

# Evaluating Gradients in Optimal Control — Continuous Adjoint versus Automatic Differentiation

Roland Griesse\*      Andrea Walther†

December 14, 2001

## 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.

## 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 minimum) principle to derive necessary conditions for the optimizer. These conditions take the

---

\*Chair of Mathematics in Engineering, University of Bayreuth, Germany, roland.griesse@uni-bayreuth.de

†Institute of Scientific Computing, Technical University Dresden, Germany, awalther@math.tu-dresden.de

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 [15]. Examples and issues on implementation are covered e.g. in Bulirsch et. al. [3] and Hiltmann [13]. A boundary value problem solver commonly used is described in Oberle [14].

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 [17] and Betts [1]. 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 [5] as well as Bock and Plitt [2] 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 chose to study only 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. Nevertheless, we intend to consider more substantial examples involving state constraints in an upcoming paper.

This paper is organized as follows: In Section 2, we introduce the optimal control problem under consideration. Section 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. We present two examples in Section 5 to illustrate the theoretical

finding, and give a conclusion in Section 6.

## 2 The Continuous Problem

We consider the following optimal control problem:

$$\text{Minimize } J(y) = \varphi(y(t_f)) \quad (1)$$

$$\text{s.t. } \dot{y}(t) = f(y(t), u(t), t) \quad t \in [0, t_f] \quad (2)$$

$$y(0) = y_0 \quad (3)$$

$$a \leq u(t) \leq b \quad a, b \in \mathbb{R}^m. \quad (4)$$

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  $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^n$ . We assume that complete initial information  $y_0 \in \mathbb{R}^n$  is given. The objective function  $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}$  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 (2)–(3) for a given control  $u$ , and define

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

Since gradients<sup>1</sup> 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  $s(t) \in \mathbb{R}^{n \times m}$  for a given control  $u$  by the following matrix ODE:

$$\dot{s}(t) = f_y(\psi(u)(t), u(t), t) s(t) + f_u(\psi(u)(t), u(t), t) \quad (5)$$

$$s(0) = 0. \quad (6)$$

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

$$D\tilde{J}(u)(t) = \nabla \varphi(\psi(u)(t)) s(t) \in \mathbb{R}^{1 \times m}.$$

---

<sup>1</sup>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  $\nabla$ .

The attribute *forward* points to the fact that, in order to find the gradient  $D\tilde{J}(u)$  at time  $t$ , the sensitivity equation has to be integrated in forward direction from 0 to  $t$ .

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

$$-\dot{\lambda}(t) = f_y(\psi(u)(t), u(t), t)^T \lambda(t) \quad (7)$$

$$\lambda(t_f) = \nabla \varphi(\psi(u)(t_f))^T. \quad (8)$$

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

$$(D\tilde{J}(u)(t))^T = f_u(\psi(u)(t), u(t), t)^T \lambda(t) \in \mathbb{R}^{m \times 1}. \quad (9)$$

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  $\bar{u}$ :

$$D\tilde{J}(u)\bar{u} = \int_0^{t_f} D\tilde{J}(u)(t)^T \bar{u}(t) dt. \quad (10)$$

Already now it becomes obvious that for the control problem under consideration, the adjoint approach is superior to the sensitivity version since the adjoint dimension  $n$  is smaller than the sensitivity dimension  $m \cdot n$ . Therefore, sensitivity-based methods will not be pursued here any further.

### 3 The Discretized Problem

In order to solve problem (1)–(4) numerically, some discretization has to be carried out. Therefore, let the time interval  $[0, t_f]$  be divided into  $N_t$  sub-intervals of equal lengths. The time grid consists of points

$$t^j = (j - 1) \cdot h \quad \text{for } j = 1, \dots, N_t + 1$$

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 (see e.g., [7]).

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

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

Similarly, we change all function names to bold-face print if they have to be adapted going from the continuous to the discretized problem.

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

$$\begin{aligned} \text{Minimize} \quad & \tilde{\mathbf{J}}(\mathbf{u}) = \varphi(\boldsymbol{\psi}^{N_t+1}(\mathbf{u})) \\ \text{s.t.} \quad & a \leq \mathbf{u}^j \leq b \end{aligned}$$

where  $\mathbf{y} = \boldsymbol{\psi}(\mathbf{u}) = (\boldsymbol{\psi}^1(\mathbf{u}), \dots, \boldsymbol{\psi}^{N_t+1}(\mathbf{u}))^T$  approximately satisfies the state ODE of the problem. In our tests, we took  $\boldsymbol{\psi}$  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  $\mathbf{y}$ :

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

for  $j = 1, \dots, N_t$ . Here,  $t^{j+\frac{1}{2}}$  denotes  $t^j + \frac{h}{2}$  and  $\mathbf{u}^{j+\frac{1}{2}}$  is either equal to  $\mathbf{u}^j$  (piecewise constant interpolation) or equal to  $\frac{\mathbf{u}^j + \mathbf{u}^{j+1}}{2}$  (piecewise linear interpolation). As mentioned in Section 1, we consider only the discretized control functions  $\mathbf{u}$  as optimization variables, which has the effect that in every evaluation of the objective  $\varphi(\boldsymbol{\psi}^{N_t+1}(\mathbf{u}))$ , 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

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

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

### 3.1 Finite Difference Evaluation of the Objective Gradient

As a first possibility, finite difference evaluation of  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  comes into mind. Denoting the  $i$ -th component of  $\mathbf{u}^j$  by  $\mathbf{u}_i^j$  and letting  $\mathbf{e}_i^j$  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

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}_i^j} \approx \frac{\tilde{\mathbf{J}}(\mathbf{u} + \epsilon \mathbf{e}_i^j) - \tilde{\mathbf{J}}(\mathbf{u})}{\epsilon}.$$

Indeed, with appropriately chosen perturbations  $\epsilon$ , the approximation is a reliable estimate of the gradient, and no additional coding is needed beyond the evaluation of  $\tilde{\mathbf{J}}(\mathbf{u})$ . However, every evaluation of  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  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 5.

### 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 (9) 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 (7)–(8), 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 (9)–(10) yields the discrete gradient  $\nabla \tilde{\mathbf{J}}(\mathbf{u}) = \left( \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^1}, \dots, \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^{N_t + 1}} \right)$ . However, as a detailed investigation in the sequel will reveal,

$$\left( \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^j} \right)^T \neq h f_u(\boldsymbol{\psi}^j(\mathbf{u}), \mathbf{u}^j, t^j)^T \boldsymbol{\lambda}^j, \quad (11)$$

i.e., the formula on the right-hand side of (11) 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 5.

### 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 (7)–(8), and assembling the gradient as suggested by (9) and (10).

The first part of this question has been addressed in Hager [12] 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.,  $\mathbf{y}^{j+1} = \mathbf{y}^j + hf(\mathbf{y}^j, \mathbf{u}^j, t^j)$ . In the spirit of adjoint information, the gradient  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  is evaluated from its last to its first components:

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

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

$$(\boldsymbol{\lambda}^{N_t+1})^T = \nabla \varphi(\boldsymbol{\psi}^{N_t+1}(\mathbf{u})) \tag{12}$$

$$(\boldsymbol{\lambda}^j)^T = \nabla \varphi(\boldsymbol{\psi}^{N_t+1}(\mathbf{u})) \frac{\partial \boldsymbol{\psi}^{N_t+1}(\mathbf{u})}{\partial \boldsymbol{\psi}^{N_t}(\mathbf{u})} \frac{\partial \boldsymbol{\psi}^{N_t}(\mathbf{u})}{\partial \boldsymbol{\psi}^{N_t-1}(\mathbf{u})} \cdots \frac{\partial \boldsymbol{\psi}^{j+1}(\mathbf{u})}{\partial \boldsymbol{\psi}^j(\mathbf{u})}, \tag{13}$$

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

Since for the explicit Euler scheme we have

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

and thus

$$\frac{\partial \boldsymbol{\psi}^{j+1}(\mathbf{u})}{\partial \boldsymbol{\psi}^j(\mathbf{u})} = I + hf_y(\boldsymbol{\psi}^j(\mathbf{u}), \mathbf{u}^j, t^j), \tag{14}$$

the  $\boldsymbol{\lambda}^j$  obey the backward recursion

$$\boldsymbol{\lambda}^j = [I + hf_y(\boldsymbol{\psi}^j(\mathbf{u}), \mathbf{u}^j, t^j)]^T \boldsymbol{\lambda}^{j+1}. \quad (15)$$

As an integration scheme applied to the adjoint ODE (7)–(8), (15) 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

$$\boldsymbol{\lambda}^j = [I - hf_y(\boldsymbol{\psi}^j(\mathbf{u}), \mathbf{u}^j, t^j)]^{-T} \boldsymbol{\lambda}^{j+1} \quad (\text{implicit Euler})$$

$$\boldsymbol{\lambda}^j = [I + hf_y(\boldsymbol{\psi}^{j+1}(\mathbf{u}), \mathbf{u}^{j+1}, t^{j+1})]^T \boldsymbol{\lambda}^{j+1} \quad (\text{explicit Euler}).$$

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

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^{N_t+1}} = 0 \quad (16)$$

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^j} = (\boldsymbol{\lambda}^{j+1})^T hf_u(\boldsymbol{\psi}^j(\mathbf{u}), \mathbf{u}^j, t^j), \quad j = 1, \dots, N_t. \quad (17)$$

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 (16)–(17) 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.

### 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 [10].

The key idea of Automatic Differentiation is the systematic application of the chain rule. For many applications, the underlying model is described by a non-linear vector function

$$F : \mathbb{R}^N \rightarrow \mathbb{R}^M, \quad x \mapsto F(x),$$

which is typically 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  $\tilde{\mathbf{J}}(\mathbf{u})$ , so that  $N = (N_t + 1)m$  and  $M = 1$ . Using the forward mode resembles a discrete sensitivity approach, cf. (5)–(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 (7)–(8). These parallels have already been hinted at in Griewank [11].

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  $\nabla F$  can be calculated at no more than five times  $O_F$  [10]. One *row* of  $\nabla F$ , 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  $O_F$ , see [10]. 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.

Analyzing our method presented in Subsection 3.3, one finds that computing the gradient  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  recursively from  $\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^{N_t+1}}$  down to  $\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^1}$  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 5.

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

The first is to generate the recursion scheme for the quantities  $\boldsymbol{\lambda}^j$  using AD. The resulting backward integration can be viewed as the appropriate adjoint scheme for the adjoint differential equation (7) with terminal condition (8). 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  $\boldsymbol{\lambda}^j$  in order to obtain the desired gradient.

An alternative approach makes use of AD to directly compute the gradient  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$ . Since we have a scalar-valued cost function  $\tilde{\mathbf{J}}$  with many input variables  $\mathbf{u}$ , the reverse mode of AD is preferable. It yields code that implicitly features the computation of  $\lambda^j$ -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 Section 5.

## 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, whereas information regarding NPSOL is presented in the Section 4.2.

### 4.1 Discretizing the continuous problem

Our software has the following main features: The code has to provide the objective value  $\tilde{\mathbf{J}}(\mathbf{u})$  to NPSOL for a given vector of optimization variables  $\mathbf{u}$  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  $\mathbf{y}^{N_t+1}$ . 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  $\mathbf{u}$  are required not only at the time grid points  $t^j$ , but also at  $t^{j+\frac{1}{2}}$  (see Section 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  $\varphi(\mathbf{y}^{N_t+1})$ . Exchanging only the problem-specific parts, a variety of problems can easily coded.

In addition to the computation of the state trajectory, derivative evaluations have to be performed. According to the previous section, 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  $\mathbf{y}$  to intermediate grid points  $t^{j+\frac{1}{2}}$  is required which can be done either in a constant fashion, i.e., ( $\mathbf{y}^{j+\frac{1}{2}} = \mathbf{y}^{j+1}$ ) or in a linear fashion, i.e., ( $\mathbf{y}^{j+\frac{1}{2}} = \frac{\mathbf{y}^{j+1} + \mathbf{y}^j}{2}$ ). Furthermore, the user has to provide subroutines to find the linearized right hand side  $f_y(\cdot)$ , the gradient of  $\varphi$  and  $f_u(\cdot)^T \lambda$ , cf. (11).

## Application of AD

AD tools offer a convenient way to automatically generate the source code to evaluate the gradient in consistence with the discrete objective. For our numerical tests we used ODYSSEE [16], developed by INRIA Sophia Antipolis, France. It is capable of differentiating FORTRAN 77 codes. Analyzing the dependencies, ODYSSEE 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  $\varphi$  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, ODYSSEE 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 ODYSSEE generates the desired code for  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  without any problems in a few seconds. ODYSSEE was also used to differentiate only the subroutine accountable for one Runge-Kutta step, as referred to adjoining one forward step, see Section 3.4.

## 4.2 The SQP solver

Being one of the most renowned SQP solvers, we use Philip Gill's code NPSOL. It 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 iterate 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. [9].

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.

## 5 Examples

### 5.1 Example 1: A 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 with limited rates. 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  $-1 \leq u(t) \leq 1$ . The price for the goods is given by

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

We obtain the following optimal control problem:

$$\begin{aligned} \text{Minimize} \quad & \varphi(y(t_f)) = -y_1(t_f) - p(t_f)y_2(t_f) \\ \text{under} \quad & \dot{y}_1(t) = -ay_2(t) - p(t)u(t) \quad y_1(0) = 100 \\ & \dot{y}_2(t) = u(t) \quad y_2(0) = 0 \end{aligned}$$

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

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

The objective value is known to be  $\varphi(y(t_f)) = -322/3 = -107.\bar{3}$ . 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

$$\lambda_1(t) = -1 \quad \lambda_2(t) = -a(t - 8) - 10. \tag{18}$$

Accordingly, the gradient (cf. (9)) 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 3.2) to differ from the true gradient.

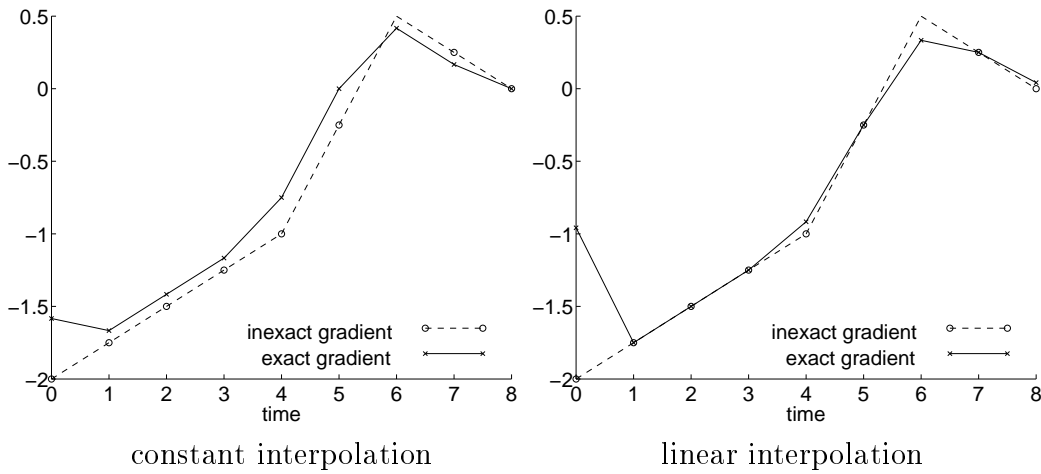


Figure 1: Inconsistency of the gradient for Example 1,  $N_t = 8$

This can be observed in Figure 1 for constant and linear interpolation of the control, respectively, at the off-grid points  $t^{j+\frac{1}{2}}$ . These computations were performed at the initial guess  $\mathbf{u}^j = 0$ .

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

$$\lambda^9 = \begin{pmatrix} -1 \\ -10 \end{pmatrix} \quad \lambda^j = \lambda^{j+1} + \begin{pmatrix} 0 \\ 0.25 \end{pmatrix}, \quad j = 8, \dots, 1. \quad (19)$$

Applying AD to provide the gradient, the derivatives of the cost function with respect to the discrete state variables  $\mathbf{y}^j$ —usually denoted by  $\bar{\mathbf{y}}^j$  in the AD

context—are calculated. For the example considered here, the  $\bar{\mathbf{y}}^j$  equal the solution of the adjoint equation given by (19), i.e.,  $\boldsymbol{\lambda}^j \equiv \bar{\mathbf{y}}^j$ . Furthermore, the  $\boldsymbol{\lambda}^j$  are the exact solution of the continuous adjoint equation at  $t = t^j$ , cf. (18). 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 Figure 1 can only result from different accumulations of the gradient. Using the straightforward approach each component of  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  is computed as

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^j} = -p(t^j)\boldsymbol{\lambda}_1^j + \boldsymbol{\lambda}_2^j \quad j = 1, \dots, 9, \quad (20)$$

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

$$\begin{aligned} \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^9} &= \frac{1}{6} \left( -p(t^9)\boldsymbol{\lambda}_1^9 + \boldsymbol{\lambda}_2^9 \right) \\ \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^j} &= \left( -\frac{2}{3}p(t^{j+\frac{1}{2}}) - \frac{1}{3}p(t^j) \right) \boldsymbol{\lambda}_1^j + \boldsymbol{\lambda}_2^j + \frac{1}{12}\boldsymbol{\lambda}_1^{j+1}, \quad j = 8, \dots, 2 \\ \frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^1} &= \left( -\frac{2}{3}p(t^{1+\frac{1}{2}}) - \frac{1}{6}p(t^1) \right) \boldsymbol{\lambda}_1^1 + \frac{5}{6}\boldsymbol{\lambda}_2^1 + \frac{1}{12}\boldsymbol{\lambda}_1^2 \end{aligned}$$

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  $\mathbf{u}^j$  on the objective value and here particularly the offset  $\frac{1}{12}\boldsymbol{\lambda}_1^{j+1}$  is responsible for the altered gradient information in the case of constant interpolation.

For linear interpolation, AD generates the formulas

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^9} = \left( -\frac{1}{6}p(t^9) - \frac{1}{3}p(t^{9-\frac{1}{2}}) \right) \boldsymbol{\lambda}_1^9 + \frac{1}{2}\boldsymbol{\lambda}_2^9 - \frac{1}{24}\boldsymbol{\lambda}_1^9 \quad (21)$$

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^j} = -\frac{1}{3} \left( p(t^{j+\frac{1}{2}}) + p(t^j) + p(t^{j-\frac{1}{2}}) \right) \boldsymbol{\lambda}_1^j + \boldsymbol{\lambda}_2^j, \quad j = 8, \dots, 1 \quad (22)$$

$$\frac{\partial \tilde{\mathbf{J}}(\mathbf{u})}{\partial \mathbf{u}^1} = \left( -\frac{1}{3}p(t^{1+\frac{1}{2}}) - \frac{1}{6}p(t^1) \right) \boldsymbol{\lambda}_1^1 + \frac{1}{2}\boldsymbol{\lambda}_2^1 + \frac{1}{24}\boldsymbol{\lambda}_1^2. \quad (23)$$

One deduces from (21) that the term  $-\frac{1}{24}\boldsymbol{\lambda}_1^9$  causes the different gradient information at time  $t^9 = 8$ . At the times  $t^7, t^5, t^3$ , and  $t^2$  the formula (22) equals the gradient calculation (20) because of the linear behaviour of the function  $p(\cdot)$ .

Hence, at these points in time the SF 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 (22) and (20). Finally, for  $t^1$ , i.e.,  $t = 0$ , the formulas (23) and (20) 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 consistence of the gradient, the run-time needed by the automatically generated code has to be analyzed. The corresponding measurements to compute  $\tilde{\mathbf{J}}(\mathbf{u})$  and  $\nabla\tilde{\mathbf{J}}(\mathbf{u})$  are given in Table 1. As can be seen, the ratio of the two run-times for evaluating

$N_t$	constant interpolation			linear interpolation		
	8	32	128	8	32	128
$\tilde{\mathbf{J}}(\mathbf{u})$	0.0001 s	0.0006 s	0.0060 s	0.0001 s	0.0006 s	0.0061 s
$\nabla\tilde{\mathbf{J}}(\mathbf{u})$	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 1: Run-time of Gradient Calculation using ODYSSÉE, Example 1

$\nabla\tilde{\mathbf{J}}(\mathbf{u})$  and  $\tilde{\mathbf{J}}(\mathbf{u})$  reaches almost the theoretical bound of five, the gap being caused by ODYSSÉE recomputing some intermediate results of the forward integration. 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 Table 2 and Table 3 for constant and linear interpolation of the control, respectively.

Here as throughout, FD denotes the finite differences approach. SF stands for the straightforward evaluation of the gradient as described in Subsection 3.2. AD<sub>j</sub> denotes the approach to apply AD to the evaluation of the objective  $\tilde{\mathbf{J}}(\mathbf{u})$ , whereas AD<sub>a</sub> marks the approach to differentiate the Runge-Kutta step using AD and to subsequently accumulate the desired gradient by hand-written code, see Subsection 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., it yields 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  $\frac{\partial\psi^j(\mathbf{u})}{\partial\mathbf{u}^i} = 0$  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

	CPU time in seconds			Major it.			objective		
$N_t$	8	32	128	8	32	128	8	32	128
FD	0.019	0.325	43.298	4	17	67	-107.1667	-107.3229	-107.3327
SF	0.014	0.037	0.693	3	13	51	-107.1667	-107.3229	-107.3327
AD $\mathbf{j}$	0.017	0.104	3.611	4	16	61	-107.1667	-107.3229	-107.3327
AD $_a$	0.018	0.099	3.207	4	16	61	-107.1667	-107.3229	-107.3327

Table 2: SQP convergence behavior, Example 1, constant interpolation of the control,  $N_t = 8|32|128$

	CPU time in seconds			Major it.			objective		
$N_t$	8	32	128	8	32	128	8	32	128
FD	0.022	0.386	46.579	6	18	69	-107.2500	-107.3281	-107.3330
SF	0.014	0.039	0.688	3	13	51	-107.2083	-107.3255	-107.3328
AD $\mathbf{j}$	0.019	0.112	3.348	5	14	52	-107.2500	-107.3281	-107.3330
AD $_a$	0.018	0.101	2.969	5	14	52	-107.2500	-107.3281	-107.3330

Table 3: SQP convergence behavior, Example 1, linear interpolation of the control,  $N_t = 8|32|128$

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.

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 runtime savings of at most 10% for Example 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 2.

## 5.2 Example 2: The 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

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

As an objective function we take, cf. Büskens [4],

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

This problem can easily be transformed into the form (1)–(4) described in Section 2 via  $y_1 = x$ ,  $y_2 = \dot{x}$ :

$$\begin{aligned} \dot{y}_1(t) &= y_2(t) \\ \dot{y}_2(t) &= -y_1(t) + y_2(t)(1.4 - 0.14y_2(t)^2) + 4u(t) \\ \dot{y}_3(t) &= u(t)^2 + y_1(t)^2 \end{aligned}$$

and objective

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

We assume initial conditions

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

and impose the control constraint  $|u(t)| \leq 1$ . 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:

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

with terminal conditions

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

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  $\mathbf{y}^{j+\frac{1}{2}}$  are necessary, provided that the same Runge-Kutta scheme is used. As described in Section 4.1, we compare both constant and linear interpolation of the state. The corresponding inconsistent gradients are shown in Figures 2 and 3.

Table 4 states the run-times needed to compute  $\nabla \tilde{\mathbf{J}}(\mathbf{u})$  and  $\tilde{\mathbf{J}}(\mathbf{u})$  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 due to the less effective compiler optimization for the forward integration. The Tables 5 and 6 show the important results concerning the optimization process. Here, SF<sub>c</sub> indicates constant interpolation of the state during the adjoint calculation using the straightforward approach, while SF<sub>l</sub> denotes linear interpolation.

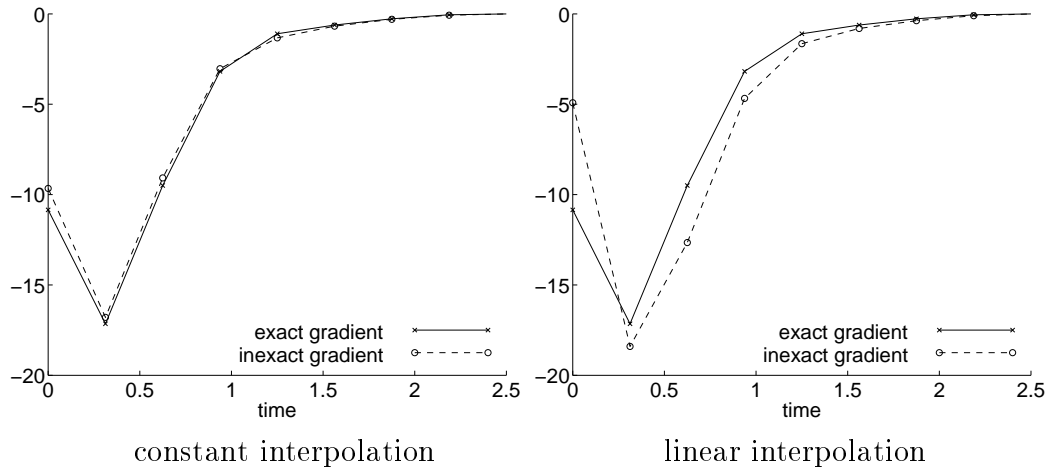


Figure 2: Inconsistency of the gradient, Example 2, constant interpolation of the control, constant and linear interpolation of the state,  $N_t = 8$

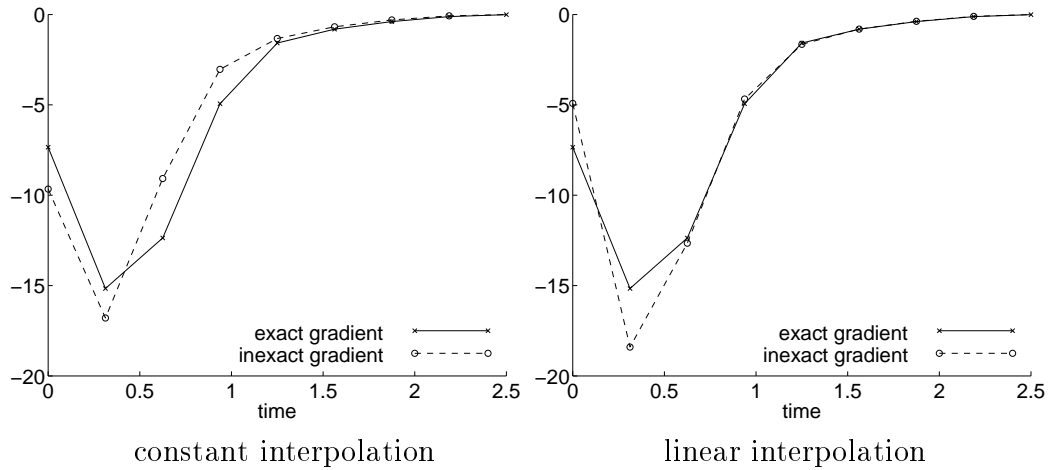


Figure 3: Inconsistency of the gradient, Example 2, linear interpolation of the control, constant and linear interpolation of the state,  $N_t = 8$

	constant interpolation			linear interpolation		
$N_t$	8	32	128	8	32	128
$\tilde{\mathbf{J}}(\mathbf{u})$	0.0001 s	0.0006 s	0.0060 s	0.0001 s	0.0007 s	0.0062 s
$\nabla \tilde{\mathbf{J}}(\mathbf{u})$	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: Run-time of Gradient Calculation using ODYSÉE, Example 2

	CPU time in seconds			Major it.			objective		
$N_t$	8	32	128	8	32	128	8	32	128
FD	0.023	0.411	21.628	9	17	19	41.17086	42.79915	42.80745
SF <sub>c</sub>	0.016	0.038	0.696	(5)	(5)	(22)	41.37857	42.81615	42.81038
SF <sub>1</sub>	0.018	0.036	0.426	(4)	(5)	(21)	41.51795	42.82070	42.81106
AD <sub><math>\tilde{\mathbf{j}}</math></sub>	0.021	0.093	1.015	9	17	19	41.17086	42.79915	42.80745
AD <sub><math>a</math></sub>	0.020	0.074	0.758	9	17	19	41.17086	42.79915	42.80745

Table 5: SQP convergence behavior, Example 2, constant interpolation of  $\mathbf{u}$

	CPU time in seconds			Major it.			objective		
$N_t$	8	32	128	8	32	128	8	32	128
FD	0.030	0.500	15.165	17	31	22	41.19445	42.79858	42.80742
SF <sub>c</sub>	0.016	0.036	0.535	(4)	(5)	(22)	41.26702	42.81044	42.81016
SF <sub>1</sub>	0.017	0.036	0.546	(4)	(5)	(22)	41.37555	42.81859	42.81074
AD <sub><math>\tilde{\mathbf{j}}</math></sub>	0.028	0.154	1.250	17	30	23	41.19445	42.79858	42.80742
AD <sub><math>a</math></sub>	0.025	0.124	1.020	17	30	23	41.19445	42.79858	42.80742

Table 6: SQP convergence behavior, Example 2, linear interpolation of  $\mathbf{u}$

It has to be mentioned that NPSOL does not terminate the optimization process until the given maximal number of major iterations is 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 paranthesis, the specified objective value is reached. After that, NPSOL chooses very small step sizes and practically stagnates. The run-times stated for SF<sub>c</sub> and SF<sub>1</sub> 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 [4] with 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. To explain this run-time behavior, we observe that the operation count  $O_{\tilde{\mathbf{j}}(\mathbf{u})}$  is asymptotically linear in the number of time grid points  $N_t$ . In addition, we have verified one of the AD's paradigms for the reverse mode: The ratio  $\nabla\tilde{\mathbf{J}}(\mathbf{u})$  vs.  $\tilde{\mathbf{J}}(\mathbf{u})$  being independent of the number of input variables  $\mathbf{u}$ , i.e., independent of  $N_t$ . Combining these two findings, we conclude that the operation count 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 it causes the SQP solver to stagnate.

## 6 Conclusion

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 2.

Despite some theoretical studies how methods should cope with inexact function and derivative information [6], 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 as explained before. In order to analyze the influence of this behaviour 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, 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 [12], finding the correct integration scheme for the continuous adjoint equation is not a trivial task. We suggest to use Automatic Differentiation tools to relieve the user of this burden, allowing to conveniently compute 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. Gill et. al. [8].

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 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 an upcoming paper, we plan to consider more realistic control problems with

complicated control and also state constraints. In order that these problems be solved efficiently by an SQP code, the constraint Jacobian will be required. We conjecture that again AD will be the best option whenever these Jacobians cannot be supplied by hand-written computer code.

## References

- [1] Betts, J. T.: *Practical Methods for Optimal Control Using Nonlinear Programming*, SIAM, Philadelphia, 2001.
- [2] Bock, H.G., Plitt, K.-J.: *A Multiple Shooting Algorithm for Direct Solution of Optimal Control Problems*, Proceedings of the 9th IFAC World Congress, Budapest, Pergamon Press, 1984.
- [3] Bulirsch, R., Nerz, E., Pesch, H.J., von Stryk, O.: *Combining Direct and Indirect Methods in Nonlinear Optimal Control: Range Maximization of a Hang Glider*, in: R. Bulirsch, A. Miele, J. Stoer, K. H. Well, eds., *Optimal Control, Calculus of Variations, Optimal Control Theory and Numerical Methods*, Birkhäuser, Basel, 1993, 273–288.
- [4] Büskens, C.: *Optimierungsmethoden und Sensitivitätsanalyse für optimale Steuerprozesse mit Steuer- und Zustandsbeschränkungen*, Dissertation, Westfälische Wilhelms-Universität Münster, 1998.
- [5] Büskens, C., Maurer, H.: *SQP-methods for Solving Optimal Control Problems with Control and State Constraints: Adjoint Variables, Sensitivity Analysis and Real-time Control*, *Journal of Computational and Applied Mathematics* 120, 2000, 85–108.
- [6] Carter, R.: *Numerical Experience with a Class of Algorithms for Nonlinear Optimization Using Inexact Function and Gradient Information*, *SIAM Journal on Scientific Computing* 14, 1993, 368–388.
- [7] Eberhard, P., Bischof, C.: *Automatic Differentiation of Numerical Integration Algorithms*. Preprint ANL/MCS-P621-1196, Argonne National Laboratory, 1996.
- [8] Gertz, M., Gill, P., Muetherig, J.: *User's Guide for SNADIOPT: A Package Adding Automatic Differentiation to SNOPT*, Technical Report NA 01-01, Department of Mathematics, University of California, San Diego, 2001.
- [9] Gill, P., Murray, W., Saunders, M., Wright, H.: *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.

- [10] Griewank, A.: *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math. 19, Phil., 2000.
- [11] Griewank, A.: *On Automatic Differentiation*, in: Iri, M. and Tanabe, K., eds., *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publishers, 1989.
- [12] Hager, W.: *Runge-Kutta Methods in Optimal Control and the Transformed Adjoint System*, Numer. Math. 87, 2000, 247–282.
- [13] Hiltmann, P.: *Numerische Lösung von Mehrpunkt-Randwertproblemen und Aufgaben der optimalen Steuerung mit Steuerfunktionen über endlichdimensionalen Räumen*, Dissertation, TU München, Mathematisches Institut, Germany, 1989.
- [14] Oberle, H., Grimm, W.: *BNDSCO — A Program for the Numerical Solution of Optimal Control Problems*, Report No. 515, Institut for Flight System Dynamics, Oberpfaffenhofen, German Aerospace Research Establishment DLR, 1989.
- [15] Pesch, H.J.: *Offline and Online Computation of Optimal Trajectories in the Aerospace Field* in: A. Miele, A. Salvetti, eds., *Applied Mathematics in Aerospace Science and Engineering*, Plenum Press, New York, *Mathematical Concepts and Methods in Science and Engineering*, 44, 1994, 165–220.
- [16] Rostaing, N., Dalmas, S. and Galligo, A.: *Automatic differentiation in ODYSSEE*, in: M. Berz, C. Bischof, G. Corliss, A. Griewank, eds., *Computational Differentiation—Techniques, Applications, and Tools*, SIAM, Philadelphia, 1996, 558–568.
- [17] von Stryk, O.: *User’s Guide for DIRCOL (Version 2.1): A Direct Collocation Method for the Numerical Solution of Optimal Control Problems*, Fachgebiet Simulation und Systemoptimierung (SIM), Technische Universität Darmstadt, 2000.