $y = f(x)$ exact Hessian-vector products can be computed by differentiating formally the results of the reverse mode of AD once more with respect to x and $\bar{y}$ using the scalar forward mode of AD. Using the notation introduced in [70], this evaluation of second order adjoints is given by

$$y = f(x) \xrightarrow[\text{reverse}]{} \bar{x} = \bar{y}f'(x) \xrightarrow[\text{forward}]{} \dot{\bar{x}} = \bar{y}f''(x)\dot{x} + \dot{\bar{y}}f'(x) \in \mathbb{R}^n, \quad (3.15)$$

for $\bar{x}, \dot{x}, \dot{\bar{x}} \in \mathbb{R}^n$ and $\bar{y}, \dot{\bar{y}} \in \mathbb{R}$. Hence, the desired Hessian-vector product $f''(x)\dot{x}$ can be computed by setting $\bar{y} = 1$ and $\dot{\bar{y}} = 0$. As shown in [70, Section 4.5], one obtains the following complexity estimate for the evaluation of a Hessian-vector product

$$\text{TIME}(f''(x)\dot{x}) \leq \omega_{soad}\text{TIME}(f(x)) \quad \text{with} \quad \omega_{soad} \in [7,\ 10].$$

Evaluating the second order adjoint procedure denoted by the subscript *soad* for the p columns which form the seed matrix, we obtain as complexity estimate for evaluating $H(x)S$

$$\text{TIME}(H(x)S) \leq \omega_{soad}\, p\, \text{TIME}(f(x)) \quad \text{with} \quad \omega_{soad} \in [7,\ 10]. \quad (3.16)$$

Again setting $\dot{\bar{y}} = 0$, i.e., ignoring the second term for computing $\dot{\bar{x}}$ in (3.15), and applying the vector forward mode for a given matrix $\dot{X} \in \mathbb{R}^{n\times p}$, one can similarly derive a vector version of the second order adjoint computation given by

$$y = f(x) \xrightarrow[\text{reverse}]{} \bar{x} = \bar{y} f'(x) \xrightarrow[\text{forward}]{} \dot{\bar{X}} = \bar{y} f''(x) \dot{X} \in \mathbb{R}^{n \times p}$$

for $\bar{x} \in \mathbb{R}^n$, $\dot{X} \in \mathbb{R}^{n \times p}$, and $\bar{y} \in \mathbb{R}$. Omitting the storage of intermediate values for simplicity, the corresponding evaluation procedure is given in Table 3.4.

Algorithm IV: Evaluation of Hessian-matrix product

```
for $i = 1, \ldots, n$
    $v_{i-n} = x_i, \quad \dot{V}_{i-n} = \dot{X}_i, \quad \bar{v}_{i-n} = 0, \quad \dot{\bar{V}}_{i-n} = 0$
for $i = 1, \ldots, l$
    $v_i = \varphi_i(v_j)_{j \prec i}, \quad \dot{V}_i = \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \dot{V}_j, \quad \bar{v}_i = 0, \quad \dot{\bar{V}}_i = 0$
$y = v_l, \quad \dot{Y} = \dot{V}_l, \quad \bar{v}_l = \bar{y}$
for $i = l, \ldots, 1$
    $\bar{v}_j \mathrel{+}= \bar{v}_i \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \quad \text{for} \quad j \prec i$
    $\dot{\bar{V}}_j \mathrel{+}= \bar{v}_i \sum_{k \prec i} \frac{\partial^2}{\partial v_j \partial v_k} \varphi_i(v_j)_{j \prec i} \dot{V}_k + \dot{\bar{V}}_i \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \quad \text{for} \quad j \prec i$
for $i = 1, \ldots, n$
    $\bar{x}_i = \bar{v}_{i-n}, \quad \dot{\bar{X}}_i = \dot{\bar{V}}_{i-n}$
```

Table 3.4: Second order adjoint vector evaluation

To analyze the computation effort needed to evaluate $H(x)S$, we will use the complexity analysis introduced in [70, Section 2.5]. Denoting the vector version of the second order adjoint computation with *soadp*, we obtain for assigning a constant, an addition or subtraction, a multiplication, and a general nonlinear function ψ as elemental function φ the operation counts given in Table 3.5, which can be directly derived from the complexity counts given in [70, Tables 3.6, 4.11]. Here, MOVES denotes the number of memory accesses. From the stated operation counts, we can derive the run-time estimate

$$\mathrm{TIME}(\dot{\bar{X}}) \leq \omega_{soadp} \mathrm{TIME}(f(x)) \tag{3.17}$$

according to [70, Section 2.5] with

$$\omega_{soadp} = \max\left\{ 2 + 2p, \frac{(12 + 6p)\mu + 3 + 3p}{3\mu + 1}, \right. \\ \frac{(11 + 11p)\mu + 2 + 5p + (3 + 6p)\pi}{3\mu + \pi}, \\ \left. \frac{(7 + 7p)\mu + 1 + 2p + (1 + 4p)\pi + 4\nu}{2\mu + \nu} \right\} \in [4 + 3p, 4 + 6p]. \tag{3.18}$$

The constants μ, π, and ν measure the complexity of a memory access, a

	Elemental function φ			
soadp	const	add/sub	mult	ψ
$MOVES$	$2+2p$	$12+6p$	$11+11p$	$7+7p$
$ADDS$	0	$3+3p$	$2+5p$	$1+2p$
$MULTS$	0	0	$3+6p$	$1+4p$
$NLOPS$	0	0	0	4

Table 3.5: Second order adjoint vector complexity

multiplication, and a nonlinear operation, respectively, where the complexity of an addition is normalized to 1. The reduction in the run-time ratio from an upper bound in $[7p, 10p]$ to an upper bound in $[4+3p, 4+6p]$ is caused by the fact that values that are independent of the directions contained in $\dot{X}$ are reused instead of recomputed. Hence, similar to the run-time reductions that can be achieved by using the vector forward mode of AD instead of the scalar forward mode for computing first derivatives, a decrease of the computing time needed for directional second derivatives can be achieved by using a vector version. The new ADOL-C driver

```
int hess_mat(short tag, int n, int p, double* x, double** S, double** B);
```

implements the vector version of the second order adjoint computation. The inputs are the identifier for the internal representation tag, the number of independent variables n for a consistency check, the current value of the independent variables x, and the seed matrix S. The result of the product $H(x)S$ is stored as output of the function call in the two-dimensional array B of size $n \times p$. More information about hess_mat including details about the memory allocation can be found in [154]. Using the three new ADOL-C drivers, it is possible to compute sparse Hessians in an efficient way as we will see in the next section. Since the Hessian entries are often required in a prescribed sparse format, ADOL-C also provides a new driver that computes the sparse Hessian and stores the entries directly in coordinate format:

```
int sparse_hess(short tag, int n, int repeat, double* x, int* nnz,
                unsigned int** r_ind, unsigned int** c_ind, double** H_val);
```

Once more, the input variables are the identifier for the internal representation tag, the number of independent variables n for a consistency check, the current value of the independent variables x. Furthermore, the flag repeat=0 indicates that a new seed matrix S has to be computed, whereas repeat=1 results in the re-usage of the previously computed seed matrix. The input/output variable nnz stores the number of the nonzero entries. Therefore, nnz also denotes the length of the arrays r_ind storing the row indices, c_ind storing the column indices, and H_val storing the values of the nonzero entries. The manual [154] contains more information about the routine sparse_hess including details about the corresponding memory management.

3.5 Numerical Examples

In this section, we will employ the AD-tool ADOL-C to present some run-time results for the proposed algorithms. For that purpose, we use optimization problems from the CUTE collection [15].

The Computation of Sparsity Patterns

In this subsection, we report on the run-time behavior of the driver hess_pat described in Subsection 3.2.4 as an implementation of Algorithm II. As test cases, we chose the Lagrange function of the CUTE problems broydnbd ($\hat{n} = 1$), chainwoo ($\hat{n} = 4$), lminsurf ($\hat{n} = 9$), and morebv ($\hat{n} = 5$) with varying dimension n. For all four examples there exists at least one index domain that contains the indices of all independent variables, i.e., there is at least one $i \in \{1, \ldots, l\}$ with $\bar{n}_i = n$. This is possible because the bounds (3.13) and (3.14) hold only for intermediate variables that are the result and the argument of a nonlinear operation, respectively. Furthermore, almost all rows of the Hessians have $\hat{n}$ nonzero entries independent of the value of n. Throughout this subsection, the figures report the run-time ratio

$$\frac{\text{TIME}\big(\text{hess\_pat}(\ldots)\big)}{(1+\hat{n})\text{TIME}(f)} \,. \tag{3.19}$$

For n varying in the interval $[1000, 10000]$ the run-time ratios obtained for the considered examples are illustrated in Figure 3.2. As can be seen, a constant

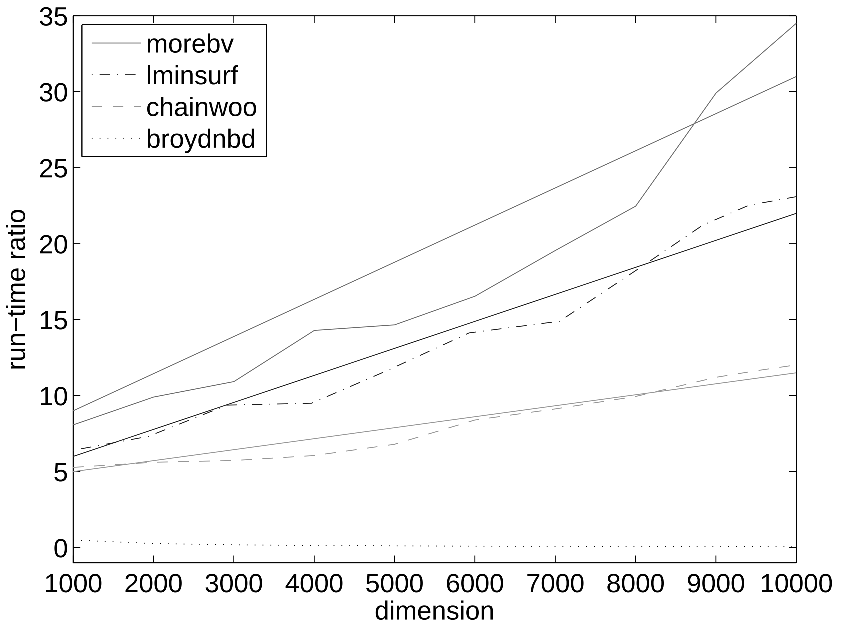

Figure 3.2: Run-time results for Algorithm II

run-time ratio is achieved for the broydnbd problem. For the problem chainwoo a small increase of the run-time ratio can be observed. For the problems lminsurf and morebv, we obtain a stronger increase of the run-time ratio (3.19). To analyze the linear behavior in more detail, lines are add to the curves illustrating the runtime results. The slopes for the added lines are well below 0.003.

To analyze the run-time behavior of Algorithm II in more detail, we performed another test by varying the problem size and the number of nonzeros $\hat{n}$. For that purpose, we enlarged the original objective function by the additional term

$$\tilde{f}(x) = f(x) + \sum_{j=1}^{u} x_i x_{i+j}.$$

Hence, the number of nonzero diagonals in the Hessian of the Lagrange function can be varied by choosing u appropriately. We generated test cases for the problem chainwoo with $\hat{n} = 7, 9, 11$ instead of $\hat{n} = 4$ as in the original version. Furthermore, we studied test cases for the morebv problem with $\hat{n} = 7, 9, 11$ instead of $\hat{n} = 5$ as in the original version. Figure 3.3 illustrates the run-time ratios achieved. For both problems, the linear behavior of the run-time ratio is almost the same if the number of nonzeros is increased. To ease the interpretation of the results, we added a black solid line with the slope 0.0001 for chainwoo and 0.0025 for morebv. As can be seen, the constant describing the linear increase of the run-time ratios is very small, which fits the expectations based on the complexity result presented in Section 3.2.3. Only for the problem chainwoo and $\hat{n} = 9$ and $\hat{n} = 11$ some cache effects disturbed the linear behavior in the range $n \in [8000, 9000]$. For the examples considered here, the linear

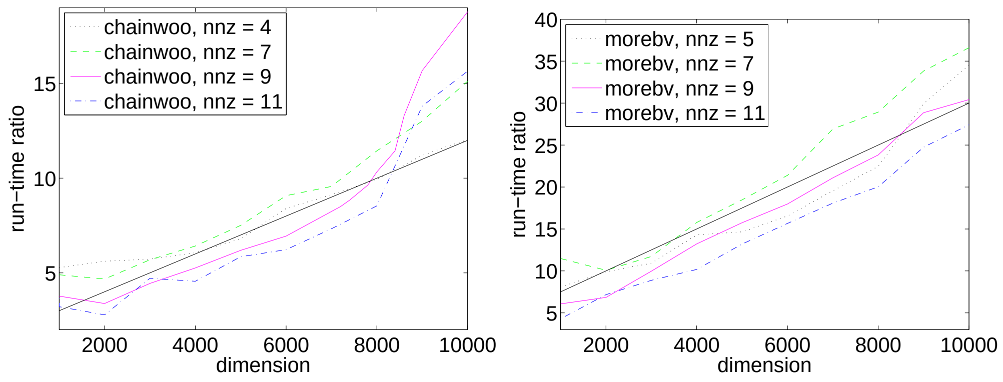

Figure 3.3: Run-time results for Algorithm II and varying number of nonzeros

increase with n of the complexity of Algorithm II can be described by a very small constant. Comparing the values observed for the run-time ratios with the complexity results given by (3.15) or (3.17), one finds that a sparsity pattern for the Hessian can be calculated at a cost that corresponds to the cost for computing a few columns of the Hessian itself.

The Evaluation of Hessian-matrix Products

A second class of run-time tests was done for the newly proposed vector version of the second order adjoint mode. Comparing the complexity bounds for scalar

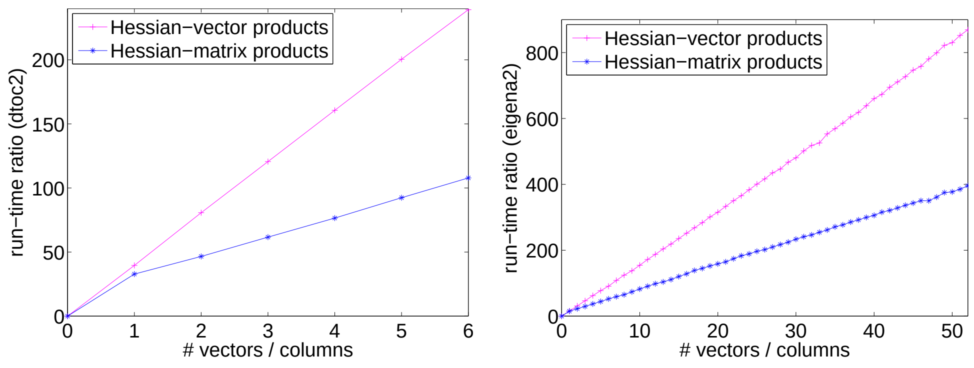

Figure 3.4: Hessian-vector and Hessian-matrix products

second order adjoints given in (3.16) and for the vector version given in (3.17) and (3.18), one obtains

$$\frac{\text{TIME}(H(x)S)}{\text{TIME}(f(x))} \leq \omega_{soad}\, p \leq 10p \qquad \text{and} \qquad \frac{\text{TIME}(\dot{\bar{X}})}{\text{TIME}(f(x))} \leq \omega_{soadp} \leq 4 + 6p,$$

respectively. We computed the stated run-time ratios using the well established driver hess_vec of ADOL-C to compute p Hessian-vector products and the new driver hess_mat of ADOL-C to compute one Hessian-matrix product. The problems dtoc2 and eigena2 serve as test cases. For dtoc2 the corresponding seed matrix has 6 columns independent of the size of the problem. For the numerical tests we set the number of variables to $n = 1485$ and the number of constraints to $m = 990$. The number of columns that form the seed matrix varies with the problem size for eigena2. To get an impression also for a higher number of columns, we set the number of independent variables to $n = 2550$ and the number of constraints $m = 1275$, resulting in a seed matrix with 52 columns.

The achieved run-time ratios are illustrated by Figure 3.4. First of all, the expected linear behavior in dependence on the number of vectors and columns, respectively, is clearly visible. Furthermore, the line with the larger slope belongs to the scalar second order adjoint computation evaluating p Hessian-vector products. Hence, so far the theory is verified by the numerical examples, since the vector version of the second order adjoint requires significantly less run-time.

Additionally, one can examine the slopes of the lines in more detail. For that purpose the slopes are stated in Table 3.6. As can be seen, the scalar mode is almost three times slower than the theory predicted for the dtoc2 example, whereas the vector mode is only about two times slower than the theory.

For the eigena2 example, the function evaluation is a little bit more complicated and the situation changes considerably in favor of ADOL-C. Here, the run-time needed by the scalar mode is only about a factor 3/2 larger than expected. So the operator-overloading tool ADOL-C comes almost close to the theory. The same is true for the vector version of the second order adjoint, where the slope is close to the theoretical bound 6.

	dtoc2	eigena2
scalar *soad*	28.6	16.5
vector *soad*	13.2	7.5

Table 3.6: Slopes obtained from run-time results

3.6 Conclusion and Outlook

This article presents the propagation of nonlinearity for determining a sparsity pattern for a Hessian matrix. The complexity of the corresponding algorithm is analyzed in detail. Once the sparsity pattern is available, well-known graph-coloring techniques can be applied to generate a seed matrix. Subsequently, the seed matrix can be used as input for a vector version of the second order adjoint mode, that is proposed and analyzed in this paper for the first time. The three ingredients – sparsity pattern, seed matrix, vector second order adjoint computation – allow an efficient evaluation of sparse Hessians. Run-time results verifying the theoretical results are presented for some problems of the CUTE collection, where the AD-tool ADOL-C is used to compute the derivatives.

Future work can be devoted to the incorporation of more sophisticated coloring techniques which are the subject of current research. This includes also the incorporation of a substitution-based seeding.

Acknowledgment

The author is grateful to Arijit Tarafdar and Assefaw Gebremedhin, Old Dominion University, for providing their graph coloring code. Furthermore, the author would like to thank the anonymous referees for their very valuable comments and constructive suggestions.

Chapter 4

Evaluating Gradients in Optimal Control — Continuous Adjoints versus Automatic Differentiation

Roland Griesse[1] and Andrea Walther

Journal of Optimization Theory and Applications,
Volume 122(1), pp. 63 – 86 (2004).

Abstract:
This paper deals with the numerical solution of optimal control problems for ODEs. The methods considered here rely on some standard optimization code to solve a discretized version of the control problem under consideration. We aim at providing the optimization software not only with the discrete objective functional, but also with its gradient. The objective gradient can be computed either from forward (sensitivity) or backward (adjoint) information.

The purpose of this paper is to discuss various ways of adjoint computation. It will be shown both theoretically and numerically that methods based on the continuous adjoint equation require a careful choice of both the integrator and gradient assembly formulae in order to obtain the gradient consistent with the discretized control problem. Particular attention will be given to automatic differentiation techniques which automatically generate a suitable integrator.

4.1 Introduction

The field of optimal control problems for ODE looks back on a rich history of books and research papers since the first facing of such problems in the 1950s. Both theoretical and numerical aspects can be considered well studied.

In principle, there are two classes of methods to treat problems of optimal control numerically, the first of which uses the Pontryagin maximum (or

[1]RADON Institute, Linz, Austria

minimum) principle to derive necessary conditions for the optimizer. These conditions take the form of a multi-point boundary value problem for the state and the additional adjoint equation. In the presence of control or state constraints, the switching structure—i.e. the points in time when the control enters or leaves the boundary of the set of feasible controls—has to be known in advance. An introductory treatment on these so-called *indirect methods* can be found in Pesch [121]. Examples and issues on implementation are covered e.g. in Bulirsch et. al. [17] and Hiltmann [91]. A boundary value problem solver commonly used is described in Oberle and Grimm [118].

The second class is termed *direct methods*. These methods have in common that, in a first step, some discretization renders the infinite-dimensional optimal control problem into a finite-dimensional optimization task, also called an NLP problem. The latter is usually solved by some optimization code, e.g. an SQP solver.

Within the class of direct methods, two techniques can be distinguished: Approaches that discretize both the control and state variables and pass the discretized state equation on to the optimization code are often described as *full discretization* concepts. Examples are collocation methods, see e.g. von Stryk [145] and Betts [12]. Note that both discretized control and state variables are treated as optimization variables by the optimization solver.

The second technique features discretization of the control variables only. Consequently, evaluating the objective by a user-provided subroutine requires forward integration of the state equation. The name *recursive discretization* is usually attributed to these methods, inspired by the fact that all classical ODE integration schemes (e.g. Runge-Kutta schemes) define the solution to the state ODE recursively, time step by time step. We refer to Büskens and Maurer [19] as well as Bock and Plitt [14] for more on these methods.

One major difference between full and recursive discretization is that the first generates a huge amount of equality constraints containing the discretized state equation (yet these have an almost block-diagonal Jacobian matrix), while the latter produces a smaller amount of optimization variables, paid for by non-trivial objective gradients.

We only study the recursive technique in the present paper. Our goal was to provide the optimization solver with the gradient consistent with the discrete objective function. Despite the long history of numerical solution of optimal control problems, this feature is still not implemented in many optimal control codes. Although the control problems covered in this work are subject to only simple control constraints, they still feature the relevant effects in gradient computation. In a companion paper [66], we consider a more substantial example involving state constraints and a rather complicated dynamical system, computing also parametric sensitivities.

This paper is organized as follows: In Section 4.2, we introduce the optimal control problem under consideration. Section 4.3 addresses discretization and the central issue of finding gradient representations for the finite-dimensional optimization problem. Main features of third-party and our own software are exposed in Section 4.4. We present two examples in Section 4.5 to illustrate the theoretical finding, and give a conclusion in Section 4.6.

4.2 Continuous Problem

We consider the following optimal control problem:

$$\min J(y) = \varphi(y(t_f)) \tag{4.1}$$
$$\text{s.t. } \dot{y}(t) = f(y(t), u(t), t) \qquad t \in [0, t_f] \tag{4.2}$$
$$y(0) = y_0 \tag{4.3}$$
$$a \leq u(t) \leq b \qquad a, b \in \mathbb{R}^m. \tag{4.4}$$

The state $y(t)$ has values in $\mathbb{R}^n$ while the control $u(t)$ maps into $\mathbb{R}^m$. Hence the dynamics are given by a right hand side function $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$. We assume that complete initial information $y_0 \in \mathbb{R}^n$ is given. The objective function $\varphi : \mathbb{R}^n \to \mathbb{R}$ evaluates the state at the given terminal time t_f. In addition, the control is subject to box constraints on each of its components. Let us assume that all functions are sufficiently smooth to carry out all differentiation necessary.

Since the state y depends on the control u, it is natural to introduce an objective in terms of the control variable only. To this end, let $y = \psi(u)$ denote the solution of the state equation (5.2)–(5.3) for a given control u, and define

$$\tilde{J}(u) = J(\psi(u)).$$

Since gradients[2] play an important role in optimization, we present two distinct representations of $D\tilde{J}(u)$. Both are based on the observation that by the chain rule,

$$D\tilde{J}(u) = DJ(\psi(u))\ D\psi(u).$$

Strictly speaking, the derivative $D\tilde{J}(u)$ is an element of the dual of the control function space. Typically, this dual space can again be identified with a space of functions depending on the time t. We will make use of this fact shortly.

Let us define the sensitivity $s(t) \in \mathbb{R}^n$ for a given control u in the direction of $\bar{u}$ by the following ODE:

$$\dot{s}(t) = f_y(\psi(u)(t), u(t), t)\, s(t) + f_u(\psi(u)(t), u(t), t)\, \bar{u}(t) \tag{4.5}$$
$$s(0) = 0. \tag{4.6}$$

This immediately leads to the *forward* or *sensitivity* representation of the gradient:

$$D\tilde{J}(u)\bar{u} = \int_0^{t_f} \nabla\varphi(\psi(u)(t))\, s(t)\, dt. \tag{4.7}$$

The attribute *forward* points to the fact that, in order to find the Gâteaux variation $D\tilde{J}(u)\bar{u}$, the sensitivity equation has to be integrated in forward direction from 0 to t. Note that in (4.7), $\nabla\varphi(\psi(u)(t))$ denotes the gradient of

[2]We will denote Fréchet derivatives in infinite-dimensional spaces by the symbol D. Partial derivatives are indicated by a subscript, e.g. f_y. Finite-dimensional Jacobian matrices and gradients (row vectors) will both be abbreviated by ∇.

φ, evaluated at $\psi(u)(t)$, and not the total derivative of the composite function $\varphi \circ \psi$. The same holds for (4.9) below.

As mentioned above, there is an alternative approach to represent the objective gradient based on the adjoint ODE

$$-\dot{\lambda}(t) = f_y(\psi(u)(t), u(t), t)^T \, \lambda(t) \tag{4.8}$$

$$\lambda(t_f) = \nabla\varphi(\psi(u)(t_f))^T. \tag{4.9}$$

The solution $\lambda(t) \in \mathbb{R}^n$ of this linear ODE is called the adjoint variable and yields the gradient's *backward* or *adjoint* representation [16, Section 2.4]

$$(D\tilde{J}(u)(t))^T = f_u(\psi(u)(t), u(t), t)^T \lambda(t) \in \mathbb{R}^{m\times 1}. \tag{4.10}$$

Now, in order to determine the gradient at time t, the adjoint equation has to be integrated in *backward* direction from t_f to t.

It should be mentioned that the connection between the function space interpretation of the objective gradient and its interpretation as a function depending on time is illustrated by the following formula for the Gâteaux variation in the direction of $\bar{u}$:

$$D\tilde{J}(u)\bar{u} = \int_0^{t_f} D\tilde{J}(u)(t)^T \, \bar{u}(t)\, dt. \tag{4.11}$$

In this article, we pursue the adjoint approach.

4.3 Discretized Problem

In order to solve problem (4.1)–(4.4) numerically, some discretization has to be carried out. Therefore, let the time interval $[0, t_f]$ be divided into N_t subintervals of equal lengths. The time grid consists of points

$$t^j = (j-1)\cdot h \quad for \quad j = 1, \ldots, N_t + 1$$

where $h = t_f/N_t$ is the time step length. For the sake of notation's simplicity we restrict the presentation to uniform grids. The use of an ODE integrator with adaptive step size control, however, introduces conceptual differences and is not considered here [45].

All control and state components will be approximated at the grid points only, and we use the notation

$$\begin{array}{lll} y^j \text{ to approximate } y(t^j), & & y = (y^1, \ldots, y^{N_t+1})^T \in \mathbb{R}^{(N_t+1)n} \\ u^j \text{ to approximate } u(t^j), & & u = (u^1, \ldots, u^{N_t+1})^T \in \mathbb{R}^{(N_t+1)m}. \end{array}$$

We end up with the following finite-dimensional NLP problem:

$$\min \tilde{J}(u) = \varphi(\psi^{N_t+1}(u))$$
$$\text{s.t. } a \le u^j \le b$$

where $y = \psi(u) = (\psi^1(u), \ldots, \psi^{N_t+1}(u))^T$ approximately satisfies the state ODE of the problem. In our tests, we took ψ to represent the classical explicit fourth-order Runge-Kutta scheme with constant step size h which leads to the following algorithm to compute the discrete state vector y:

1. Let $y^1 = y_0$.
2. Given y^j, compute y^{j+1} by
   (a) $k_1 = f(y^j, u^j, t^j)$
   (b) $k_2 = f(y^j + \frac{1}{2}hk_1, u^{j+\frac{1}{2}}, t^{j+\frac{1}{2}})$
   (c) $k_3 = f(y^j + \frac{1}{2}hk_2, u^{j+\frac{1}{2}}, t^{j+\frac{1}{2}})$
   (d) $k_4 = f(y^j + hk_3, u^{j+1}, t^j + h)$
   (e) $y^{j+1} = y^j + \frac{h}{6}[k_1 + 2k_2 + 2k_3 + k_4]$

for $j = 1, \ldots, N_t$. Here, $t^{j+\frac{1}{2}}$ denotes $t^j + \frac{h}{2}$ and $u^{j+\frac{1}{2}}$ is either equal to u^j (piecewise constant interpolation) or equal to $\frac{u^j+u^{j+1}}{2}$ (piecewise linear interpolation).

As mentioned in Section 4.1, we consider only the discretized control functions u as optimization variables, which has the effect that in every evaluation of the objective $\varphi(\psi^{N_t+1}(u))$, the forward equation has to be solved using the Runge-Kutta integration scheme. We obtain a relatively small NLP problem whose objective gradient is not trivial to evaluate: The term in question is the gradient of the function

$$\mathbb{R}^{(N_t+1)m} \ni u \mapsto \tilde{J}(u) = \varphi(\psi^{N_t+1}(u)) \in \mathbb{R}$$

where $y^{N_t+1} = \psi^{N_t+1}(u)$ is a rather intricate function of u. We now turn to some possibilities to compute this gradient $\nabla \tilde{J}(u)$.

4.3.1 Finite–Difference Evaluation of the Objective Gradient

As a first possibility, finite difference evaluation of $\nabla \tilde{J}(u)$ comes into mind. Denoting the i-th component of u^j by u_i^j and letting e_i^j be the unit vector of length $(N_t+1)m$ with the $[(j-1)m+i]$-th entry equal to one, a one-sided finite difference approximation is given by

$$\frac{\partial \tilde{J}(u)}{\partial u_i^j} \approx \frac{\tilde{J}(u + \epsilon\, e_i^j) - \tilde{J}(u)}{\epsilon}.$$

Indeed, with appropriately chosen perturbations ϵ, the approximation is a reliable estimate of the gradient, and no additional coding is needed beyond the evaluation of $\tilde{J}(u)$. However, every evaluation of $\nabla \tilde{J}(u)$ is as expensive as $(N_t+1)m$ evaluations of the objective itself, leading to unacceptable performance for fine discretizations. This observation is confirmed in the numerical experiments of Section 4.5.

4.3.2 Straightforward Evaluation of the Objective Gradient Using the Continuous Adjoint Equation

An alternative idea is to use the continuous gradient's adjoint representation (4.10) as a basis for the gradient computation. While in this subsection we introduce a straightforward approach, we will find in the sequel that this simple method yields an objective gradient that is not exactly consistent with the objective value itself.

It is a natural idea to start by obtaining a discretized solution of the continuous adjoint equation (4.8)–(4.9), and it is tempting to use the same integration scheme as for the forward equation, working with step size $-h$ instead of h since integration is backwards in time.

Once the discrete adjoint has been computed, it is readily assumed that a discretization of (4.10)–(4.11) yields the discrete gradient

$$\nabla \tilde{J}(u) = \left(\frac{\partial \tilde{J}(u)}{\partial u^1}, \ldots, \frac{\partial \tilde{J}(u)}{\partial u^{N_t+1}} \right).$$

However, as a detailed investigation in the sequel will reveal,

$$\left(\frac{\partial \tilde{J}(u)}{\partial u^j} \right)^T \neq h f_u(\psi^j(u), u^j, t^j)^T \lambda^j, \tag{4.12}$$

i.e., the formula on the right-hand side of (4.12) is *not* the correct representation of the discrete gradient. The usage of this straightforward procedure leads to incorrect gradients and can even cause an SQP solver to fail to converge as shown in Section 4.5.

4.3.3 Consistent Evaluation of the Objective Gradient Using the Continuous Adjoint Equation

In the previous subsection, we have introduced a straightforward and appealing method, aiming at computing the objective gradient, simply by discretizing the continuous equations. We will now show that the quantity obtained in this way is *not* the objective gradient. In order to get correct discrete gradient information, two major issues have to be taken into account: Finding a suitable integration scheme for the continuous adjoint equation (4.8)–(4.9), and assembling the gradient as suggested by (4.10) and (4.11).

The first part of this question has been addressed in Hager [84] for a general Runge-Kutta integration scheme. In contrast to our presentation, Hager pursues a full discretization approach for the underlying control problem, treating the integration scheme as additional constraints. Moreover, he introduces additional optimization variables, corresponding to the control functions at the intermediate time grid points required by the Runge-Kutta scheme. While this eliminates the need for control interpolation, it also leads to a larger optimization problem and therefore is not very common in practical algorithms.

We will illustrate the relevant effects using for simplicity the explicit Euler scheme for the forward state equation $\dot{y} = f(y, u, t)$, i.e.,

$$y^{j+1} = y^j + h f(y^j, u^j, t^j).$$

In the spirit of adjoint information, the gradient $\nabla\tilde{J}(u)$ is evaluated from its last to its first components:

$$\begin{aligned}
\frac{\partial\tilde{J}(u)}{\partial u^{N_t+1}} &= \nabla\varphi(\psi^{N_t+1}(u))\frac{\partial\psi^{N_t+1}(u)}{\partial u^{N_t+1}} = \nabla\varphi(\psi^{N_t+1}(u))\cdot 0\\
\frac{\partial\tilde{J}(u)}{\partial u^{N_t}} &= \nabla\varphi(\psi^{N_t+1}(u))\,\frac{\partial\psi^{N_t+1}(u)}{\partial u^{N_t}}\\
\frac{\partial\tilde{J}(u)}{\partial u^{N_t-1}} &= \nabla\varphi(\psi^{N_t+1}(u))\,\frac{\partial\psi^{N_t+1}(u)}{\partial\psi^{N_t}(u)}\,\frac{\partial\psi^{N_t}(u)}{\partial u^{N_t-1}}\\
&\vdots\\
\frac{\partial\tilde{J}(u)}{\partial u^{1}} &= \nabla\varphi(\psi^{N_t+1}(u))\,\frac{\partial\psi^{N_t+1}(u)}{\partial\psi^{N_t}(u)}\,\frac{\partial\psi^{N_t}(u)}{\partial\psi^{N_t-1}(u)}\cdots\frac{\partial\psi^{3}(u)}{\partial\psi^{2}(u)}\frac{\partial\psi^{2}(u)}{\partial u^{1}}.
\end{aligned}$$

These steps can be carried out efficiently by first aggregating the adjoint-like quantities

$$(\lambda^{N_t+1})^T = \nabla\varphi(\psi^{N_t+1}(u)) \tag{4.13}$$

$$(\lambda^{j})^T = \nabla\varphi(\psi^{N_t+1}(u))\frac{\partial\psi^{N_t+1}(u)}{\partial\psi^{N_t}(u)}\frac{\partial\psi^{N_t}(u)}{\partial\psi^{N_t-1}(u)}\cdots\frac{\partial\psi^{j+1}(u)}{\partial\psi^{j}(u)}, \tag{4.14}$$

where $j = N_t, N_t-1, \ldots, 1$.

Since for the explicit Euler scheme we have

$$\psi^{j+1}(u) = \psi^{j}(u) + hf(\psi^{j}(u), u^j, t^j)$$

and thus

$$\frac{\partial\psi^{j+1}(u)}{\partial\psi^{j}(u)} = I + hf_y(\psi^{j}(u), u^j, t^j), \tag{4.15}$$

the λ^j obey the backward recursion

$$\lambda^j = [I + hf_y(\psi^{j}(u), u^j, t^j)]^T\lambda^{j+1}. \tag{4.16}$$

As an integration scheme applied to the adjoint ODE (4.8)–(4.9), (4.16) represents an *explicit* method backwards in time with right hand side information taken partly from an *implicit* method.

For comparison, the implicit and explicit Euler schemes are

$$\begin{aligned}
\lambda^j &= [I - hf_y(\psi^{j}(u), u^j, t^j)]^{-T}\lambda^{j+1} && \text{(implicit Euler)}\\
\lambda^j &= [I + hf_y(\psi^{j+1}(u), u^{j+1}, t^{j+1})]^{T}\lambda^{j+1} && \text{(explicit Euler).}
\end{aligned}$$

Finally, the gradient sought can be obtained from the staggered formulae

$$\frac{\partial\tilde{J}(u)}{\partial u^{N_t+1}} = 0 \tag{4.17}$$

$$\frac{\partial\tilde{J}(u)}{\partial u^{j}} = (\lambda^{j+1})^T hf_u(\psi^{j}(u), u^j, t^j), \qquad j = 1, \ldots, N_t. \tag{4.18}$$

Summarizing, one has to exercise great caution when using the continuous adjoint equation to generate discrete gradient information. In fact, it is rather tedious to find the appropriate adjoint integration scheme and the gradient evaluation formulae (4.17)–(4.18) by hand. This is the reason why this method has not been included in our numerical tests: While it is assumed to be accurate and fast, it is difficult to implement. In contrast to that, the approach using automatic differentiation described in the following subsection generates the same computational steps, avoiding error-prone hand-coding at only slightly increased run-time.

4.3.4 Evaluation of the Objective Gradient Using Automatic Differentiation

Over the last decades the technique of automatic differentiation (AD) has been developed. This method, which is still not well known, offers an opportunity to provide derivative information for a given code segment. A comprehensive introduction to AD can be found in Griewank [70].

The key idea of automatic differentiation is the systematic application of the chain rule. For many applications, the underlying model is described by a nonlinear vector function $F : \mathbb{R}^N \to \mathbb{R}^M, x \mapsto F(x)$, defined and evaluated by a computer program. The computation of such a function F can be decomposed into a (typically very long) sequence of simple evaluations, e.g. additions, multiplications, and calls to elementary function such as $\sin(x_i)$ or $\exp(x_i)$. The derivatives with respect to the arguments of these operations can be easily calculated. A systematic application of the chain rule then yields the derivatives of the whole sequence with respect to the input variables $x \in \mathbb{R}^N$. Depending on the starting point of this methodology—either at the beginning or at the end of the respective chain of computational steps—one distinguishes between the forward mode and the reverse mode of AD. In our context of discretized optimal control problems, $F(x)$ is nothing else than the objective $\tilde{J}(u)$, so that $N = (N_t + 1)m$ and $M = 1$. Using the forward mode resembles a discrete sensitivity approach, cf. (4.5)–(4.6). I.e., the forward mode recursively computes approximations to $s(t^j)$. Conversely, the reverse mode involves a discrete analog to the continuous adjoint equation (4.8)–(4.9). These parallels have already been hinted at in Griewank [68].

For both modes, the time-complexity results are based on the operation count O_F of the underlying vector function F. Using the forward mode of AD, one *column* of the Jacobian ∇F can be calculated at no more than five times O_F [70]. One *row* of ∇F, i.e., the gradient of a scalar-valued component function of F, is obtained using the reverse mode in its basic form also at no more than five times O_F [70]. It is important to realize that this bound for the reverse mode is completely independent of the number of input variables, N. From the results above, one obtains immediately that the forward mode of AD allows the computation of Jacobians at an operation count of at most five times the number of *input* variables N times O_F. Conversely, the reverse mode allows the computation of Jacobians for at most five times the number of *output* variables M times O_F. However, the memory requirement of the basic reverse mode is proportional to the time needed to evaluate the function F itself. However, there are several strategies based on checkpointing to reduce this memory requirement and a remarkable decrease can be obtained effortless. In particular, for the time-integration problem considered in this paper there are optimal checkpointing schedules that allow a logarithmic increase in the memory requirement [76].

Analyzing our method presented in Subsection 4.3.3, one finds that computing the gradient $\nabla \tilde{J}(u)$ recursively from $\partial \tilde{J}(u)/\partial u^{N_t+1}$ down to $\partial \tilde{J}(u)/\partial u^1$ exactly matches the reverse mode of AD. The key observation is that the consistent objective gradient presentation—which depends on the forward integration scheme—can be generated automatically. Hence, the usage of an AD tool eliminates the hand-coding of derivative calculations, a rather involved and error-prone process. Moreover, it is to be expected that the SQP solver performs better when having access to the correct discrete gradient information. This conjecture is confirmed by the numerical results presented in Section 4.5. Nevertheless, one has to note that AD is applied here to compute the exact derivative of an approximation of the objective and do not yield an approximation to the exact derivatives of the objective. Hence, in numerous cases AD provides exactly what the user of AD requires, but one has to keep this difference in mind.

Derivatives have surfaced at many different locations in the discussion so far. At first sight, there are two obvious possibilities of where to apply AD:

The first is to generate the recursion scheme for the quantities λ^j using AD. The resulting backward integration can be viewed as the appropriate adjoint scheme for the adjoint differential equation (5.6) with terminal condition (4.9). We refer to this technique as *adjoining one forward step*. Note that AD automatically takes care of computing f_y and f_u. However, one has to give some thought to appropriately accumulate the adjoint-like quantities λ^j in order to obtain the desired gradient.

An alternative approach makes use of AD to directly compute the gradient $\nabla \tilde{J}(u)$. Since we have a scalar-valued cost function $\tilde{J}$ with many input variables u, the reverse mode of AD is preferable. It yields a code that implicitly features the computation of λ^j-like quantities from $j = N_t + 1$ down to $j = 1$.

In order to use the first technique some insight is needed to accumulate the desired gradient information. The benefit is a decrease in the overall run-time because some recomputations and storage operations can be avoided. The second technique represents an easy-to-use method, but leads to a larger temporal and/or spatial complexity, depending on the AD tool applied. Run-time results for both approaches are presented in Sec. 4.5.

4.4 Software

We used our own experimental software as an interface to the SQP solver NPSOL to conduct the numerical tests. This interface is described in the following subsection. Information regarding NPSOL is presented in the Sec. 4.4.2.

4.4.1 Discretizing the Continuous Problem

Our software has the following main features: The code has to provide the objective value $\tilde{J}(u)$ to NPSOL for a given vector of optimization variables u which correspond to the discretized control functions. To this end, the forward equation has to be solved because the objective value depends on the terminal state y^{N_t+1}. The integration of the forward equation is carried out using the classical fourth-order Runge-Kutta scheme over an equidistant time grid. Since values of f and thus of u are required not only at the time grid points t^j, but also at $t^{j+\frac{1}{2}}$ (see Section 4.3), linear or constant interpolation of the control is utilized. In order to allow flexibility there are problem-specific subroutines to compute the right hand side f, the initial values y_0 and to assess the terminal state via $\varphi(y^{N_t+1})$. Exchanging only the problem-specific parts, a variety of problems can easily be coded.

In addition to the computation of the state trajectory, derivative evaluations have to be performed. Our code offers four possibilities:

Finite Differences

When using finite differences to evaluate the objective gradient, the above ingredients to compute a solution of the forward equation are already sufficient to solve the optimization problem.

Straightforward Discretization of Continuous Adjoint Equation

In this case, after integrating the forward equation, the adjoint equation must be solved. This is done using the same Runge-Kutta scheme again. Whenever the right hand side of the adjoint equation $f_y(\cdot)$ depends on the state y, this requires interpolation of the discrete state y to intermediate grid points $t^{j+\frac{1}{2}}$ which can be done either in a constant fashion, i.e., $(y^{j+\frac{1}{2}} = y^{j+1})$ or in a linear fashion, i.e., $(y^{j+\frac{1}{2}} = (y^{j+1} + y^j)/2)$. Furthermore, the user has to provide subroutines to find the linearized right hand side $f_y(\cdot)$, the gradient of φ and $f_u(\cdot)^T\lambda$, cf. (4.12).

Application of AD

AD tools offer a convenient way to automatically generate the source code to evaluate the gradient in consistency with the discrete objective. For our numerical tests we used ODYSSÉE [127], developed by INRIA Sophia Antipolis, France. It is capable of differentiating FORTRAN 77 codes. Analyzing the dependencies, ODYSSÉE finds that the ODE integrator (i.e., the Runge-Kutta scheme) and the user-provided routines for the right hand side f and the objective φ have to be differentiated. The resulting code can be used with no further changes to compute the gradient of the objective. According to our experience, ODYSSÉE is an easy-to-use and reliable tool. Only two minor changes in our software were necessary: On the one hand, a goto-statement had to be eliminated. On the other hand, a subroutine call which contained the same parameter twice (once as input and once as output) had to be changed, using an auxiliary variable. After these rather minor modifications ODYSSÉE generates the desired code for $\nabla\tilde{J}(u)$ without any problems in a few seconds. ODYSSÉE was also used to differentiate only the subroutine accountable for one Runge-Kutta step, as referred to adjoining one forward step, see Section 4.3.4.

4.4.2 SQP Solver

For solving the optimization problem, we use Philip Gill's code NPSOL, which is one of the most renowned SQP codes. NPSOL employs a dense SQP algorithm. An augmented Lagrangian merit function ensures convergence from an arbitrary starting point. NPSOL is designed to minimize an arbitrary function subject to constraints which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints. The problems to solve may contain up to a few hundred constraints and variables, depending on the amount of memory available. NPSOL employs three tests of convergence only two of which apply for our examples due to the absence of nonlinear constraints. A sequence of optimization variables is considered converged when the norm of the search direction is small compared to the norm of the current control values and when the norm of the reduced gradient is small compared to the current objective value. For the numerical examples in the subsequent section, we left all the default tolerances unaltered. A detailed description of NPSOL is contained in Gill et. al. [62].

The user has to provide subroutines that define the objective and nonlinear constraint functions as well as optionally their gradients. Our software described in the previous subsection has exactly this functionality.

4.5 Examples

Example 4.5.1: Simple Problem from Economics

A trading company aims at maximizing its profit for the 8 months to come. Let the price of the trading goods $p(t)$ be known in advance. The company initially has assets K. It can buy and sell the goods only at a limited rate. To keep the good in stock, it must pay fees proportional to the amount in stock. The constant $a = 0.25$ describes the fee to store one item for one time unit (month).

Let $y_1(t)$ denote the assets at time $t \in [0,8]$ and let $y_2(t)$ be the amount of goods on stock. The control $u(t)$ denotes buying and selling activity, subject to the constraint $-1 \le u(t) \le 1$. The price for the goods is given by

$$p(t) = \begin{cases} 6 + 0.5t & 0 \le t \le 4 \\ 4 + t & 4 \le t \le 6 \\ 10 & 6 \le t \le 8. \end{cases}$$

We obtain the following optimal control problem:

$$\begin{aligned} \min \varphi(y(t_f)) &= -y_1(t_f) - p(t_f)y_2(t_f) & & \\ \text{s.t. } \dot{y}_1(t) &= -ay_2(t) - p(t)u(t) & \qquad y_1(0) &= 100 \\ \dot{y}_2(t) &= u(t) & \qquad y_2(0) &= 0 \end{aligned}$$

It is possible to derive the optimal control of this minimization problem analytically. It is of bang-bang type:

$$u(t) = \begin{cases} 1 & \text{for } 0 \le t \le \frac{16}{3} \\ -1 & \text{for } \frac{16}{3} < t \le 8. \end{cases}$$

The objective value is known to be $\varphi(y(t_f)) = -322/3 = -107.\overline{3}$. In this example, one obtains the adjoint equation (independent of the state and control)

$$\begin{aligned} \dot{\lambda}_1(t) &= 0 & \lambda_1(8) &= -1 \\ \dot{\lambda}_2(t) &= -a\lambda_1(t) & \lambda_2(8) &= -10. \end{aligned}$$

Its solution is

$$\lambda_1(t) = -1 \qquad\qquad \lambda_2(t) = -a(t-8) - 10. \tag{4.19}$$

Accordingly, the gradient (cf. (4.10)) of the continuous problem is

$$\begin{aligned} (D\tilde{J}(u)(t))^T &= f_u(\psi(u)(t), u(t), t)^T \lambda(t) \\ &= -p(t)\lambda_1(t) + \lambda_2(t) \\ &= p(t) - a(t-8) - 10. \end{aligned}$$

In the sequel we present some numerical results for the time discretization using $N_t = 8$ points in time. We expect the gradient obtained from straightforward application of the adjoint approach (Section 4.3.2) to differ from the true gradient. This can be observed in Fig. 4.1 for constant and linear interpolation of

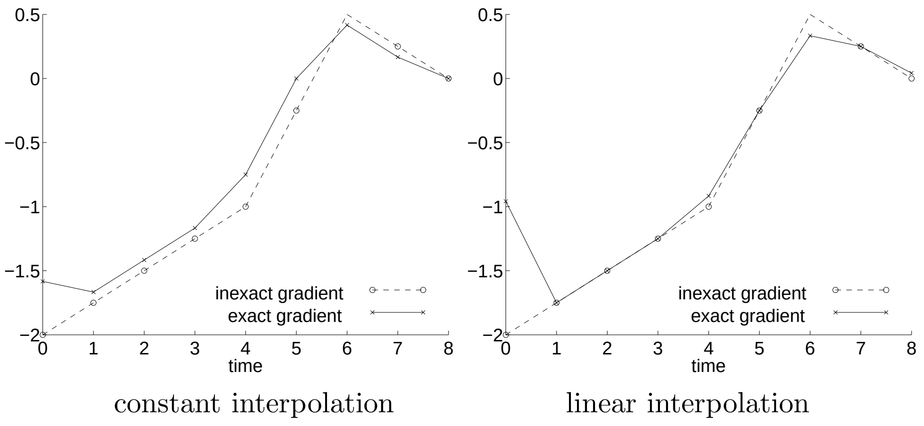

Figure 4.1: Inconsistency of the gradient for Example 4.5.1, constant and linear interpolation of the control, $N_t = 8$

the control, respectively, at the off-grid points $t^{j+\frac{1}{2}}$. These computations were performed at the initial guess $u^j = 0$.

It is interesting to note that the gradient in the case of constant interpolation is inexact everywhere except at $t = t_f$. In contrast to that, in the case of linear interpolation, the gradient is correct except at $t \in \{t_0, t_f\}$ and at the points of non-differentiability of p which appears in f_u. For this simple example, it is possible to trace back this incorrect gradient information.

Let us first consider the calculation of the adjoint quantities $\lambda^1, \dots, \lambda^9$. Using the fourth-order classical Runge-Kutta scheme we obtain

$$\lambda^9 = \begin{pmatrix} -1 \\ -10 \end{pmatrix} \qquad \lambda^j = \lambda^{j+1} + \begin{pmatrix} 0 \\ 0.25 \end{pmatrix} \quad j = 8, \dots, 1. \tag{4.20}$$

Applying AD to provide the gradient, the derivatives of the cost function with respect to the discrete state variables y^j—usually denoted by $\bar{y}^j$ in the AD context—are calculated. For the example considered here, the $\bar{y}^j$ equal the solution of the adjoint equation given by (4.20), i.e., $\lambda^j \equiv \bar{y}^j$. Furthermore, the λ^j are the exact solution of the continuous adjoint equation at $t = t^j$, cf. (4.19). This can be attributed to the fact that our Runge-Kutta scheme integrates an ODE with constant right hand side without error.

Hence, the inconsistency of the gradient shown in Fig. 4.1 can only result from different accumulations of the gradient. Using the straightforward approach, each component of $\nabla \tilde{J}(u)$ is computed as

$$\frac{\partial \tilde{J}(u)}{\partial u^j} = -p(t^j)\lambda_1^j + \lambda_2^j \qquad j = 1, \dots, 9, \tag{4.21}$$

for $N_t = 8$. It is important to note that this formula stays the same, whether constant interpolation or linear interpolation for the control is used. Therefore, the different influence of the two control interpolation modes on the computed objective value is completely neglected by the straightforward approach. This is not true if AD is utilized to compute the gradient. Here, we obtain for constant interpolation the formulas

$$\begin{aligned}
\frac{\partial \tilde{J}(u)}{\partial u^9} &= \frac{1}{6}\Bigg(- p(t^9)\lambda_1^9 + \lambda_2^9 \Bigg) \\
\frac{\partial \tilde{J}(u)}{\partial u^j} &= \left(-\frac{2}{3}p(t^{j+\frac{1}{2}}) - \frac{1}{3}p(t^j) \right) \lambda_1^j + \lambda_2^j + \frac{1}{12}\lambda_1^{j+1}, \qquad && j = 8, \dots, 2 \\
\frac{\partial \tilde{J}(u)}{\partial u^1} &= \left(-\frac{2}{3}p(t^{1+\frac{1}{2}}) - \frac{1}{6}p(t^1) \right) \lambda_1^1 + \frac{5}{6}\lambda_2^1 + \frac{1}{12}\lambda_1^2
\end{aligned}$$

for $N_t = 8$. Hence, the influence at the intermediate times $t^{j+\frac{1}{2}}$ comes into play. The consideration of this influence of u^j on the objective value and here particularly the offset $\frac{1}{12}\lambda_1^{j+1}$ is responsible for the altered gradient information in the case of constant interpolation.

For linear interpolation, AD generates the formulas

$$\frac{\partial \tilde{J}(u)}{\partial u^9} = \left(-\frac{1}{6}p(t^9) - \frac{1}{3}p(t^{9-\frac{1}{2}}) \right) \lambda_1^9 + \frac{1}{2}\lambda_2^9 - \frac{1}{24}\lambda_1^9 \tag{4.22}$$

$$\frac{\partial \tilde{J}(u)}{\partial u^j} = -\frac{1}{3}\Bigg(p(t^{j+\frac{1}{2}}) + p(t^j) + p(t^{j-\frac{1}{2}}) \Bigg)\lambda_1^j + \lambda_2^j, \qquad j = 8, \dots, 1 \tag{4.23}$$

$$\frac{\partial \tilde{J}(u)}{\partial u^1} = \left(-\frac{1}{3}p(t^{1+\frac{1}{2}}) - \frac{1}{6}p(t^1) \right) \lambda_1^1 + \frac{1}{2}\lambda_2^1 + \frac{1}{24}\lambda_1^2. \tag{4.24}$$

One deduces from (4.22) that the term $-\frac{1}{24}\lambda_1^9$ causes the different gradient information at time $t^9 = 8$. At the times t^7, t^5, t^3, and t^2 the formula (4.23) equals the gradient calculation (4.21) because of the linear behavior of the function $p(\cdot)$. Hence, at these points in time the straightforward approach and the AD approach yield the same gradient information. At t^8 and t^4, the piecewise linear structure of $p(\cdot)$ leads to different values of the formulas (4.23) and (4.21). Finally, for t^1, i.e., $t = 0$, the formulas (4.24) and (4.21) are completely different. This fact explains the large deviation of the gradient information at $t = 0$.

As described before, we apply the AD tool ODYSSÉE to derive code for the computation of the discrete gradient. Therefore, besides the consistency of the gradient, the run-time needed by the automatically generated code has to be analyzed. The corresponding measurements to compute $\tilde{J}(u)$ and $\nabla\tilde{J}(u)$ are given in Tab. 4.1. As can be seen, the ratio of the two run-times for

	constant interpolation			linear interpolation		
N_t	8	32	128	8	32	128
$\tilde{J}(u)$	0.0001 s	0.0006 s	0.0060 s	0.0001 s	0.0006 s	0.0061 s
$\nabla\tilde{J}(u)$	0.0005 s	0.0037 s	0.0324 s	0.0006 s	0.0040 s	0.0324 s
ratio	5.00	6.17	5.4	6.00	6.67	5.31

Table 4.1: Run-time of gradient calculation using ODYSSÉE, Example 4.5.1

evaluating $\nabla\tilde{J}(u)$ and $\tilde{J}(u)$ reaches almost the theoretical bound of five, the gap being caused by ODYSSÉE recomputing some intermediate results of the forward integration. These recomputations are caused by the checkpointing strategy implemented in ODYSSÉE.

As a next step, we investigate how the inconsistency of the gradient affects the SQP solver convergence. For this purpose, we did several test runs with $N_t \in \{8, 32, 128\}$ time grid points. The corresponding results, including run-time, major iteration count, and the objective value are presented in Tab. 4.2 and Tab. 4.3 for constant and linear interpolation of the control, respectively.

	CPU Time [sec]			Major it.			Objective		
N_t	8	32	128	8	32	128	8	32	128
FD	0.019	0.325	43.298	4	17	67	-107.166	-107.322	-107.332
SF	0.014	0.037	0.693	3	13	51	-107.166	-107.322	-107.332
AD$_{\tilde{J}}$	0.017	0.104	3.611	4	16	61	-107.166	-107.322	-107.332
AD$_a$	0.018	0.099	3.207	4	16	61	-107.166	-107.322	-107.332

Table 4.2: SQP convergence behavior, Example 4.5.1, constant interpolation of the control, $N_t = 8|32|128$

Here as throughout, FD denotes the finite differences approach. SF stands for the straightforward evaluation of the gradient as described in Subsec. 4.3.2. AD$_{\tilde{J}}$ denotes the approach to apply AD to the evaluation of the objective $\tilde{J}(u)$,

	CPU Time [sec]			Major it.			Objective		
N_t	8	32	128	8	32	128	8	32	128
FD	0.022	0.386	46.579	6	18	69	-107.250	-107.328	-107.333
SF	0.014	0.039	0.688	3	13	51	-107.208	-107.325	-107.332
AD$_{\tilde{J}}$	0.019	0.112	3.348	5	14	52	-107.250	-107.328	-107.333
AD$_a$	0.018	0.101	2.969	5	14	52	-107.250	-107.328	-107.333

Table 4.3: SQP convergence behavior, Example 4.5.1, linear interpolation of the control, $N_t = 8|32|128$

whereas AD$_a$ marks the approach to differentiate the Runge-Kutta step using AD and to subsequently accumulate the desired gradient by hand-written code, see Subsection 4.3.4.

From the results achieved one can conclude several things. First of all, it is verified that the finite difference approach to compute the gradient is quite exact, i.e., converges to the analytical solution, but inefficient. This observation still holds even if the finite difference approximation of the objective gradient is done more efficiently, using the trivial fact that $\partial\psi^j(u)/\partial u^i = 0$ whenever $i > j$. This can save up to 50% of computational effort off the times given in the run-time tables.

Secondly, the straightforward approach is fast but leads to an inexact solution if the control is linearly interpolated. This is surprising because the gradient information is exact at four points of time. Conversely, the inexactness at eight points for the constant interpolation does not seem to influence the optimization process. The low CPU times result on the one hand from the efficient (hand-coded) computation of the adjoint information, and on the other hand from the reduced major iteration numbers since the stopping criteria are fulfilled earlier.

Thirdly, the AD approach performs best for the linear interpolation of the control. It is much faster than finite differences and yet exact, i.e., computes the analytical solution. Furthermore, one sees that automatic differentiation of the Runge-Kutta steps and accumulation of the gradient by hand leads to only minor run-time savings of at most 10% for Example 4.5.1.

While for this simple example the straightforward approach—at least in the case of constant control interpolation—is competitive, it will cause the SQP solver fail to converge in Example 4.5.2.

Example 4.5.2: Rayleigh Equation

The Rayleigh equation describes a so-called tunnel diode oscillator. This circuit contains a power source whose voltage $u(t)$ is the control variable. The current $x(t)$ obeys the ODE

$$\ddot{x}(t) = -x(t) + \dot{x}(t)(1.4 - 0.14\dot{x}(t)^2) + 4u(t).$$

As an objective function we take , cf. [18, 140, 99],

$$I(y,u) = \int_0^{t_f} u(t)^2 + x(t)^2 \, dt.$$

This problem can easily be transformed into the form (4.1)–(4.4) described in Section 4.2 via $y_1 = x$, $y_2 = \dot{x}$:

$$\begin{aligned}
\dot{y_1}(t) &= y_2(t) \\
\dot{y_2}(t) &= -y_1(t) + y_2(t)(1.4 - 0.14 y_2(t)^2) + 4u(t) \\
\dot{y_3}(t) &= u(t)^2 + y_1(t)^2
\end{aligned}$$

and objective

$$J(y) = \varphi(y(t_f)) = y_3(t_f).$$

We assume initial conditions

$$y_1(0) = -5, \qquad y_2(0) = -5, \qquad y_3(0) = 0,$$

and impose the control constraint $|u(t)| \le 1$. The final time is $t_f = 2.5$. Note that in contrast to the previous example, the right-hand side is non-linear in the state variable y, so the adjoint equation depends on the state:

$$-\dot{\lambda} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 1.4 - 0.42 y_2^2 & 0 \\ 2y_1 & 0 & 0 \end{bmatrix}^T \lambda$$

with terminal conditions

$$\lambda_1(t_f) = 0, \qquad \lambda_2(t_f) = 0, \qquad \lambda_3(t_f) = 1.$$

Again, we compare the consistent gradient to the inconsistent gradient from the straightforward adjoint approach. Since the right hand side of the adjoint equation $f_y(\cdot)$ now depends on the state, evaluations of $y^{j+\frac{1}{2}}$ are necessary, provided that the same Runge-Kutta scheme is used. As described in Section 4.4.1, we compare both constant and linear interpolation of the state. The corresponding inconsistent gradients are shown in Figs. 4.2 and 4.3. Tab. 4.4 states the run-times needed to compute $\nabla\tilde{J}(u)$ and $\tilde{J}(u)$ as well as their ratio. Once more, the results achieved reach almost the theoretical bound of 5 and are even better than in the first example. The second fact may be caused by the more complex model to be differentiated. This is typically observed and is due to the less effective compiler optimization for the forward integration.

Tabs. 4.5 and 4.6 show the important results concerning the optimization process. Here, SF_c indicates constant interpolation of the state during the adjoint calculation using the straightforward approach, while SF_l denotes linear interpolation.

It has to be mentioned that in this example NPSOL does not terminate the optimization process until the given maximal number of major iterations is

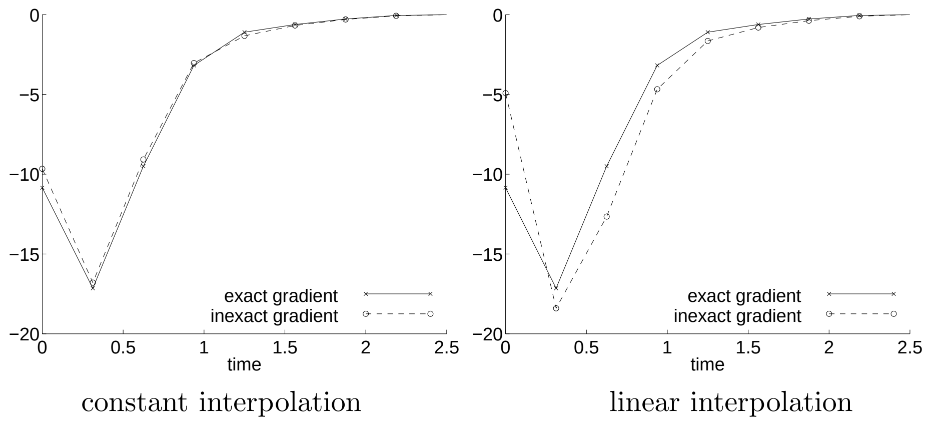

Figure 4.2: Inconsistence of the gradient, Example 4.5.2, constant interpolation of the control, constant and linear interpolation of the state, $N_t = 10$

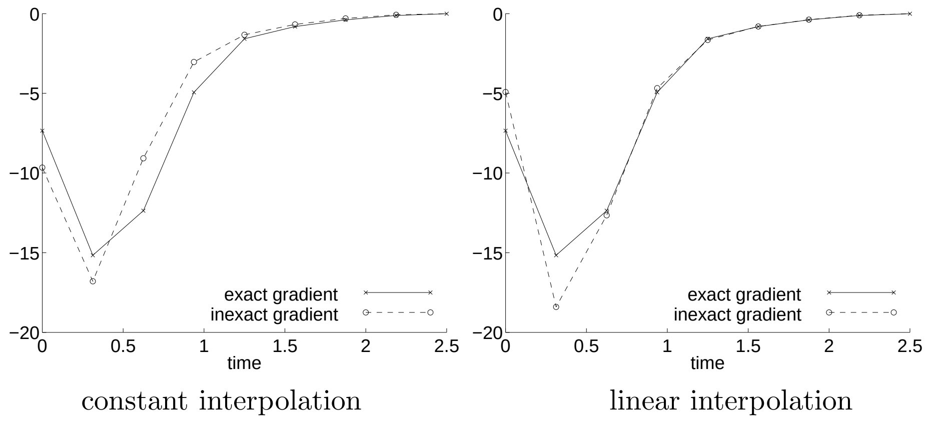

Figure 4.3: Inconsistence of the gradient, Example 4.5.2, linear interpolation of the control, constant and linear interpolation of the state, $N_t = 10$

reached. This is true for all combinations of constant and linear interpolation of the state for the straightforward adjoint calculation. Nevertheless, after the (comparably few) major iterations stated in parenthesis, the specified objective value is reached. After that, NPSOL chooses very small step sizes and practically stagnates. The run-times stated for $\mathrm{SF_c}$ and $\mathrm{SF_l}$ are for the number of iterations given, forcibly terminating NPSOL.

The finite difference approach and the AD method yield a solution close to the one presented in [18] which has an objective value of 42.80743761. Furthermore, both ways of gradient calculations lead to the same objective value that is smaller than the one computed with the SF technique. However, the derivative calculation based on AD accelerates the computation enormously relative to FD. To explain this run-time behavior, we observe that the operation count $O_{\tilde{J}(u)}$ is asymptotically linear in the number of time grid points N_t. In addition, we have verified one of the AD paradigms for the reverse mode, viz:

	constant interpolation			linear interpolation		
N_t	8	32	128	8	32	128
$\tilde{J}(u)$	0.0001 s	0.0006 s	0.0060 s	0.0001 s	0.0007 s	0.0062 s
$\nabla\tilde{J}(u)$	0.0005 s	0.0033 s	0.0309 s	0.0006 s	0.0034 s	0.0321 s
ratio	5.00	5.50	5.15	6.00	4.85	5.18

Table 4.4: Run-time of gradient calculation using ODYSSÉE, Example 4.5.2

	CPU Time [sec]			Major it.			Objective		
N_t	8	32	128	8	32	128	8	32	128
FD	0.023	0.411	21.628	9	17	19	41.1708	42.7991	42.8074
SF$_c$	0.016	0.038	0.696	(5)	(5)	(22)	41.3785	42.8161	42.8103
SF$_l$	0.018	0.036	0.426	(4)	(5)	(21)	41.5179	42.8207	42.8110
AD$_{\tilde{J}}$	0.021	0.093	1.015	9	17	19	41.1708	42.7991	42.8074
AD$_a$	0.020	0.074	0.758	9	17	19	41.1708	42.7991	42.8074

Table 4.5: SQP convergence behavior, Ex. 4.5.2, constant interpolation of u

The ratio $\nabla\tilde{J}(u)$ vs. $\tilde{J}(u)$ being independent of the number of input variables u, i.e., independent of N_t. Combining these two findings, we conclude that the operation count for computing the objective plus its gradient using AD's reverse mode is asymptotically *linear* in the number of grid points. In contrast, it is easily verified that the corresponding operations count using finite differences is asymptotically *quadratic* in N_t.

Therefore, the statements made for the first example concerning the various methods remain mainly true here. However, the use of the straightforward approach is completely ruled out since the inaccuracy of the supplied derivatives causes the SQP solver to stagnate.

4.6 Conclusions

We have studied the solution of discretized optimal control problems for ODEs. The discretization was carried out by a Runge-Kutta scheme. While it is appealing to use adjoint information generated by the same scheme, this straightforward approach leads to inconsistent gradients in general. However, SQP solvers rely on consistent and exact gradient information and can fail to converge if inexact gradients are provided as was shown in Example 4.5.2.

Despite some theoretical studies how methods should cope with inexact function and derivative information [26], one should look for alternatives. One obvious variant is given by the finite difference approach. However, using finite differences the run-time needed to compute the gradient grows quadratically in the number of controls. In order to analyze the influence of this behavior on the optimization process for our examples, a profiling of the optimization on a Origin 2000 using ssrun was done. We find that NPSOL needed no more than 10% of the computing time. This small portion is caused by the easy constraints,

	CPU Time [sec]			Major it.			Objective		
N_t	8	32	128	8	32	128	8	32	128
FD	0.030	0.500	15.165	17	31	22	41.1944	42.7985	42.8074
$\mathrm{SF_c}$	0.016	0.036	0.535	(4)	(5)	(22)	41.2670	42.8104	42.8101
$\mathrm{SF_l}$	0.017	0.036	0.546	(4)	(5)	(22)	41.3755	42.8185	42.8107
$\mathrm{AD}_{\tilde{J}}$	0.028	0.154	1.250	17	30	23	41.1944	42.7985	42.8074
AD_a	0.025	0.124	1.020	17	30	23	41.1944	42.7985	42.8074

Table 4.6: SQP convergence behavior, Ex. 4.5.2, linear interpolation of u

namely only box-constraints for the controls. Because of the small percentage required for Npsol, the computing time needed for calculation gradient information plays an important role for the overall run-time. Hence, all savings that can be achieved calculating derivatives have a direct and important impact on the total run-time. Therefore, other ways to compute the gradient come into play.

As can be seen also from Hager [84], finding the correct integration scheme for the continuous adjoint equation is not a trivial task. We suggest using automatic differentiation tools to relieve the user of this burden, allowing convenient computation of the objective gradient.

It is interesting to note that automatic differentiation is receiving an increasing amount of attention in the optimization community. The code Snopt, a successor to Npsol suitable for large sparse optimization problems, has recently been furnished with an interface to Adifor, another AD tool capable of the forward mode, cf. Gertz et. al. [58].

While finding the correct adjoint integration scheme (i.e., by adjoining the forward scheme by hand) and implementing it to solve the continuous adjoint equation is still the ideal solution in terms of CPU time, it is not feasible in practical situations: It requires the right hand side of the adjoint equation $f_y(\cdot)$ to be available symbolically, which can not be assumed for complicated dynamical systems in contrast to our examples.

In a companion paper, we consider a realistic control problem with complicated control and also state constraints [66]. In order that this problem be solved efficiently by an SQP code, the constraint Jacobian will be computed using AD. In addition, second order derivatives are generated applying AD for the calculation of parametric sensitivities.

Acknowledgements

We would like to thank the anonymous referees for their careful reading of the manuscript.

Chapter 5

Automatic Differentiation for Explicit Runge-Kutta Methods for Optimal Control

Andrea Walther

Journal of Computational Optimization and Applications,
Volume 36, pp. 83 – 108 (2007)

Abstract:
This paper considers the numerical solution of optimal control problems based on ODEs. We assume that an explicit Runge-Kutta method is applied to integrate the state equation in the context of a recursive discretization approach. To compute the gradient of the cost function, one may employ Automatic Differentiation (AD). This paper presents the integration schemes that are automatically generated when differentiating the discretization of the state equation using AD. We show that they can be seen as discretization methods for the sensitivity and adjoint differential equation of the underlying control problem. Furthermore, we prove that the convergence rate of the scheme automatically derived for the sensitivity equation coincides with the convergence rate of the integration scheme for the state equation. Under mild additional assumptions on the coefficients of the integration scheme for the state equation, we show a similar result for the scheme automatically derived for the adjoint equation. Numerical results illustrate the presented theoretical results.

5.1 Introduction

For numerous real-world applications, it is possible to influence the state or process under consideration by varying the value of several parameters. Quite often, this relation is modeled by a system of ordinary differential equations that involves control functions. The control functions can be used to minimize a cost function to obtain a desired state or process. There are many books and research papers on optimal control problems based on ODEs and many theoretical as well as numerical aspects are well studied.

In order to compute an optimal control for a given problem, we may employ derivative information in several representations. In general we distinguish two classes. The first one comprises approaches that are based on the Pontryagin maximum principle and yields a multi-point boundary value problem for the state and the additional adjoint equation. Hence, these so-called *indirect methods* correspond to the solution of the continuous KKT system. An introduction to this technique can be found in Pesch [121]. Implementation issues and examples of applications are contained e.g. in Bulirsch et. al. [17] and Hiltmann [91]. A software package based on an indirect method is described in Oberle [118].

The second class is termed *direct methods*. Here, in a first step one transforms the infinite-dimensional optimal control problem into a finite-dimensional optimization task. For that purpose, one may use collocation methods, see e.g. von Stryk [145] and Betts [12]. Subsequently, one solves the complete resulting discrete KKT system. Consequently, both the discretized state as well as the discretized control are treated as optimization variables by the optimization algorithm. These methods, which are known as *full discretization* concepts, lead to a huge number of equality constraints that have an almost block-diagonal Jacobian matrix. As alternative, only the discretized controls can be treated as optimization variables. Hence, the objective value can only be obtained by one integration of the state equation and the optimization is based on the non-trivial gradient of the cost function with respect to the discretized controls. These approaches are usually called *recursive discretization*. The name is inspired by the fact that all classical ODE integration schemes (e.g. Runge-Kutta schemes) define the solution to the state ODE recursively, time step by time step. We point the reader to Büskens and Maurer [19] as well as Bock and Plitt [14] for more information on these methods. One major problem of this approach is to find a correct method to compute the gradient information. For that purpose, one may derive a suitable integration scheme for the adjoint differential equation. Usually, these adjoint discretization schemes do not coincide with the integration schemes used for the state equation (see e.g. [18, 152]). Therefore, their computation is by no means obvious.

Over the last decades, several research groups have developed the technique of Automatic or Algorithmic Differentiation (AD), which allows the generation of exact derivative information for a given code segment. A comprehensive introduction to this method can be found in Griewank [70]. Furthermore, there are numerous examples for the application of AD in the optimal control context, see e.g. [25, 27, 94]. Theoretical aspects were for example studied in [47, 48]. Applying recursive discretizations, we can use AD to compute the consistent objective gradient which depends on the forward integration scheme as presented, for example, in [48]. That is, AD yields the exact discrete gradient information for the chosen discretization of the state equation. Therefore, the usage of an AD tool eliminates the hand-coding of derivative calculations, a rather involved and error-prone process. However, one has to keep in mind that AD computes the exact derivative of an approximation of the objective and may not yield an approximation to the exact derivatives of the objective. The purpose of this paper is to analyze this discrepancy and its impacts for the recursive discretization in more detail. That means, we analyze for a fixed control function the convergence rate of the discrete adjoint associated with the corresponding discretized control problem to the corresponding continuous adjoint. For that purpose, we present the discretization schemes automatically derived by AD to integrate the sensitivity equation and the adjoint equation of the underlying optimal control problem. Subsequently, we prove that the discretization method obtained for the sensitivity equation inherits the convergence properties of the discretization scheme used for the state equation. A similar result can be proved for the adjoint equation provided that some additional assumptions on the coefficients of the original discretization method are fulfilled. Moreover, we sketch the computation of the required gradient information using the approximations of the sensitivity equation and the adjoint equation, respectively. However, we also show that the gradient information provided by AD is the exact discrete derivative information, which need not coincide with the one based on the approximation of the sensitivity and adjoint equation, respectively. This aspect is subject of ongoing research and was studied for example in [48, 67].

Hence, we choose for a given fixed control the discretize-then-optimize approach and compare the approximation that we achieve with the continuous solutions of the optimize-then-discretize method. The analysis in this paper can be considered as a continuation of [67], where the effect of this difference is examined by means of two numerical examples. Here, analyzes the discrete adjoint information arising for one subproblem of a gradient-based optimization procedure. Therefore, it is related to [44, 84], where the convergence rate of the solution of the discretized control problem to the solution of the continuous problem is studied for the full discretization approach. That is, these papers study the difference between the optimal solutions of the continuous optimization problem and the discrete version.

The present paper has the following structure. In Section 5.2 we introduce the continuous optimal control problem to be considered. Furthermore, we discuss two possibilities of computing the objective gradient for the continuous formulation. Subsequently, we present the corresponding finite-dimensional optimization problem in Section 5.3. Section 5.4 introduces discretization schemes for computing an approximate solution of the sensitivity equation. We prove some convergence properties of the schemes generated by the forward mode of AD. Convergence results for the recursive discretization applying the reverse more of AD are proved in Section 5.5. Section 5.6 gives a numerical illustration of the obtained results. Finally, we draw some conclusions in Section 5.7.

5.2 The Continuous Optimal Control Problem and its Gradient

We consider the following optimal control problem:

$$\begin{aligned} \text{Minimize} \quad & J(y) &&= \varphi(y(t_f)) && && (5.1)\\ \text{s.t.} \quad & \frac{dy}{dt}(t) &&= f(y(t), u(t)) && t \in [0, t_f] && (5.2)\\ & y(0) &&= y^0 && && (5.3)\end{aligned}$$

where $y(t) \in \mathbb{R}^n$ denotes the state and $u(t) \in \mathbb{R}^m$ denotes the control. The dynamics are given by a right-hand side function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and $y^0 \in \mathbb{R}^n$ describes the initial conditions. To compute the value of the objective function, $\varphi : \mathbb{R}^n \to \mathbb{R}$ evaluates the state at the prescribed final time t_f. For simplicity, we assume that all functions are sufficiently smooth such that the existence of a solution is ensured and all necessary differentiations can be performed. Furthermore, no constraints on the control are considered yet. Box constraints and more complicated constraints on the control will be subject of further studies.

The state y is fully determined by the control u. Therefore, we may introduce an objective in terms of the control variable only. To this end, we define for a given control u and a corresponding solution $y = \psi(u)$ of the state equation (5.2)–(5.3) the objective function

$$\tilde{J}(u) = J(\psi(u))$$

depending only on u. Since optima are locally characterized as roots of the gradient, we have a closer look at its representation. For that purpose, let the symbol D denote Fréchet derivatives in infinite-dimensional spaces. Applying the chain rule, one obtains

$$D\tilde{J}(u) = DJ(\psi(u))\, D\psi(u).$$

Now, one can employ two distinct ways for calculating $D\tilde{J}(u)$. The first one determines for a given control u the sensitivity $s(t) \in \mathbb{R}^n$ in the direction of $d(t) \in \mathbb{R}^m$ using the sensitivity equation:

$$\frac{ds}{dt}(t) = f_y(\psi(u)(t), u(t))\, s(t) + f_u(\psi(u)(t), u(t))\, d(t), \qquad s(0) = 0. \tag{5.4}$$

One obtains immediately the *forward* or *sensitivity* representation of the gradient:

$$D\tilde{J}(u)d = \nabla\varphi(\psi(u)(t_f))\, s(t_f)\,. \tag{5.5}$$

Here, the attribute *forward* refers to the fact that the sensitivity equation has to be integrated in forward direction from 0 to t_f to find the Gâteaux variation $D\tilde{J}(u)d$. It should be noted that in (5.5), $\nabla\varphi(\psi(u)(t_f))$ denotes the gradient of φ with respect to y evaluated at $\psi(u)(t)$, and not the total derivative of the composite function $\varphi \circ \psi$.

The following alternative computes the complete $D\tilde{J}(u)$ by employing the adjoint differential equation

$$-\frac{d\lambda}{dt}(t) = f_y(\psi(u)(t), u(t))^T\, \lambda(t), \qquad \lambda(t_f) = \nabla\varphi(\psi(u)(t_f))^T. \tag{5.6}$$

The solution $\lambda(t) \in \mathbb{R}^n$ of this linear ODE is called the adjoint variable and yields the gradient's *backward* or *adjoint* representation [16, Section 2.4]

$$(D\tilde{J}(u)(t))^T = f_u(\psi(u)(t), u(t))^T \lambda(t) \in \mathbb{R}^{m\times 1}. \tag{5.7}$$

It follows that one has to integrate the adjoint equation in *backward* direction from t_f to t for calculating the gradient at time t. The connection between the interpretation of the objective gradient as a function depending on time and its function space interpretation is illustrated by the following formula for the Gâteaux variation in the direction of $\bar{u}$:

$$D\tilde{J}(u)^T \bar{u} = \int_0^{t_f} D\tilde{J}(u)(t)^T \, \bar{u}(t) \, dt. \tag{5.8}$$

All AD-tools that the author is aware of provide the forward mode which can be seen as a discrete version of the sensitivity equation. On the other hand, not all AD-tools allow the application of the reverse mode that forms a discrete analog of the adjoint method. Therefore, we will consider sensitivity-based methods as well as adjoint-based methods to compute gradient information in this paper.

5.3 The Discrete Optimal Control Problem

To solve the problem (5.1)–(5.3) numerically, we have to perform some discretizations. Therefore, assume that the time interval $[0, t_f]$ is divided into $N-1$ sub-intervals of equal length. Then the time grid consists of points

$$t_i = (i-1) \cdot h \quad \text{for} \quad i = 1, \ldots, N \tag{5.9}$$

where $h = t_f/(N-1)$ is the time step length. For simplicity, we restrict here the presentation to uniform grids.

All control and state components will be approximated at the grid points only, where we use the notation

$$\begin{array}{lll} y_i & \text{to approximate} & y(t_i), \quad \mathbf{y} = (y_1, \ldots, y_N)^T \in \mathbb{R}^{Nn}, \\ u_i & \text{to approximate} & u(t_i), \quad \mathbf{u} = (u_1, \ldots, u_N)^T \in \mathbb{R}^{N}. \end{array}$$

Here, we restrict the analysis to one distributed control for notational simplicity. The argumentation for several distributed controls only complicates the formulation and can be easily integrated. We obtain the finite-dimensional NLP problem

$$\text{Minimize} \quad \tilde{\mathbf{J}}(\mathbf{u}) \quad = \quad \varphi(\boldsymbol{\psi}_N(\mathbf{u}))$$

where $\tilde{\mathbf{J}} : \mathbb{R}^{Nm} \to \mathbb{R}$ and $\mathbf{y} = \boldsymbol{\psi}(\mathbf{u}) = (\boldsymbol{\psi}_1(\mathbf{u}), \ldots, \boldsymbol{\psi}_N(\mathbf{u}))^T$ approximately satisfies the state ODE (5.2) of the problem. Throughout the paper, we took $\boldsymbol{\psi}$ to represent an explicit Runge-Kutta scheme with constant step size h.

In dependence on the specific Runge-Kutta scheme applied, the control function u has to be evaluated at arguments $t_i + c_j h$ that may not lie on the time grid (5.9). Then, one can consider these intermediate values as independent optimization variables or as auxiliary values that are computed by an interpolation scheme using the values of $\mathbf{u}$ on the grid points. The first approach is applied for example in [84] for the full discretization. The latter is common for recursive discretization methods. Therefore, we assume throughout that the intermediate values $\tilde{u}_j$ of the control function at time $t_i + c_j h$, $0 < c_j < 1$, are computed by an interpolation polynomial p.

For constant and linear interpolation at time $\tilde{t}$ with $t_i < \tilde{t} < t_{i+1}$, only the values of u_i and u_{i+1} are required. Hence, the number of unknowns equals Nm. Quadratic interpolation is rarely used due to the resulting oscillations in the approximation of the control. Instead, one applies more frequently cubic interpolation based on a cubic spline, see e.g. [124]. This approach requires in addition to the Nm unknowns for linear interpolation only the values of u'_1 and u'_N. The corresponding coefficients of the polynomials are defined by

$$\alpha_i = u_i, \quad \beta_i = \frac{u_{i+1} - u_i}{h} - h\frac{2\gamma_i + \gamma_{i+1}}{6}, \quad \delta_i = \frac{\gamma_{i+1} - \gamma_i}{h}$$

where $\gamma = (\gamma_1, \dots, \gamma_N)$ is determined by the solution of a tridiagonal system

$$D\gamma = r \tag{5.10}$$

with a system matrix D independent of u. The right-hand side r is defined as

$$r_1 = \frac{6}{h}\left(\frac{u_2 - u_1}{h} - u'_1\right), \qquad r_N = \frac{6}{h}\left(u'_N - \frac{u_N - u_{N-1}}{h}\right),$$
$$r_i = \frac{3(u_{i+1} - 2u_i + u_{i-1})}{h^2}, \qquad i = 2, \dots, N-1.$$

Hence, the cubic interpolation is available at low cost. For the sake of notational simplicity, we will assume that the values of u'_1 and u'_N are fixed. However, the presented analysis can be easily extended to allow also varying derivatives. The discussed interpolation methods up to fourth order are shown in Table 5.1 and seem to cover numerous applications of the recursive discretization. To abbreviate the notation, we combine the first two arguments of the interpolation polynomial that belong to the control to one argument $v_i \equiv (u_i, u_{i+1}) \in \mathbb{R}^2$.

Type	constant	linear	cubic
κ	1	2	4
$p(v_i, t)$	u_i	$(u_{i+1} - u_i t)/h + u_i$	$\alpha_i + \beta_i t + \gamma_i t^2/2 + \delta_i t^3/6$

Table 5.1: Interpolation polynomials $p(v_i, t)$ of order ν

Then, we obtain the following algorithm for computing the discrete state vector $\mathbf{y}$ and the value of the objective function:

Algorithm 1: Integration of state equation

```
$y_1 = y^0$
For $i = 1, N-1, 1$
    For $j = 1, s, 1$
        $\tilde{y}_j = y_i + h \sum_{l=1}^{j-1} a_{jl}\, k_l, \qquad \tilde{u}_j = p(v_i, c_j h)$
        $k_j = f(\tilde{y}_j, \tilde{u}_j)$
    $y_{i+1} = y_i + h \sum_{j=1}^{s} b_j k_j$
```

$$\tilde{\mathbf{J}} = \varphi(y_N)$$

with $c_j = \sum_{l=1}^{j-1} a_{jl}$ and $a_{jl} = 0$ for $l \geq j$. In the following sections we will extend Algorithm 1 to compute in addition to the state and the cost function also derivative information.

5.4 Forward Mode of AD in Recursive Discretization

For a given vector function F evaluated at a certain point x and a direction $\dot{x}$, the scalar forward mode of AD yields the product $F'(x)\dot{x}$, i.e. the Jacobian $F'(x)$ multiplied by the vector $\dot{x}$ from the right. The evaluation cost for such a product can be bounded above by the product of a small constant and the cost to evaluate the function itself. In the literature the value of the constant varies between three and five depending on the considered computational complexity estimates [70].

In our context of an optimal control problem, the discrete evaluation of the cost function $\tilde{\mathbf{J}}$ given by Algorithm 1 defines the vector function F that has to be differentiated. Using the direction $\dot{\mathbf{u}} = (\dot{u}_1, \ldots, \dot{u}_N)^T \in \mathbb{R}^N$, the forward mode of AD applied to F yields the following integration procedure [70]:

Algorithm 2: Applying AD, forward differentiation of state equation

```
$y_1 = y^0, \quad \dot{y}_1 = \mathbf{0}_n$
For $i = 1, N-1, 1$
  For $j = 1, s, 1$
    $\tilde{y}_j = y_i + h \sum_{l=1}^{j-1} a_{jl} k_l, \quad \tilde{u}_j = p(v_i, c_j h),$
    $\dot{\tilde{y}}_j = \dot{y}_i + h \sum_{l=1}^{j-1} a_{jl} \dot{k}_l \quad \dot{\tilde{u}}_j = p_u(v_i, c_j h)\dot{\mathbf{u}}$
    $k_j = f(\tilde{y}_j, \tilde{u}_j), \quad \dot{k}_j = f_y(\tilde{y}_j, \tilde{u}_j)\dot{\tilde{y}}_j + f_u(\tilde{y}_j, \tilde{u}_j)\dot{\tilde{u}}_j$
  $y_{i+1} = y_i + h \sum_{j=1}^{s} b_j k_j, \quad \dot{y}_{i+1} = \dot{y}_i + h \sum_{j=1}^{s} b_j \dot{k}_j$
$\tilde{\mathbf{J}} = \varphi(y_N), \quad \dot{\tilde{\mathbf{J}}} = \nabla\varphi(y_N)\dot{y}_N$
```

Here, the customary notation in AD literature is used. Note that for cubic interpolation, the derivative $p_u(v_i, c_j h)$ takes also the differentiation of the coefficients $\alpha_i, \beta_i, \gamma_i, \delta_i$ with respect to $\mathbf{u}$ into account.

Employing the recursive discretization for solving the optimal control problem (5.1)–(5.3), one is interested in computing the complete gradient $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ of the cost function with respect to the control variables $\mathbf{u} \in \mathbb{R}^N$. One obtains the exact discrete derivative information $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ using the forward mode by choosing the N unit vectors as directions $\dot{\mathbf{u}}$ because of the following argument. Applying the chain rule, it follows that the derivative of $\tilde{\mathbf{J}}$ with respect to the ith component u_i has the representation

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial u_i} = \sum_{l=2}^{N} \nabla\varphi(\boldsymbol{\psi}_N(\mathbf{u})) \frac{\partial \boldsymbol{\psi}_N(\mathbf{u})}{\partial \boldsymbol{\psi}_{N-1}(\mathbf{u})} \cdots \frac{\partial \boldsymbol{\psi}_{l+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_l(\mathbf{u})} \frac{\partial \boldsymbol{\psi}_l(\mathbf{u})}{\partial u_i}. \tag{5.11}$$

Let $e_i \in \mathbb{R}^N$ be the ith unit vector. Then, one can conclude from the statements for $\dot{\tilde{u}}_j$, $\dot{k}_j$ and $\dot{y}$ that in the step $i = 1$ the derivative of $\boldsymbol{\psi}_2$ with respect to u_i is added to $\dot{y}_2$. Furthermore from the statements for $\dot{\tilde{y}}_j$ and $\dot{k}_j$, one can see that in step $i > 1$ the vector $\dot{y}_i$ is first used to compute the derivative of $\boldsymbol{\psi}_{i+1}(\mathbf{u})$ with respect to $\boldsymbol{\psi}_i(\mathbf{u})$ as well as the derivative of $\boldsymbol{\psi}_{i+1}(\mathbf{u})$ with respect to u_i. Subsequently, these values together with $\dot{y}_i$ form $\dot{y}_{i+1}$, i.e. one has

$$\dot{y}_{i+1} = \sum_{l=2}^{i+1} \frac{\partial \boldsymbol{\psi}_{i+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_i(\mathbf{u})} \cdots \frac{\partial \boldsymbol{\psi}_{l+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_l(\mathbf{u})} \frac{\partial \boldsymbol{\psi}_l(\mathbf{u})}{\partial u_i}.$$

Finally, the vector $\dot{y}_N$ is multiplied by $\nabla\varphi(\boldsymbol{\psi}_N(\mathbf{u}))$ to obtain the component of the gradient belonging to u_i. Hence, for evaluating $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ one has to propagate N unit vectors through Algorithm 2. Obviously, this way to calculate $\nabla\tilde{\mathbf{J}}$ is very costly for high dimensions N. Some savings can be obtained by using the vector forward mode of AD, where a bundle of directions is propagated. Nevertheless, the reverse mode of AD should be used for gradient calculation for moderate and high dimensions whenever possible as illustrated in Section 5.5.

However, our aim was to analyze the approximate solution of the sensitivity equation (5.4) generated by AD. For that purpose, Algorithm 3 states one possibility to integrate the sensitivity equation applying the same Runge-Kutta scheme used for integrating the state equation:

Algorithm 3: Integration of state and sensitivity equation

```
$y_1 = y^0, \quad s_1 = \mathbf{0}_n$
For $i = 1, N-1, 1$
    For $j = 1, s, 1$
        $\tilde{y}_j = y_i + h \sum_{l=1}^{j-1} a_{jl} k_l, \qquad \tilde{u}_j = p(v_i, c_j h)$
        $\tilde{s}_j = s_i + h \sum_{l=1}^{j-1} a_{jl} \tilde{k}_l \qquad \tilde{d}_j = q(w_i, c_j h)$
        $k_j = f(\tilde{y}_j, \tilde{u}_j), \qquad \tilde{k}_j = f_y(\tilde{y}_j, \tilde{u}_j)\tilde{s}_j + f_u(\tilde{y}_j, \tilde{u}_j)\tilde{d}_j$
    $y_{i+1} = y_i + h \sum_{j=1}^{s} b_j k_j, \qquad s_{i+1} = s_i + h \sum_{j=1}^{s} b_j \tilde{k}_j$
$\tilde{\mathbf{J}} = \varphi(y_N)$
```

Here, $q(w_i, c_j h)$ with $w_i = (d_i, d_{i+1})$ interpolates the chosen direction $\mathbf{d} \in \mathbb{R}^N$ at the intermediate times $t_i + c_j h$ with the same interpolation order κ as p. Let us assume that the Runge-Kutta scheme applied in Algorithm 1 for the state integration is consistent of order ν and an appropriate interpolation order $\kappa \geq \nu$ is used. Then the resulting approximations y_i, $i = 1, \ldots, N$, are consistent of order ν. In addition, it follows that also the approximate solution s of the sensitivity equation obtained by Algorithm 3 is consistent of order ν. Hence, one obtains that the convergence order of the integration method proposed in Algorithm 3 for the sensitivity equation equals ν since the right-hand side f is assumed to be sufficiently smooth.

To obtain a related convergence result for Algorithm 2, we examine now the relation between Algorithm 2 and Algorithm 3. As can be seen, there are similar recursions for $\dot{\tilde{y}}_j$ and $\tilde{s}_j$ as well as for $\dot{k}_j$ and $\tilde{k}_j$. The only difference lies in the direction $\dot{\tilde{u}}_j$ and $\tilde{d}_j$. Therefore, we have to examine the computation of $\dot{\tilde{u}}_j$ in more detail. For constant and linear interpolation it is very easy to check that $\dot{\tilde{u}}_j = p_u(v_i, c_j h)\dot{\mathbf{u}}$ also yields a constant and linear interpolation of $\dot{\mathbf{u}}$. For cubic interpolation, the system matrix D of (5.10) is constant with respect to $\mathbf{u}$. Let r_u denote the differentiation of r with respect to all components of $\mathbf{u}$. Then, the product $\tilde{r} = r_u\dot{\mathbf{u}}$ agrees with r if one substitutes $\dot{u}_i$ with u_i since r is only linear in $\mathbf{u}$. Furthermore, one has for the derivative γ_u of γ with respect to $\mathbf{u}$ the identity

$$\gamma_u\dot{\mathbf{u}} = D^{-1}r_u\dot{\mathbf{u}} = D^{-1}\tilde{r} = \tilde{\gamma}. \tag{5.12}$$

Analyzing also the remaining coefficient vectors of the cubic interpolation polynomial, e.g. $\tilde{\alpha} \equiv \alpha_u\dot{\mathbf{u}} = \dot{\mathbf{u}}$, one finds easily that

$$\dot{\tilde{u}}_j = p_u(v_i, c_j h)\dot{\mathbf{u}} = \tilde{\alpha}_i + \tilde{\beta}_i t + \tilde{\gamma}_i t^2/2 + \tilde{\delta}_i t^3/6 \tag{5.13}$$

is a cubic interpolation scheme for the given direction $\dot{\mathbf{u}}$ and the slopes $\dot{\mathbf{u}}'_1 = \dot{\mathbf{u}}'_N = 0$. It follows that Algorithm 2 and Algorithm 3 yield the same approximate solution $\dot{y}_i$ and s_i, respectively, when $\dot{\mathbf{u}} = \mathbf{d}$ and $d'_1 = d'_N = 0$ holds. Hence, we have proved the following convergence result:

Theorem 5.4.1 (Convergence of Forward Mode Discretization)**.**
Assume that f and u are ν times continuously differentiable with $\nu \le 4$. Suppose that the integration method applied in Algorithm 1 is convergent of order ν. Furthermore, let the order of the interpolation polynomial taken from Table 5.1 be not less than ν. Then one has for a given direction d with $d'_1 = d'_N = 0$ that the approximate solution $\dot{y}_i$, $i = 1, \dots, N$, generated by the forward mode of AD (Algorithm 2) converges with order ν to the continuous solution of the sensitivity equation (5.4) if $\dot{\mathbf{u}} = \mathbf{d}$.

This theorem ensures that the discretization scheme generated by the forward mode of AD applied to the state equation inherits the convergence rate of the discretization scheme used for the state equation without any additional assumptions. Note that nonzero values for the derivative of $\mathbf{d}$ at the times t_0 and t_N can be easily taken into account. Using the gradient representation (5.5), we are now able to compute an approximation of the continuous gradient of order $\nu \le 4$ based on the approximate solution $\dot{y}_i$, $i = 1, \dots, N$, of the sensitivity equation by choosing the Nm unit vectors as directions $\dot{\mathbf{u}}$ in Algorithm 2. Obviously, this approach is usually very costly for reasonable discretizations of the original problem. Therefore, we will now turn to the reverse mode of AD that provides a cheap computation of gradient information.

5.5 Reverse Mode of AD in Recursive Discretization

For a given vector function F evaluated at a certain point x with $y = F(x)$ and a weight vector $\bar{y}$, the scalar reverse mode of AD yields the product $F'(x)^T\bar{y}$, i.e. the transposed Jacobian $F'(x)^T$ multiplied by the vector $\bar{y}$ from the right.

Once more the evaluation cost for such a product can be bounded above by the product of a small constant and the cost to evaluate the function itself. Depending on the considered complexity estimates, the value of the constant varies again between three and five [70]. Hence, the exact gradient of a scalar valued function given by a computer program can be computed via AD with an effort that is independent of the number of optimization variables. It seems that this impressive result is yet not well enough appreciated compared to the inexactness of finite differences and the effort to evaluate them.

In the context of an optimal control problem, the weight vector $\bar{\mathbf{J}} \in \mathbb{R}$ is set to $\bar{\mathbf{J}} = 1$ and we are interested in the gradient $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ with $\mathbf{u} \in \mathbb{R}^N$. Then, the scalar reverse mode of AD applied to the state integration given by Algorithm 1 corresponds to the following integration procedure [70]:

Algorithm 4: Applying AD, reverse differentiation of state equation

```
$y_1 = y^0$
For $i = 1, N-1, 1$
    For $j = 1, s, 1$
        $\tilde{y}_j = y_i + h \sum_{l=1}^{j-1} a_{jl}\, k_l$
        $\tilde{u}_j = p(v_i, c_j h)$
        $k_j = f(\tilde{y}_j, \tilde{u}_j)$
    $y_{i+1} = y_i + h \sum_{j=1}^{s} b_j k_j$
$\tilde{\mathbf{J}} = \varphi(y_N)$
$\bar{y}_N = \nabla\varphi(y_N)^T$, $\bar{u}_i = \mathbf{0}_m \quad 1 \le i \le N$
For $i = N-1, 1, -1$
    $\bar{y}_i = \bar{y}_{i+1} \qquad \bar{k}_j = h\, b_j \bar{y}_{i+1}, \quad j = 1, \ldots, s$
    For $j = s, 1, -1$
        $\bar{\tilde{y}}_j = f_y(\tilde{y}_j, \tilde{u}_j)^T \bar{k}_j, \qquad \bar{\tilde{u}}_j = f_u(\tilde{y}_j, \tilde{u}_j)^T \bar{k}_j$
        $\bar{u} \mathrel{+}= p_u(v_i, c_j h)^T \bar{\tilde{u}}_j,$
        $\bar{y}_i \mathrel{+}= \bar{\tilde{y}}_j, \qquad \bar{k}_l \mathrel{+}= h\, a_{jl}\, \bar{\tilde{y}}_j, \quad l = 1, \ldots, j-1$
$\bar{y}_1 = 0.$
```

Once more, the common notation in AD literature is used, where $a \mathrel{+}= b$ denotes $a = a + b$. Applying this basic form of the scalar reverse mode, first the forward integration is performed to calculate the values $\tilde{y}_j$ and $\tilde{u}_j$. Then, the second for-loop computes the desired gradient $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ assuming that the intermediate values $\tilde{y}_j$ and $\tilde{u}_j$ of state i are still available. These intermediate values can be stored sequentially onto a data structure and retrieved in reverse order when they are needed. This approach is easy to implement but results in a memory requirement that is proportional to the number N of time steps. To reduce this potential enormous memory requirement, optimal checkpointing procedures are available that can easily be incorporated into Algorithm 4 [76].

After these general remarks about the reverse mode of AD we return to the analysis of the approximate solutions of the state equation (5.2) and the adjoint equation (5.6) generated by the reverse mode of AD. First, one has to note that the integration method for the state is not changed. Therefore, the convergence order of the integration method applied for the state integration does not change. This is also true for other approaches to compute the gradient information $\nabla\tilde{\mathbf{J}}(\mathbf{u})$ using the recursive discretization. Therefore, we face the first important difference between the full discretization and the recursive discretization. Using the full discretization, it may happen that the discrete solution of the state does not converge to the continuous solution if a second order Runge-Kutta method is used, see [84, Section 6]. In the remaining part of this section, we will prove that for a fixed control function the approximate solution of the continuous adjoint equation converges to the exact solution of the continuous adjoint equation with order ν if the state approximation converges already with order ν to the exact solution of the state equation. Here, we face the second important difference between the full discretization and the recursive discretization: The usual order conditions for Runge-Kutta schemes are shown in Table 5.2, where the conditions for order ν comprise those listed in Table 5.2 for that specific order as well as those for all lower orders. Using the full discretization, it was proved by Hager that additional conditions have to be fulfilled for $\nu \geq 3$ to ensure the order ν also in the optimal control context [84].

ν	Conditions with $c_i = \sum_{j=1}^{s} a_{ij}$, $d_i = \sum_{j=1}^{s} b_j a_{ji}$
1	$\sum b_i = 1$
2	$\sum b_i c_i = \frac{1}{2}$
3	$\sum c_i d_i = \frac{1}{6}$, $\sum b_i c_i^2 = \frac{1}{3}$
4	$\sum b_i c_i^3 = \frac{1}{4}$, $\sum b_i c_i a_{ij} c_j = \frac{1}{8}$, $\sum b_i a_{ij} c_j^2 = \frac{1}{12}$, $\sum b_i a_{ij} a_{jk} c_k = \frac{1}{24}$

Table 5.2: Order ν of a Runge-Kutta discretization for recursive discretization

With respect to the approximate solution of the adjoint equation given by the reverse mode of AD, one obtains from Algorithm 4 the recursion

$$\begin{aligned} \bar{y}_i &= \bar{y}_{i+1} + \sum_{j=1}^{s} \bar{\tilde{y}}_j, \qquad \bar{y}_N = \nabla\varphi(y_N)^T, \qquad (5.14)\\ \bar{\tilde{y}}_j &= f_y(\tilde{y}_j, \tilde{u}_j)^T \bar{k}_j = f_y(\tilde{y}_j, \tilde{u}_j)^T \left(h\, b_j\, \bar{y}_{i+1} + h \sum_{l=j+1}^{s} a_{lj}\, \bar{\tilde{y}}_l \right). \end{aligned}$$

Now, we transform this recursion into a formulation that is better suited for the analysis. This modification is similar to the one used by Hager [84]. We introduce a new variable $\hat{y}_j$ by setting

$$\hat{y}_j = \bar{y}_{i+1} + \sum_{l=j+1}^{s} \frac{a_{lj}}{b_j} \bar{\tilde{y}}_l, \qquad j = 1, \ldots, s, \qquad (5.15)$$

where we assume $b_j > 0$, $j = 1, \ldots, s$. It follows that

$$\bar{\bar{y}}_j \;=\; h\, b_j\, f_y(\tilde{y}_j, \tilde{u}_j)^T \hat{y}_j. \tag{5.16}$$

Exploiting the identities (5.15) and (5.16) yields

$$\hat{y}_j - \bar{y}_{i+1} \;=\; \sum_{l=j+1}^{s} \frac{a_{lj}}{b_j} \bar{\bar{y}}_l \;=\; \sum_{l=j+1}^{s} \frac{a_{lj}}{b_j} \left(h\, b_l\, f_y(\tilde{y}_l, \tilde{u}_l)^T \hat{y}_l\right).$$

Using (5.14) and (5.16), we obtain the recursion

$$\begin{aligned} \bar{y}_i &= \bar{y}_{i+1} + h \sum_{j=1}^{s} b_j\, f_y(\tilde{y}_j, \tilde{u}_j)^T \hat{y}_j, \qquad \bar{y}_N \;=\; \nabla\varphi(y_N)^T, \\ \hat{y}_j &= \bar{y}_{i+1} + \sum_{l=j+1}^{s} \frac{a_{lj}}{b_j} \left(h\, b_l\, f_y(\tilde{y}_l, \tilde{u}_l)^T \hat{y}_l\right) \end{aligned}$$

which marches still backwards. Therefore, we now change the direction of the integration. This yields

$$\begin{aligned} \bar{y}_{i+1} &= \bar{y}_i - h \sum_{j=1}^{s} b_j\, f_y(\tilde{y}_j, \tilde{u}_j)^T \hat{y}_j, \qquad \bar{y}_N \;=\; \nabla\varphi(y_N)^T \\ \hat{y}_j &= \bar{y}_i - h \sum_{l=1}^{s} b_l\, f_y(\tilde{y}_l, \tilde{u}_l)^T \hat{y}_l + \sum_{l=j+1}^{s} \frac{a_{lj}}{b_j} \left(h\, b_l\, f_y(\tilde{y}_l, \tilde{u}_l)^T \hat{y}_l\right) \\ &= \bar{y}_i - h \sum_{l=1}^{s} \bar{a}_{jl} f_y(\tilde{y}_l, \tilde{u}_l)^T \hat{y}_l \quad \text{with} \quad \bar{a}_{jl} \;=\; \frac{b_j b_l - b_l a_{lj}}{b_j} \end{aligned}$$

assuming $a_{lj} = 0$ if $l \leq j$. Defining $g(y, \bar{y}, u) = -f_y(y, u)^T \bar{y}$, we obtain

$$\bar{y}_{i+1} \;=\; \bar{y}_i + h \sum_{j=1}^{s} b_j\, g(\tilde{y}_j, \hat{y}_j, \tilde{u}_j), \quad \bar{y}_N \;=\; \nabla\varphi(y_N)^T \tag{5.17}$$

$$\hat{y}_j \;=\; \bar{y}_i + h \sum_{l=1}^{s} \bar{a}_{jl} g(\tilde{y}_l, \hat{y}_l, \tilde{u}_l), \quad \tilde{y}_j \;=\; y_i + h \sum_{l=1}^{j-1} a_{jl} f(\tilde{y}_l, \tilde{u}_l), \tag{5.18}$$

which requires the knowledge of the intermediate values $y_1, \ldots, y_N$. The derived integration scheme (5.17)–(5.18) for the adjoint equation (5.6) is very similar to the one obtained by Hager for the full discretization. However, the important difference is given by the fact that for the recursive discretization the function f depends on the state and the control as well as the function g on the state, the control, and the costate whereas for the full discretization the functions f, i.e. the right-hand side of the state equation, and ϕ, i.e. the right-hand side of the costate equation, both depend on the state and the costate. In addition, for the recursive discretization we interpolate the control at the intermediate points $t_i + c_j h$ and do not consider the intermediate values as additional control variables. Hence, we can not simply refer to the consistency proof by Hager [84], although there will be a significant similarity of the proofs.

Since the summations for $\hat{y}_j$ and $\tilde{y}_j$ involve different coefficients $\bar{a}_{jl}$ and a_{jl}, it is also not possible to apply the usual consistency theory for Runge-Kutta schemes. Therefore, we use a similar approach to that of Butcher [21] but take the specific situation with the different coefficients $\bar{a}_{jl}$ and a_{jl} as well as the interpolation of the control into account. Despite the fact that the analysis works also for higher order, we restrict the proof to schemes of order less than 5 since is seems quite unusual to have an appropriate interpolation that maintains a higher order of the applied Runge-Kutta method.

Defining the function $\tilde{g} = (f, g, u')^T$ acting on $(y, \bar{y}, u)$, the solution λ of the adjoint equation (5.6) has the Taylor expansion

$$\begin{aligned}\lambda(t+h) &= \lambda(t) + gh + \frac{1}{2}g'\tilde{g}\,h^2 + \frac{1}{6}\Big[g''\tilde{g}^2 + g'\tilde{g}'\Big]h^3 + \\ &\quad \frac{1}{24}\Big[g'''\tilde{g}^3 + 3g''\tilde{g}\tilde{g}' + g'\tilde{g}''\Big]h^4 + \mathcal{O}(h^5).\end{aligned}$$

Here, the functions g and $\tilde{g}$ as well as their derivatives are evaluated at the point $(y(t), \lambda(t), u(t))$. The derivatives should be viewed in an operator context introduced for example in [21] or [136]. Hence, the derivative g' operates on a vector and yields a scalar. The second derivative g'' acts on a pair of vectors to give a vector. Finally, one has $\tilde{g}' = (f_y f + f_u u', g_y f + g_{\bar{y}} g + g_u u', u'')^T$ and so on.

For the Taylor expansion around the approximations $\bar{y}_i$, we define

$$\zeta_1(h) = \begin{pmatrix} y_i \\ \bar{y}_i \\ u_i \end{pmatrix} \quad \text{and} \quad \zeta_j(h) = \begin{pmatrix} \tilde{y}_j \\ \hat{y}_j \\ \tilde{u}_j \end{pmatrix} \qquad 1 < j \le s$$

as well as the functions

$$G(\zeta) = \sum_{j=1}^{s} b_j g(\zeta_j) \quad \text{and} \quad F_j(\zeta) = \begin{pmatrix} \sum_{l=1}^{j-1} a_{jl} f(\tilde{y}_l, \tilde{u}_l) \\ \sum_{l=1}^{s} \bar{a}_{jl} g(\tilde{y}_l, \hat{y}_l, \tilde{u}_l) \\ (p(v_i, c_j h) - u_i)/h \end{pmatrix}.$$

Then, one has

$$\zeta(h) = \zeta(0) + hF(\zeta(h)) \quad \text{and} \quad \bar{y}_{i+1} = \bar{y}_i + hG(\zeta(h)).$$

Expanding $G(\zeta(h))$ in a Taylor series around $h = 0$, we obtain

$$\begin{aligned}\bar{y}_{i+1} &= \bar{y}_i + hG + \frac{1}{2}\Big[2G'F\Big]h^2 + \frac{1}{6}\Big[3G''F^2 + 6G'F'F\Big]h^3 + \\ &\quad \frac{1}{24}\Big[4G'''F^3 + 24G''FF'F + 24G'(F')^2F + 12G'F''F^2\Big]h^4 + \mathcal{O}(h^5),\end{aligned}$$

where G, F, and their derivatives are evaluated at $\zeta(0)$. Now, we can prove the main result of this section:

Theorem 5.5.1 (Convergence of Reverse Mode Discretization I).
Assume that f and u are ν times continuously differentiable with $\nu \le 4$. Let the integration method applied in Algorithm 1 be of order ν with coefficients

$b_j > 0$, $1 \leq j \leq s$, i.e., it fulfills the conditions of Table 5.2 up to order ν. Furthermore, let the order of the interpolation polynomial taken from Table 5.1 be not less than ν. Then, the approximate solution $\bar{y}_i$, $i = 1, \ldots, N$, generated by the reverse mode of AD (Algorithm 4) converges with order ν to the continuous solution of the adjoint equation (5.6).

Proof: We have to check for a given ν that the Taylor expansions of λ and $\bar{y}$ agree through all terms of order h^κ, $\kappa = 1, \ldots, \nu$. For that purpose, the arguments of the functions f and g are skipped since they are always evaluated at the point $(y_i, \bar{y}_i, u_i)$.

For $\nu = 1$, the equation

$$g \;=\; \sum b_j g$$

has to be fulfilled, which yields with $\sum b_j = 1$ the condition for order 1 as stated in Table 5.2. Here, we employ the summation convention that if an index range does not appear on a summation sign then the summation is over each index, taking values from 1 to s.

For $\nu = 2$, we introduce abbreviations $c_j = \sum_{l=1}^{j-1} a_{jl}$, $\bar{c}_j = \sum_{l=1}^{s} \bar{a}_{jl}$, and $\hat{u}_j = (p(v_i, c_j h) - u_i)/h = (\tilde{u}_j - u_i)/h$. One has to check whether

$$g'\tilde{g} \;=\; g_y f + g_{\bar{y}} g + g_u u_i' \tag{5.19}$$

agrees with

$$2G'F \;=\; 2\sum_{j=1}^{s} b_j \big(c_j g_y f + \bar{c}_j g_{\bar{y}} g + g_u \hat{u}_j\big) \tag{5.20}$$

up to terms of order 1. One obtains immediately the conditions

$$\sum b_j c_j \;=\; \frac{1}{2} \;=\; \sum b_j \bar{c}_j.$$

The first equality holds because of the usual order conditions for Runge-Kutta schemes. Hence, it is fulfilled because the scheme for the forward integration (Algorithm 1) is of order 2. The second equality is obtained by

$$\sum b_j \bar{c}_j = \sum b_j \frac{b_j b_l - b_l a_{lj}}{b_j} = \sum b_j - \sum b_j c_j = \frac{1}{2}.$$

Additionally we have to take the interpolation of the control into account. For the interpolation polynomials of order $\kappa \geq 2$ that are given in Table 5.2, we have

$$\hat{u}_j = c_j u_i' + \mathcal{O}(h). \tag{5.21}$$

Hence, the Taylor expansions of λ and $\bar{y}$ agree through all terms of order κ, $\kappa = 1, 2$ if the conditions for order $\kappa \leq 2$ as stated in Table 5.2 are fulfilled.

For $\nu = 3$, one obtains that $g_{\bar{y}\bar{y}} = 0$ since g depends only linearly on $\bar{y}$. Hence, we have to consider

$$g''\tilde{g}^2 = g_{yy} f f + 2 g_{y\bar{y}} f g + (2 g_{yu} f + 2 g_{\bar{y}u} g + g_{uu} u_i') u_i' \tag{5.22}$$

and

$$3G''F^2 = 3\sum b_j\Big(c_j^2 g_{yy}ff + 2c_j\bar{c}_j g_{y\bar{y}}fg + \\ (2c_j g_{yu}f + 2\bar{c}_j g_{\bar{y}u}g + g_{uu}\hat{u}_j)\hat{u}_j\Big). \tag{5.23}$$

For the terms $g_{yy}ff$ and $g_{y\bar{y}}fg$ the identities

$$\frac{1}{3} = \sum b_j c_j^2 = \sum b_j c_j \bar{c}_j \tag{5.24}$$

must hold. The first one is again an usual order condition for Runge-Kutta schemes. Therefore it is valid since the scheme applied in Algorithm 1 is supposed to be of order 3. For the next equality, one has

$$\sum b_j c_j \bar{c}_j = \sum b_j c_j \frac{b_j b_l - b_l a_{lj}}{b_j} = \sum b_j c_j \Big(1 - \frac{1}{b_j}\sum_{l=1}^{s} b_l a_{lj}\Big) \\ = \frac{1}{2} - \sum b_l a_{lj} c_j = \frac{1}{6}.$$

With respect to the control u, condition (5.21) holds for the interpolation polynomial of order $\kappa \geq 3$. This fact together with the equality (5.24) yields that (5.22) and (5.23) agree except for terms of higher order. Second, we examine

$$g'\tilde{g}' = g_y(f_y f + f_u u_i') + g_{\bar{y}}(g_y f + g_{\bar{y}} g + g_u u_i') + g_u u_i'' \tag{5.25}$$

and

$$G'F'F = \sum b_j\big(a_{jl}\, g_y\, (c_l f_y f + f_u \hat{u}_l) + \bar{a}_{jl}\, g_{\bar{y}}\, (c_l g_y f + \bar{c}_l g_{\bar{y}} g + g_u \hat{u}_l)\big), \tag{5.26}$$

where no term $b_j g_u$ occurs in (5.26) since the derivative of the third entry of F_j, i.e., of $(p(v_i, 0) - u_i)/h$, with respect to u_i vanishes for the interpolation polynomial of order κ with $\kappa \geq 3$ at $t = 0$. It follows from the Taylor expansions of λ and $\bar{y}$ that the equations

$$\frac{1}{6} = \sum b_j a_{jl} c_l \;=\; \sum b_j \bar{a}_{jl} c_l \;=\; \sum b_j \bar{a}_{jl} \bar{c}_l \tag{5.27}$$

must be valid. These equalities cover three terms of (5.25). The first one is an usual condition for Runge-Kutta methods and holds because of the order of the forward integration. For the second and third equation, we obtain

$$\sum b_j \bar{a}_{jl} c_l = \sum b_j \frac{b_j b_l - b_l a_{lj}}{b_j} c_l = \sum b_j b_l c_l - \sum b_l a_{lj} c_l = \frac{1}{6},$$

$$\sum b_j \bar{a}_{jl} \bar{c}_l = \sum\; b_j \frac{b_j b_l - b_l a_{lj}}{b_j}(1 - \frac{1}{b_l}\sum_{k=1}^{s} b_k a_{kl}) = \frac{1}{6}.$$

Applying again the property (5.21) for the interpolation polynomials of order $\kappa \geq 3$ yields for the terms $g_y f_u u_i'$ and $g_{\bar{y}} g_u u_i'$ of (5.25) two equalities of (5.27) both of which are valid. The remaining term $g_u u_i''$ of (5.26) together with the term $g_u u_i'$ of (5.19) forms the sum

$$g_u \left(\frac{h^2}{2} u_i' + \frac{h^3}{6} u_i'' \right)$$

in the Taylor expansion of λ. Due to the interpolation order $\nu \geq 3$, we obtain

$$p(v_i, c_j h) = u_i + c_j h u_i' + \frac{1}{2} c_j^2 h^2 u_i'' + \frac{1}{6} c_j^3 h^3 u_i''' + \mathcal{O}(h^4) \tag{5.28}$$

(see e.g. [124, Property 8.3]). It follows that we have for the term $g_u \hat{u}_j$ of (5.20) not only $g_u \hat{u}_j = g_u c_j u_i' + \mathcal{O}(h)$, but for the corresponding term in the Taylor expansion of $\bar{y}$ that

$$\begin{aligned} h^2 \sum b_j g_u \hat{u}_j &= h^2 g_u \sum b_j (c_j u_i' + c_j^2 h u_i''/2 + \mathcal{O}(h^2)) \\ &= g_u \left(\frac{h^2}{2} u_i' + \frac{h^3}{6} u_i'' + \mathcal{O}(h^4) \right). \end{aligned} \tag{5.29}$$

Hence, for an interpolation order $\kappa \geq 3$ the equations (5.25) and (5.26) agree except for terms of higher order.

For $\nu = 4$, first one has

$$\begin{aligned} g''' \tilde{g}^3 = g_{yyy} f^3 + 3 g_{yy\bar{y}} f^2 g + (3 g_{yyu} f^2 + 6 g_{y\bar{y}u} f g) u_i' + \\ (3 g_{yuu} f + 3 g_{\bar{y}uu} g)(u_i')^2 + g_{uuu} g (u_i')^3 \end{aligned} \tag{5.30}$$

and

$$\begin{aligned} 4G''' F^3 = 4 \sum b_j \Big(& c_j^3 g_{yyy} f^3 + 3 c_j^2 \bar{c}_j g_{yy\bar{y}} f^2 g + \\ & (3 c_j^2 g_{yyu} f^2 + 6 c_j \bar{c}_j g_{y\bar{y}u} f g) u_i' + \\ & (3 c_j g_{yuu} f + 3 \bar{c}_j g_{\bar{y}uu} g)(u_i')^2 + g_{uuu} g (u_i')^3 \Big) \end{aligned} \tag{5.31}$$

when taking $g_{\bar{y}\bar{y}} = 0$ into account. Hence, the equalities

$$\frac{1}{4} = \sum b_j c_j^3 = \sum b_j c_j^2 \bar{c}_j \tag{5.32}$$

must hold for the first four terms of (5.30). Once more, the first one is an usual order condition for Runge-Kutta schemes and holds because of the order of the forward integration. For the next equality, one obtains

$$\sum b_j c_j^2 \bar{c}_j = \sum b_j c_j^2 \Big(1 - \frac{1}{b_j} \sum b_l a_{lj} \Big) = \frac{1}{2} - \sum b_l a_{lj} c_j^2 = \frac{1}{4}.$$

Employing condition (5.21) for the interpolation polynomial of order $\nu = 4$ together with the equality (5.32) yield that (5.30) and (5.31) agree except for terms of higher order. Second, we consider the term

$$\begin{aligned} 3 g'' \tilde{g} \tilde{g}' = 3 \Big(& (g_{yy} f + g_{y\bar{y}} g + g_{yu} u_i')(f_y f + f_u u_i') + \\ & (g_{y\bar{y}} f + g_{\bar{y}u} u_i')(g_y f + g_{\bar{y}} g + g_u u_i') + \\ & u_i'' (g_{yu} f + g_{\bar{y}u} g + g_{uu} u_i') \Big) \end{aligned} \tag{5.33}$$

as well as

$$24G''FF'F =$$

$$24\sum\sum\Big(b_ja_{jl}(c_jg_{yy}f+\bar{c}_jg_{y\bar{y}}g+g_{yu}\hat{u}_j)(c_lf_yf+f_u\hat{u}_l)\Big)+ \qquad (5.34)$$
$$24\sum\Big(b_j\bar{a}_{jl}(c_jg_{y\bar{y}}f+g_{\bar{y}u}\hat{u}_j)(c_lg_yf+\bar{c}_lg_{\bar{y}}g+g_u\hat{u}_l)\Big).$$

It follows that the equalities

$$\frac{1}{8}=\sum b_jc_ja_{jl}c_l=\sum b_j\bar{c}_ja_{jl}c_l=\sum b_jc_j\bar{a}_{jl}c_l=\sum b_jc_j\bar{a}_{jl}\bar{c}_l \qquad (5.35)$$

must be fulfilled. The first one is again an usual order condition for Runge-Kutta schemes that holds since the order of the forward integration is assumed to be 4, whereas the next three equalities can be proved by

$$\sum b_j\bar{c}_ja_{jl}c_l=\sum b_j\Big(1-\frac{1}{b_j}\sum_{k=1}^{s}b_ka_{kj}\Big)a_{jl}c_l=\sum b_ja_{jl}c_l-\sum b_ka_{kj}a_{jl}c_l=\frac{1}{8},$$

$$\sum b_jc_j\bar{a}_{jl}c_l=\sum b_jc_j\frac{b_jb_l-b_la_{lj}}{b_j}c_l=\sum b_jc_jb_lc_l-\sum b_lc_la_{lj}c_j=\frac{1}{8},$$

$$\sum b_jc_j\bar{a}_{jl}\bar{c}_l=\sum b_jc_j\frac{b_jb_l-b_la_{lj}}{b_j}(1-\frac{1}{b_l}\sum_{k=1}^{s}b_ka_{kl})$$
$$=\sum b_jc_jb_l-\sum b_la_{lj}c_j-\sum b_jc_jb_ka_{kl}+\sum b_ka_{kl}a_{lj}c_j=\frac{1}{8}.$$

Employing condition (5.21) for the interpolation polynomial of order $\nu=4$ together with the equality (5.35) yields that (5.33) and (5.34) agree up to terms of higher order except for the part involving u_i'', namely $u_i''(g_{yu}f+g_{\bar{y}u}g+g_{uu}u_i')$. We will examine this one later. Next, we analyze

$$\begin{aligned} g'\tilde{g}'' = {} & g_y\left[f_{yy}f^2+2f_{yu}fu_i'+f_y(f_yf+f_uu_i')+f_{uu}(u_i')^2\right]+\\ & g_{\bar{y}}\left[g_{yy}f^2+2g_{y\bar{y}}fg+2g_{yu}fu_i'+2g_{\bar{y}u}gu_i'+\right.\\ & \quad \left. g_{uu}(u_i')^2+g_y(f_yf+f_uu_i')+g_{\bar{y}}(g_yf+g_{\bar{y}}g+g_uu_i')\right]+\\ & u_i''(g_yf_u+g_{\bar{y}}g_u)+g_uu_i''' \end{aligned} \qquad (5.36)$$

as well as

$$\begin{aligned} 12G'F''F^2 = {} & 12\sum b_ja_{jl}\,g_y\left[c_l^2f_{yy}f^2+2c_lf_{yu}f\hat{u}_l+f_{uu}(\hat{u}_l)^2\right]+\\ & 12\sum b_j\bar{a}_{jl}\,g_{\bar{y}}\left[c_l^2g_{yy}f^2+2c_l\bar{c}_lg_{y\bar{y}}fg+2c_lg_{yu}f\hat{u}_l+\right.\\ & \qquad \left.+2\bar{c}_lg_{\bar{y}u}g\hat{u}_l+g_{uu}(\hat{u}_l)^2\right] \end{aligned} \qquad (5.37)$$

and

$$\begin{aligned} G'(F')^2F = {} & \sum b_ja_{jl}a_{lk}g_yf_y(c_kf_yf+f_u\hat{u}_k)+\\ & \sum b_j\bar{a}_{jl}a_{lk}g_{\bar{y}}g_y(c_kf_yf+f_u\hat{u}_k)+\\ & \sum b_j\bar{a}_{jl}\bar{a}_{lk}g_{\bar{y}}^2(c_kg_yf+\bar{c}_kg_{\bar{y}}g+g_u\hat{u}_k). \end{aligned} \qquad (5.38)$$

From (5.36) and (5.37), we can conclude that

$$\frac{1}{12} = \sum b_j a_{jl} c_l^2 = \sum b_j \bar{a}_{jl} c_l^2 = \sum b_j \bar{a}_{jl} c_l \bar{c}_l$$

must be valid, where the first one is an usual order condition for Runge-Kutta schemes. Hence it is fulfilled because of the order of the forward integration. The remaining two equalities hold since we have

$$\sum b_j \bar{a}_{jl} c_l^2 = \sum b_j \frac{b_j b_l - b_l a_{lj}}{b_j} c_l^2 = \sum b_j b_l c_l^2 - \sum b_l c_l^3 = \frac{1}{12},$$

$$\sum b_j \bar{a}_{jl} c_l \bar{c}_l = \sum (b_j - a_{lj}) c_l (b_l - \sum_{k=1}^{s} b_k a_{kl})$$

$$= \sum b_l c_l - \sum b_k a_{kl} c_l b_j - \sum b_l c_l a_{lj} + \sum b_k a_{kl} c_l a_{lj} = \frac{1}{12}.$$

From (5.36), (5.38), the Taylor expansions of λ and $\bar{y}$ as well as the property (5.21), it follows that

$$\frac{1}{24} = \sum b_j a_{jl} a_{lk} c_k = \sum b_j \bar{a}_{jl} a_{lk} c_k = \sum b_j \bar{a}_{jl} \bar{a}_{lk} c_k = \sum b_j \bar{a}_{jl} \bar{a}_{lk} \bar{c}_k$$

must hold. Once more, the first one is fulfilled because the scheme applied in Algorithm 1 is assumed to be of order 4. The next three identities are valid because

$$\sum b_j \bar{a}_{jl} a_{lk} c_k = \sum (b_j b_l - b_l a_{lj}) a_{lk} c_k = \sum b_j b_l a_{lk} c_k - \sum b_l a_{lj} a_{lk} c_k = \frac{1}{24},$$

$$\sum b_j \bar{a}_{jl} \bar{a}_{lk} c_k = \sum (b_j - a_{lj})(b_l b_k - b_k a_{kl}) c_k$$

$$= \sum b_k c_k - \sum b_k a_{kl} c_k - \sum b_l a_{lj} b_k c_k + \sum b_k c_k a_{kl} a_{lj} = \frac{1}{24},$$

$$\sum b_j \bar{a}_{jl} \bar{a}_{lk} \bar{c}_k = \sum (b_j - a_{lj})(b_l - a_{kl})(b_k - \sum_{i=1}^{s} b_i a_{ik})$$

$$= 1 - \sum b_i c_i - \sum b_l c_l + \sum d_k c_k - \sum b_l c_l + \sum b_l c_l b_i c_i$$

$$+ \sum d_l c_l - \sum b_i a_{ik} a_{kl} c_l = \frac{1}{24}.$$

The remaining terms are $u_i''(3g_{yu}f + 3g_{\bar{y}u}g + 3g_{uu}u_i' + g_y f_u + g_{\bar{y}} g_u)$ and $g_u u_i'''$. The terms involving u_i' in (5.22) and (5.25) as well as the terms involving u_i'' in (5.33) and (5.36) contribute via the sum

$$\begin{aligned} &\frac{u_i'}{6}(2g_{yu}f + 2g_{\bar{y}u}g + g_{uu}u_i' + g_y f_u + g_{\bar{y}} g_u)h^3 + \\ &\frac{u_i''}{24}(3g_{yu}f + 3g_{\bar{y}u}g + 3g_{uu}u_i' + g_y f_u + g_{\bar{y}} g_u)h^4 \end{aligned} \tag{5.39}$$

to the Taylor expansion of λ. On the other hand, we have from (5.23) and (5.26)

$$\frac{h^3}{6} \sum_{j=1}^{s} b_j \left((6c_j g_{yu} f + 6\bar{c}_j g_{\bar{y}u} g + 3g_{uu} \hat{u}_j)\hat{u}_j + \sum_{l=1}^{s} 6(a_{jl}\, g_y f_u + \bar{a}_{jl} g_{\bar{y}} g_u)\hat{u}_l \right) \tag{5.40}$$

as contribution to the Taylor expansion of $\bar{y}$. The equation (5.28) yields for the interpolation order $\nu \geq 4$

$$\hat{u}_j = c_j u_i' + \frac{c_j^2 h u_i''}{2} + \frac{c_j^3 h^2 u_i'''}{6} + \mathcal{O}(h^3). \tag{5.41}$$

Therefore, it is easy to check that (5.39) and (5.40) agree except for terms of higher order. Furthermore, in the Taylor expansion of λ, the terms $g_u u_i'$, $g_u u_i''$, and $g_u u_i'''$ occur in the sum

$$g_u \biggl(\frac{h^2}{2} u_i' + \frac{h^3}{6} u_i'' + \frac{h^4}{24} u_i''' \biggr).$$

Because of (5.41), we have for the term $g_u \hat{u}_j$ of (5.20) not only (5.29), but for the corresponding term in the Taylor expansion of $\bar{y}$ that

$$\begin{aligned} h^2 \sum b_j g_u \hat{u}_j &= h^2 g_u \sum b_j \biggl(c_j u_i' + \frac{c_j^2 h u_i''}{2} + \frac{c_j^3 h^2 u_i'''}{6} + \mathcal{O}(h^3) \biggr) \\ &= g_u \biggl(\frac{h^2}{2} u_i' + \frac{h^3}{6} u_i'' + \frac{h^4}{24} u_i''' + \mathcal{O}(h^5) \biggr). \end{aligned}$$

Therefore, also the remaining terms agree except for terms of higher order.

We proved so far that the approximate solution generated by the scalar reverse mode of AD has the consistency order ν. Since we assume also that the functions f and u are sufficiently smooth on the whole time interval, one has that the convergence order of the integration scheme obtained by applying the scalar reverse mode of AD equals ν. ∎

Hence, we have proved that for a fixed control the approximate solution of the adjoint equation generated by the reverse mode of AD converges at the same rate as the approximate solution of the state equation. That is, the discretization scheme generated automatically for the adjoint differential equation by applying the technique of Automatic Differentiation has the same convergence order as the discretization scheme that is used for the forward integration under fairly mild additional assumptions. This result differs from the convergence result for the full discretization where the convergence rate of the optimal solution of the discretized control problem to the optimal solution of the continuous problem is considered. Analyzing the proof, this difference is due to the fact that the terms involving second- or higher-order derivatives of g with respect to $\bar{y}$ vanish for the recursive discretization. For that reason, no additional conditions on the coefficients are needed. However, it may happen that due to constraints on the control the right-hand sides f and g are only Lipschitz continuous. This problem will not be discussed here but is subject to further investigations.

It is possible to eliminate the restriction $b_i > 0$ for Runge-Kutta schemes up to second order using the full discretization approach [44]. We are able to prove a related result for the recursive discretization. Instead of the recursion (5.17)–(5.18), we exploit another formulation of the discretization scheme. It can be shown by inspection for $s \leq 4$ and using a rather technical induction for $s > 4$ that (5.17)–(5.18) is equivalent to the recursion

$$\begin{aligned} \bar{y}_{i+1} &= \bar{y}_i + h \sum b_j\, g(\tilde{y}_j, \breve{y}_j, \tilde{u}_j), \quad \bar{y}_N \;=\; \nabla\varphi(y_N)^T && (5.42)\\ \breve{y}_j &= \breve{y}_i + h \sum_{l=1}^{s} \breve{a}_{jl} g(\tilde{y}_l, \hat{y}_l, \tilde{u}_l), \quad \breve{a}_{jl} = b_l - a_{jl} && (5.43)\\ \tilde{y}_j &= y_i + h \sum_{l=1}^{j-1} a_{jl} f(\tilde{y}_l, \tilde{u}_l), && (5.44)\end{aligned}$$

which does not require the restriction $b_j > 0$. Using this Runge-Kutta scheme, we obtain the following result:

Theorem 5.5.2 (Convergence of Reverse Mode Discretization II).
Assume that f and u are ν times continuously differentiable with $\nu \leq 4$. Let the integration method applied in Algorithm 1 be of order ν, i.e., it fulfills the conditions of Table 5.2 up to order ν. Furthermore, let the order of the interpolation polynomial taken from Table 5.1 be not less than ν. Then, the approximate solution $\bar{y}_i$, $i = 1, \ldots, N$, generated by the reverse mode of AD (Algorithm 4) converges at least with order $\kappa = \min\{2, \nu\}$ to the continuous solution of the adjoint equation (5.6).

Proof: Once more, we assume throughout, that the functions f and u are sufficiently smooth on the whole domain such that convergence follows immediately from consistency. The difference between the two formulations is caused by the different values of $\bar{c}_j = \sum \bar{a}_{jl}$ and $\breve{c}_j = \sum \breve{a}_{jl} = 1 - \sum a_{jl} = 1 - c_j$ that need not to coincide.

For $\nu = 1$ only the condition $\sum b_j = 1$ has to be fulfilled as shown in the proof of Theorem 5.1. This equality holds since the integration scheme applied in Algorithm 1 is supposed to be of order 1 and we obtain $\kappa = 1$.

For $\nu = 2$, we have to check whether

$$\sum b_j c_j \;=\; 1/2 \;=\; \sum b_j \breve{c}_j.$$

holds. The first equality holds because of the usual order conditions for Runge-Kutta schemes in the ODE context. The second equality is valid since we have

$$\sum b_j \breve{c}_j \;=\; \sum b_j(1 - c_j) \;=\; \sum b_j - \sum b_j c_j \;=\; 1/2.$$

For $\nu = 3$, the equation $\sum b_j c_j \breve{c}_j = 1/3$ must hold in analogy to (5.24). However, the recursion (5.42)–(5.44) yields

$$\sum b_j c_j \breve{c}_j \;=\; \sum b_j c_j (1 - c_j) \;=\; \sum b_j c_j - \sum b_j c_j^2 \;=\; 1/2 - 1/3 \;=\; 1/6.$$

Hence, using (5.42)-(5.44) it is not possible to prove a convergence order that exceeds 2. ■

Using the approximate solution $\bar{y}_i$ of the adjoint equation (5.6) of order ν, we are able to compute the desired gradient information with order ν using the adjoint representation (5.7) given in Section 5.2. Furthermore, one has to note the following important fact: Comparing (5.7) with the formulas for $\bar{\mathbf{u}}$ of Algorithm 4, we obtain that the standard scalar reverse mode yields an approximation of the continuous gradient of order ν if the influence of the control on the interpolation procedure at intermediate times $t_i + c_j h$ with $0 < c_j < 1$ is completely neglected. For that purpose, it is required to passivate the interpolation procedure during the application of AD to ignore the dependence of the intermediate values on the control variables. The passivation of variables is possible for some AD-tools, e.g. for ADOL-C [154]. However, this approach to compute gradient information has not been exploited so far. Instead, usually the vector $\bar{\mathbf{u}} = (\bar{u}_1, \ldots, \bar{u}_N)$ is used for the optimization since $\bar{u}$ is directly generated by the reverse mode of AD without passivation. Then the values $\bar{u}_j$ represent the exact discrete gradient information for the integration method given by Algorithm 1. To verify this fact, we compare the discrete gradient representation (5.11) with the computation of $\bar{\mathbf{u}}$. For the Runge-Kutta method of Algorithm 1, we have

$$\frac{\partial \boldsymbol{\psi}_{i+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_i(\mathbf{u})} = I + h \sum_{j=1}^{s} f_y(\tilde{y}_j, \tilde{u}_j) \frac{\partial \tilde{y}_j}{\partial y_i}.$$

Hence, we obtain by inspecting the statements for $\bar{\tilde{y}}_j$ and $\bar{k}_j$ of Algorithm 4

$$\bar{y}_i = \left(I + h \sum_{j=1}^{s} f_y(\tilde{y}_j, \tilde{u}_j) \frac{\partial \tilde{y}_j}{\partial y_i} \right)^T \bar{y}_{i+1} = \left(\frac{\partial \boldsymbol{\psi}_{i+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_i(\mathbf{u})} \right)^T \bar{y}_{i+1}.$$

Since $\bar{y}_N$ is initialized to $\nabla\varphi(y_N)^T$, one obtains the formula

$$\bar{y}_i = \left(\frac{\partial \boldsymbol{\psi}_{i+1}(\mathbf{u})}{\partial \boldsymbol{\psi}_i(\mathbf{u})} \right)^T \cdots \left(\frac{\partial \boldsymbol{\psi}_N(\mathbf{u})}{\partial \boldsymbol{\psi}_{N-1}(\mathbf{u})} \right)^T \nabla\varphi(y_N)^T.$$

Now, this derivative vector is used to compute the derivative of $\psi_i(\mathbf{u})$ with respect to $\mathbf{u}$. For that purpose, the derivative of $\boldsymbol{\psi}_i(\mathbf{u})$ with respect to $\mathbf{u}$ is composed for each intermediate time $c_j h$ because of the definition of $\bar{k}_j$ and $\bar{\tilde{u}}_j$. Subsequently $\bar{\tilde{u}}_j$ is multiplied by the derivative of $\tilde{u}_j$ with respect to $\mathbf{u}$ and the result is added to the appropriate components of $\bar{\mathbf{u}}$. Therefore, Algorithm 4 yields the exact discrete derivatives (5.11) of the discrete cost function $\mathbf{J}$ evaluated by Algorithm 1 with respect to the control vector $\mathbf{u}$. However, since this gradient computation does not coincide with a discrete version of the adjoint representation (5.7) it is not possible to deduce an order of convergence for the discrete gradient information $\bar{\mathbf{u}}$ generated directly by the reverse mode of AD.

5.6 Numerical Illustration

For the verification of the theoretical results presented in the last sections, consider the following simple test problem taken from [84]:

$$\min \frac{1}{2} \int_0^1 u(t)^2 + 2x(t)^2 dt \quad \text{s.t.} \quad \frac{dx}{dt}(t) = 0.5x(t) + u(t), \quad x(0) = 1.$$

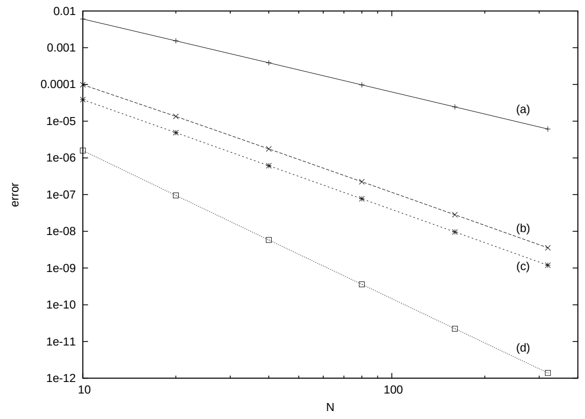

Figure 5.1: Discrete state error in L^∞

This problem can easily be transformed into the form (5.1)–(5.3) described in Section 5.2 using $y_1 = x$, $y_2 = \frac{dx}{dt}$ and

$$\frac{dy_1}{dt}(t) = 0.5y_1(t) + u(t), \qquad y_1(0) = 1, \tag{5.45}$$

$$\frac{dy_2}{dt}(t) = y_1(t)^2 + 0.5u(t)^2, \qquad y_2(0) = 0. \tag{5.46}$$

Then the objective is given by $J(y) = \varphi(y(1)) = y_2(1)$. This optimization problem has the optimal solution [84]

$$y_1^*(t) = \frac{2e^{3t} + e^3}{e^{3t/2}(2 + e^3)}, \qquad u^*(t) = \frac{2(e^{3t} - e^3)}{e^{3t/2}(2 + e^3)},$$

$$y_2^*(t) = \frac{2e^{3t} - e^{6-3t} - 2 + e^6}{(2 + e^3)^2}.$$

To approximate y^*, we use u^* as control and the Runge-Kutta schemes

$$(a)\ A = \begin{bmatrix} 0 & 0 \\ \frac{1}{2} & 0 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \qquad (b)\ A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, b = \begin{bmatrix} \frac{1}{6} \\ \frac{2}{3} \\ \frac{1}{6} \end{bmatrix},$$

$$(c)\ A = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ 0 & \frac{3}{4} & 0 \end{bmatrix}, b = \begin{bmatrix} \frac{2}{9} \\ \frac{1}{3} \\ \frac{4}{9} \end{bmatrix}, \quad (d)\ A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, b = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{6} \end{bmatrix},$$

where scheme (a) is of order 2, schemes (b) and (c) are of order 3, and scheme (d) is of order 4. These schemes were already used for numerical studies in [84]. For scheme (a), we apply linear interpolation, for the remaining schemes cubic interpolation. The resulting L^∞ errors for the discrete state at the grid points using the step size $h = 1/N$ with $N = 10, 20, 40, 80, 160, 320$ are given in Fig. 5.1.

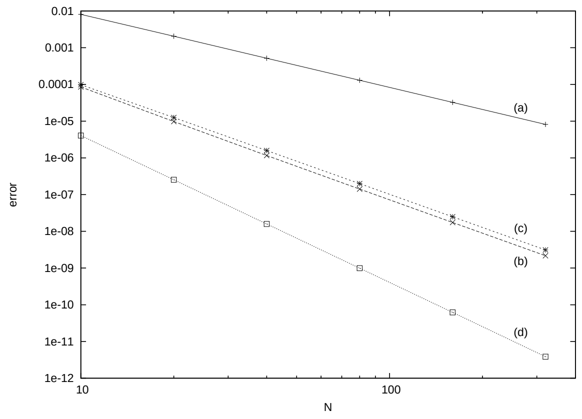

Figure 5.2: Discrete sensitivity error in L^∞

It follows that the computed errors correspond very well to the order of the applied Runge-Kutta scheme. Note, that in the context of full discretization methods, the scheme (b) may not even yield a convergent approximation for the state equation (c.f. [84]).

From the state equations (5.45) – (5.46) follows that the sensitivity equations are given by

$$\begin{aligned} \frac{ds_1}{dt}(t) &= 0.5s_1(t) + d(t), & s_1(0) &= 0, \\ \frac{ds_2}{dt}(t) &= 2x_1(t)s_1(t) + u(t)d(t), & s_2(0) &= 0. \end{aligned}$$

For the numerical experiments, we set $d(t) = 1$, which yields the analytical solution

$$s_1(t) = 2e^{-t/2} - 2, \qquad s_2(t) = \frac{4(e^{2t} - e^{3t/2} - e^{3-t} + e^{3-3t/2})}{2 + e^3}.$$

Applying the AD-tool ADOL-C, we implement Algorithm 2 to obtain the approximate solution $\dot{\mathbf{y}}$ of the sensitivity equation. The L^∞ errors for this discrete sensitivity at the grid points are illustrated by Fig. 5.2. Once more, the resulting error plot illustrates perfectly the order of the applied Runge-Kutta scheme.

Finally for the state equations (5.45) – (5.46), the corresponding adjoint equation is given by

$$\begin{aligned} \frac{\lambda_1}{dt}(t) &= -0.5\lambda_1(t) - 2y_1(t)\lambda_2(t), & \lambda(1) &= 0, \\ \frac{\lambda_2}{dt}(t) &= 0, & \lambda_2(1) &= 1, \end{aligned}$$

with the analytical solution

$$\lambda_1(t) = \frac{e^{3-t} - 2e^{2t}}{e^{t/2}(2 + e^3)}, \qquad \lambda_2(t) = 1.$$

Employing the reverse mode of ADOL-C, we implement Algorithm 4 to obtain the approximate solution $\bar{\mathbf{y}}$ of the continuous adjoint equation. The L^∞ errors

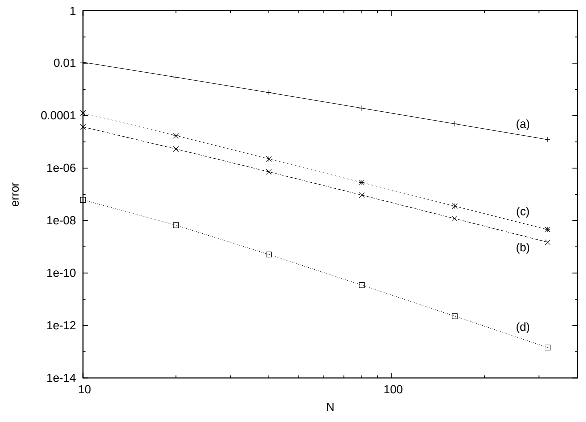

Figure 5.3: Discrete adjoint error in L^∞

for this discrete adjoint at the grid points are shown in Fig. 5.3. This error plot illustrates the convergence rate of the adjoint Runge-Kutta methods corresponding to the schemes (a), (b), and (c) very well. Only for scheme (d), the corresponding adjoint scheme shows the expected convergence behavior only for the finer discretizations. This may be due to the inexact information of the forward computation required by Algorithm 4. However, it is important to note that no additional conditions are required to obtain the convergence order 3 for scheme (c) which differs from the full discretization context.

5.7 Conclusion

This paper presents an analysis of the derivative information generated by the forward mode and reverse mode of AD for the recursive discretization in optimal control.

It has been shown that the forward mode of AD generates an approximate solution to the corresponding sensitivity equation of order ν if the Runge-Kutta method used for the integration of the state equation is already of order ν. Exploiting the approximate solution, we can compute an approximation of the continuous gradient of order ν. The standard initialization using unit vectors yields the exact discrete gradient information for the forward integration procedure as illustrated in this paper.

Furthermore, it is shown that the approximate solution of the adjoint equation obtained by the reverse mode of AD is of order ν if the integration method of the discrete version of the state equation is of order ν provided that one has $b_i > 0$. These results for the recursive discretization differ remarkably from the results obtains for the full discretization by Hager [84]. The less restrictive conditions for the coefficients of the Runge-Kutta scheme result from the fixed control for the recursive discretization whereas the control depends on the state and adjoint for the full discretization. Once more, the approximate solution presented here can be applied to compute an approximation of the continuous gradient of the same order ν. However, this gradient information is not generated directly by a black-box AD approach. The gradient information obtained by applying the standard reverse mode of AD equals again the exact discrete gradient as illustrated by this paper.

The question whether it is preferable to use the continuous gradient or the discrete gradient in an optimization procedure is still open and the answer depends certainly on the underlying problem to be solved. So far, all applications of AD known to the author provide the exact discrete gradient information. Based on the results of this paper, one may use now an approximation to the continuous gradient as alternative. Hence, it is possible to study the influences of the different derivative information on the optimization process. This subject will be investigated in the future. Furthermore, one has to examine the case when control constraints destroy the differentiability of the right-hand sides, which is ignored in this paper.

Acknowledgments

The author is indebted to Roland Griesse because the joint work with him raise several questions examined in this paper. Furthermore, the author likes to thank Andreas Griewank and Christian Grossmann for many discussions on the subject.

Chapter 6

An Optimal Memory-reduced Procedure for Calculating Adjoints of the Instationary Navier-Stokes Equations

Michael Hinze[1], Andrea Walther, and Julia Sternberg[1]

Optimal Control Applications and Methods
Volume 27(1), pp. 19 – 40 (2006).

Abstract:
This paper discusses approximation schemes for adjoints in control of the instationary Navier-Stokes system. It tackles the storage problem arising in the numerical calculation of the appearing adjoint equations by proposing a low-storage approach which utilizes optimal checkpointing. For this purpose, a new proof of optimality is given. This new approach gives so far unknown properties of the optimal checkpointing strategies and thus provides new insights. The optimal checkpointing allows a remarkable memory reduction by accepting a slight increase in run-time caused by repeated forward integrations as illustrated by means of the Navier-Stokes equations. In particular, a memory reduction of two orders of magnitude causes only a slow down factor of 2-3 in run-time.

6.1 Introduction

Adjoints are the most important tool for computing sensitivities in control problems for the time dependent Navier-Stokes equations. In particular, this is the case when the number of control variables is large and thus forbids approaches based on finite difference methods. To illustrate the role of adjoints for calculating sensitivities let us consider the following abstract optimal control problem: *Find an optimal control $u \in U$ that minimizes the functional*

$$\hat{J}(u) := J(y(u), u), \tag{6.1}$$

[1]Department of Mathematics, University of Hamburg, Germany

where $y \in Y$ and u are related through the equality constraints

$$G(y,u) = 0 \tag{6.2}$$

in Z^*. Here and throughout, the superscript * either denotes duals of functions spaces or adjoint operators, where it follows from the context which operation is performed. The superscript $^{-*}$ stands for the corresponding inverse operator. Furthermore, $(U, (\cdot,\cdot)_U)$, $(Y, (\cdot,\cdot)_Y)$ and $(Z, (\cdot,\cdot)_Z)$ denote Hilbert spaces. The functions $J : Y \times U \to \mathbb{R}$ and $G : Y \times U \to Z^*$ are sufficiently smooth mappings. For simplicity, from here onwards we assume that equation (6.2) admits a unique solution $y(u)$ for every $u \in U$. For the same reason, we assume that $J(y,u) = J_1(y) + J_2(u)$, $G(y,u) = G_1(y) - Bu$ and that $B : U \to Z^*$ is a bounded linear mapping. Let the derivative $G_y(y,u)$ have a bounded inverse for every $(y,u) \in Y \times U$. In Section 6.2 we will show that control problems for the instationary Navier-Stokes equations fit into this setting.

The first and second derivatives of the functional $\hat{J}$ are the main ingredients of modern optimization algorithms for the numerical solution of (6.1) – (6.2). For the action of $\hat{J}'$ at point u in direction δu, we obtain

$$(\hat{J}'(u), \delta u)_U = \langle J_y(y,u), y'(u)\delta u\rangle_{Y^*,Y} + (J_u(y,u), \delta u)_U$$

utilizing the chain rule. The implicit function theorem applied to equation (6.2) yields

$$y'(u) = -G_y(y,u)^{-1} G_u(y,u),$$

which implies

$$\langle J_y(y,u), y'(u)\delta u\rangle_{Y^*,Y} = (-G_u(y,u)^* G_y(y,u)^{-*} J_y(y,u), \delta u)_U.$$

Introducing the *adjoint variable* $\lambda := -G_y(y,u)^{-*} J_y(y,u) \in Z$ we finally can express the gradient of $\hat{J}$ as

$$\hat{J}'(u) = J_u(y(u),u) + G_u(y(u),u)^* \lambda. \tag{6.3}$$

In other words, all directional derivatives of the functional $\hat{J}$ can be calculated with the help of the adjoint variable λ, which in turn is obtained by one solve of the *adjoint equation*

$$G_y(y,u)^* \lambda = -J_y(y,u). \tag{6.4}$$

By similar calculations and transformations we obtain the reduced Hessian, i.e. the second derivative, of the functional $\hat{J}$:

$$\begin{aligned}\hat{J}''(u) = {} & G_u(y(u),u)^* G_y(y(u),u)^{-*} \{J_{yy}(y(u),u) + \\ & \langle G_{yy}(y(u),u)(\cdot,\cdot), \lambda\rangle_{Z^*,Z}\} \, G_y(y(u),u)^{-1} G_u(y(u),u) + J_{uu}(y(u),u).\end{aligned} \tag{6.5}$$

From its structure we conclude that the application of $\hat{J}''(u)$ to an element $\delta u \in U$ amounts to the following algorithm

Algorithm I: Reduced Hessian times control

1. Solve $G_y(y(u),u)v = G_u(y(u),u)\delta u$ for $v \in Y$,
2. form right-hand side
   $g(y,u,\lambda,v) := J_{yy}(y(u),u)v + \langle G_{yy}(y(u),u)(v,\cdot),\lambda\rangle_{Z^*,Z}$,
3. solve $G_y(y(u),u)^*\mu = g(y,u,\lambda,v)$ for $\mu \in Z$,
4. evaluate $\tilde{\mu} := G_u(y(u),u)^*\mu$, and finally,
5. set $\hat{J}''(u)\delta u = \tilde{\mu} + J_{uu}(y(u),u)\delta u$.

One observes, that the adjoint operator $G_y(y(u),u)^*$ is needed twice: One time to evaluate λ defined by equation (6.4) for a given $u \in U$ and the second time to provide μ in step 3 for applying $\hat{J}''(u)$. Here we note that in the context of inexact Newton methods the evaluation of Hessian times increment plays the central role.

We think of (6.2) as an abstract realization of the instationary Navier-Stokes equations on the time horizon $[0,T]$ with y denoting the state variables and u serving as control variables. For a given control u, the computation of λ and μ requires knowledge of the state $y(u)$ on the whole time horizon. Therefore, the storage of $y(u)$ forms a serious bottleneck for large time horizons where y represents a 2- or 3-dimensional flow velocity field.

As partial remedy to this storage problem, this paper presents an optimal checkpointing strategy that is applied to compute adjoints for the instationary Navier-Stokes equations. Using the proposed checkpointing technique, a storage reduction of two orders of magnitude is observed, whereas the resulting slow down factor of the adjoint calculation caused by repeated forward integrations of the Navier-Stokes equations lies only between 2 and 3. The main algorithmical tool utilized was revolve which Griewank and the second author developed in [76]. This binomial checkpointing strategy features the adjoint computation within a minimal run time for a given number of checkpoints. In the present paper, a new proof of optimality is given in Theorem 6.3.5. It uses the new concept of frequency numbers first introduced in [135]. The new technique simplifies the proof significantly and yields in addition so far unknown properties of the optimal checkpointing strategies as for example Theorem 6.3.4.

As far as the authors know there are only few contributions to control of the instationary Navier-Stokes equations that tackle the storage problem. The technique of equidistant checkpointing and repeated forward integration, also known as windowing, is discussed by Berggren and coauthors in [9, 10]. A detailed comparison of the windowing technique, also called multi-level checkpointing, with the checkpointing strategy proposed in the present paper is contained in [153]. The analysis yields that for a given number of checkpoints and a given upper bound on the number of repeated forward integrations, the checkpointing approach used in this paper allows the adjoint computation for a larger time horizon. Furthermore, it is shown that the time needed for the adjoint computation coincides for both approaches if the windowing is not used recursively. Moreover, it is also proved in [153] that for a given number of checkpoints the time required for the adjoint computation with a recursive windowing exceeds the run time needed for the adjoint computation with the checkpointing method proposed in this paper. Therefore, windowing can be seen as not optimal.

The application of the optimal checkpointing strategy for the Burgers equation is discussed in [152]. This approach has been extended to an adaptive memory and run-time reduced checkpointing strategy by the third author in [135]. Its application for calculating adjoints of the instationary Navier-Stokes system is examined in [95], where the case of a-priory unknown number of time steps with different computational complexity is considered. For this purpose, the extended package a-revolve for adaptive checkpointing is utilized.

For a general analytical framework of first and second order derivatives for control of the instationary Navier-Stokes equations we refer to the work [93] by Kunisch and the first author. For further contributions to (distributed) control of the instationary Navier-Stokes system we refer to Gunzburger and Manservisi [82].

This paper is organized as follows. In Section 6.2 the instationary Navier-Stokes equations are adapted to the setting introduced at the beginning of Section 6.1. Appropriate discretizations of the direct and the adjoint PDEs are given. Section 6.3 introduces reversal schedules including a new proof of optimality for the proposed checkpointing strategies. For this purpose, we introduce for the first time the concept of frequency numbers. They are used to derive new properties of the optimal reversal schedules, which presumably can be exploited in more detail in further studies. Section 6.4 illustrates the capabilities of the checkpointing algorithm proposed here with respect to memory reduction. The resulting storage requirements and run-times are given and discussed. Finally, Section 6.5 presents a summary.

6.2 Adjoints for the Navier Stokes Equations

As model application we now illustrate how derivatives of the reduced functional $\hat{J}$ in control of the instationary Navier-Stokes equations can be realized utilizing the formalism developed in the previous section. We begin with stating an appropriate functional analytic setting (for details see [92, 93]).

6.2.1 Analytical setting

To define the spaces and operators required for the investigation of the control problem (6.1) – (6.2), let $\Omega \subset \mathbb{R}^2$ denote a bounded domain and $[0,T]$ a time horizon. Further, set $Q := \Omega \times (0,T)$. We introduce the solenoidal spaces

$$H = \overline{\{v \in C_0^\infty(\Omega)^2 \colon \text{ div } v = 0\}}^{|\cdot|_{L^2}}, V = \overline{\{v \in C_0^\infty(\Omega)^2 \colon \text{ div } v = 0\}}^{|\cdot|_{H^1}},$$

with the superscripts denoting closures in the respective norms. For the subsequent considerations it is convenient to introduce the spaces

$$W = \{v \in L^2(V) \colon v_t \in L^2(V^*)\} \quad \text{and} \quad Z := L^2(V) \times H,$$

where W is endowed with the norm

$$|v|_W = |v|_{L^2(V)} + |v_t|_{L^2(V^*)}$$

and V^* denotes the dual space of V. Here $L^2(V)$ is an abbreviation for $L^2(0,T;V)$ and similarly $L^2(V^*) = L^2(0,T;V^*)$. We recall that up to a set of measure zero in $(0,T)$ elements $v \in W$ can be identified with elements in $C([0,T];H)$. By U we denote the Hilbert space of controls which is identified with its dual U^* and we set $\langle\cdot,\cdot\rangle := \langle\cdot,\cdot\rangle_{L^2(V^*),L^2(V)}$. Finally we assume that the cost functional $J\colon (y,u) \in W \times U \to J(y,u) = J_1(y) + J_2(u)$ is bounded from below, weakly lower semi-continuous, twice Fréchet differentiable with locally Lipschitzean second derivative, and radially unbounded in u, i.e. $J(y,u) \to \infty$ as $|u|_U \to \infty$, for every $y \in W$.

Example 6.2.1. These assumptions are satisfied for cost functionals including tracking type functionals (also with tracking of states at the final time T)

$$J(y,u) = \frac{1}{2}\int\limits_Q |y-z|^2 dx\,dt + \frac{\gamma}{2}\int\limits_\Omega |y(T)-z(T)|^2 dx + \frac{\alpha}{2}\,|u|_U^2, \tag{6.6}$$

and functionals involving the vorticity of the fluid

$$J(y,u) = \frac{1}{2}\int\limits_Q |\nabla_x \times y(t,\cdot)|^2\,dx\,dt + \frac{\alpha}{2}\,|u|_U^2, \tag{6.7}$$

where $\alpha,\gamma > 0$ and $z \in W$ are given. Of course, these functionals are even infinitely Fréchet differentiable on $W \times U$. ■

We define the nonlinear mapping

$$G\colon W \times U \to Z^*$$

by

$$G(y,u) = (\tfrac{\partial y}{\partial t} + (y\cdot\nabla)y - \nu\Delta y - Bu, y(0) - y_0),$$

where $B \in \mathcal{L}(U, L^2(V^*))$ denotes the control extension operator, $y_0 \in H$ and $\nu = 1/\mathrm{Re}$ with Re denoting the Reynolds number. In variational form the instationary Navier-Stokes equations can be expressed as: Given $u \in U$ find $y \in W$ such that $y(0) = y_0$ in H and

$$\langle y_t, v\rangle + \langle (y\cdot\nabla)y, v\rangle + \nu(\nabla y, \nabla v)_{L^2(L^2)} = \langle Bu, v\rangle \;\; \forall\, v \in L^2(V), \tag{6.8}$$

which in turn equivalently can be expressed by

$$G(y,u) = 0.$$

Next we note that for every $u \in U$ equation (6.8) admits a unique solution $y = y(u)$, see Temam [138]. Utilizing the control-to-state mapping $u \mapsto y(u)$ and defining $Y := W$, the optimal control problem for the Navier-Stokes equations may be formulated as

$$\min\ \hat{J}(u) = J(y(u), u) \ \text{ subject to } \ u \in U, \tag{6.9}$$

which is exactly of the form (6.1) – (6.2). It is proven by Abergel et al. [1] that (6.9) admits a solution $(y^*(u), u^*) \in W \times U =: X$. Furthermore, the operator $G = (G^1, G^2)\colon X \to Z^*$ is infinitely Fréchet differentiable with Lipschitz continuous first derivative, constant second derivative and vanishing third and higher derivatives, see [92, 93]. As a consequence the differentiability properties of the functional J are carried forward to the functional $\hat{J}$ and it is now clear how the expressions for $\hat{J}'$ and $\hat{J}''$ in (6.3) and (6.5), respectively, are to be understood once the actions of the operator $G_y(y,u)^{-1}$ and $G_y(y,u)^{-*}$ are known. To describe these actions for given $J_y(y,u)$, let the function $\lambda \in Z$ be defined by

$$\lambda \ = \ (\lambda^1, \lambda^2) \ = \ -G_y(y,u)^{-*} J_y(y,u),$$

and for $(f, v_0) \in Z^*$ set

$$v := G_y(y,u)^{-1}(f, v_0). \tag{6.10}$$

From now onwards, we assume that all functions are sufficiently smooth. Precise regularity statements of the variables y, v, and λ can be found in [92, 93].

For numerical considerations it is convenient to formulate the equations (6.8) in the primitive setting given by

$$\left.\begin{array}{rcl} y_t - \nu\Delta y + (y\cdot\nabla)y + \nabla p &=& Bu \text{ in } Q, \\ -\text{div } y &=& 0 \text{ in } Q, \\ y(x,t) &=& 0 \text{ on } \partial\Omega \times (0,T), \\ y(x,0) &=& y_0(x) \text{ in } \Omega, \end{array}\right\} \tag{6.11}$$

where p denotes the pressure. Together with the adjoint pressure ξ the adjoint variables λ of (6.4) satisfies the system

$$\left.\begin{array}{rcl} -\lambda^1_t - \nu\Delta\lambda^1 - (y\cdot\nabla)\lambda^1 + (\nabla y)^t\lambda^1 + \nabla\xi &=& -J^{(t)}_{1_y}(y) \text{ in } Q, \\ -\text{div } \lambda^1 &=& 0 \text{ in } Q, \\ \lambda^1(x,t) &=& 0 \text{ on } \partial\Omega \times (0,T), \\ \lambda^1(x,T) &=& -J^{(T)}_{1_y}(y) \text{ in } \Omega, \end{array}\right\} \tag{6.12}$$

and $\lambda^2 = \lambda^1(0)$. The superscripts (t), (T) refer to a possible dependence of the representative of the functional J_{1_y} on the time instances (t) and (T), respectively. Finally, together with some pressure h the function v of (6.10) satisfies

$$\left.\begin{array}{rcl} v_t - \nu\Delta v + (y\cdot\nabla)v + (v\cdot\nabla)y + \nabla h &=& f \text{ in } Q, \\ -\text{div } v &=& 0 \text{ in } Q, \\ v(x,t) &=& 0 \text{ on } \partial\Omega \times (0,T), \\ v(x,0) &=& v_0(x) \text{ in } \Omega. \end{array}\right\} \tag{6.13}$$

6.2.2 Discretization

In order to provide a numerical approximation of $\hat{J}'(u)$ for given $u \in U$, the partial differential equations (6.11) for $y(u)$ and (6.12) for λ have to be discretized appropriately. For the numerical tests presented in Section 6.4, Taylor-Hood finite elements are used for spatial discretization, i.e. continuous, piecewise quadratic polynomials for the velocity approximation and continuous, piecewise linear polynomials for the pressure approximation for both y and λ. To discretize w.r.t. time, we fix an equidistant time grid $0 = t_0 < \cdots < t_l = T$, where $t_k := k\Delta t$ and $\Delta t := T/l$.

As time discretization scheme for (6.11) we apply a semi-implicit Euler-scheme, i.e. implicit w.r.t. diffusive terms, explicit w.r.t. convective terms. The resulting scheme for states y^j with $0 \le j < l$ is given by

$$\begin{aligned} \tfrac{y^{j+1}-y^j}{\Delta t} - \nu\Delta y^{j+1} + \nabla p^{j+1} &= (Bu)^j - (y^j\nabla y^j) \text{ in } \Omega, \\ -\operatorname{div} y^{j+1} = 0 \text{ in } \Omega,\ y^{j+1} &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{6.14}$$

where $y^0 = y(0)$. To be prepared for Section 6.3.2 we rewrite this forward integration scheme in the form

$$y^{j+1} = F(y^j, (Bu)^j), \tag{6.15}$$

where the time step function $F(y^j, (Bu)^j)$ is defined as

$$F(y,z) := \Delta t\,(P - \nu\Delta t S)^{-1}\,(z - (y\nabla)\,y) + Py. \tag{6.16}$$

Here, S denotes the Stokes operator. The operator P defines the Leray projection $L^2(\Omega)^{2,3} \to \{v \in L^2(\Omega)^{2,3},\ \operatorname{div}\, v = 0\}$ [37]. As will be seen in the subsequent Section 6.3, equation (6.15) matches exactly the time step definition (6.22) that forms the basis for the reversal schedules.

The time discretization scheme of the adjoint variables λ in (6.12) is more involved. Here we take the transpose of the semi-implicit time discretization of equation (6.13) yielding

$$\begin{aligned} \tfrac{v^{j+1}}{\Delta t} - \nu\Delta v^{j+1} + \nabla h^{j+1} &= f^j + \tfrac{v^j}{\Delta t} - (y^j\nabla v^j) - (v^j\nabla y^j) \text{ in } \Omega, \\ -\operatorname{div} v^{j+1} = 0 \text{ in } \Omega,\ v^{j+1} &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{6.17}$$

for $0 \le j < l$, where $v^0 = v(0)$. To derive the time integration scheme for the adjoint variables, it is convenient to reformulate this initial condition in the form

$$\begin{aligned} \tfrac{v^0}{\Delta t} - \nu\Delta v^0 + \nabla h^0 &= \tfrac{v_0}{\Delta t} - \nu\Delta v_0 \text{ in } \Omega, \\ -\operatorname{div} v^0 = 0 \text{ in } \Omega,\ v^0 &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{6.18}$$

where we set $h^0 \equiv 0$ and require $v_0 \in V \cap H^2(\Omega)^2$.

Let now $\lambda \equiv \lambda^1$. Formally transposing the scheme (6.17), (6.18), we obtain for the adjoint equations and $j = l-k \ge 0$ the integration scheme

$$\begin{aligned} \tfrac{\lambda^{l-k}}{\Delta t} - \nu\Delta\lambda^{l-k} &= \tfrac{\lambda^{l-k+1}}{\Delta t} - \nabla\xi^{l-k} - J_{1_y}^{(t_{l-k})}(y^{l-k}) \\ &\quad + (\lambda^{l-k+1}\nabla y^{l-k}) - (\nabla y^{l-k})^t\lambda^{l-k+1} \text{ in } \Omega, \\ -\operatorname{div}\lambda^{l-k} = 0 \text{ in } \Omega,\ \lambda^{l-k} &= 0 \text{ on } \partial\Omega \end{aligned} \tag{6.19}$$

where the states y^{l-k} are given by the solution of (6.14) and $\lambda^i \equiv 0$ for all $i \geq l+1$. We note that the initial state λ^l in this integration scheme satisfies the quasi-Stokes system

$$\begin{aligned} \tfrac{\lambda^l}{\Delta t} - \nu\Delta\lambda^l - \nabla\xi^l = \; & -J_{1_y}^{(t_l)}(y^l, u^l) \text{ in } \Omega, \\ -\text{div } \lambda^l = 0 \text{ in } \Omega, \; \lambda^{l-k} \; = \; & \; 0 \text{ on } \; \partial\Omega. \end{aligned}$$

This adjoint integration scheme may be rewritten as

$$\lambda^j = \bar{F}(t_j, y^j, \lambda^{j+1}), \tag{6.20}$$

where

$$\bar{F}(t, y, z) := \Delta t\,(P - \nu\Delta t S)^{-1}\left(-J_{1_y}^{(t)}(y) + (z\nabla)\,y - (\nabla y)^t\,z\right) + Pz. \tag{6.21}$$

Next let us briefly discuss the time discretization of $\hat{J}''(u)\delta u$ described in Algorithm I. To begin with we note that

$$\langle G_{yy}(y(u), u)(a, b), (\lambda, \lambda^2)\rangle_{Z^*,Z} = \langle (a\nabla)b + (b\nabla a), \lambda\rangle_{L^2(V^*),L^2(V)},$$

which is independent of y and u, and $G_u(y, u) = (-B, 0)$. To discretize step 1 of Algorithm I we propose to apply scheme (6.17) – (6.18) for the computation of v with $v_0 \equiv 0$. In step 2 the numerical results of schemes (6.14) for y and (6.19) for λ are utilized to evaluate the right-hand side $g(y, u, \lambda, v)$. Finally, for the numerical computation of μ in step 3, we propose to utilize again scheme (6.19), where the terms containing the functional J have to be replaced by the corresponding terms of the right-hand side $g(y, u, \lambda, v)$ from step 2. We emphasize that in general both, the state y and the variable v enter into the right-hand side $g(y, u, \lambda, v)$.

Remark 6.2.2. *The application of the time-discretization scheme* (6.19) *for the adjoint equations guarantees among other things symmetry of the discretized reduced Hessian, c.f.* (6.5). *It also follows immediately from this approach that the states y in the convective terms in* (6.19) *are taken at the same time instances as the unknown adjoint variables.*

We finally note that $v_0 \in V \cap H^2(\Omega)^2$ is the minimal regularity requirement for proving error estimates for numerical approximation schemes of the Navier-Stokes system, compare [40, 90].

In practical applications of flow control, the storage of the states y (and v) needed for the adjoint calculations forms a serious bottleneck, in particular for large time horizons $[0, T]$. As remedy to this memory requirement problem, in the next section we propose an optimal memory reduced procedure for calculating adjoints which allows a memory reduction of two orders of magnitude for our numerical example while only causing a slow down factor of 2-3 in run-time.

6.3 Reversal Schedules

In this section we give a general introduction of reversal schedules based on checkpointing strategies. They can be applied for the numerical integration of backward-in-time problems such as (6.12). To this end, we refer to (6.2) as realization of a forward PDE and to (6.4) as realization of an adjoint PDE, respectively. Furthermore, we present a new proof of optimality. Using this new technique, one can derive properties of the optimal reversal schedules that have been unknown so far. They provide new insight and may be used for further theoretical studies.

6.3.1 Subfunctions, Taping, and Adjoint Variables

For calculating an approximation of the state $y(u)$ corresponding to the control u one usually has to evaluate subfunctions F_j, $0 \leq j < l$, called time steps, that act on the state y^j to calculate the subsequent intermediate state y^{j+1} for $0 \leq j < l$ depending on a control action $\bar{u}^j$, i.e.,

$$y^{j+1} \;=\; F_j(y^j, \bar{u}^j)\,. \tag{6.22}$$

In our application we have $F_j(y^j, \bar{u}^j) = F(y^j, (Bu)^j)$ with F from (6.16). To compute a solution of the adjoint PDE the discretization yield adjoint time steps $\bar{F}_j$ for $l > i \geq 0$ with

$$\lambda^j \;=\; \lambda^{j+1} F_j'(y^j, \bar{u}^j) \;\equiv\; \bar{F}_j(y^j, \bar{u}^j, \lambda^{j+1})\,. \tag{6.23}$$

In our model application, we have $\bar{F}_j(y^j, \bar{u}^j, \lambda^{j+1}) = \bar{F}(t_j, y^j, \lambda^{j+1})$ with $\bar{F}$ given by (6.21), where the prime denotes differentiation with respect to y. Note that in our application λ^{j+1} does not directly depend on the control action $\bar{u}^j$, which is the case in many control problems for partial differential equations. The evaluation of $\bar{F}_j$ may require some intermediate results calculated during the computation of y^{j+1} from the previous state y^j. Hence, it is supposed that for each $j \in \{0, \ldots, l-1\}$, there exists a preparation step $\hat{F}_j$ that combines the recording of intermediate values onto a data structure called *tape* and the evaluation of F_j. In our application, in addition to the evaluation of F_j the recording step $\hat{F}_j$ performs the storage of y^j and $(Bu)^j$ on disk and/or memory stacks.

Using the recording step $\hat{F}_j$ and the adjoint time step $\bar{F}_j$ the basic way to integrate the adjoint PDE reads as follows.

Algorithm II: Basic approach

```
Initialization: Set $y$ to the initial value $y_0$.

Recording: do $j = 0, l-1$
               Perform $y^{j+1} = \hat{F}_j(y^j, \bar{u}^j)$
           end do

Reverse: Initialize the adjoint $\lambda^{l+1} = 0$
         do $j = l, 0, -1$
             Perform $\lambda^j = \bar{F}_j(y^j, \bar{u}^j, \lambda^{j+1})$
         end do
```

Remark 6.3.1. *Let us note that the basic approach does not require a-priori discretization of the controls. Rather it requires evaluation of the control actions $\bar{u}$, respectively at the time instances t_j. In our application, we have $\bar{u} = Bu$.*

It is clear that the storage requirement of the basic approach is proportional to the number l of time steps because intermediate data of l time steps are stored during the recording steps. Suppose that we want to calculate the adjoint of a real-world problem with thousand of time steps. Then the memory requirement of the basic approach becomes a problem even if only the intermediate states y_j, $j = 0, \dots, l$, have to be saved on the tape. For example, for computing 3D flows with unstructured grids one may need easily 10 to 100 MBytes to store one state vector y^j [102]. Therefore, it is reasonable to assume that due to their size, only a very limited number of intermediate states can be kept in memory. They will be used as checkpoints. Applying a checkpointing technique, the required intermediate values are generated piecewise by restarting the evaluation repeatedly from these suitably placed checkpoints, according to requests by the reversal process. Therefore, the calculation of F can be reversed based on a checkpointing strategy, even in such cases where the basic method to calculate adjoints fails due to excessive memory requirement (see e.g. [76, 80]).

6.3.2 Reversals of Function Evaluations

To derive an optimal reversal strategy, one has to take into account four parameters:

1.) the number l of time steps to be reversed;

2.) the number c of checkpoints that can be accommodated;

3.) the number p of processors that are available; and

4.) the step costs: $\tau_j = TIME(F_j)$, $\hat{\tau}_j = TIME(\hat{F}_j)$, $\bar{\tau}_j = TIME(\bar{F}_j)$.

If the number of time steps l is known a-priori, one very popular checkpointing strategy is to distribute the checkpoint equidistantly over the time interval, see e.g. [87]. This technique is also known as windowing, see [10]. However, it was shown in [153] that this checkpointing approach is not optimal. In this paper, we apply well known optimal reversal schedules for serial machines, i.e. $p = 1$, and constant step costs $\tau_j = \tau \in \mathbb{R}$, also known as offline checkpointing. However, we present a new proof of optimality. The optimal offline checkpointing allows an enormous reduction of the memory required to reverse a given evolution in comparison with the basic approach (see e.g. [69, 80, 70]). This achievable memory reduction will be illustrated by the numerical results in Section 6.4. Even if the step costs $\tau_j = TIME(F_j)$ are not constant it is possible to compute optimal reversal schedules for one processor machines [147]. However in both cases, i.e. constant and non constant step costs, one has to pay for the improvements in the form of a greater temporal complexity because of repeated forward integrations. If a multi-processor can be applied to perform the reversal then optimal parallel reversal schedules ensure that the adjoint values are obtained in minimal wall-clock time [147].

Nevertheless, our motivation was to study the run-time behavior of adjoint calculations using a serial machine. Furthermore, the step costs τ_j are nearly constant. Therefore, we apply the offline checkpointing provided by the routine revolve for constant step costs [76]. To reverse a given time integration, revolve initiates a do-loop similar to the following one:

Algorithm III: Offline checkpointing (reversal schedule)

```
Initialization: Reserve space for $c$ checkpoints and set the first one to the
initial state.

do $p = l$, 1, -1

    Advance: Starting from the last checkpoint assigned, advance to
    state $y^{p-1}$, performing forward time steps without recording of in-
    termediates. If one or more checkpoints are free, set as many of them
    as possible to intermediate states along the way.

    Reverse: Perform the recording step $y^p = \hat{F}_{p-1}(y^{p-1}, \bar{u}^{p-1})$ and the
    reverse step $\bar{F}_{p-1}(y^{p-1}, \bar{u}^{p-1}, \lambda^p)$ to calculate the adjoint $\lambda^{p-1}$. If
    state $y^{p-1}$ is stored in a checkpoint, free the checkpoint up for sub-
    sequent use.

end do
```

The question arises, where to place the checkpoints in the Advances of Algorithm III. The strategy implemented in revolve ensures that the initiated reversal process is provably optimal with respect to the run-time increase for a given number of checkpoints. Moreover, it can be shown that the resulting reversal schedules allow a growth of memory and run-time that is only logarithmic in the number l of time steps [69]. One such optimal reversal schedule computed with revolve for ten time steps and three checkpoints is shown in Fig. 6.1.

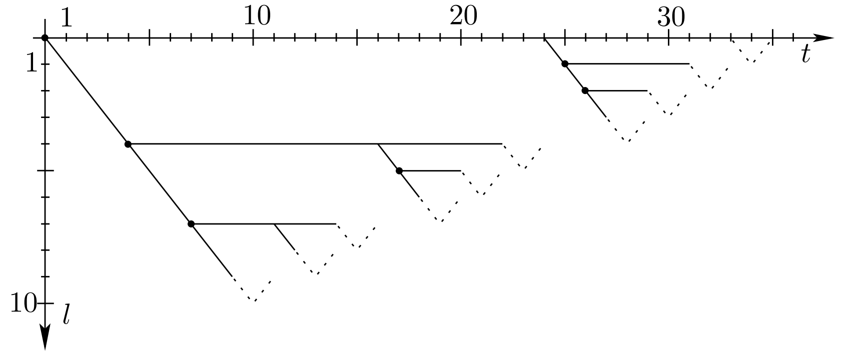

Figure 6.1: Optimal Reversal Strategy for $l = 10$ and $c = 3$

Here, the time steps are plotted along the vertical axis. The computing time required for the reversal is represented by the horizontal axis that can be thought of as the computational axis. Each solid horizontal line including the computational axis itself represents a checkpoint. The solid slanted lines represent the Advances of Algorithm III. Here, the setting of a checkpoint is marked with a dot. The corresponding Reverses are visualized by dotted slanted lines.

6.3.3 Minimal Reversal Cost

Naturally, one wants to analyze the increase in run-time caused by the checkpointing approach in more detail. Usually, the number of checkpoints that fit into the available memory is fixed and the number of time steps to be reversed varies depending on the current problem specification. For given values of the number l of time steps and the number of checkpoints c, let $t(l, c)$ denote the minimal number of time steps F_j that are performed during the execution of a reversal schedule.

To establish an explicit formula for the minimal cost $t(l, c)$ we introduce some characteristic quantities. First, we will examine the number of times one particular time step is evaluated during the execution of a given reversal schedule S:

Definition 6.3.2 (Repetition Number)**.** *Let a reversal schedule S for the reversal of l time steps be given. The repetition number $r_i(S)$ counts how often the time step F_i is evaluated during the execution of the reversal schedule S. The maximal repetition number $r_{max}(S)$ is defined by*

$$r_{max}(S) = \max_{0 \le i \le l-1} r_i(S).$$

The repetition numbers can be used to group the time steps to be reversed in the following way:

Definition 6.3.3 (Frequency Number)**.** *Suppose a reversal schedule S for the reversal of l time steps is given. The frequency numbers $m_j(S)$, $0 \le j \le r_{max}(S)$, defined by*

$$m_j(S) \ = | \ \{r_i(S) = j, \ 0 \le i \le l-1\} \ |,$$

count how many time steps are evaluated exactly j times during the execution of the reversal schedule S.

Hence, the operator $|\cdot|$ in Definition 6.3.3 determines the cardinal number of a given set. To establish an explicit formula for the frequency numbers, a given reversal schedule S will be decomposed into two smaller substructures S^1 and S^2 as illustrated in Fig. 6.2. As shown in [70, Lemma 12.2], each reversal schedule S can be decomposed into two subschedules S^1 and S^2 without affecting the overall execution cost. This can be done by first storing the intermediate state $y^{\check{l}}$ into a checkpoint. Then the time steps $F_{\check{l}}, ..., F_{l-1}$ are reversed using up to $c-1$ checkpoints at any time with the subschedule S^2. Subsequently, the time steps $F_0, ..., F_{\check{l}-1}$, are reversed using the subschedule S^1. This can be defined by the following binary composition

$$S = S^1 \circ S^2. \tag{6.24}$$

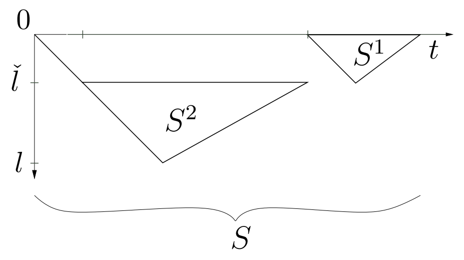

Figure 6.2: Decomposition of reversal schedule $S = S^1 \circ S^2$

Since a reversal schedule S for reversing l time steps can also be used to reverse $\tilde{l}$ time steps with $\tilde{l} < l$, we now define $l_{max}(S)$ as the maximal number of time steps that can be reversed with the reversal schedule S and the corresponding $r_{max}(S)$ as defined in Def. 6.3.2. Using the decomposition (6.24), one can then easily derive the relations

$$l_{max}(S^1 \circ S^2) = l_{max}(S^1) + l_{max}(S^2), \tag{6.25}$$
$$r_{max}(S^1 \circ S^2) = \max\left\{r_{max}(S^1) + 1, r_{max}(S^2)\right\}, \tag{6.26}$$
$$m_j(S) = m_j(S^2) + m_{j-1}(S^1), \tag{6.27}$$

see, e.g. [135]. We will employ the identities (6.25) - (6.27) to derive an explicit formula for the minimal number $t(c, l)$ of time step evaluations that are required for computing the adjoint of an given sequence with l time steps with up to c checkpoints accommodated at any time.

As is proven in [69], the maximal number $l(c, r)$ of time steps, which can be reversed with up to c checkpoints and at most r evaluations of each time step is given by

$$l(c, r) = \beta(c, r) \equiv \binom{c+r}{c} \approx \frac{1}{\sqrt{2\pi}} \left[1 + \frac{r}{c}\right]^c \left[1 + \frac{c}{r}\right]^r \sqrt{\frac{1}{c} + \frac{1}{r}}, \tag{6.28}$$

using (6.25). Let $t(S)$ denote the number of time step evaluations that are required by S to compute the adjoint of a sequence with $l_{max}(S)$ time steps. For a reversal schedule S using c checkpoints with

$$\beta(c, r-1) < l_{max}(S) \leq \beta(c, r), \tag{6.29}$$

and $t(S) = t(c, l_{max}(S))$, i.e., S is optimal, one has obviously

$$r_{max}(S) = r. \tag{6.30}$$

Using the concept of repetition numbers, now one can prove additionally the following property:

Theorem 6.3.4 (Explicit Formula for Frequency Numbers)**.** *Assume that l time steps are reversed using Algorithm III with up to c checkpoints accommodated at any time. Let the number of time step evaluations be minimal, i.e. equal to $t(c, l)$. Then, one has*

$$m_j(c, l) = \beta(c-1, j) = \binom{c-1+j}{c-1}, \qquad 0 \leq j < r, \tag{6.31}$$

where $m_j(c,l)$ denotes the number of time steps evaluated exactly j times for reversing l time steps using up to c checkpoints and the minimal number of time step evaluations $t(c,l)$. The integer r is determined by the equations (6.29) *and* (6.30)*.*

Proof: The relation (6.31) can be proven by induction:

Trivial Case I: $c = 1$.
To achieve the minimal number $t(1,l)$ of time step evaluations, we can explicitly derive the corresponding reversal schedule: The only checkpoint available must be set to the initial state y^0. For the first Reverse action of Algorithm III, the time steps F_j, $0 < j < l-1$ have to be evaluated. To perform the next Reverse action, the time steps F_j, $0 < j < l-2$ have to be evaluated and so on. Hence, for the overall reversal, the first time step F_0 is evaluated $l-1$ times, the second time step F_1 is evaluated $l-2$ times and so on. Therefore, we obtain $m_j(c,l) = 1,\ 0 \le j \le l-1 = r$ since

$$l(1,r-1) = \beta(1,r-1) = r-1 < l \le r = \beta(1,r) = l(c,r).$$

Using the definition of the function $\beta(c,j)$, it follows that

$$m_j(1,l) = 1 = \beta(0,j), \qquad 0 \le j \le r.$$

Trivial Case II: $j = 0$.
As can be seen from Algorithm III, all time steps except the last one, i.e., $\hat{F}_{l-1}$, are evaluated at least one time, since during the reversal schedule only the preparing step $\hat{F}_{l-1}$ and the reverse step $\bar{F}_{l-1}$ but not the original time step F_{l-1} are executed. Therefore, one obtains $m_0(c,l) = 1$ and

$$m_0(c,l) = 1 = \beta(c-1,0).$$

using once more the definition of the function $\beta(c,j)$. Hence, for the two trivial cases, the identity (6.31) holds.

Induction step in lexicographical order of (c,j):
The numbers $c > 1$ and $j > 0$ are given. Assume that the assertion (6.31) is true for all $(\tilde{c},\tilde{j})$ with $\tilde{j} < j$ or $\tilde{c} = c$ and $\tilde{j} < j$. Now it will be shown that equality (6.31) is valid for the pair (c,j).

The applied optimal reversal schedule S can be split into two subschedules $S = S^1 \circ S^2$, where the subschedule S^1 uses up to c checkpoints and the subschedule S^2 up to $(c-1)$ checkpoints as explained above. Furthermore, since S is optimal also the subschedules S^1 and S^2 have to be optimal since otherwise S could be improved. Then we have due to the induction assumption that

$$\begin{aligned} m_{j-1}(S^1) &= m_{j-1}(c,\check{l}) = \beta(c-1,j-1), \\ m_j(S^2) &= m_j(c-1,l-\check{l}) = \beta(c-2,j). \end{aligned} \tag{6.32}$$

Applying the recursive formula (6.27) for frequency numbers, we obtain

$$\begin{aligned} m_j(S) &= m_j(S^2) + m_{j-1}(S^1) = m_j(c-1, l-\check{l}) + m_{j-1}(c, \check{l}) \\ &= \beta(c-2, j) + \beta(c-1, j-1) = \frac{(c+j-2)!}{(c-2)!\, j!} + \frac{(c+j-2)!}{(c-1)!\,(j-1)!} \\ &= \frac{(c+j-2)!}{(c-2)!\,(j-1)!}\left(\frac{1}{j} + \frac{1}{c-1}\right) = \frac{(c+j-2)!}{(c-2)!\,(j-1)!}\,\frac{(c+j-1)}{j\,(c-1)} \\ &= \frac{(c+j-1)!}{(c-1)!\, j!} = \beta(c-1, j) = m_j(c, l). \end{aligned}$$

Therefore, the relation (6.31) is proven. ∎

Now, we can present a new proof for the explicit formula of $t(c, l)$ that is based on the frequency numbers and much simpler in the argumentation than the original one presented in [76]:

Theorem 6.3.5 (Minimal Evaluation Cost)**.** *The minimal evaluation cost to reverse l time steps with up to c checkpoints accommodated at any time has the explicit form*

$$t(l, c) = rl - \beta(c+1, r-1), \tag{6.33}$$

with r being the unique integer satisfying $\beta(c, r-1) < l \leq \beta(c, r)$.

Proof: We will employ the frequency numbers to prove the explicit formula of the minimal cost $t(l, c)$. For given values of l and c, we have

$$t(l, c) = \sum_{j=0}^{r} j\, m_j(c, l) \qquad \text{and} \qquad l = \sum_{j=0}^{r} m_j(c, l). \tag{6.34}$$

The last equation yields

$$m_r(c, l) = l - \sum_{j=0}^{r-1} m_j(c, l). \tag{6.35}$$

We derive equation (6.33) in two steps: First, identity (6.33) will be proven for the case $l = \beta(c, r)$. Second, we will generalize this result for situations, where the number l of time steps satisfies $l(c, r-1) < l < l(c, r)$.

Step 1: $l = \beta(c, r)$
Using Theorem 6.3.4 and the identity (6.35), we have

$$\begin{aligned} m_r(c, l) &= l - \sum_{j=0}^{r-1} m_j(c, l) = l - \sum_{j=0}^{r-1} \beta(c-1, j) = \beta(c, r) - \sum_{j=0}^{r-1} \binom{c-1+j}{c-1} \\ &= \binom{c+r}{c} - \binom{c-1+r}{c} = \binom{c+r-1}{c-1} = \beta(c-1, r). \end{aligned}$$

Using (6.34), it follows that

$$\begin{aligned}
t(l,c) &= \sum_{j=0}^{r} j\, m_j(c,l) = \sum_{j=0}^{r} j\, \beta(c-1,j) = \sum_{j=0}^{r} j \frac{(c+j-1)!}{j!(c-1)!} \\
&= \sum_{j=0}^{r} \frac{(c+j-1)!}{(j-1)!(c-1)!} = \sum_{j=0}^{r} c \,\frac{(c+j-1)!}{(j-1)!c!} = c\sum_{j=0}^{r} \beta(c,j-1) \\
&= c\sum_{j=1}^{r} \binom{c+j-1}{c} = c\binom{c+r-1}{c+1} = c\frac{(c+r)!}{(c+1)!(r-1)!} \\
&= \frac{(c+r)!}{c!(r-1)!} - \frac{(c+r)!}{(c+1)!(r-1)!} = r\binom{c+r}{c} - \binom{c+r}{c+1} \\
&= rl - \beta(c+1,r-1).
\end{aligned}$$

Step 2: $\beta(c,r-1) = l(c,r-1) < l < l(c,r) = \beta(c,r)$
For this case, one can derive

$$\begin{aligned}
m_r(c,l) &= l - \sum_{j=0}^{r-1} m_j(c,l) = l - \binom{c-1+r}{c} \\
&= l - \beta(c,r-1) + \beta(c,r) - \beta(c,r) = l + \beta(c-1,r) - \beta(c,r).
\end{aligned}$$

Using once more (6.34) yields

$$\begin{aligned}
t(c,l) &= \sum_{j=0}^{r} j\, m_j(c,l) = \sum_{j=0}^{r-1} j\, \beta(c-1,j) + r\, m_r(c,l) \\
&= \sum_{j=0}^{r-1} j\, \frac{(c+j-1)!}{j!(c-1)!} + r\beta(c-1,r) - r(\beta(c,r)-l) \\
&= \sum_{j=0}^{r-1} \frac{(c+j-1)!}{(j-1)!(c-1)!} + r\frac{(c+r-1)!}{r!(c-1)!} - r(\beta(c,r)-l)
\end{aligned}$$

$$\begin{aligned}
&= \sum_{j=0}^{r} \frac{(c+j-1)!}{(j-1)!(c-1)!} - r(\beta(c,r)-l) = c\beta(c+1,r-1) - r(\beta(c,r)-l) \\
&= rl + c\frac{(c+r)!}{(c+1)!(r-1)!} - \frac{(c+r)!}{c!(r-1)!} = rl + \frac{(c+r)!}{c!(r-1)!}(\frac{c}{c+1} - 1) \\
&= rl - \beta(c+1,r-1).
\end{aligned}$$

Hence, equation (6.33) was proven for any number l of time steps that satisfies the inequality $\beta(c,r-1) < l \leq \beta(c,r)$. ∎

Using Theorem 6.3.5 and Stirling's formula, the equation (6.28) yields

$$l \sim \begin{cases} \exp(r)/\sqrt{r} & \text{if} \quad c \sim r \\ c^r & \text{if} \quad c \gg r = const \\ r^c & \text{if} \quad r \gg c = const \end{cases}$$

as well as the asymptotic behavior

$$\lim_{l\to\infty} \frac{t(l,c)}{l^{1+c}} = \sqrt[c]{c!} \approx \frac{c}{e}.$$

Thus we see that binomial checkpointing allows a reduction of the spatial complexity by the factor of size c/l at the expense of an increase in the temporal complements of size $\sqrt[c]{l}$ which can be seen as an attractive way to reduce the enormous memory requirement caused by the basic approach as given by Algorithm II.

6.4 Numerical results

To illustrate the potential of the checkpointing approach with respect to memory reduction, a cavity flow problem serves as numerical example. The domain is given by the unit square $\Omega := (0,1) \times (0,1)$. We normalized the final time to 1, i.e., $T = 1$, and set $\mathrm{Re} = \frac{1}{\nu} = 10$. The equation

$$y_0(x) \quad = \quad e \begin{bmatrix} (\cos 2\pi x_1 - 1)\sin 2\pi x_2 \\ -(\cos 2\pi x_2 - 1)\sin 2\pi x_1 \end{bmatrix}$$

with the Euler number e is used as initial condition. The time-dependent function to be approximated is given by

$$z(x,t) \quad = \quad \begin{bmatrix} \varphi_{x_2}(x_1,x_2,t) \\ -\varphi_{x_1}(x_1,x_2,t) \end{bmatrix}.$$

Here, φ is defined through the stream function

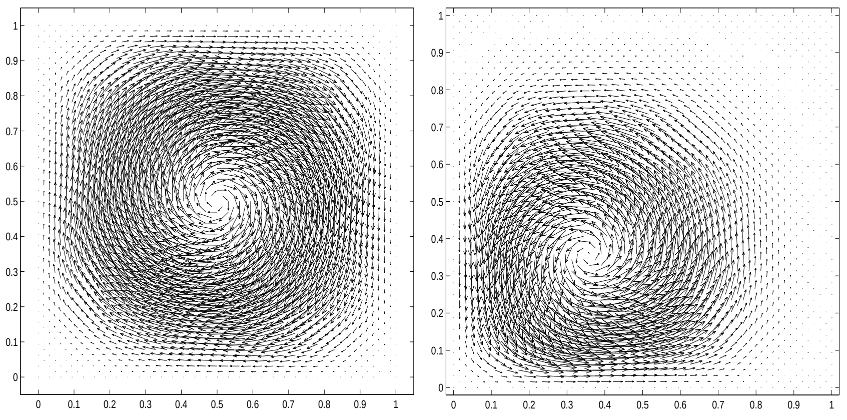

Figure 6.3: Left: Cavity flow at $t = 0.01$, right: Desired flow at $T = 1$

$$\varphi(x_1,x_2,t) = \theta(x_1,t)\theta(x_2,t)$$

with $\theta(y,t) = (1-y)^2(1-\cos 2k\pi t)$ for $y \in [0,1]$. Fig. 6.3 shows the cavity flow at $t = 0.01$ together with the desired flow at $T = 1$. As cost function we chose

$$J(y,u) \quad = \quad \frac{1}{2}\int_0^T \int_\Omega |y-z|^2 dx\, dt + \frac{c}{2}\int_0^T \int_\Omega |u|^2 dx\, dt$$

with $c = 0.01$. In what follows we focus on the numerical evaluation of $\hat{J}'(u)$, where we recall that $\hat{J}(u) = J(y(u), u)$.

For this application, using the recording steps $\hat{F}(y^j, (Bu)^j)$ and the adjoint steps $\bar{F}(t_j, y^j, \lambda^{j+1})$, one may compute the gradient $\hat{J}'(u)$ applying the basic approach as given by Algorithm II. Then, a complete trajectory is stored onto the tape during the forward integration. Subsequently, the tape is read backward during the adjoint calculation. Using a discretization with 2113 velocity nodes and 545 pressure nodes, the storage of the full state information causes a memory requirement of 3.8 MByte for $l = 100$ time steps. If 10.000 time steps have to be performed the memory requirement equals 380 MByte. Note, that only a 2D problem is considered. For computing the solution of a 3D problem the situation gets even worse.

The reversal schedules presented in Section 6.3 now offer the opportunity to reduce this memory requirement drastically. For that purpose, the routine revolve is employed, which realizes a do-loop like that illustrated by Algorithm III in Section 6.3. The implementation of revolve is described in detail in [76], to where we refer for detailed implementational issues. Nevertheless, one has to note that for the performed do-loop, in addition to the functions $F(y^j, (Bu)^j)$, $\hat{F}_j(y^j, (Bu)^j)$, and $\bar{F}(t_j, y^j, \lambda^{j+1})$ which are required also for the basic approach to calculate adjoints, only coding of the routines for storing and retrieving a checkpoint is necessary. Therefore, it is usually no problem to incorporate a reversal schedule into the adjoint calculation. The resulting enormous reduction of memory requirement at the cost of a comparably slight increase in run-time is illustrated in the remainder of this section.

The desired gradient $\hat{J}'(u)$ was computed applying revolve and several numbers of checkpoints. Figure 6.4 shows the observed run-time behavior for $l = 100$ time steps. Here, the vertical axis gives the ratio of the run-time needed to compute $\hat{J}'(u)$ and the run-time to compute $\hat{J}(u)$. The horizontal axis denotes the number of checkpoints used by the reversal schedule.

To compare the achieved results with the basic approach for computing adjoints, one has to note that the run-time for computing $\hat{J}'(u)$ is bounded above by a small constant times the run-time to compute $\hat{J}(u)$. The value of the constant varies between three and five depending on the specific operation counts and memory accesses [70]. As can be seen, the run-time ratio for the checkpointing approach varies between 2 and 5 for a reasonable number of checkpoints. This behavior results in a slow down factor up to 2 compared to the basic approach, where a complete trajectory is stored to compute the adjoint values. That is, using the checkpointing approach the computation of the adjoints is at most twice as slow as the basic approach, where a complete forward trajectory is stored. Nevertheless, the achieved memory requirement can be reduced enormously. Using the same example discretization as mentioned above, namely 2113 velocity nodes and 545 pressure nodes one needs 38 kByte to store one checkpoint. Hence, if the reversal schedule utilize 5 checkpoints, the memory requirement equals 228 kByte for calculating adjoints with reversal schedules. If the basic approach is applied, the memory requirement amount to 3.8 MByte. The dependence of the run-time ratio on the size of the grid as shown in Fig. 6.4 is remarkable. According to the theory developed in

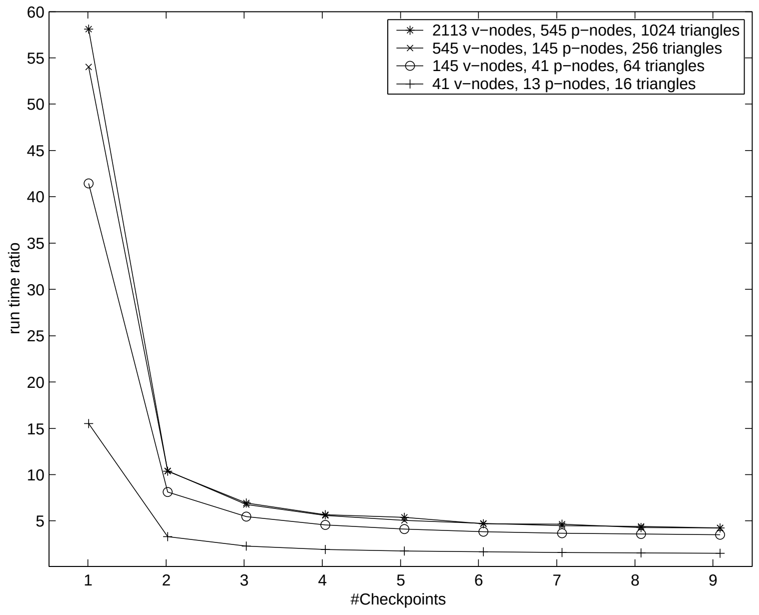

Figure 6.4: Run-time behavior, 100 Time Steps

[70] the displayed run-time ratio should be mesh independent. However, so far, there is no explanation for the mesh-dependent behavior and further studies are needed.

The possible memory reduction gets even more impressive if the number of time steps grows. Figure 6.5 shows the observed run-time behavior for $l = 1000$ time steps. Here, the run-time ratio varies between 2 and 7 for a reasonable number of checkpoints. Compared to the basic approach, where a complete trajectory is stored and read backwards to calculate the adjoints, these run-times represent a slow down factor of up to three. On the other hand, the memory requirement can be reduced drastically in comparison to the basic approach. For example, using 20 checkpoints the adjoint calculation based on the reversal schedule requires 798 kByte for the same discretization as above. Applying the basic approach, the used memory adds up to 38 MByte. Hence, there are two orders of magnitude between the two storage requirements. Therefore, it is shown that reversal schedules enable an immense memory reduction at a slight increase in run-time.

The zig-zagging of the run-time curves in Fig. 6.5 can be interpreted in the following way: Other processes had to be performed on the machine during the calculation of the gradient is performed. Therefore, the run-time depends also naturally on the current load of the machine. Since the resulting variations vary around a certain run-time ratio if the number of checkpoints exceeds an lower bound one may draw the conclusion that the number of checkpoints is not the main influence on the run-time behavior. Hence above the lower bound, the number of checkpoints can be varied without a big influence on the run-time. The fact is also illustrated by the flat development of the run-time ratios in

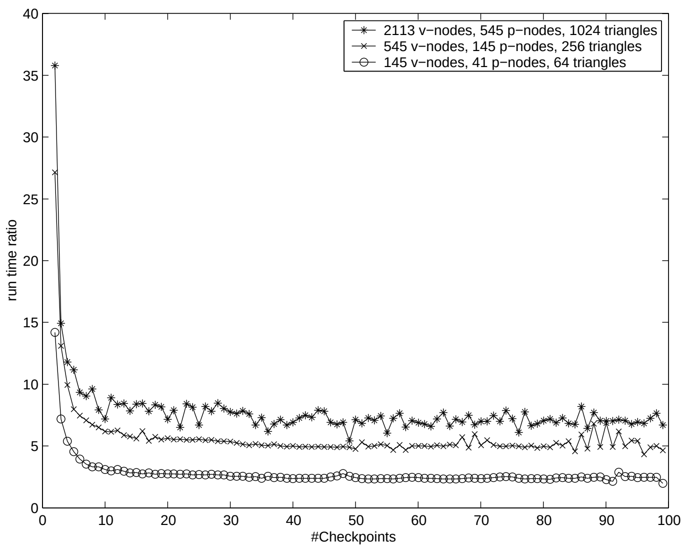

Figure 6.5: Run-Time behavior, 1000 Time Steps

Fig. 6.4 if the number of checkpoints exceeds three.

Figure 6.6 shows two frequency number profiles for 1000 time steps, where $c = 5$ and $c = 10$ checkpoints are available resulting in $r = 8$ in the first case and $r = 4$ in the second case. As we observe, the frequency number profiles for these numbers of checkpoints are quite different. For $c = 5$ checkpoints, we have that $m_8(5, 1000) < m_7(5, 1000)$. This is due to the fact $1000 \ll l_{max}(5, 8) = 1287$. As a consequence, for $c = 5$ the maximal frequency number $\beta(4, 8) = 495$ for 5 checkpoints and 8 repetitions is not attained. Instead, we have $m_8(5, 1000) = 208$ that is given by

$$m_8(5, 1000) = m_8(5, 1000) - (l_{max}(5, 8) - l) = 495 - (1287 - 1000) = 208,$$

see the proof of Theorem 6.3.5. Hence, only 208 time steps are executed exactly eight times. In the second case, i.e., for $c = 10$ checkpoints and $r = 4$, the inequality $m_4(10, 1000) > m_3(10, 1000)$ is valid. We have $l = 1000 \approx l_{max}(10, 4) = 1001$, which is the maximal number of time steps that can be reversed with $c = 10$ and at most $r = 4$ executions of each time step. Therefore, in this case, almost the maximal frequency number $\beta(9, 4) = 715$ is required, since $l_{max}(10, 4) - l = 1001 - 1000 = 1$. Thus, 714 time steps are executed four times.

All computations were performed on an two-processor personal computer with ADLON 1GHz CPU and 512 MB memory.

6.5 Summary

This paper in its first part presents recipes for the numerical treatment of reduced gradients and reduced Hessians times increments for cost functionals

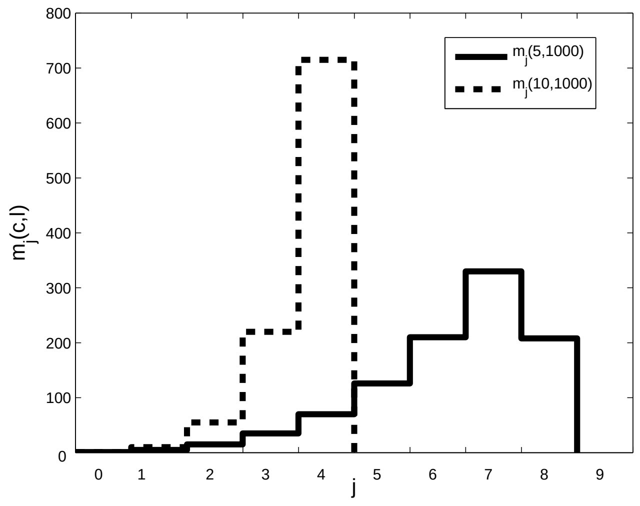

Figure 6.6: Frequency number profiles for 1000 time steps with 5 and 10 checkpoints

involving the incompressible Navier-Stokes systems as state equations. Among other things these recipes guarantee symmetry of the discretized reduced Hessian. Their core is a balanced numerical treatment of the Navier-Stokes equations, their derivative and the adjoint of the latter.

For adjoint calculations, one has to provide information computed during the forward integration in reverse order. The basic approach, namely the complete recording of the required information onto one stack, causes an enormous memory requirement.

As remedy, in its second part the paper presents optimal reversal schedules that allow a drastic reduction of the memory requirement at only a comparable slight increase in run-time. In this context, we present a new proof for the optimal complexity yielding also so far unknown properties of the optimal reversal schedules. The resulting memory reduction and run-time behavior is studied for the calculation of adjoints of the incompressible Navier-Stokes equations by applying the checkpointing routine revolve. The achieved results are quite promising. Nevertheless, one has to study further the dependence of the run-time ratios on the size of the discretization.

Acknowledgment

The first author acknowledges support of the Sonderforschungsbereich 609 that is funded by the Deutsche Forschungsgemeinschaft. Furthermore, we would like to thank the anonymous referees for their valuable comments and hints.

Chapter 7

A First-order Convergence Analysis of Trust-region Methods with Inexact Jacobians

Andrea Walther
Submitted to SIAM Journal on Optimization (in revision)

Abstract:
A class of trust-region sequential quadratic programming algorithms for the solution of minimization problems with nonlinear equality constraints is analyzed. The considered class of optimization methods does not require the exact evaluation of the constraint Jacobian in each optimization step but uses only an approximation of this first-order derivative information. Hence, the presented approach is especially well suited for equality constrained optimization problems where the Jacobian of the constraints is dense.

The accuracy requirements for the presented first-order global convergence result are based on the feasibility and the optimality of the iterates. The corresponding criteria can be verified easily during the optimization process to adjust the approximation quality of the constraint Jacobian.

7.1 Introduction

Trust-region successive quadratic programming (SQP) algorithms have been applied efficiently to solve a wide range of nonlinear optimization problems given by

$$\min_{x \in \mathbb{R}^N} f(x) \qquad \text{subject to } c(x) = 0, \tag{7.1}$$

where the objective $f : \mathbb{R}^N \to \mathbb{R}$ and the vector of the constraints $c : \mathbb{R}^N \to \mathbb{R}^M$ with $N \geq M$ are given smooth functions. For the majority of the trust-region SQP type algorithms, the computation of the next iterate requires the solution of a linear system of the form

$$A(x_k)A(x_k)^T v = b,$$

where

$$A(x) = (\nabla c_1(x), \dots, \nabla c_M(x))^T \in \mathbb{R}^{M \times N}$$

is the exact matrix of the constraint gradients at x. Furthermore, a representation $Z(x)$ of the null space of $A(x)$ is needed frequently for the computation of the next step. For these reasons, the explicit forming and factoring of the constraint Jacobian $A(x)$ provides an efficient step calculation if $A(x)$ is sparse and well structured, see, e.g., [4]. As an alternative, one may use iterative system solves up to a certain accuracy, for example Krylov subspace or multigrid methods, for the step calculation in each iteration, see, e.g., [88, 106, 155]. However, both approaches may result in very time-consuming computations, especially if the Jacobian of the constraints is dense or unstructured. Therefore, we present and analyze in this paper a class of trust-region SQP algorithms that does not require the exact evaluation of the constraint Jacobian or an iterative solution of a linear system with a system matrix that involves the constraint Jacobian. Instead the algorithm proposed works only with an approximation of this first-order information. Hence, the algorithm presented here is well suited for optimization problems of moderate size but with a special structure of the constraint Jacobian. The corresponding applications cover the wide range of periodic adsorption processes including for example the purification of hydrogen. In these cases, the Jacobian of the equality constraints is dense due to the periodicity of the underlying chemical process. As a consequence, the runtime needed for the optimization process may be dominated significantly by the computation of the dense Jacobian and its factorization, see, e.g., [101]. For these optimization tasks and problems with a similar structure, the algorithm proposed in this paper may allow a considerable reduction of the computing time required to calculate a solution.

For numerous optimization problems, the considered system is described by ordinary or partial differential equations the discretization of which yields the equality constraints. Exploiting the direct sensitivity equation or the adjoint differential equation, one can evaluate products of the Jacobian $A(x)$ and a given vector v, i.e., $A(x)v$ and $A(x)^T v$. Related derivative information can be computed also by applying automatic differentiation [70]. Hence, it is reasonable to assume that one can evaluate exact products of the Jacobian multiplied from the right or from the left by a given vector. However, the computation of the complete Jacobian matrix $A(x)$ may be very time-consuming, especially if $A(x)$ is dense or unstructured, since many Jacobian-vector products are required to build the full matrix $A(x)$ in these cases. Therefore, we present an algorithm that uses only Jacobian-vector and vector-Jacobian products but avoids the calculation and factorization of $A(x)$ in each optimization step or the iterative solve of a linear system involving $A(x)$ as part of the system matrix.

To solve the optimization problem (7.1), we follow the approach proposed by Byrd [22] and Omojokun [119]. For composite-step trust-region methods that employ exact information, a comprehensive treatment of the convergence properties can be found in [36]. Implementations of the Byrd-Omojokun trust-region method are used successfully to solve equality constrained NLPs [4, 106]. Related implementations using augmented Lagrangian merit functions are proposed and analyzed in [46]. Extensions of this approach to a more general class of trust-region methods can be found in [42]. Box trust-region methods are analyzed in [64]. More recently, trust-region methods without penalty functions have been developed by Fletcher et al. [49, 50, 51] as well as Ulbrich and Ulbrich [141].

The effects of inexact problem information on the global convergence of inexact SQP methods can be found, for example, in [100, 107, 144]. In a line search setting, the effects of inexact information on the global convergence are studied in [23]. For an inexact composite step trust-region SQP method a first proof of global convergence is given in [88], where the analysis is focused on inexactness arising from iterative system solves. Our analysis and assumptions on inexactness differ from [88] in the following way: We do not consider a splitting of the variables into state and control variables. Hence, we allow general unstructured approximations of the Jacobian $A(x)$ and the corresponding null space representation as well as inexactness due to iterative solves. The proofs of first-order convergence given in this paper are based on ideas presented in [24]. Since we concentrate our analysis on the effects of inexact Jacobian information, the present paper does not examine the performance of the algorithm in the presence of dependent constraint gradients. Therefore, we assume in contrast to [24] throughout that the constraint Jacobian has full rank. Furthermore, we do not incorporate inequality constraints as in [24], since the efficient handling of inequalities in the case of inexact constraint Jacobians is subject of further research.

This paper has the following structure. In Section 7.2 we introduce the notation and the main assumptions that are used for the proof of global first-order convergence. Subsequently, we discuss our inexact trust-region SQP algorithm in Section 7.3. The well-posedness of this algorithm will be shown in Section 7.4. Section 7.5 contains the proof of global convergence to first-order critical points. Finally, some conclusions and possible extensions are presented in Section 7.6.

7.2 Notations and Assumptions

The Lagrangian of (7.1) is defined by

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T c(x).$$

Assuming that a suitable constraint qualification is fulfilled, the first-order optimality conditions yield for an optimal solution x_* of (7.1) that

$$\begin{aligned}\nabla_x \mathcal{L}(x_*, \lambda_*) &= \nabla f(x_*) + A(x_*)^T \lambda_* = 0\\ \nabla_\lambda \mathcal{L}(x_*, \lambda_*) &= c(x_*) = 0\end{aligned}$$

holds for a certain Lagrange multiplier $\lambda_* \in \mathbb{R}^M$. To apply an SQP trust-region algorithm, we approximately solve in the kth iteration the quadratic program

$$\begin{aligned} \min_{d \in \mathbb{R}^N} \ & \nabla f(x_k)^T d + \frac{1}{2} d^T B_k d \\ \text{subject to } & A(x_k) d + c(x_k) = 0 \\ & \|d\| \leq \Delta_k \end{aligned} \tag{7.2}$$

to compute a new step d_k for a given iterate x_k, a given trust-region radius Δ_k, and Lagrange multipliers λ_k. Here and throughout, B_k may stand for the exact second-order information $\nabla^2_{xx}\mathcal{L}(x_k, \lambda_k)$. Then, the functions $f(\cdot)$ and $c(\cdot)$ have to be twice continuously differentiable. Alternatively, one may use a symmetric matrix approximating the Hessian $\nabla^2_{xx}\mathcal{L}(x_k, \lambda_k)$. Furthermore, $\|\cdot\|$ denotes the Euclidean norm $\|\cdot\|_2$.

Since problem (7.2) may have no feasible solution, relaxation strategies were studied, see, e.g., [28, 123, 133]. As alternative to overcome this difficulty, one can use a composite-step method. Following the approach of Byrd and Omojokun, we define the merit function

$$\phi(x; \mu) = f(x) + \mu \|c(x)\|$$

with the penalty parameter $\mu > 0$ to judge the progress toward the solution. This merit function is exact but non-differentiable due to the Euclidean norm in the second term. A model of $\phi(\cdot; \mu_k)$ around an iterate x_k is given by the function

$$m_k(d) = f(x_k) + \nabla f(x_k)^T d + \frac{1}{2} d^T B_k d + \mu_k \|c(x_k) + A(x_k) d\|.$$

For measuring the progress of our algorithm, we define for a given iterate x_k and a step d the actual reduction in the merit function as

$$\text{ared}_k(d) = \phi(x_k; \mu_k) - \phi(x_k + d; \mu_k). \tag{7.3}$$

The predicted reduction in the merit function is defined as the change of the model m_k caused by a step d, i.e.,

$$\begin{aligned} \text{pred}_k(d) &= m_k(0) - m_k(d) \\ &= -\nabla f(x_k)^T d - \frac{1}{2} d^T B_k d + \mu_k (\|c(x_k)\| - \|c(x_k) + A(x_k) d\|). \end{aligned} \tag{7.4}$$

We suppose that for each iteration k one can provide an approximation A_k of the exact Jacobian $A(x_k)$ and an approximation Z_k of an exact null space basis $Z(x_k)$ with $A(x_k)Z(x_k) = 0$ and $A_k Z_k = 0$. Hence, we refer to the exact matrix information as $A(x_k)$ and $Z(x_k)$ and to the corresponding approximation as A_k and Z_k, respectively. The approximation of the derivative matrices using quasi-Newton update formulas fits into this setting. For this purpose, one may employ the well-known symmetric rank one (SR1) update formula to approximate the Hessian $\nabla^2_{xx}\mathcal{L}(x_k, \lambda_k)$. This approach is examined for unconstrained optimization in [35]. The two-sided rank one (TR1) update formula as proposed in [77] can be used to approximate the constraint Jacobian. Another possibility would be to compute the required Hessian-vector products exactly employing for example automatic differentiation. For the first-order information, the exact information $A(x_k)$ and $Z(x_k)$ could be computed for the iterate k and used for the following iterates as long as the restrictions on the inexactness are fulfilled. This is a promising approach since the iterates converge frequently in a tangential way toward the optimal solution. This observation holds when the Hessian is approximated for example by a quasi-Newton formula and the exact Jacobian of the constraints is used. We observe the same behavior in our first numerical experiments for numerous test problems using the TR1 update to approximate the Jacobian. Therefore, the changes in the null space will be hopefully rather small at the end of the optimization procedure.

To prove the convergence results presented in this paper, we define $D \equiv N - M$ and make the following assumptions:

(AS1) $A(x_k)$ has full row rank for all iterates x_k with $\sigma_D(A(x_k)) \geq \hat{\sigma} > 0$, where $\sigma_D(A(x_k))$ denotes the smallest singular value of $A(x_k)$.

(AS2) A_k has full row rank for all iterations with $\sigma_D(A_k) \geq \tilde{\sigma} > 0$.

(AS3) $Z_k \in \mathbb{R}^{N \times D}$ has full column rank D for all iterates x_k with $\sigma_D(Z_k) \geq \check{\sigma} > 0$ and remains bounded.

(AS4) The sequence $\{f(x_k)\}$ is bounded below. The sequences $\{\nabla f(x_k)\}$, $\{c(x_k)\}$, $\{A(x_k)\}$, and $\{B_k\}$ are bounded.

(AS5) The functions $\nabla f(\cdot)$, $c(\cdot)$, and $A(\cdot)$ are Lipschitz continuous on an open convex set $\mathcal{X}$ containing all iterates.

(AS6) The gradients $\nabla f(x)$, $\nabla_x \mathcal{L}(x, \lambda)$, the gradient-vector product $\nabla f(x)^T d$ and the products $A(x)v$, $w^T A(x)$ can be evaluated exactly.

(AS7) For fixed x_k, the approximation Z_k can be improved in a finite number of steps such that an exact null space representation $Z(x_k)$ is obtained.

Assumption (AS1) is needed to prove the feasibility of all limit points and to derive upper bounds for the normal steps in Sec. 7.5. A similar assumption is made in [88, Sec. 3.3] to prove first-order global convergence. In the paper [24], the upper bound for the normal step is derived using also an assumption similar to (AS1). Furthermore, the analysis in [24] explicitly studies the rank deficiency of the constraint Jacobian $A(x_k)$ and its influence on the overall algorithm. That is, the iterates could converge either to a feasible point or to a limit point failing the linear independence constraint qualification. Therefore, an assumption similar to (AS1) is not made for this part of [24]. However, the present paper focuses mainly on the convergence of a trust-region algorithm with inexact Jacobian information. For that reason, we decided not to explore the possibility that $A(x_k)$ is rank deficient since this would complicate the analysis considerably. The convergence to a limit point not satisfying the linear independence constraint qualification may be the subject of future research.

In (AS7), we assume that we can improve the approximation Z_k such that it represents an exact null space $Z(x_k)$ of $A(x_k)$ after a finite number of improvement steps. This is possible, for example, for the TR1 approach by performing M rank one updates without changing the current iterate x_k since the TR1 update procedure yields the exact Jacobian $A(x_k)$ for fixed x_k after at most M

updates. This can be verified in the following way: Starting with an approximation $\tilde{A}_0 = A_k$ one performs M TR1 updates of the form

$$\tilde{A}_{i+1} \equiv \tilde{A}_i + \frac{(y_i - \tilde{A}_i v_i)(\tau_i^T - w_i^T \tilde{A}_i)}{(\tau_i^T - w_i^T \tilde{A}_i)v_i}$$

with $y_i \equiv A(x_k)v_i$ and $\tau_i \equiv w_i^T A(x_k)$ for arbitrary linearly independent vectors v_i and w_i chosen such that $(\tau_i^T - w_i^T \tilde{A}_i)v_i \neq 0$ holds. Due to the heredity of the rank one update, one obtains after M updates that

$$w_i^T \tilde{A}_M = w_i^T A(x_k) \qquad \text{for all} \qquad i = 0, \dots, M-1.$$

The proof of this identity is similar to the proof of a related result for the SR1 update and can be found in [151]. Since the w_i, $0 \leq i < M$, are M linearly independent vectors, it follows that $\tilde{A}_M = A(x_k)$. Using an equivalent update procedure for a factorized null space representation, one obtains an exact null space representation $Z_k = Z(x_k)$ after at most M updates [79]. If one freezes the Jacobian and null space information as proposed above, one can evaluate new exact Jacobian information if the restrictions on the inexactness are not valid any more. This approach ensures that assumption (AS7) holds. Hence, one can use the approximation $Z_k = Z_{k-1}$ and improve the approximation of the null space if required.

All other assumptions are either standard assumptions required also for the global convergence analysis in other papers, i.e., (AS4) and (AS5), or motivated by the applications that we had in mind when designing the algorithm, i.e., (AS2), (AS3), and (AS6).

7.3 A Jacobian-free Trust-Region Method

To apply a composite step trust-region method as proposed by Byrd and Omojokun, we first compute a normal step n that lies well inside the trust-region radius and that attempts to satisfy the linear constraints in (7.2). Subsequently, we take a tangential step t toward optimality. Putting both steps together, we obtain the total step $d = n + t$.

7.3.1 The Normal Subproblem

For the current iterate x_k, we compute a normal step n_k that best satisfies the linearized constraints by solving the *normal subproblem*

$$\begin{aligned} \min_{n \in \mathbb{R}^N} \quad & \|c(x_k) + A(x_k)n\|^2 \\ \text{subject to } & \|n\| \leq \tilde{\Delta}_k \end{aligned} \tag{7.5}$$

with $\tilde{\Delta}_k = \kappa \Delta_k$ and $\kappa \in (0, 1)$. This optimization problem may have infinitely many solutions. The exact Cauchy step for (7.5) is given by

$$n_k^C = -\alpha_k^C A(x_k)^T c(x_k) \tag{7.6}$$

where α_k^C is the optimal solution of the problem

$$\begin{aligned} &\min_{\alpha \geq 0} \|c(x_k) - \alpha A(x_k) A(x_k)^T c(x_k)\| \\ &\text{subject to } \|\alpha A(x_k)^T c(x_k)\| \leq \tilde{\Delta}_k. \end{aligned} \tag{7.7}$$

Hence, due to our assumption (AS6) that we can evaluate $A(x_k)v$ and $A(x_k)^T w$ for given v and w exactly, we are able to compute the exact Cauchy step. Nevertheless, employing only the exact Cauchy step may yield very slow convergence [117]. To accelerate the optimization process, one could use in addition also the exact Newton step. This global minimizer of the unconstrained version of (7.5) is given by

$$n^N(x_k) = -A(x_k)^+ c(x_k) = -A(x_k)^T (A(x_k) A(x_k)^T)^{-1} c(x_k).$$

However, we do not want to compute the vector $(A(x_k)A(x_k)^T)^{-1}c(x_k)$ exactly. Alternatively, if one assumes that an approximation $(A_k A_k^T)^{-1} c(x_k)$ can be evaluated at low computational cost, for example, by maintaining a factorized approximation of $A(x_k)$ as described in [79], then one could use the approximation

$$n_k^N = -A(x_k)^T (A_k A_k^T)^{-1} c(x_k)$$

of the exact Newton step. In combination with the exact Cauchy step, then one may compute the inexact dogleg step of Powell by setting

$$n_k^D = \eta n_k^N + (1 - \eta) n_k^C$$

with $\eta = 1$ if $\|n_k^N\| \leq \tilde{\Delta}_k$. Otherwise $\eta \in [0, 1]$ would be adjusted such that the length of n_k^D is equal to $\tilde{\Delta}_k$.

For obtaining convergence, one has to analyze the reduction in the linearized constraints caused by the normal step. For that purpose, we define the *normal predicted reduction* for a vector n as

$$\text{npred}_k(n) = \|c(x_k)\| - \|c(x_k) + A(x_k)n\| \tag{7.8}$$

and require that the normal step n_k computed in the kth iteration satisfies the following condition:

Normal Cauchy Decrease Condition. *An approximate solution n_k of the normal subproblem* (7.5) *must satisfy*

$$\text{npred}_k(n_k) \geq \gamma_n \text{npred}_k(n_k^C), \tag{7.9}$$

for some constant $\gamma_n > 0$.

To guarantee that a sufficient normal Cauchy decrease is achieved, one may either use the exact Cauchy step itself as normal step. Then (7.9) is obviously fulfilled with $\gamma_n = 1$. If one uses the inexact dogleg step, one can ensure that (7.9) holds by maximizing $\text{npred}_k(.)$ over the dogleg path. For our convergence analysis, the normal steps n_k have to fulfill the *range space condition*

$$\exists\, v_k \in \mathbb{R}^M \quad \text{such that} \quad n_k = A^T(x_k) v_k, \quad \text{i.e.,} \quad n_k \perp \ker(A(x_k)), \tag{7.10}$$

holds for all iterations $k \in \mathbb{N}$. Note that the normal steps n_k^D, n_k^C, and a linear combination of n_k^D and n_k^C are of the form $A^T(x_k)v_k$ such that they fulfill (7.10).

Since $\alpha = 0$ is feasible for the optimization problem (7.7), it follows from (7.9) that

$$\text{npred}_k(n_k) \geq 0 \tag{7.11}$$

holds. One can improve the bound on the normal predicted reduction as shown for example in [36, Lemma 15.4.17] and [24, Lemma 2]. The main ingredients of the proofs are the normal Cauchy decrease condition and the property $\|A(x_k)u_k^C\| > 0$ for $u_k^C \equiv -A(x_k)^T c(x_k) \neq 0$ due to the full rank of $A(x_k)$, i.e., assumption (AS1), and $u_k^C \perp \ker(A(x_k))$. Since the inexactness of the Jacobian does not influence the derivation of the result, we skip the proof of the following lemma. It can be proved exactly along the lines of Lemma 2 in [24].

Lemma 7.3.1. Suppose that assumption (AS1) holds. Let n_k be an approximate solution of the normal subproblem (7.5) such that (7.9) holds. Then

$$\|c(x_k)\|\text{npred}_k(n_k) \geq \frac{\gamma_n}{2}\|A(x_k)^T c(x_k)\| \min\left\{\tilde{\Delta}_k, \frac{\|A(x_k)^T c(x_k)\|}{\|A(x_k)\|^2}\right\}. \tag{7.12}$$

7.3.2 The Tangential Subproblem

Given a current iterate x_k, we compute the tangential step towards optimality. Usually, one tries to maintain linearized feasibility, i.e., the exact tangential step $t(x_k) = Z(x_k)p_k$ should be in the exact null space of the constraints. Since we have only an approximation Z_k of the exact null space $Z(x_k)$ available, we will have to safeguard the computation of the tangential step $t_k = Z_k p_k$ by limiting the amount of inexactness as will be explained later.

However, first we concentrate on computing an approximate solution of the inexact *tangential subproblem*

$$\begin{gathered} \min_{p \in \mathbb{R}^{N-M}} (\nabla f(x_k) + B_k n_k)^T Z_k p + \frac{1}{2} p^T Z_k^T B_k Z_k p \\ \|Z_k p\| \leq \hat{\Delta}_k \end{gathered} \tag{7.13}$$

with $\hat{\Delta}_k = (1-\kappa)\Delta_k$.

The steepest descent direction in the null space basis variables for this optimization problem at $p = 0$ is given by

$$p_k^C = -Z_k^T(\nabla f(x_k) + B_k n_k), \tag{7.14}$$

see, e.g., [24, 88]. For judging the improvement provided by the tangential step, we define the *tangential predicted reduction* produced by a tangential step $t = Z_k p$ as change in the objective function of the tangential subproblem. Hence, we have

$$\text{tpred}_k(t) = -(\nabla f(x_k) + B_k n_k)^T t - \frac{1}{2} t^T B_k t.$$

To ensure global convergence of our trust-region algorithm, we will impose the following condition on the tangential step:

Tangential Cauchy Decrease Condition. *An approximate solution t_k of the tangential subproblem (7.13) must satisfy*

$$\operatorname{tpred}_k(t_k) \geq \gamma_t \operatorname{tpred}_k(\theta_k^C Z_k p_k^C), \tag{7.15}$$

for some constant $\gamma_t > 0$, where θ_k^C solves the problem

$$\begin{aligned} &\min_{\theta \geq 0} \left[-\operatorname{tpred}_k(\theta Z_k p_k^C) \right] \\ &\text{subject to } \; \|\theta Z_k p_k^C\| \leq \hat{\Delta}_k. \end{aligned} \tag{7.16}$$

Since $\theta = 0$ is feasible for the optimization problem (7.16), it follows that

$$\operatorname{tpred}_k(t_k) \geq 0. \tag{7.17}$$

For deriving a sharper bound on the tangential predicted reduction that is needed for the convergence analysis, we cite the following result [24, Lemma 1]:

Lemma 7.3.2. Consider the one-dimensional problem

$$\begin{aligned} &\min_{z \geq 0} \psi(z) \equiv \frac{1}{2} a z^2 - bz \\ &\text{subject to } z \leq y \end{aligned}$$

where $b \geq 0$ and $y > 0$. Then the optimal value ψ_* satisfies

$$\psi_* \leq -\frac{b}{2} \min\left\{ y, \frac{b}{|a|} \right\} \text{ if } a \neq 0 \qquad \text{and} \qquad \psi_* \leq -by \text{ if } a = 0.$$

The derivation of a tighter lower bound for the tangential predicted reduction is based also on the representation of the null space of the constraint Jacobian. In the corresponding proofs of [36, Lemma 15.4.2] and [24, Lemma 3], the steepest descent direction is computed with an exact null space representation. The same holds true for the corresponding estimate in [88, Section 3.1.2]. We do not require that an exact null space representation is available but use only the inexact tangential subproblem (7.13). Therefore, we state the full proof of the following result, where we use ideas applied to prove Lemma 3 in [24].

Lemma 7.3.3. Suppose that assumptions (AS3) and (AS4) hold. Let t_k be an approximate solution of the tangential subproblem (7.13) that satisfies (7.15). Then

$$\operatorname{tpred}_k(t_k) \geq \hat{\gamma} \|p_k^C\| \min\left\{ \hat{\Delta}_k, \|p_k^C\| \right\} \tag{7.18}$$

for a constant $\hat{\gamma} > 0$.

Proof: Inequality (7.18) clearly holds if $p_k^C = 0$. Hence, we now assume that $p_k^C \neq 0$. Then, problem (7.16) is equivalent to

$$\begin{aligned} &\min_{\theta \geq 0} -\frac{1}{2}(p_k^C)^T Z_k^T B_k Z_k p_k^C \theta^2 - \|p_k^C\|^2 \theta \\ &\text{subject to } \theta \leq \frac{\hat{\Delta}_k}{\|Z_k p_k^C\|}. \end{aligned} \tag{7.19}$$

First assume that $(p_k^C)^T Z_k^T B_k Z_k p_k^C \neq 0$. Lemma 7.3.2 applied to problem (7.19) yields

$$-\text{tpred}_k(\theta_k^C Z_k p_k^C) \leq -\frac{1}{2}\|p_k^C\|^2 \min\left\{\frac{\hat{\Delta}_k}{\|Z_k p_k^C\|}, \frac{\|p_k^C\|^2}{|(p_k^C)^T Z_k^T B_k Z_k p_k^C|}\right\}.$$

Combining this inequality with (7.15) and using norm inequalities, we obtain

$$\text{tpred}_k(t_k) \geq \frac{\gamma_t}{2}\|p_k^C\| \min\left\{\frac{\hat{\Delta}_k}{\|Z_k^T Z_k\|^{1/2}}, \frac{\|p_k^C\|}{\|Z_k^T B_k Z_k\|}\right\}.$$

Since we assume that the approximations $\{Z_k\}$ remain bounded, we have that $\{Z_k^T Z_k\}$ are bounded. In addition, $\{B_k\}$ is bounded, which yields that $Z_k^T B_k Z_k$ is bounded. Hence, we can deduce from the last inequality that there exists a positive constant $\hat{\gamma}$ such that (7.18) holds.

We do not assume that B_k has full rank. Therefore, it may happen that $(p_k^C)^T Z_k^T B_k Z_k p_k^C = 0$ even if $p_k^C \neq 0$. Then the solution of (7.19) is given by $\theta_k^C = \frac{\hat{\Delta}_k}{\|Z_k p_k^C\|}$. It follows that

$$-\text{tpred}_k(\theta_k^C Z_k p_k^C) \leq -\|p_k^C\|^2 \frac{\hat{\Delta}_k}{\|Z_k p_k^C\|} \leq -\|p_k^C\| \frac{\hat{\Delta}_k}{\|Z_k^T Z_k\|^{1/2}}.$$

Since $\{Z_k\}$ remains bounded, this inequality proves the assertion. □

To accelerate the convergence, one may use not the steepest descent direction given by (7.14) but an approximation of the Newton step. For this purpose, we may apply the Steihaug CG algorithm, see, e.g., [24, 117], as long as (7.15) is fulfilled for the tangential step t_k.

The matrix Z_k only approximates the null space $Z(x_k)$ of the exact Jacobian $A(x_k)$. Hence, one has for the combined step $d_k = n_k + t_k$ that the identity $A(x_k)d_k = A(x_k)n_k$ is not necessarily valid. Therefore, we obtain for the predicted reduction (7.4) of the function m_k the equation

$$\begin{aligned} \text{pred}_k(d_k) = &-\nabla f(x_k)^T(n_k + t_k) - \frac{1}{2}(n_k + t_k)^T B_k (n_k + t_k) \\ &+ \mu_k(\|c(x_k)\| - \|c(x_k) + A(x_k)(n_k + t_k)\|) \\ = &\ \text{tpred}(t_k) + \mu_k \text{npred}(n_k) + \chi_k + \text{err}_k(d_k, \mu_k), \end{aligned}$$

where

$$\begin{aligned} \chi_k &= -\nabla f(x_k)^T n_k - \frac{1}{2} n_k^T B_k n_k \\ \text{err}_k(d_k, \mu_k) &= \mu_k(\|c(x_k) + A(x_k)n_k\| - \|c(x_k) + A(x_k)d_k\|). \end{aligned} \tag{7.20}$$

As can be seen, $\mathrm{err}_k(d_k, \mu_k)$ is a measure for the error in Z_k, i.e., in the approximation of $Z(x_k)$. Since the usual identity for the predicted reduction is not valid, we define an inexact predicted reduction

$$\mathrm{ipred}_k(d_k) = \mathrm{tpred}(t_k) + \mu_k \mathrm{npred}(n_k) + \chi_k \tag{7.21}$$

by omitting the error term. We will use this inexact measure for our trust-region algorithm. However, to ensure well-posedness and convergence for the considered class of trust-region methods, we need a bound on the error term $\mathrm{err}_k(d_k, \mu_k)$. Obviously, one can derive that

$$\begin{aligned} |\mathrm{err}_k(d_k, \mu_k)| &= \mu_k \big| \|c(x_k) + A(x_k)n_k\| - \|c(x_k) + A(x_k)d_k\| \big| \\ &\le \mu_k \|A(x_k)t_k\| \le \mu_k \, \nu \Delta_k^2. \end{aligned}$$

Hence, one may use a criterion like

$$\|A(x_k)t_k\| \le \nu \Delta_k^2 \tag{7.22}$$

for a constant $\nu > 0$ to bound the inexactness that is due to the tangential step. This inequality can be easily verified by evaluating one Jacobian-vector product. Similar requirements on the inexactness can be found in [88, Section 4.1.4] in the context of the convergence analysis of inexact trust-region methods for PDE-constrained optimization problems and in [36, Section 10.4] for trust-region methods in the unconstrained case. However, using (7.22) it may happen that $\mathrm{pred}_k(d_k)$ may become negative if $\mathrm{err}_k(d_k, \mu_k)$ is large relative to $\mathrm{ipred}_k(d_k)$. For this reason, we will use the direct criterion

$$-\mathrm{err}_k(d_k, \mu_k) < \left(1 - \eta - \frac{1-\eta}{2}\right) \mathrm{ipred}_k(d_k) \tag{7.23}$$

for a constant $\eta \in (0, 1)$. This inequality can be used to control the error in the inexact predicted reduction and therefore allows to ensure well-posedness of the algorithm. Note that one only has to bound a negative $\mathrm{err}_k(d_k, \mu_k)$ since a positive error even lead to a larger $\mathrm{pred}_k(d_k)$. If (7.23) holds, one has

$$\begin{aligned} \mathrm{pred}_k(d_k) &= \mathrm{ipred}_k(d_k) + \mathrm{err}_k(d_k, \mu_k) \\ &> \mathrm{ipred}_k(d_k) - \left(1 - \eta - \frac{1-\eta}{2}\right) \mathrm{ipred}_k(d_k) \\ &> \left(\eta + \frac{1-\eta}{2}\right) \mathrm{ipred}_k(d_k) \ge 0 \end{aligned} \tag{7.24}$$

if $\mathrm{ipred}_k(d_k) \ge 0$. Once more, (7.23) can be easily verified by evaluating two Jacobian-vector products.

7.3.3 The Trust-Region Algorithm

After specifying the computation of the normal and tangential step, we can now state a detailed description of our algorithm for solving the NLP (7.1):

Algorithm I:

```
Start: Set initial values $x_0$, $\lambda_0$, $\mu_{-1} > 0$, $A_0$, $Z_0$, $\Delta_0$, $\rho \in (0,1)$, $\eta \in (0,1)$,
       $\omega \in (0, \frac{1}{2})$, and $\nu > 0$
for $k = 0, 1, \dots$
  1. Compute a normal step $n_k$ such that (7.9) and (7.10) hold.
  2. Compute a tangential step $t_k$ such that (7.15) holds.
     Compute the total step $d_k = n_k + t_k$.
  3. Compute the smallest value $\tilde{\mu}_k$ such that
     $\text{ipred}_k(d_k) = \text{tpred}(t_k) + \tilde{\mu}_k \text{npred}(n_k) + \chi_k \geq \rho\, \tilde{\mu}_k\, \text{npred}_k(n_k).$   (7.25)
     If $\tilde{\mu}_k \leq \mu_{k-1}$, set $\mu_k = \mu_{k-1}$, otherwise set $\mu_k = \max\{\tilde{\mu}_k, 1.5\mu_{k-1}\}$.
  4. If (7.23) does not hold, update $A_k$ and $Z_k$ and go to step 1.
  5. If
        $\text{ared}_k(d_k) < \eta\, \text{ipred}_k(d_k)$
     decrease $\Delta_k$ by a constant factor and go to 1.
  6. Set $x_{k+1} = x_k + d_k$ and choose a $\Delta_{k+1}$ such that $\Delta_{k+1} \geq \Delta_k$
  7. Compute new $A_{k+1}$, $Z_{k+1}$, and Lagrange multipliers $\lambda_{k+1}$ using
        $\lambda_{k+1} = -(A_{k+1}A_{k+1}^T)^{-1} A(x_{k+1}) \nabla f(x_{k+1})$   (7.26)
     such that $\|Z_{k+1}^T A(x_{k+1})^T \lambda_{k+1}\| \leq \omega \|Z_{k+1}^T \nabla f(x_{k+1})\|$.
  8. If $Z_{k+1}^T \nabla f(x_{k+1}) = 0$ and $c(x_{k+1}) = 0$ go to 7 to improve $Z_{k+1}$,
     else increase $k$ by 1 and go to 1.
```

Algorithm I represents a Byrd-Omojokun trust-region algorithm that takes the inexactness of the Jacobian and its null space representation into account. To clarify this point we will discuss now each step of Algorithm I and compare it to a standard Byrd-Omojokun approach. The computation of a normal direction in Step 1 is identical to a standard approach where the normal Cauchy decrease condition and the range space condition have to be fulfilled. Note that the inexactness of the Jacobian may enter into the normal direction due to the choice of the normal step. The tangential direction computed in Step 2 has to fulfill the tangential Cauchy decrease condition, i.e., a standard requirement for a Byrd-Omojokun algorithm.

In Step 3, χ_k can be of any sign. Furthermore, we have that $\text{npred}_k(n_k)$ and $\text{tpred}_k(t_k)$ are nonnegative due to (7.11) and (7.17). Hence, if $\text{npred}_k(n_k) > 0$ holds it follows that $\text{ipred}_k(d_k) \geq \rho\, \mu_k\, \text{npred}_k(n_k)$ is valid when

$$\mu_k \geq \frac{-\chi_k}{(1-\rho)\text{npred}_k(n_k)}.$$

This lower bound is a sufficient condition, but not necessary, as condition (7.25) may hold also for smaller values of μ_k. If $\text{npred}_k(n_k) = 0$ one can conclude from Lemma 7.3.1 that $c(x_k) = 0$ due to assumption (AS1). Therefore, $n_k = 0$ solves the normal subproblem (7.5). The solution of (7.5) must be unique because of the range space condition (7.10). It follows for $\text{npred}_k(n_k) = 0$ that $n_k = 0$, $\chi_k = 0$, and that (7.25) is satisfied for any value of μ_k.

The additional test on (7.23) in Step 4 ensures that the inexactness of the Jacobian and its null space representation does not harm the tangential direction too much. Due to assumption (AS7), we need only a finite number of improvement steps for fixed x_k to obtain an exact $Z_k = Z(x_k)$ such that (7.23) is fulfilled.

The Steps 5 and 6 are standard update procedures of a trust-region algorithm. One only has to remember that $\text{ipred}(d_k)$ is not equal to the predicted reduction $\text{pred}(d_k)$ due to the inexactness allowed here. We will see later that the algorithm converges despite this inexactness.

In Step 7, we compute an approximation Z_{k+1} of the exact null space such that the inexactness is limited to a certain amount in the direction λ_{k+1}. Such an approximation can be found due to assumption (AS7). Subsequently, we test whether the approximation Z_{k+1} is good enough. A stationary point of the NLP (7.1) would satisfy the equations

$$Z(x_{k+1})^T \nabla f(x_{k+1}) = 0 \qquad c(x_{k+1}) = 0$$

due to the first-order optimality condition. However, we do not have an exact null space representation $Z(x_{k+1})$. Therefore, in Step 8 we check whether x_{k+1} is a stationary point of the inexact problem, i.e., whether the equations

$$Z_{k+1}^T \nabla f(x_{k+1}) = 0 \qquad c(x_{k+1}) = 0$$

hold. If this is the case but x_{k+1} is not a KKT point of the NLP (7.1), we have that $Z(x_{k+1})^T \nabla f(x_{k+1}) \neq 0$. Hence, our approximation Z_{k+1} of the null space $Z(x_{k+1})$ must be improved to obtain well-posedness. Due to assumption (AS7), we need only a finite number of improvement steps for fixed x_k to obtain $Z_{k+1}^T \nabla f(x_{k+1}) \neq 0$. Hence, it follows that there can be only an infinite cycling between Step 7 and 8 if x_{k+1} is an KKT point of the NLP (7.1).

7.4 Well-posedness of Algorithm I

An important property of a trust-region algorithm is the well-posedness. Here, one has to show that the trust-region radius cannot shrink to zero if an iterate x_k is not a stationary point of the NLP (7.1). For this purpose, we analyze the relation of the actual and predicted reduction. We will employ ideas used in the proof of Lemma 4 in [24]. In addition, we must take the inexactness of the Jacobian and its null space representation into account. That is, we have to ensure that the error term $\text{err}_k(d_k, \mu_k)$ does not dominate the model. In Step 4 of Algorithm I, we require that (7.23) holds. Employing this inequality, we can prove the following result that is related to Lemma 4 in [24].

Lemma 7.4.1. Assume that the assumptions (AS4), (AS5), and (AS7) hold on the open convex set $\mathcal{X}$ containing all iterates. Then there exists a positive constant ζ such that for any iterate x_k and any step $d_k = n_k + t_k$ generated by Algorithm I with $[x_k, x_k + d_k] \subset \mathcal{X}$ and $\text{ared}_k(d_k) \leq \eta\, \text{ipred}_k(d_k)$, it follows that

$$0 \leq \eta \text{ipred}_k(d_k) - \text{ared}_k(d_k) \leq \zeta(1 + \mu_k)\Delta_k^2 \tag{7.27}$$

Proof: Since $A(\cdot)$ is Lipschitz continuous, there exists a constant $\zeta' > 0$ such that

$$\begin{aligned}\big|\, \|c(x_k + d_k)\| - \|c(x_k) + A(x_k)d_k\,\| \big| &\le \|c(x_k + d_k) - c(x_k) - A(x_k)d_k\,\| \\ &\le \sup_{\tilde{x}\,\in[x_k, x_k+d_k]} \|A(\tilde{x}) - A(x_k)\|\,\|d_k\| \\ &\le \zeta' \Delta_k^2\,.\end{aligned}$$

As in Lemma 4 of [24], the last inequality, the definitions (7.3) and (7.4), the Lipschitz continuity of ∇f, and the boundedness of B yield

$$\begin{aligned}|\,\mathrm{pred}_k(d_k) - \mathrm{ared}_k(d_k)\,| \le \Big| & f(x_k + d_k) - f(x_k) - \nabla f(x_k)^T d_k - \frac{1}{2} d_k^T B_k d_k \\ & + \mu_k(\|c(x_k + d_k)\| - \|c(x_k) + A(x_k)d_k\|)\Big| \\ \le\ & \zeta(1 + \mu_k)\Delta_k^2\end{aligned}$$

for some positive constant ζ. Combining the last two inequalities with the bound (7.23) on the error and therefore (7.24), we obtain

$$\begin{aligned}0 < \eta\mathrm{ipred}_k(d_k) - \mathrm{ared}_k(d_k) &\le \left(\eta + \frac{1-\eta}{2}\right) \mathrm{ipred}_k(d_k) - \mathrm{ared}_k(d_k) \\ &\le \mathrm{pred}_k(d_k) - \mathrm{ared}_k(d_k) \le \zeta(1 + \mu_k)\Delta_k^2\,.\end{aligned}$$

□

Next, we have to prove that Algorithm I can not generate an infinite cycling between Steps 1 and 5. To show that an acceptable step is determined with a finite number of reductions of Δ_k even if the Jacobian and its null space representation are inexact, we employ two properties: First, it follows for $c(x_k) = 0$ from (7.8), (7.11), and assumption (AS1) that $\mathrm{npred}_k(n_{k,i}) = 0$, $n_{k,i} = 0$, and therefore $p_k^C = -Z_k^T \nabla f(x_k) \neq 0$ due to Steps 7 and 8 of Algorithm I. Second, it follows for $c(x_k) \neq 0$ from assumption (AS1) that $A(x_k)^T c(x_k) \neq 0$. Using these properties of our inexact setting, the proof of the following result is similar to the one of Proposition 1 in [24] taking the modified estimate (7.27) into account. Therefore, we only will state the parts of the proof that differ from the proof of [24, Proposition 1].

Proposition 7.4.2. *Let assumption (AS1) hold. Suppose that x_k is not a stationary point of the NLP (7.1). Then there exists a Δ_k^0 such that*

$$\mathrm{ared}_k(d_k) \ge \eta\,\mathrm{ipred}_k(d_k)$$

for any $\Delta \in (0, \Delta_k^0)$.

Proof: Let the iterate x_k be fixed. To prove the assertion, we assume that there is a subsequence indexed by i of trust radii $\Delta_{k,i}$ such that $\Delta_{k,i}$ converges to zero and that $\mathrm{ared}_k(d_{k,i}) < \eta\,\mathrm{ipred}_k(d_{k,i})$ for the corresponding steps $d_{k,i} = n_{k,i} + t_{k,i}$ and the penalty parameter $\mu_{k,i}$ for all i.

For $\eta \in (0,1)$, the inequality $\mathrm{ared}_k(d_{k,i}) < \eta\, \mathrm{ipred}_k(d_{k,i})$ yields

$$\left(\eta + \frac{1-\eta}{2}\right) \mathrm{ipred}_k(d_{k,i}) - \mathrm{ared}_k(d_{k,i}) > \frac{1-\eta}{2} \mathrm{ipred}_k(d_{k,i}) \geq 0.$$

Then, it follows from Lemma 7.4.1 in combination with $\Delta_{k,i} \to 0$ that

$$\mathrm{ipred}_k(d_{k,i}) = (1 + \mu_{k,i}) o(\|d_{k,i}\|). \tag{7.28}$$

This equation can be used exactly along the lines of the proof of Proposition 1 in [24] to produce a contradiction proving the assertion of the proposition. Therefore, we skip the rest of the proof here. □

Hence, to obtain well-posedness of Algorithm I even in the presence of inexact first-order information one has to ensure that the approximation Z_k of the exact null space representation is not too bad. In our approach the effects of the inexactness are bounded for the tangential step by the additional condition (7.23). This suffices to show the bound (7.27). Additionally, the test on the quality of Z_k in Step 7 and 8 of Algorithm 1 ensures that there can not be an infinite cycling between Step 1 and 5, i.e., an acceptable step can be computed with a finite number of iterations. Note that only the inexactness of the null space approximation Z_k but not the inexactness of the constraint Jacobian approximation A_k has to be controlled to achieve well-posedness.

7.5 Convergence Analysis

Comparing the following theorem with its counterpart in [24], one finds that the result presented here is less general. This is due to the fact that we concentrate the analysis in this paper mainly on the influence of inexact Jacobian information. That is, we do not want to study the performance of Algorithm I in the presence of dependent constraint gradients as in [24] but focus on the effects caused by inexact constraint Jacobian information. Therefore, we assume in contrast to [24] that the exact constraint Jacobian $A(x_k)$ has full row rank, i.e., assumption (AS1) holds. Otherwise, the iterates generated by Algorithm I may converge to a limit point failing the linear independence constraint qualification. For the derivation of the next result, it is not required to handle the inexactness of A_k and Z_k directly. The inexact first-order information are taken into account by Lemma 7.4.1 which is used in the proof of the following assertion. Due to the estimate in Lemma 7.4.1 that differs from [24, Proposition 1], we state the parts of the proof that differ from [24, Lemma 7], but skip the rather long remaining parts of the proof.

Theorem 7.5.1 (Feasibility of all limit points). *Assume that (AS1) – (AS7) hold. Then, we have*

$$\lim_{k \to \infty} c(x_k) = 0.$$

Proof: We define the function

$$\Psi(x) = \|A(x)^T c(x)\| \,.$$

Using the assumptions (AS4) and (AS5), we obtain that there are constants $\epsilon_1, \epsilon_2, \epsilon_3 > 0$ such that

$$\begin{aligned} |\Psi(x) - \Psi(x_l)| &= \|A(x)^T c(x) - A(x)^T c(x_l) + A(x)^T c(x_l) - A(x_l)^T c(x_l)\| \\ &\leq \epsilon_1 \|x - x_l\| \; + \epsilon_2 \|x - x_l\| \leq \epsilon_3 \|x - x_l\| \end{aligned} \tag{7.29}$$

holds for any two points x and x_l in $\mathcal{X}$. Now, consider an arbitrary iterate x_l with $\Psi_l \equiv \Psi(x_l) \neq 0$. First, we will show that Algorithm I accepts all sufficiently small steps that are in a neighborhood of the iterate x_l. If the current step d_k is acceptable nothing has to be shown, otherwise one has $\text{ared}_k(d_k) < \eta\, \text{ipred}_k(d_k)$ and Lemma 7.4.1 can be applied. We define the ball

$$\mathcal{B}_l = \{x \,:\, \|x - x_l\| < \Psi_l/(2\epsilon_3)\} \,.$$

Applying (7.29) yields for any $x \in \mathcal{B}_l$ that $\Psi(x) \geq \Psi_l/2 > 0$. It follows that there exists a constant $\bar{c}$ with $\|c(x)\| \geq \bar{c} > 0$. Using Lemma 7.3.1 and the assumption (AS4) yields the existence of a constant $\epsilon_4 > 0$ such that for any iterate $x_k \in \mathcal{B}_l$ the inequality

$$\text{ipred}_k(d_k) \geq \rho\, \mu_k \text{npred}_k(n_k) \geq \mu_k \epsilon_4 \Psi_l \min\{\tilde{\Delta}_k, \Psi_l\} \tag{7.30}$$

holds. For sufficient small Δ_k it follows that

$$\text{ipred}_k(d_k) \geq \mu_k \epsilon_4 \Psi_l \tilde{\Delta}_k. \tag{7.31}$$

Employing this inequality together with the estimate derived in the proof of Lemma 7.4.1, we have

$$0 \leq \frac{\left(\eta + \frac{1-\eta}{2}\right) \text{ipred}_k(d_k) - \text{ared}_k(d_k)}{\text{ipred}_k(d_k)} \leq \frac{\zeta(1+\mu_k)\Delta_k^2}{\mu_k \epsilon_4 \Psi_l \tilde{\Delta}_k}$$

and therefore

$$\text{ared} \geq \eta \text{ipred}_k(d_k) + \left(\frac{1-\eta}{2} - \frac{\zeta(1+\mu_k)\Delta_k}{\mu_k \epsilon_4 \Psi_l}\right) \text{ipred}_k(d_k) \,.$$

For Δ_k sufficiently small, the second term on the right-hand side is non-negative. Hence, for all $x_k \in \mathcal{B}_l$ and all such Δ_k, we have

$$\text{ared}_k(d_k) \geq \eta\, \text{ipred}_k(d_k) \tag{7.32}$$

which results in acceptance of d_k due to Step 4 of Algorithm I. The remainder of this proof follows exactly along the lines of [24, Lemma 7]. $\square$

To prove the first-order optimality of all limit points, we need that the normal step can be bounded by the normal predicted reduction and that the penalty factor μ_k eventually becomes constant. For that purpose, we present the next two lemmas. For the proofs of the following two results, it is not necessary to handle the inexactness of A_k and Z_k directly. Nevertheless, we state the two proofs since the derivation differs slightly from the proofs contained in [24] due to the different setting.

Lemma 7.5.2 (Upper bound on normal step). Let assumptions (AS1) and (AS4) be fulfilled. Then there exists a positive constant γ such that

$$\|n_k\| \leq \gamma \,\text{npred}_k(n_k) \tag{7.33}$$

Proof: Using Lemma 7.3.1, we have for the normal step

$$\|c(x_k)\|\text{npred}_k(n_k) \geq \frac{\gamma_n}{2}\|A(x_k)^T c(x_k)\| \min\left\{\tilde{\Delta}_k, \frac{\|A(x_k)^T c(x_k)\|}{\|A(x_k)\|^2}\right\}.$$

If $c(x_k) = 0$ then inequality (7.33) is trivially satisfied. Therefore assume that $c(x_k) \neq 0$. Since $A(x_k)$ is supposed to remain bounded there exists a constant $\bar{\sigma} = \sup_k \|A(x_k)\|$. Together with assumption (AS1), this gives

$$\text{npred}_k(n_k) \geq \frac{\gamma_n}{2}\hat{\sigma} \min\left\{\tilde{\Delta}_k, \frac{\hat{\sigma}\|c(x_k)\|}{\bar{\sigma}^2}\right\}. \tag{7.34}$$

Now, we have to consider two cases. First, let $\|c(x_k)\| \geq \hat{\sigma}\tilde{\Delta}_k/2$. Using $\bar{\sigma} \geq \hat{\sigma}$ and the trust-region constraint, we obtain

$$\text{npred}_k(n_k) \geq \frac{\gamma_n}{2}\hat{\sigma} \min\left\{1, \frac{\hat{\sigma}^2}{2\bar{\sigma}^2}\right\}\tilde{\Delta}_k \geq \frac{\gamma_n \hat{\sigma}^3}{4\bar{\sigma}^2}\|n_k\|.$$

This yields (7.33). Second, assume that $\|c(x_k)\| < \hat{\sigma}\tilde{\Delta}_k/2$. To derive the upper bound (7.33) in this case, we employ (7.10) and Lemma 7.3.1. Hence, there exists a vector $v_k \in \mathbb{R}^M$ such that

$$\begin{aligned}\|c(x_k)\|^2 &\geq \|c(x_k) + A(x_k)n_k\|^2 \\ &= \|c(x_k)\|^2 + 2c(x_k)^T A(x_k)n_k + \|A(x_k)A(x_k)^T v_k\|^2.\end{aligned}$$

One obtains

$$\|A(x_k)A(x_k)^T v_k\|^2 \leq -2c(x_k)^T A(x_k)n_k.$$

Using the Cauchy-Schwarz inequality, it follows that

$$\|A(x_k)A(x_k)^T v_k\| \leq 2\|c(x_k)\|.$$

Due to assumption (AS1), this inequality implies that

$$\|n_k\| = \|A(x_k)^T v_k\| \leq \frac{2}{\hat{\sigma}}\|c(x_k)\|.$$

Employing the last inequality and (7.34), we have

$$\begin{aligned}\text{npred}_k(n_k) &\geq \frac{\gamma_n}{2}\hat{\sigma} \min\left\{\tilde{\Delta}_k, \frac{\hat{\sigma}\|c(x_k)\|}{\bar{\sigma}^2}\right\} \geq \frac{\gamma_n}{2}\hat{\sigma} \min\left\{\frac{2}{\hat{\sigma}}, \frac{\hat{\sigma}}{\bar{\sigma}^2}\right\}\|c(x_k)\| \\ &\geq \gamma_n \min\left\{\frac{2}{\hat{\sigma}}, \frac{\hat{\sigma}}{\bar{\sigma}^2}\right\}\|n_k\|,\end{aligned}$$

which concludes the proof. □

Lemma 7.5.3 (Bound on hpred and constant μ_k for $k \geq k_1$). Suppose that the assumptions (AS1) and (AS4) are satisfied. Then the sequence of penalty parameters $\{\mu_k\}$ is bounded. Furthermore, there exist an index k_1 and positive constants $\bar{\mu}$ and ξ, such that $\mu_k = \bar{\mu}$ holds for all $k \geq k_1$ and

$$\text{ipred}_k(d_k) \geq \xi \text{tpred}_k(t_k). \tag{7.35}$$

Proof: The sequences $\{\nabla f(x_k)\}$ and $\{B_k\}$ are bounded due to assumption (AS4). It follows from (7.8) that $\text{npred}_k(n_k) \leq \|c(x_k)\|$. Furthermore, $\|c(x_k)\|$ is bounded due to assumption (AS4). Hence, $\text{npred}_k(n_k)$ is bounded. Using (7.33), we obtain that there exists a constant ξ_1 such that

$$-\nabla f(x_k)^T n_k - \frac{1}{2} n_k^T B_k n_k \geq -\xi_1 \text{npred}_k(n_k).$$

Then, we can deduce from the definition (7.21) of $\text{ipred}_k(d_k)$ that

$$\text{ipred}_k(d_k) \geq \text{tpred}(t_k) + \mu_k \text{npred}(n_k) - \xi_1 \text{npred}_k(n_k). \tag{7.36}$$

Employing that $\text{npred}_k(n_k) \geq 0$ and $\text{tpred}_k(t_k) \geq 0$, we can derive from this inequality that (7.25) in Step 3 of Algorithm I holds for $\mu_k \geq \xi_1/(1-\rho)$. Hence, if μ_k becomes larger than $\xi_1/(1-\rho)$, it will never be increased. Taking into account that Algorithm I increases μ_k by a constant factor this yields that after some iterate, e.g. k_1, μ_k will remain unchanged at some value $\bar{\mu}$.

Then, it follows from (7.21) and (7.25) that

$$\text{ipred}_k(d_k) \geq \text{tpred}(t_k) - \xi_1 \text{npred}_k(n_k) \geq \text{tpred}(t_k) - \frac{\xi_1}{\rho\, \mu_k} \text{ipred}_k(d_k).$$

Hence, (7.35) is satisfied with $1/\xi = 1 + \xi_1/(\rho\, \bar{\mu})$. □

Now, the field is prepared to prove the main result of this paper, namely the convergence to a first-order critical point from an arbitrary starting point. That is, we prove global convergence for our trust-region method given by Algorithm I. For this purpose, we have to take the inexactness of Z_k explicitly into account: The bound on the error in the null space representation provided by Step 7 of Algorithm I is directly required to prove the following result. Therefore, we state the full proof, where we also employ ideas from [24, Lemma 12].

Theorem 7.5.4 (All limit points are first-order optimal). *Suppose that (AS1) – (AS7) hold. Then, it follows that*

$$\lim_{k\to\infty} \nabla_x \mathcal{L}(x_k, \lambda_k) = \lim_{k\to\infty} (\nabla f(x_k) + A(x_k)^T \lambda_k) = 0$$

where the multipliers λ_k are defined as in (7.26).

Proof: Step 7 of Algorithm I ensures that

$$\|Z_k^T A(x_k)^T \lambda_k\| \leq \omega \|Z_k^T \nabla f(x_k)\|$$

for $\omega \in (0, \frac{1}{2})$ and $k > 0$. This yields for $q_k = \nabla_x \mathcal{L}(x_k, \lambda_k)$

$$\begin{aligned}\|Z_k^T q_k\| &= \|Z_k^T(\nabla f(x_k) + A(x_k)^T \lambda_k)\| \geq \|Z_k^T \nabla f(x_k)\| - \|Z_k^T A(x_k)^T \lambda_k\| \\ &\geq (1-\omega)\|Z_k^T \nabla f(x_k)\| \geq \frac{1-\omega}{\omega}\|Z_k^T A(x_k)^T \lambda_k\|.\end{aligned}$$

Setting $\varrho = \omega/(1-\omega) \in (0,1)$, we obtain

$$\varrho \|Z_k^T q_k\| \geq \|Z_k^T A(x_k)^T \lambda_k\|.$$

It follows that there exists a constant γ_1' such that (AS3) and (AS4) yield

$$\begin{aligned}\|p_k^C\| &= \| - Z_k^T(\nabla f(x_k) + B_k n_k)\| \\ &= \| - Z_k^T \nabla f(x_k) - Z_k^T A(x_k)^T \lambda_k + Z_k^T A(x_k)^T \lambda_k - Z_k^T B_k n_k\| \\ &= \| - Z_k^T q_k + Z_k^T A(x_k)^T \lambda_k - Z_k^T B_k n_k\| \\ &\geq \breve{\sigma}\|q_k\| - \varrho\breve{\sigma}\|q_k\| - \gamma_1'\|n_k\| = (1-\varrho)\breve{\sigma}\|q_k\| - \gamma_1'\|n_k\|\end{aligned}$$

is valid.

To obtain a contradiction, suppose that $\lim_{k\to\infty} q_k = 0$ does not hold. Then, there exists a constant $\vartheta > 0$ such that $0 < \vartheta \leq \frac{1}{4}\limsup_{k\to\infty}\|q_k\|$. Lemma 7.5.1 ensures that $c(x_k) \to 0$. Together with Lemma 7.5.2 this yields $\|n_k\| \to 0$. Hence, there is an arbitrarily large l such that for the iterate x_l and all $k \geq l$, we have $\|q_l\| > 3\vartheta$ and $\gamma_1'\|n_k\| < (1-\varrho)\breve{\sigma}\vartheta$. Let γ_L be the Lipschitz constant for q_k. We define the ball $\mathcal{B}_l = \{x : \|x - x_l\| \leq \vartheta/\gamma_L\}$. Now, assume that the iterates x_k with $k > l$ do not leave $\mathcal{B}_l$. Then, it follows for all k that

$$\begin{aligned}\|p_k^C\| &\geq (1-\varrho)\breve{\sigma}(\|q_l\| - \|q_l - q_k\|) - \gamma_1'\|n_k\| \\ &\geq (1-\varrho)\breve{\sigma}(3\vartheta - \vartheta - \vartheta) = (1-\varrho)\breve{\sigma}\vartheta > 0.\end{aligned}$$

Employing Lemma 7.3.3 and Lemma 7.5.3 gives with $\gamma_2' = \xi\,\hat{\gamma}(1-\varrho)\breve{\sigma}$ that

$$\operatorname{ipred}_k(d_k) \geq \xi\operatorname{tpred}_k(t_k) \geq \gamma_2'\,\vartheta\min\{\hat{\Delta}_k, (1-\varrho)\breve{\sigma}\,\vartheta\}. \tag{7.37}$$

Now, we define the scaled merit function

$$\tilde{\phi}(x;\mu) \equiv \frac{1}{\mu}\phi(x;\mu) = \frac{1}{\mu}f(x) + \|c(x)\|$$

as proposed in [24]. Since the values $\{f(x_k)\}$ of the objective function are bounded below, we can add a constant to f such that $f(x_k) \geq 0$ holds at all iterates. Then, we can deduce from Step 4 of Algorithm I and the fact that μ_k is nondecreasing, see Lemma 7.5.3, that

$$\begin{aligned}\tilde{\phi}(x_k;\mu_k) - \frac{\eta\operatorname{ipred}_k(d_k)}{\mu_k} &\geq \tilde{\phi}(x_{k+1};\mu_k) \\ &= \tilde{\phi}(x_{k+1};\mu_{k+1}) + \left(\frac{1}{\mu_k} - \frac{1}{\mu_{k+1}}\right) f(x_{k+1}) \\ &\geq \tilde{\phi}(x_{k+1};\mu_{k+1}).\end{aligned}$$

That is, the sequence $\tilde{\phi}(x_k;\mu_k)$ is decreasing. Furthermore, the boundedness of $f(x_k)$ gives

$$\tilde{\phi}(x_k;\mu_k) = \frac{1}{\mu_k}f(x_k) + \|c(x_k)\| \geq K \tag{7.38}$$

for a constant $K \in \mathbb{R}$. The fact that $\{\tilde{\phi}(x_k;\mu_k)\}$ is bounded and Lemma 7.5.3 imply that $\operatorname{ipred}_k(d_k) \to 0$ since otherwise $\{\tilde{\phi}(x_k;\mu_k)\}$ would not be bounded.

Together with (7.37) this implies $\hat{\Delta}_k \to 0$. Taking l sufficiently large yields for any $k \geq l$ with $x_k \in \mathcal{B}_l$ that $\hat{\Delta}_k \leq \min\{1, (1-\varrho)\check{\sigma}\vartheta\}$ and therefore

$$\text{ipred}_k(d_k) \geq \gamma_2' \vartheta \hat{\Delta}_k. \tag{7.39}$$

If $x_k \in \mathcal{B}_l$, we employ the same argument as in the proof of Theorem 7.5.1 to show that an acceptable step is generated for sufficiently small Δ_k. Hence, if $x_k \in \mathcal{B}_l$ for all $k > l$ then Δ_k would eventually stop decreasing. However, this contradicts the fact that $\hat{\Delta}_k \to 0$. Thus the sequence $\{x_k\}$ must leave $\mathcal{B}_l$ for some $k > l$.

In that case, suppose that x_{k+1} is the first iterate after x_l that is not contained in $\mathcal{B}_l$. We deduce from (7.39) and $\hat{\Delta}_k = (1-\kappa)\Delta_k$ that

$$\begin{aligned} \phi(x_{k+1};\mu_{k+1}) &\leq \phi(x_l;\mu_l) - \eta \sum_{j=l}^{k} \text{ipred}(x_j,\mu_j) \\ &\leq \phi(x_l;\mu_l) - \gamma_2'\vartheta(1-\kappa)\sum_{j=l}^{k}\Delta_j \\ &\leq \phi(x_l;\mu_l) - \gamma_2'(1-\kappa)\vartheta^2/\gamma_L. \end{aligned} \tag{7.40}$$

One can derive the last inequality from the fact x_{k+1} has left the ball $\mathcal{B}_l$ with radius ϑ/γ_L. The sequence $\{\phi(x_k;\mu_k)\}$ is decreasing and bounded below due to (7.38). Hence, it converges. This is a contradiction to the fact that l can be chosen arbitrarily large in (7.40) and the fact that $\vartheta > 0$. Therefore, $q_k \to 0$. □

Once more, one only has to limit the error due to the inexact null space representation Z_k for the proof of global convergence. Therefore, an implementation of Algorithm I will have to handle the approximation of the null space representation carefully. One possibility is to employ the TR1 update of the Jacobian that also provides an approximation of the null space representation [79]. We will present corresponding numerical results in a forthcoming paper [151].

7.6 Conclusion

In this paper, we have proposed and analyzed for the first time a class of trust-region methods based only on inexact information on the constraint Jacobian and the null space representation without any assumption on the method to approximate these matrices. Using two conditions measuring the inexactness of the null space representation, we prove global first-order convergence for the presented algorithm under quite mild conditions. The two required conditions on the inexactness can be easily verified during the optimization process.

Due to the non-differentiable merit function and the weak assumptions on the inexactness, one may need to accelerate the convergence rate using additional safe-guard strategies for the inexactness possibly in combination with a second order correction or a watch-dog technique.

In addition to this subject, future work will also comprise the handling of inequality constraints. The introduction of slack variables in combination with interior point techniques would be one possibility. Alternatively, one may analyze projection methods to incorporate, for example, bound constraints.

Acknowledgments

The author is very thankful to Lorenz T. Biegler for numerous discussions on trust-region methods. Furthermore, the author is grateful to Jorge Nocedal and Hubert Schwetlick for their motivating comments on the subject of this paper. The valuable comments and hints of the anonymous referees are also gratefully acknowledged.

Bibliography

[1] F. Abergel and R. Temam. On some control problems in fluid mechanics. *Theoret. Comput. Fluid Dynamics*, 1:303–325, 1990.

[2] F. Abraham, M. Behr, and M. Heinkenschloss. The effect of stabilization in finite element methods for the optimal boundary control of the Oseen equations. *Finite Elements in Analysis and Design*, 41:229–251, 2004.

[3] A. Albertson, G. Chappell, H. Kierstead, A. Kündgen, and R. Ramamurthu. Coloring with no 2-colored P_4's. *Electron. J. Comb.*, 11(1):R26, 2004.

[4] A. Arora and L. Biegler. A trust region SQP algorithm for equality constrained parameter estimation with simple parameter bounds. *Comput. Optim. Appl.*, 28(1):51–86, 2004.

[5] G. Bärwolff and M. Hinze. Optimization of semiconductor melts. *ZAMM*, 86:423–437, 2006.

[6] R. Becker, D. Meidner, and B. Vexler. Efficient numerical solution of parabolic optimization problems by finite element methods. *Optim. Methods Softw.*, 2007. To appear.

[7] R. Becker and B. Vexler. Optimal control of the convection-diffusion equation using stabilized finite element methods. *Num. Math.*, 2007. To appear.

[8] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical report, Technical University of Denmark, 1996.

[9] M. Berggren. Numerical solution of a flow control problem: Vorticity reduction by dynamic boundary action. *SIAM J. Sci. Comput.*, 19(3):829–860, 1998.

[10] M. Berggren, R. Glowinski, and J.L. Lions. A computational approach to controllability issues for flow-related models. I: Pointwise control of the viscous Burgers equation. *Int. J. Comput. Fluid Dyn.*, 7(3):237–252, 1996.

[11] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Phil., 1996.

[12] J.T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. SIAM, Philadelphia, 2001.

[13] C. Bischof, A. Carle, P. Hovland, P. Khademi, and A. Mauer. ADIFOR 2.0 user's guide (Revision D). Technical Report CRPC-95516-S, Argonne National Laboratory, USA, 1998.

[14] H.G. Bock and K.-J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings of the 9th IFAC World Congress*, pages 243–247. Pergamon Press, 1984.

[15] I. Bongartz, A.R. Conn, N. Gould, and Ph. Toint. CUTE: Constrained and unconstrained testing environment. *ACM Trans. Math. Softw.*, 21:123–160, 1995.

[16] A.E. Bryson and Y. Ho. *Applied Optimal Control—Optimization, Estimation, and Control*. Hemisphere Publishing Corporation, New York, 1975.

[17] R. Bulirsch, E. Nerz, H.J. Pesch, and O. von Stryk. Combining direct and indirect methods in nonlinear optimal control: Range maximization of a hang glider. In R. Bulirsch, A. Miele, J. Stoer, and K.H. Well, editors, *Optimal Control, Calculus of Variations, Optimal Control Theory and Numerical Methods*, pages 273–288. Birkhäuser, 1993.

[18] C. Büskens. *Optimierungsmethoden und Sensitivitätsanalyse für optimale Steuerprozesse mit Steuer- und Zustandsbeschränkungen*. PhD thesis, Westfälische Wilhelms-Universität Münster, 1998.

[19] C. Büskens and H. Maurer. SQP-methods for solving optimal control problems with control and state constraints: Adjoint variables, sensitivity analysis and real-time control. *J. Comp. App. Math.*, 120:85–108, 2000.

[20] C. Büskens and H. Maurer. Sensitivity analysis and real-time optimization of parametric nonlinear programming problems. In M. Gröschel, S. Krumke, and J. Rambau, editors, *Online Optimization of Large Scale Systems: State of the Art*, pages 3–16. Springer, 2001.

[21] J.C. Butcher. *The numerical analysis of ordinary differential equations*. John Wiley, New York, 1987.

[22] R. Byrd. Robust trust region methods for constrained optimization, Houston, USA. Third SIAM Conference on Optimization, 1987.

[23] R. Byrd, F. Curtis, and J. Nocedal. Inexact SQP methods for equality constrained optimization. Technical report, Northwestern University, USA, 2006.

[24] R. Byrd, J. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Math. Program.*, 89A:149–185, 2000.

[25] J.-B. Caillau and J. Noailles. Continuous optimal control sensitivity analysis with AD. In Corliss et al. [38], pages 109–117.

[26] R. Carter. Numerical experience with a class of algorithms for nonlinear optimization using inexact function and gradient information. *SIAM J. Sci. Comput.*, 14:368–388, 1993.

[27] D. Casanova, R.S. Sharp, M. Final, B. Christianson, and P. Symonds. Application of automatic differentiation to race car performance optimisation. In Corliss et al. [38], pages 109–117.

[28] M.R. Celis, J.E. Dennis, and R.A. Tapia. A trust region strategy for nonlinear equality constrained optimization. In *Numerical optimization*, Proc. SIAM Conf., pages 71–82, 1985.

[29] I. Charpentier. Checkpointing schemes for adjoint codes: Application to the meteorological model meso-nh. *SIAM J. Sci. Comput.*, 22:2135–2151, 2001.

[30] G. Chavent. Identification of distributed parameter systems: About the output least square method, its implementation, and identifiability. Identification and system parameter estimation, Proc. 5th IFAC Symp., Darmstadt 1979, Vol. 1, 85-97, 1980.

[31] B. Christianson. Reverse accumulation and implicit functions. *Optim. Methods Softw.*, 9(4):307–322, 1998.

[32] T. Coleman and J. Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7:221–235, 1986.

[33] T. Coleman and J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program.*, 28:243–270, 1984.

[34] S.S. Collis and M. Heinkenschloss. Analysis of the streamline upwind/Petrov Galerkin method applied to the solution of optimal control problems. Technical Report CAAM TR02-01, CAAM, 2002.

[35] A. Conn, N. Gould, and Ph. Toint. Convergence of quasi-Newton matrices generated by the symmetric rank one update. *Math. Program.*, 50A(2):177–196, 1991.

[36] A. Conn, N. Gould, and Ph. Toint. *Trust-region methods.* SIAM, 2000.

[37] P. Constantin and C. Foias. *Navier-Stokes Equations.* The University of Chicago Press, 1988.

[38] G.F. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation: From Simulation to Optimization.* Springer, New York, 2001.

[39] CppAD. web-site: http://www.seanet.com/~bradbell/CppAD/.

[40] K. Deckelnick and M. Hinze. Error estimates in space and time for tracking-type control of the instationary Stokes system. *International Series on Numerical Mathematics*, 143:87–103, 2002.

[41] K. Deckelnick and M. Hinze. Semidiscretization and error estimates for distributed control of the instationary Navier-Stokes equations. *Numer. Math.*, 97:297–320, 2004.

[42] J. Dennis, M. El-Alem, and M. Maciel. A global convergence theory for general trust-region-based algorithms for equality constrained optimization. *SIAM J. Optim.*, 7(1):177–207, 1997.

[43] L. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Griewank and Corliss [71], pages 114–125.

[44] A.L. Dontchev, W. Hager, and V. Veliov. Second-order Runge-Kutta approximations in control constrained optimal control. *SIAM J. Numer. Anal.*, 38:202–226, 2000.

[45] P. Eberhard and C. Bischof. Automatic differentiation of numerical integration algorithms. *Math. Comput.*, 68(226):717–731, 1999.

[46] M. El-Alem. A global convergence theory for the Celis-Dennis-Tapia trust-region algorithm for constrained optimization. *SIAM J. Numer. Anal.*, 28(1):266–290, 1991.

[47] Y.G. Evtushenko. Automatic differentiation viewed from optimal control theory. In Griewank and Corliss [71], pages 25–30.

[48] Y.G. Evtushenko. Computation of exact gradients in distributed dynamic systems. *Optim. Methods Softw.*, 9(1-3):45–75, 1998.

[49] R. Fletcher, N. Gould, S. Leyffer, P. Toint, and A. Wächter. Global convergence of a trust-region SQP-filter algorithm for general nonlinear programming. *SIAM J. Optim.*, 13(3):635–659, 2003.

[50] R. Fletcher and S. Leyffer. Nonlinear programming without a penalty function. *Math. Program.*, 91A(2):239–269, 2002.

[51] R. Fletcher, S. Leyffer, and P. Toint. On the global convergence of a filter-SQP algorithm. *SIAM J. Optim.*, 13(1):44–59, 2002.

[52] N. Gauger. Aerodynamic shape optimization using the adjoint Euler equations. In T. Sonar and I. Thomas, editors, *Proceedings of the GAMM workshop*, pages 87–96. Logos Verlag, 2001.

[53] N. Gauger, A. Walther, C. Moldenhauer, and M. Widhalm. Automatic differentiation of an entire design chain with applications. Technical report, TU Dresden, 2006. To appear in Jahresbericht 2007 der Arbeitsgemeinschaft Strömungen mit Ablösung STAB.

[54] D. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In Berz et al. [11], pages 173–184.

[55] A. Gebremedhin, A. Pothen, A. Tarafdar, and A. Walther. Efficient computation of sparse Hessians: An experimental study using ADOL-C. Technical report, Old Dominion University, 2006. Submitted (INFORMS Journal on Computing, in revision).

[56] A.H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[57] U. Geitner, J. Utke, and A. Griewank. Automatic computation of sparse Jacobians by applying the method of Newsam and Ramsdell. In Berz et al. [11], pages 161–172.

[58] M. Gertz, P. Gill, and J. Muetherig. User's guide for SNADIOPT: A package adding automatic differentiation to SNOPT. Technical Report NA 01-01, Department of Mathematics, University of California, 2001.

[59] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. Math. Software*, 24:437–474, 1998.

[60] R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.

[61] M.B. Giles. On the iterative solution of adjoint equations. In Corliss et al. [38], pages 145–151.

[62] P. Gill, W. Murray, M. Saunders, and M. Wright. User's guide for NPSOL 5.0: A fortran package for nonlinear programming. Technical Report NA 98-2, Department of Mathematics, University of California, San Diego, 1998.

[63] M. Gockenbach, D. Reynolds, and W. Symes. Efficient and automatic implementation of the adjoint state method. *ACM Trans. Math. Software*, 28:22–44, 2002.

[64] F. Gomes, M. Maciel, and J. Martinez. Nonlinear programming algorithms using trust regions and augmented Lagrangians with nonmonotone penalty parameters. *Math. Program.*, 84A(1):161–200, 1999.

[65] R. Griesse. Parametric sensitivity analysis in optimal control of a reaction-diffusion system. II: Practical methods and examples. *Optim. Methods Softw.*, 19(2):217–242, 2004.

[66] R. Griesse and A. Walther. Parametric sensitivities for optimal control problems using automatic differentiation. *Opt. Cont. Appl. Meth.*, 24:297–314, 2003.

[67] R. Griesse and A. Walther. Evaluating gradients in optimal control — continuous adjoints versus automatic differentiation. *J. Opt. Theo. Appl.*, 122(1):63–86, 2004.

[68] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–107. Kluwer Academic Publishers, 1989.

[69] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optim. Methods Softw.*, 1:35–54, 1992.

[70] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.

[71] A. Griewank and G. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications*, Philadelphia, 1991. SIAM.

[72] A. Griewank and C. Faure. Piggyback differentiation and optimization. In L. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, editors, *Large-scale PDE-constrained optimization*, LNCSE 30, pages 148–164. Springer, 2003.

[73] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22:131–167, 1996.

[74] A. Griewank and C. Mitev. Verifying Jacobian sparsity. In Corliss et al. [38], pages 271–279.

[75] A. Griewank, S. Schlenkrich, and A. Walther. A quasi-Newton method with optimal R-order without independence assumption. Technical Report 340, MATHEON, 2006. Submitted (Optim. Methods Softw., in revision).

[76] A. Griewank and A. Walther. Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software*, 26:19–45, 2000.

[77] A. Griewank and A. Walther. On constrained optimization by adjoint-based quasi-Newton methods. *Optim. Methods Softw.*, 17:869–889, 2002.

[78] A. Griewank and A. Walther. On the efficient generation of Taylor expansions for DAE solutions by automatic differentiation. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Proceedings of PARA'04*, LNCS 3732, pages 1089 – 1098, 2006.

[79] A. Griewank, A. Walther, and M. Korzec. Maintaining factorized KKT systems subject to rank-one updates of Hessians and Jacobians. *Optim. Methods Softw.*, 22(2):279–295, 2007.

[80] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Berz et al. [11], pages 95–106.

[81] M.D. Gunzburger. *Perspectives in flow control and optimization.* SIAM, 2003.

[82] M.D. Gunzburger and S. Manservisi. Analysis and approximation of the velocity tracking problem for Navier-Stokes flows with distributed control. *SIAM J. Numer. Anal.*, 37:1481–1512, 2000.

[83] E. Haber. Quasi-Newton methods for large scale electromagnetic inverse problems. *Inverse Problems*, 21:305–317, 2004.

[84] W. Hager. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numer. Math.*, 87:247–282, 2000.

[85] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Tech. Rep. 300, INRIA, 2004.

[86] S.B. Hazra, V. Schulz, J. Brezillon, and N. Gauger. Aerodynamic shape optimization using simultaneous pseudo-timestepping. *J. Comput. Phys.*, 204(1):46–64, 2005.

[87] P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21:1356–1371, 2005.

[88] M. Heinkenschloss and L.N. Vicente. Analysis of inexact trust-region SQP algorithms. *SIAM J. Optim.*, 12(2):283–302, 2001.

[89] V. Heuveline and A. Walther. Online checkpointing for adjoint computation in PDEs: Application to goal-oriented adaptivity and flow control. In W.E. Nagel, W.V. Walter, and W. Lehner, editors, *Proceeding of Euro-Par 2006*, LNCS 4128, pages 689 – 699. Springer, 2006.

[90] J.G. Heywood and R. Rannacher. Finite-element approximation of the nonstationary Navier-Stokes problem, I-IV. *SIAM J. Numer. Anal.*, 19:275-311, 23:750-777, 25:489-512, 27:353-384, 1982-1990.

[91] P. Hiltmann. *Numerische Lösung von Mehrpunkt-Randwertproblemen und Aufgaben der optimalen Steuerung mit Steuerfunktionen über endlichdimensionalen Räumen.* PhD thesis, TU München, Mathematisches Institut, 1989.

[92] M. Hinze. Optimal and instantaneous control of the instationary Navier-Stokes equations. Habilitationsschrift, Fachbereich Mathematik, TU Berlin, 1999.

[93] M. Hinze and K. Kunisch. Second order methods for optimal control of time-dependent fluid flow. *SIAM J. Control Optim.*, 40:925–946, 2001.

[94] M. Hinze and T. Slawig. Adjoint gradients compared to gradients from algorithmic differentiation in instantaneous control of the Navier-Stokes equations. *Optim. Methods Softw.*, 18(3):299–315, 2003.

[95] M. Hinze and J. Sternberg. A-revolve: An adaptive memory and run-time-reduced procedure for calculating adjoints; with an application to the instationary Navier-Stokes system. *Optim. Methods Softw.*, 20(6):645–663, 2005.

[96] M. Hinze, A. Walther, and J. Sternberg. An optimal memory-reduced procedure for calculating adjoints of the instationary Navier-Stokes equations. *Opt. Cont. Appl. Meth.*, 27(1):19–40, 2005.

[97] S. Hossain and T. Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optim. Methods Softw.*, 10:33–48, 1998.

[98] A. Iollo, G. Kuruvila, and S. Ta'asan. Pseudotime method for shape design of Euler flows. *AIAA J.*, 34(9):1807–1813, 1996.

[99] D.H. Jacobson and D.Q. Mayne. *Differential Dynamic Programming.* American Elsevier Publishing Company, 1970.

[100] H. Jäger and E. Sachs. Global convergence of inexact reduced SQP methods. *Optim. Methods Softw.*, 7:83–110, 1997.

[101] L. Jiang, L.T. Biegler, and G. Fox. Optimization of pressure swing adsorption systems for air separation. *AIChE Journal*, 49:1140–1157, 2003.

[102] H.-J. Kaltenbacher, W. Jürgens, and A. Spille. Numerische Simulation, Beeinflussung und Eigenmoden-Analyse einer abgelösten Strömung mit Querkomponente. Ergebnisberichte SFB 557 TP A6, 2001.

[103] M. Knauer and C. Büskens. Real-time trajectory planning of the industrial robot IRB 6400. In *PAMM*, volume 3, pages 515–516, 2003.

[104] A. Kowarz and A. Walther. Optimal checkpointing for time-stepping procedures in ADOL-C. In V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, editors, *Proceedings of ICCS 2006*, LNCS 3994, pages 541–549. Springer, 2006.

[105] K. Kubota. A fortran 77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. *Optim. Methods Softw.*, 10:319–335, 1998.

[106] M. Lalee, J. Nocedal, and T. Plantenga. On the implementation of an algorithm for large-scale equality constrained optimization. *SIAM J. Optim.*, 8(3):682–706, 1998.

[107] F. Leibfritz and E. Sachs. Inexact SQP interior point methods and large scale optimal control problems. *SIAM J. Cont. Opt.*, 38(1):272–293, 1999.

[108] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT (Nordisk Tidskrift for Informationsbehandling)*, 16:146 – 160, 1976.

[109] S. Nadarajah and A. Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. AIAA-2000-0667, 2000.

[110] U. Naumann. Cheaper Jacobians by simulated annealing. *SIAM J. Optim.*, 13:660–674, 2002.

[111] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Program.*, 99A:399–421, 2004.

[112] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Prog.*, 2006. In Press. Appeared on Springer's "Online First" Web portal.

[113] U. Naumann and P. Gottschling. Angel - automatic differentiation nested graph elimination library. http://angellib.sourceforge.net/.

[114] U. Naumann and J. Riehme. A differentiation-enabled fortran 95 compiler. *ACM Trans. Math. Softw.*, 31(4):458–474, 2005.

[115] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of SCAM 2004*, pages 55–64. IEEE Computer Society, 2004.

[116] N.S. Nedialkov and J.D. Pryce. Solving differential-algebraic equations by Taylor series. I: Computing Taylor coefficients. *BIT*, 45(3):561–591, 2005.

[117] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 1999.

[118] H. Oberle and W. Grimm. Bndsco — a program for the numerical solution of optimal control problems. Technical Report 515, Institute for Flight System Dynamics, Oberpfaffenhofen, German Aerospace Research Establishment DLR, 1989.

[119] E. Omojokun. *Trust region algorithms for optimization with nonlinear equality and inequality constraints*. PhD thesis, Dept. of Computer Science, University of Colorado, 1989.

[120] D.B. Özyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. Sci. Comput.*, 26(5):1725–1743, 2005.

[121] H.J. Pesch. Offline and online computation of optimal trajectories in the aerospace field. In A. Miele and A. Salvetti, editors, *Applied Mathematics in Aerospace Science and Engineering*, volume 44 of *Mathematical Concepts and Methods in Science and Engineering*, pages 165–220, 1994.

[122] M. Powell and Ph. Toint. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal.*, 16:1060–1074, 1979.

[123] M. Powell and Y. Yuan. A trust region algorithm for equality constrained optimization. *Math. Program.*, 49A(2):189–211, 1990.

[124] A. Quarteroni, R. Sacco, and F. Saleri. *Numercial Mathematics.* Springer, New York, 2000.

[125] K. Röbenack. Automatic differentiation and nonlinear controller design by exact linearization. *Future Generation Computer Systems*, 21(8):1372–1379, 2005.

[126] K. Röbenack and K. J. Reinschke. The computation of Lie derivatives and Lie brackets based on automatic differentiation. *ZAMM*, 84(2):114–123, 2004.

[127] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in ODYSSEE. In Berz et al. [11], pages 558–568.

[128] S. Schlenkrich, A. Griewank, and A. Walther. Local convergence analysis of TR1 updates for solving nonlinear equations. Technical Report Preprint 337, MATHEON, 2006. Submitted (Math. Program.).

[129] S. Schlenkrich and A. Walther. Global convergence of quasi-Newton methods based on adjoint tangent rank-1 updates. Technical report, TU Dresden, 2006. Submitted (Appl. Num. Math., in revision).

[130] S. Schlenkrich, A. Walther, N. Gauger, and R. Heinrich. Differentiating fixed point iterations with ADOL-C: Gradient calculation for fluid dynamics. Technical report, TU Dresden, 2006. To appear in proceedings of HPSC 2006.

[131] S. Schlenkrich, A. Walther, and A. Griewank. Application of AD-based quasi-Newton-methods to stiff ODEs. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, LNCSE 50, pages 89–98. Springer, 2005.

[132] R. Serban and A.C. Hindmarsh. CVODES: An ODE solver with sensitivity analysis capabilities. UCRL-JP-20039, LLNL, 2003.

[133] G. Shultz, R. Schnabel, and R. Byrd. A family of trust region based algorithms for unconstrained minimization with strong global convergence properties. *SIAM J. Numer. Anal.*, 22(1):47–67, 1985.

[134] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, University of Illinois, 1980.

[135] J. Sternberg. Adaptive Umkehrschemata für Schrittfolgen mit nicht-uniformen Kosten, 2002. Diplomarbeit.

[136] K. Strehmel and R. Weiner. *Numerik gewöhnlicher Differentialgleichungen.* Teubner Studienbücher: Mathematik. Teubner, Stuttgart, 1995.

[137] O. Talagrand and P. Courtier. Variational assimilation of meteorological observations with the adjoint vorticity equation – Part I. Theory. *Q. J. R. Meteorol. Soc.*, 113:1311–1328, 1987.

[138] R. Temam. *Navier-Stokes Equations.* North-Holland, 1979.

[139] F. Tröltzsch. *Optimale Steuerung partieller Differentialgleichungen.* Vieweg Verlag, 2005.

[140] T. Tun and T.S. Dillon. Extensions of the differential dynamic programming method to include systems with state dependent control constraints and state variable inequality constraints. *Journal of Applied Science and Engineering*, 3A:171–192, 1978.

[141] M. Ulbrich and S. Ulbrich. Non-monotone trust region methods for nonlinear equality constrained optimization without a penalty function. *Math. Program.*, 95B(1):103–135, 2003.

[142] J. Utke. OpenAD: Algorithm implementation user guide. Tech. Mem. ANL/MCS–TM–274, MCS, Arg. Nat. Lab., Argonne, Ill., 2004. online at ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf.

[143] R. Vanderbei and D. Shanno. An interior-point algorithm for nonconvex nonlinear programming. *Comput. Optim. Appl.*, 13:231–252, 1999.

[144] S. Volkwein and M. Weiser. Affine invariant convergence analysis for inexact augmented Lagrangian-SQP methods. *SIAM J. Cont. Opt.*, 41(3):875–899, 2002.

[145] O. von Stryk. User's guide for dircol (version 2.1): A direct collocation method for the numerical solution of optimal control problems. Technical report, TU Darmstadt, 2000.

[146] A. Wächter and L. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[147] A. Walther. *Program Reversal Schedules for Single- and Multi-processor Machines.* Ph.D. thesis, Institute of Scientific Computing, 1999.

[148] A. Walther. A first-order convergence analysis of trust-region methods with inexact Jacobians. Technical Report MATH-WR-01-2005, TU Dresden, 2005. Submitted (SIAM J. Optim., in revision).

[149] A. Walther. Automatic differentiation of explicit Runge-Kutta methods for optimal control. *J. Comp. Opt. Appl.*, 36:83 – 108, 2007.

[150] A. Walther. Computing sparse Hessians with automatic differentiation. *ACM Trans. Math. Softw.*, 2007. To appear.

[151] A. Walther and L. Biegler. A trust-region algorithm for nonlinear programming problems with dense constraint Jacobians. Technical Report MATH-WR-01-2007, TU Dresden, 2007. Submitted (J. Comp. Opt. Appl.).

[152] A. Walther and A. Griewank. Applying the checkpointing routine treeverse to discretizations of Burgers' equation. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing*, LNCSE 8, pages 13–24. Springer, 1999.

[153] A. Walther and A. Griewank. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In M. Feistauer et al., editor, *Numerical mathematics and advanced applications*, pages 834–843. Springer, 2004. Proceedings of ENUMATH 2003.

[154] A. Walther, A. Kowarz, and A. Griewank. *Documentation of ADOL-C: version 1.10.1*, 2005. Updated version of [73].

[155] R. Waltz and J. Nocedal. KNITRO user's manual. Technical Report OTC 05/2003, Optimization Technology Center, Northwestern University, Evanston, IL 60208, USA, 2003.

[156] G. Wanner. *Integration gewöhnlicher Differentialgleichnugen, Lie Reihen,Runge-Kutta-Methoden*, volume XI, 831/831a of *B.I-Hochschulskripten.* Bib. Institut, 1969.

[157] R.E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7:463–464, 1964.

[158] P.J. Werbos. Application of advances in nonlinear sensitivity analysis. In R.F. Drenick and F. Kozin, editors, *System Modeling and Optimization: Proceedings of the 19th IFIP Conference New York*, volume 38 of *Lecture Notes in Control Inform. Sci.*, pages 762–770. Springer Verlag, New York, 1982.

[159] GJ.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, UK, 1965.

Erklärung

gemäß §6 Abs. 2, Ziffer 2 der Habilitationsordnung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Bestandteil der Habilitationsschrift sind die Veröffentlichungen

[1] R. Griesse und A. Walther. Evaluating gradients in optimal control - continuous adjoints versus automatic differentiation. *Journal of Optimization Theory and Applications*, 122(1), pp. 63 – 86 (2004).

[2] M. Hinze, A. Walther und J. Sternberg. An optimal memory-reduced procedure for calculating adjoints of the instationary Navier-Stokes equations. *Optimal Control Applications and Methods*, 27(1), pp. 19 – 40 (2006).

[3] A. Walther. Automatic differentiation of explicit Runge-Kutta methods for optimal control. *Journal of Computational Optimization and Applications*, 36:83 – 108 (2007).

[4] A. Walther. Computing Sparse Hessians with Automatic Differentiation. Preprint MATH-WR-03-2005, TU Dresden (2005). Erscheint in *ACM Transaction on Mathematical Software.*

[5] A. Walther. A first-order convergence analysis of trust-region methods with inexact Jacobians. Modifizierte Version von Preprint MATH-WR-01-2005, TU Dresden (2005). Eingereicht (SIAM Journal of Optimization, in Revision).

[6] A. Walther und A. Griewank. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In M. Feistauer, V. Dolejší, P. Knobloch, and K. Najzar, eds., *Numerical Mathematics and Advanced Applications*, ENUMATH 2003, Prag, pp. 834 – 843, Springer (2004).

Meine Beiträge in den Veröffentlichungen [1], [2] und [6] ordnen sich wie folgt ein:

- In [1], d.h. Kapitel 4 der Habilitationsschrift:
  Die im Kapitel 4, Abschnitt 4.3, enthaltene Darstellung und Untersuchung der diskreten Gradienten-Berechnung habe ich hergeleitet. Das gleiche trifft auf den Vergleich der diskreten Gradienten mit den analytischen

Gradienten im Kapitel 4, Abschnitt 4.5, zu. Die dafür erforderlichen numerischen Ergebnisse, d.h. auch die Umsetzung als Computerprogramm wie im Kapitel 4, Abschnitt 4.4 beschrieben, waren das Ergebnis einer engen Zusammenarbeit von Roland Griesse und mir. Eine noch genauere genauer Einordnung der eigenen Leistungen auch im Vergleich zu bereits vorhandenen Literatur befindet sich auf den Seiten 27 und 28 der vorgelegten Habilitationsschrift.

- In [2], d.h. Kapitel 6 der Habilitationsschrift:
  An der Herleitung der im Kapitel 6, Abschnitt 6.3, dargestellen neuen Komplexitätstheorie für das binomiale Checkpointing habe ich durch wesentliche Beiträge mitgewirkt. Die im Kapitel 6, Abschnitt 6.4 enthaltenen numerischen Ergbnisse entstanden in enger Zusammenarbeit von Michael Hinze und mir. Die Analyse und Interpretation der numerischen Ergebnisse (Kapitel 6, Abschnitt 6.4) stammt von mir. Eine noch genauere genauer Einordnung der eigenen Leistungen auch im Vergleich zu bereits vorhandenen Literatur befindet sich auf den Seiten 29 und 30 der vorgelegten Habilitationsschrift.

- In [6], d.h. Kapitel 2 der Habilitationsschrift:
  An der Herleitung der im Kapitel 2, Abschnitt 2.3, dargestellten theoretischen Ergebnissen zum binomialen Checkpointing war ich massgeblich mitbeteiligt. Die im Kapitel 2, Abschnitt 2.4, dargelegten theoretischen Ergebnisse zum Vergleich des äquidistanten und des binomialen Checkpointing habe ich alleine erarbeitet. Eine noch genauere genauer Einordnung der eigenen Leistungen auch im Vergleich zu bereits vorhandenen Literatur befindet sich auf den Seiten 17 und 18 der vorgelegten Habilitationsschrift.

Dresden, den 10. Mai 2007