MATHEMATICS OF COMPUTATION
Volume 68, Number 226, April 1999, Pages 717–731
S 0025-5718(99)01027-3

AUTOMATIC DIFFERENTIATION OF NUMERICAL INTEGRATION ALGORITHMS

PETER EBERHARD AND CHRISTIAN BISCHOF

ABSTRACT. Automatic differentiation (AD) is a technique for automatically augmenting computer programs with statements for the computation of derivatives. This article discusses the application of automatic differentiation to numerical integration algorithms for ordinary differential equations (ODEs), in particular, the ramifications of the fact that AD is applied not only to the solution of such an algorithm, but to the solution procedure itself. This subtle issue can lead to surprising results when AD tools are applied to variable-stepsize, variable-order ODE integrators. The computation of the final time step plays a special role in determining the computed derivatives. We investigate these issues using various integrators and suggest constructive approaches for obtaining the desired derivatives.

1. INTRODUCTION

Typically, the description of technical systems or natural phenomena leads to complicated mathematical models involving ordinary differential equations, differential-algebraic equations, or partial differential equations. For example, many problems in mechanical, chemical, and electrical engineering can be formulated as an initial value problem using ODEs:

> For given values of system parameters $\mathbf{p} \in \mathbb{R}^h$, find the trajectories $\mathbf{x}(t, \mathbf{p}), \mathbf{x} \in \mathbb{R}^n$ for $t^0 \leq t \leq t^1$, where $\mathbf{x}$ is the state vector, t the time, t^0 the initial time, and t^1 the final time. The state is determined by the solution of the initial value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{p}, t), \quad \mathbf{x}(t = t^0, \mathbf{p}) = \mathbf{x}^0, \tag{1.1}$$

> where $\mathbf{f}$ is the vector of state derivatives and $\mathbf{x}^0$ is the initial state.

Problem (1.1) typically is solved by using a numerical integration algorithm, and a large body of literature is devoted to this subject (see, e.g., [6, 13, 14, 15, 19, 20, 21]). Also, in many engineering applications, one is interested not only in the final state, but also in performance criteria ψ computed from the trajectories $\mathbf{x}$.

Received by the editor November 11, 1996 and, in revised form, July 24, 1997.

1991 *Mathematics Subject Classification.* Primary 34A12, 65L05, 65L06; Secondary 68N99.

This work was partially completed while the first author was visiting the Department of Mechanical Engineering at the University of California at Berkeley supported by the German Research Council (DFG) under grant EB195/1-1.

The second author was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

If optimization procedures are applied in order to choose optimal design variables with respect to certain performance criteria, or if parameter estimation techniques are used to identify model parameters from measurements (see, e.g., [2, 8]), then, with the final state

$$\mathbf{x}^1 := \mathbf{x}(t^1, \mathbf{p}), \tag{1.2}$$

one typically has to compute

$$\frac{\partial \mathbf{x}^1}{\partial \mathbf{p}^T}. \tag{1.3}$$

While the sensitivities at the final time step are of interest, their numerical value may depend on the whole time history of the system (e.g., when a performance criterion is some integral over $\mathbf{x}$ from t^0 to t^1).

During the past decade, so-called automatic differentiation (AD) tools have been developed that make it possible to augment general Fortran or C codes with statements for the computation of derivatives in an automated fashion. AD relies on the fact that every computer program employs only simple operations such as additions or multiplications or intrinsics such as `sin()`, whose derivatives are known, and then computes derivatives for the whole program by judicious composition of these derivatives using the chain rule of differential calculus. In our experiments, we used the ADIFOR[1] tool [4]; the ADIFOR reference, as well as [1, 12, 17], provides some basics on AD. The impact of the associativity of the chain rule on the cost of computing derivatives is discussed in [3, 11], and a collection of currently available AD tools can be found on the World-Wide Web.[2]

While AD has been used successfully in many large-scale applications and inherently computes accurate derivatives, the black-box application of AD can lead to surprising results, because AD will differentiate not only the solution computed by a computer program, *but also the algorithm by which the solution is being derived.* That is, while AD will compute a derivative, *the value of this derivative may well depend on the algorithm chosen to compute the solution.*

In this article we investigate the automatic differentiation of numerical integration algorithms. In the next section, we consider the automatic differentiation of a prototypical integration algorithm and illustrate the impact that error-adaptive schemes can have on the computed derivatives—that is, different integrators can lead to very different values for the computed derivatives. This realization also leads us to suggest two approaches to suppress the impact of the solution algorithm on derivatives, thus resulting in the computation of derivatives that are defined by the nondiscretized solution $\mathbf{x}(t, \mathbf{p})$ and, in general, are the desired ones.

We illustrate these effects and our remediation methodology in Section 3 on a relatively simple ODE with known explicit solution, using single-step Runge-Kutta integrators with and without adaptive stepsize control, as well as the multistep Shampine-Gordon algorithm. The computation of the final time step has a fundamental effect on the overall derivatives, and this issue is also investigated. In Section 4, we then apply our approach to a complicated problem from multibody dynamics and verify the results using the adjoint variable approach, an approximation-free method to efficiently compute sensitivities for multibody systems. Finally, in Section 5, we summarize our results.

[1] See `http://www.mcs.anl.gov/adifor` and `http://www.cs.rice.edu/~adifor`.

[2] See `http://www.mcs.anl.gov/Projects/autodiff/adtools`.

2. Automatic differentiation of prototypical numerical integration algorithms

The numerical integration of ODEs is one of the basic problems in numerical computing, and many research groups are working on the development of reliable and efficient algorithms. These algorithms can be categorized into several classes, including single step algorithms, multistep algorithms, and extrapolation algorithms.

Special algorithms also exist, for example, for stiff systems, highly or loosely coupled systems, and systems composed of several subsystems with different frequency ranges or real-time requirements.

2.1. **Derivatives of time.** To illustrate the issues relevant to the interplay of AD and integration algorithms, we choose some explicit single-step algorithms of Euler and Runge-Kutta type with and without stepsize control, as well as a sophisticated multistep integration algorithm with adaptive stepsize and interpolation order control. The following discussion of integration algorithms is intentionally kept simple to emphasize the salient points. Details on numerical integration algorithms and their implementation can be found, for example, in the aforementioned references and in the many codes available from netlib.[3]

With single-step algorithms, the time discretization that typically is applied to solve (1.1) leads to a recursive scheme

$$\mathbf{x}_{i+1} = \mathbf{x}_i + h_i \dot{\tilde{\mathbf{x}}}_i, \qquad t_{i+1} = t_i + h_i, \tag{2.1}$$

where the subscript i denotes the ith integration step. That is, $\mathbf{x}_i := \mathbf{x}(t_i)$, h_i is the actual stepsize, and $\dot{\tilde{\mathbf{x}}}_i$ denotes a slope estimation. The simplest case $h_i = h =$ *constant* and $\dot{\tilde{\mathbf{x}}}_i = \dot{\mathbf{x}}_i$ yields the explicit Euler scheme; with $\dot{\tilde{\mathbf{x}}}_i = \dot{\mathbf{x}}_{i+1}$, on the other hand, we have the implicit Euler scheme. Usually $\dot{\tilde{\mathbf{x}}}_i$ is composed from different evaluations of the ODE at different times and approximations. The (nonunique) weighting coefficients of its different components have to satisfy a Taylor series approximation with certain order.

Most advanced integrators employ adaptive stepsize control that, based on local error estimates (e.g., available from a doublestep technique or the simultaneous evaluation of different-order schemes), dynamically adapts the stepsize. This stepsize control is essential for efficiency, enabling one to choose the stepsize as big as possible, yet at the same time sufficiently small enough to restrict the integration error. Multistep algorithms additionally use information from former steps to predict appropriate stepsizes and slopes. Many sophisticated modifications, such as the use of variable extrapolation order or projections, may further improve the efficiency. As a result, *all variables in* (2.1) *may depend on the system parameters* $\mathbf{p}$, that is,

$$\mathbf{x}_{i+1}(\mathbf{p}) = \mathbf{x}_i(\mathbf{p}) + h_i(\mathbf{p})\dot{\tilde{\mathbf{x}}}_i(\mathbf{p}). \tag{2.2}$$

The procedure defined by (2.2) is the blueprint of a numerical algorithm, which starts from the initial values $\mathbf{x}_0 = \mathbf{x}^0$ and the system parameters $\mathbf{p}$ and computes some final values $\mathbf{x}^1(\mathbf{p})$ via an often extremely large number of intermediate steps.

[3]See `http://netlib.att.com/netlib/master/readme.html`.

Differentiating (2.2) with respect to $\mathbf{p}$, with

$$\nabla \mathbf{x} := \frac{d\mathbf{x}}{d\mathbf{p}^T}, \tag{2.3}$$

we obtain

$$\nabla \mathbf{x}_{i+1} = \nabla \mathbf{x}_i + h_i \nabla \dot{\mathbf{x}}_i + \nabla h_i \dot{\mathbf{x}}_i. \tag{2.4}$$

Note that, depending on how h_i is computed, we may obtain very different values for the total derivative $\nabla \mathbf{x}^1$. If the initial values $\mathbf{x}^0$ are independent of $\mathbf{p}$, then $\nabla \mathbf{x}^0 = 0$; otherwise, $\nabla \mathbf{x}^0$ has nonzero components.

2.2. **Computation of the desired derivatives for the state variables.** To obtain the desired derivatives, we can consider two choices, which are illustrated in Figure 1.

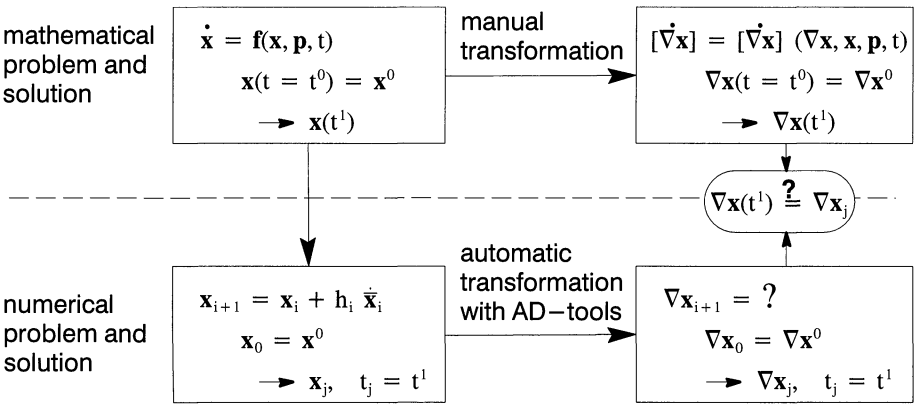

FIGURE 1. Manual transformation versus automatic transformation with AD tools

1. **Differentiate the ODE and integrate:**
   Differentiating (1.1) with respect to $\mathbf{p}$, we obtain with $dt/d\mathbf{p}^T = 0$

$$\frac{d}{d\mathbf{p}^T}(\dot{\mathbf{x}}) = \frac{d}{d\mathbf{p}^T}\left(\frac{d\mathbf{x}}{dt}\right) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^T}\frac{d\mathbf{x}}{d\mathbf{p}^T} + \frac{\partial \mathbf{f}}{\partial \mathbf{p}^T}. \tag{2.5}$$

   Exchanging the order of differentiation, we thus obtain a new ODE for $\nabla \mathbf{x}$:

$$\frac{d}{dt}\left(\frac{d\mathbf{x}}{d\mathbf{p}^T}\right) = \frac{d}{dt}(\nabla \mathbf{x}) = [\dot{\nabla}\mathbf{x}] = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^T}\nabla \mathbf{x} + \frac{\partial \mathbf{f}}{\partial \mathbf{p}^T}, \quad \nabla \mathbf{x}(t = t^0) = \nabla \mathbf{x}^0, \tag{2.6}$$

   which we can integrate alongside our original solution. AD techniques could, for example, be employed to compute $\partial \mathbf{f}/\partial \mathbf{x}^T$ and $\partial \mathbf{f}/\partial \mathbf{p}^T$, *but we would not differentiate through the integration algorithm for* $\mathbf{x}$.

   Given a suitable integrator, this approach will deliver the desired sensitivities $\nabla \mathbf{x}^1$ with suitable accuracy, since its behavior is governed jointly by (1.1) and (2.6). Up to the chosen tolerance, the actual time discretization will not impact either $\mathbf{x}^1$ or $\nabla \mathbf{x}^1$.
2. **Differentiate the integrator for the ODE:**
   Viewing the integration procedure for solving (1.1) as a black box that, given values for $\mathbf{p}$, produces values for $\mathbf{x}^1$, we can employ an AD tool to

augment *both the problem-independent code for the numerical time integration algorithm and the problem-specific code for the evaluation of the ODE* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{p}, t)$ with statements for the computation of derivatives.

We observe two facts:

(a) Since AD tools do not alter the control flow of the original program, the time discretization chosen is determined solely by the integration of (1.1).

(b) As observed in (2.2), the stepsize h_i is likely to depend on the parameters $\mathbf{p}$, and an AD tool will associate a gradient ∇h_i with h_i. Thus, the update (2.4), which will be computed by an AD tool, leads to an inconsistent integrator for $\nabla \mathbf{x}$, as the final result depends on the discretization strategy chosen. Hence, it is also unlikely that $\nabla \mathbf{x}^1 = \nabla \mathbf{x}\big|_{t_i = t^1}$ equals the desired $\partial \mathbf{x}^1 / \partial \mathbf{p}^T$.

The automatically differentiated integration algorithm computes $\mathbf{x}^1(t^1(\mathbf{p}), \mathbf{p})$, where the physically implausible dependence of t^1 on $\mathbf{p}$ results only from the adaptive time discretization. Differentiating with respect to $\mathbf{p}$, we obtain

$$\nabla \mathbf{x}^1 = \frac{\partial \mathbf{x}^1}{\partial t^1} \nabla t^1 + \frac{\partial \mathbf{x}^1}{\partial \mathbf{p}^T}. \tag{2.7}$$

The total derivatives $\nabla \mathbf{x}^1 = d\mathbf{x}^1 / d\mathbf{p}^T$ and $\nabla t^1 = dt^1 / d\mathbf{p}^T$ depend on the time discretization and are computed by the AD-generated code. Thus, to arrive at the desired solution $\partial \mathbf{x}^1 / \partial \mathbf{p}^T$, we can pursue one of the following strategies:

- Perform an a posteriori error correction: From (2.7), taking (1.1) into account, we realize that the desired derivatives $\partial \mathbf{x}^1 / \partial \mathbf{p}^T$, *which do not depend on the time discretization chosen*, can be computed as

$$\frac{\partial \mathbf{x}^1}{\partial \mathbf{p}^T} = \nabla \mathbf{x}^1 - \mathbf{f}(\mathbf{x}^1, \mathbf{p}, t^1) \nabla t^1. \tag{2.8}$$

- Use an integrator with fixed stepsize: In this case, we have $\nabla h_i \equiv 0, \forall i$. Thus $\nabla t \equiv 0$, and hence $\nabla \mathbf{x}^1 \equiv \partial \mathbf{x}^1 / \partial \mathbf{p}^T$ in (2.7). The AD-computed derivative trajectories are the desired ones, and thus, no modification is required for fixed-stepsize integration algorithms (see also [18], which explores this issue in more detail in the context of a so-called quasi-steady-state integrator).
- Modify the AD-generated code to enforce $\nabla h_i = 0, \forall i$: For the first step, the user must guess an initial stepsize h_0. Because h_0 is independent of $\mathbf{p}$, we have $\nabla h_0 = 0$. Assume that the correct stepsize is known in advance for each step. Then $\nabla h_i = 0, \forall i$ (which implies $\nabla t_i = 0, \forall i$), and one gets the same correct results as for the fixed stepsize algorithms. Thus, by modifying the AD-generated code manually to ensure

$$\nabla h_i = 0 \tag{2.9}$$

we can ensure consistency. While this procedure could be done easily when the code is developed, and may in fact be desirable because of the potentially unpredictable nature of $\nabla \mathbf{x}$ and ∇t even when $\partial \mathbf{x} / \partial \mathbf{p}^T$ is well behaved, it is likely to be a nontrivial task after the fact, since in-depth knowledge of the differentiated code may be required in order not to miss subtle dependencies.

Perhaps surprising, a commonly used strategy for determining $\mathbf{x}^1$ for a predetermined t^1 results in $\nabla t^1 = 0$ and hence $\partial \mathbf{x}^1 / \partial \mathbf{p}^T = \nabla \mathbf{x}^1$. If, for a suggested stepsize

h_i, we have in the last step $t_i + h_i > t^1$, we are likely to set

$$h_{i+1} = t^1 - t_i \quad (\text{implying } \nabla h_{i+1} = -\nabla t_i), \tag{2.10}$$

since t^1 is a user-selected constant. Thus, the computation

$$t_{i+1} = t_i + h_{i+1} \quad \text{implies} \quad \nabla t^1 = \nabla t_{i+1} = \nabla t_i + \nabla h_{i+1} = 0. \tag{2.11}$$

Hence, for the last step (and most likely only for the last step), we have $\nabla t_i = 0$, and therefore the a posteriori correction is not required for $\nabla \mathbf{x}^1$, although it is most likely required for any other point on the derivative trajectory.

However, there is no guarantee that $\nabla t^1 = 0$ if t^1 has been preselected. Another frequently chosen approach to compute $\mathbf{x}^1$ is to terminate the integration at the first time point beyond t^1 and then interpolate the value of $\mathbf{x}$ for t^1. In this case, ∇t^1 is unlikely to be zero, and the error correction is required. Thus, for the general user, who most likely is not familiar with the internals of the algorithm, we suggest checking whether $\nabla t|_{t=t^1}$ is zero. If it is not, the a posteriori correction (2.8) should be applied.

3. Experimental results with a one-mass oscillator

The simplest multibody system is a horizontal one-mass oscillator shown in Figure 2. As shown, for example, in [2], one can derive closed-form solutions for both the state and its gradients, and thus the system is well suited as an example to illustrate the issues outlined in the preceding section.

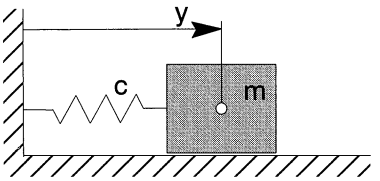

Figure 2. One-mass oscillator

3.1. **Mechanical model.** A body with mass m can slide on the horizontal ground. It is coupled to the wall with a linear spring with spring stiffness c. The position is described by $y(t)$. From Newton's equation one can derive the equation of motion

$$m\ddot{y} + cy = 0 \tag{3.1}$$

or, with $\mathbf{x} = [y, \dot{y}]^T$, the corresponding set of first-order ODEs

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{c}{m} x_1 \end{bmatrix}. \tag{3.2}$$

With the initial condition $\mathbf{x}(t = t^0) = \mathbf{x}^0 = [0, v^0]^T$, the solution of the ODE is

$$y(t) = v^0 \sqrt{\frac{m}{c}} \sin \sqrt{\frac{c}{m}} t. \tag{3.3}$$

For $m = 1$ and the system parameters $\mathbf{p} = [c, v^0]^T$, we find

$$y(t) = v^0 \frac{1}{\sqrt{c}} \sin \sqrt{c}t, \quad \dot{y}(t) = v^0 \cos \sqrt{c}t, \quad \ddot{y}(t) = -v^0 \sqrt{c} \sin \sqrt{c}t. \tag{3.4}$$

Two criteria are now defined: The criterion ψ_1 contains the position of the body at time $t^1 = \pi/2$, and for $\mathbf{p} = [1, 0.5]^T$ we have

$$\begin{aligned} \frac{d\psi_1}{dp_1} &= \frac{d\psi_1}{dc} = \frac{v^0}{2c}\left(\frac{\pi}{2}\cos\left(\sqrt{c}\frac{\pi}{2}\right) - \frac{1}{\sqrt{c}}\sin\left(\sqrt{c}\frac{\pi}{2}\right)\right) = -0.25, \\ \frac{d\psi_1}{dp_2} &= \frac{d\psi_1}{dv^0} = \frac{1}{\sqrt{c}}\sin\left(\sqrt{c}\frac{\pi}{2}\right) = 1.0. \end{aligned} \tag{3.5}$$

The criterion ψ_2 integrates the position over the whole interesting simulation time interval $[t^0, t^1]$:

$$\psi_2 = \int_{t^0}^{t^1} y(t)\,dt = \int_{t^0=0}^{t^1=\pi/2} \frac{v^0}{\sqrt{c}}\sin(\sqrt{c}t)\,dt = \frac{v^0}{\sqrt{c}}\left(1 - \cos\sqrt{c}\frac{\pi}{2}\right). \tag{3.6}$$

Here, the integral type criterion can still be computed analytically, but for more complicated systems it has to be evaluated numerically together with the ODE of the mechanical system, yielding an extended state

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{c}{m}x_1 \\ \dot{\psi}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{c}{m}x_1 \\ x_1 \end{bmatrix}. \tag{3.7}$$

Again we find explicit solutions for the gradients

$$\frac{d\psi_2}{dp_1} = \frac{d\psi_2}{dc} = \frac{\frac{v^0}{2\sqrt{c}}\frac{\pi}{2}\sin\left(\sqrt{c}\frac{\pi}{2}\right) + v^0\cos\left(\sqrt{c}\frac{\pi}{2}\right)}{c^2} - \frac{v^0}{c^2}, \tag{3.8}$$

$$\frac{d\psi_2}{dp_2} = \frac{d\psi_2}{dv^0} = \frac{1}{c} - \frac{1}{c}\cos\left(\sqrt{c}\frac{\pi}{2}\right), \tag{3.9}$$

and, for the given numerical values, the results

$$\frac{d\psi_2}{dp_1} = \frac{\pi}{8} - \frac{1}{2}, \qquad \frac{d\psi_2}{dp_2} = 1.0. \tag{3.10}$$

3.2. **Single-step integration algorithms without stepsize control.** We investigate three similar integration schemes: the explicit Euler scheme, the Heun algorithm, and the fourth-order Runge-Kutta algorithm; see [16]. Because the stepsize h does not change during the integration, ADIFOR does not generate a gradient ∇h.

Table 1 shows the relative errors for the criteria and the gradients for different stepsizes and integration algorithms. Only minor differences between the reference gradient and the AD gradients exist. The relative error in the criteria is about the size of the error in the gradients. As expected, automatic differentiation of single-step integration algorithms without stepsize control leads to the desired result without any need for user modifications.

The differences can be explained by the choice of the stepsize and the algorithm; no additional errors are introduced by the automatic differentiation procedure. The higher-order algorithms yield higher accuracies, and the errors are at least of order $O(h)$ for the Euler integrator, $O(h^2)$ for the Heun integrator and $O(h^4)$ for the Runge-Kutta integrator. Some components yield even higher error order; for example, for the Runge-Kutta integrator ψ_1 and $d\psi_1/dp_2$ seems to be of order $O(h^5)$.

TABLE 1. Relative errors for different stepsizes

Step h	Method	ψ_1	ψ_2	$\frac{d\psi_1}{dp_1}$	$\frac{d\psi_1}{dp_2}$	$\frac{d\psi_2}{dp_1}$	$\frac{d\psi_2}{dp_2}$
0.25	Euler	.20e-0	.36e-1	.28e-0	.20e-0	.63e-0	.36e-1
	Heun	.28e-2	.15e-1	.16e-1	.28e-2	.45e-1	.15e-1
	Ru-Ku	.10e-4	.48e-4	.25e-4	.10e-4	.37e-3	.48e-4
0.025	Euler	.20e-1	.33e-3	.21e-1	.20e-1	.72e-1	.33e-3
	Heun	.30e-5	.16e-3	.25e-3	.30e-5	.40e-3	.16e-3
	Ru-Ku	.11e-9	.51e-8	.74e-8	.11e-9	.36e-7	.51e-8
0.0025	Euler	.20e-2	.33e-5	.20e-2	.20e-2	.72e-2	.33e-5
	Heun	.31e-8	.16e-5	.26e-5	.31e-8	.38e-5	.16e-5
	Ru-Ku	.14e-14	.49e-12	.76e-12	.14e-14	.36e-11	.49e-12

3.3. **Single-step integration algorithms with adaptive stepsize control.** The next algorithm we investigate is a mixed fourth- and fifth-order Runge-Kutta algorithm with stepsize control; see [7]. To get an estimate for the absolute local error, a fifth-order Runge-Kutta method

$$\mathbf{x}_{i+1} = \mathbf{x}_i + h(\dots) + O(h^6) \tag{3.11}$$

and an embedded fourth-order Runge-Kutta method

$$\mathbf{x}^\star_{i+1} = \mathbf{x}_i + h(\dots) + O(h^5) \tag{3.12}$$

are evaluated. The error is of magnitude $O(h^5)$ and follows from the difference

$$\Delta = ||\mathbf{x}_{i+1} - \mathbf{x}^\star_{i+1}||_\infty. \tag{3.13}$$

Because the error Δ is of magnitude h^5, we can estimate the required stepsize $\bar{h}$ from the desired error bound $\bar{\Delta}$, the actual stepsize h, and the actual error Δ:

$$\frac{\bar{h}^5}{\bar{\Delta}} \approx \frac{h^5}{\Delta} \quad \rightarrow \quad \bar{h} \approx h \sqrt[5]{\frac{\bar{\Delta}}{\Delta}}. \tag{3.14}$$

If $\bar{h} > h$, the actual stepsize was too big, and the step has to be repeated with decreased stepsize $\bar{h}$ until the error estimate is acceptable. Otherwise, the next (increased) stepsize is computed, and the integration proceeds. Because the actual stepsize h_i and the actual time t_i depend on the state $\mathbf{x}_i$ and therefore on the system parameters $\mathbf{p}$, an automatic differentiation tool will compute gradients $\nabla h_i = dh_i/d\mathbf{p}^T$ and $\nabla t_i = dt_i/d\mathbf{p}^T$, respectively, as suggested in (2.4).

We then employ relation (2.8) to compute, at every time step, the desired $\partial\mathbf{x}/\partial\mathbf{p}^T$ from $\nabla\mathbf{x}$ and ∇t. Figure 3 shows some of the trajectories from $\partial\mathbf{x}/\partial\mathbf{p}^T$, $\nabla\mathbf{x}$, and ∇t, where the error tolerance was chosen to be 10^{-8}. Note that the trajectories for $\partial\mathbf{x}/\partial\mathbf{p}^T$ and the AD-computed $\nabla\mathbf{x}$ are very different, but the gradients computed with the correction (2.8) lead to the correct results. (The derivatives with respect to p_2 and the derivatives related to x_2 and x_3 show the same behavior but, for clarity of presentation, are not drawn here.)

The error can be controlled by the user-defined error bound; see Table 2, where the final time is set equal to $t^1 = 10$. It can be seen that the relative errors in $\psi_1 = x_1, \psi_2 = x_3$ and their derivatives are both within the prescribed error bound. Of course, the stricter the prescribed error bounds, the higher the number of steps required. In Figure 4 the trajectories are shown for two different prescribed errors

TABLE 2. Relative errors in the states for different prescribed error bounds

$\bar{\Delta}$	# Steps	ψ_1	ψ_2	$\frac{d\psi_1}{dp_1}$	$\frac{d\psi_1}{dp_2}$	$\frac{d\psi_2}{dp_1}$	$\frac{d\psi_2}{dp_2}$
10^{-5}	21	.11e-4	.36e-4	.82e-4	.32e-4	.73e-4	.11e-4
10^{-8}	72	.89e-7	.36e-7	.11e-6	.54e-7	.35e-7	.22e-7
10^{-11}	281	.83e-10	.37e-10	.88e-10	.51e-10	.54e-10	.19e-11

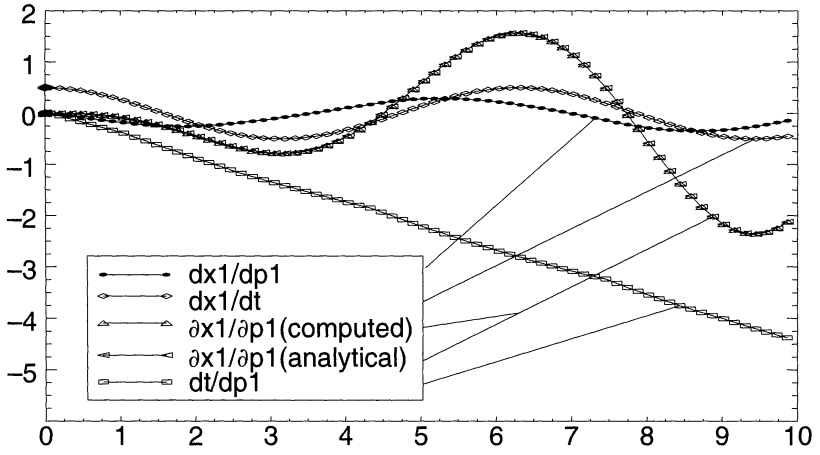

FIGURE 3. Trajectories $\partial\psi_1/\partial p_1 = \partial x_1/\partial p_1$, $\nabla\psi_1 = \nabla x_1$, ∇t_1 and $\dot{\psi}_1 = \dot{x}_1$

bounds (10^{-5} and 10^{-8}). The trajectories for $\dot{\mathbf{x}}$ and $\partial\mathbf{x}/\partial\mathbf{p}^T$ nearly coincide for different error bounds, but because the time discretization depends on the prescribed error bound, we get significantly different trajectories for $\nabla\mathbf{x}$ and ∇t.

Note that in the unmodified AD-created code the stepsize is controlled only by the integration of the n ODEs for $\mathbf{x}_i$ and is *not* affected by the nh ODEs for $\nabla\mathbf{x}_i$. Therefore, the prescribed error bound $\bar{\Delta}$ is valid only for $\mathbf{x}$.

By a manual modification of the generated code, we can also include the extended state $\nabla\mathbf{x}_i$ in the error estimation. This allows us to guarantee correct results within the error bounds also for the sensitivities, but of course the integration may require more time steps because of the larger dimension $n + nh$ of the extended ODE.

Different methods to compute the criteria and their derivatives at the final timestep have been investigated. The simplest approach is to restrict the size of the final timestep. This can easily be done by checking the new proposed time $t_{i+1} = t_i + h_{i+1}$ after every step. If $t_{i+1} > t^1$, we restrict h_{i+1} to be $t^1 - t_i$. As described in (2.11), ∇t^1 will be zero and no correction (2.8) is required to obtain the correct results.

As an alternative, we used interpolation. Here the integration is stopped as soon as $t_i > t^1$, and the criteria and derivatives at the final time t^1 are computed by an interpolation between the last two intgration steps, for example, the linear

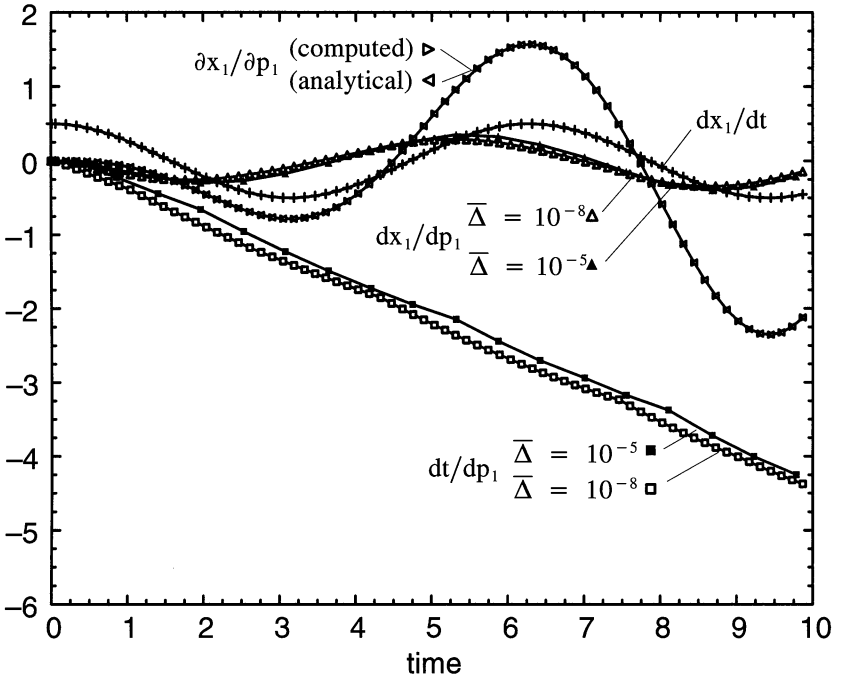

FIGURE 4. Trajectories for different time discretizations

interpolation

$$\mathbf{x}^1 = \mathbf{x}_{i-1} + (\mathbf{x}_i - \mathbf{x}_{i-1})\frac{t^1 - t_{i-1}}{t_i - t_{i-1}}. \tag{3.15}$$

As expected, the results for the criteria are correct, but the results for their gradients are wrong if the correction (2.8) is not applied. The use of higher-order interpolation schemes leads to different values for $\nabla \mathbf{x}^1$ and ∇t^1, but the value for $\partial \mathbf{x}^1 / \partial \mathbf{p}^T$ computed from (2.8) does not change appreciably.

If the final time is not known explicitly, but is determined implicitly by a final condition $H^1(t^1, \mathbf{x}^1, \mathbf{p}) = 0$, a root search has to be performed as described in [20]. The final timestep is not actually performed until the root search determined the final time with sufficient accuracy. Therefore, not the root search algorithm itself but only its result influences the final state. Because the final timestep will be chosen to reach t^1, the results for both the criteria and their gradients need no correction (2.8). This issue is described in detail in [9].

3.4. **Multistep integration algorithm.** The four algorithms described so far are robust, but for production codes, more sophisticated algorithms typically are used. In multibody dynamics the Shampine-Gordon algorithm [20] serves as standard solver and has proved its reliability and efficiency in many applications. It is a multistep algorithm, in which the information already available from previous steps is used to predict further steps. Not only is the stepsize adjusted adaptively, but also the order of extrapolation polynomials is controlled by local error estimates. For trajectories that are not too rough (i.e., nonstiff systems), high polynomial orders and large stepsizes are obtained.

TABLE 3. Relative errors in the states for different prescribed error bounds

Error Bound	ψ_1	ψ_2	$\frac{\partial\psi_1}{\partial p_1}$	$\frac{\partial\psi_1}{\partial p_2}$	$\frac{\partial\psi_2}{\partial p_1}$	$\frac{\partial\psi_2}{\partial p_2}$
.1e-2	.11e-2	.66e-2	.12e-1	.11	.11e-1	.18e-1
.1e-5	.17e-4	.66e-5	.16e-5	.82e-4	.22e-4	.23e-4
.1e-8	.15e-7	.15e-8	.59e-8	.79e-7	.14e-7	.60e-8

The integration algorithm consists of about 900 lines of rather complicated Fortran code. Therefore, a manual modification of the code to ensure $\nabla h_i = 0$ is not deemed to be a reliable approach, and the a posteriori error correction (2.8) is applied to the ADIFOR-generated derivative code. This leads to the correct results; see Table 3.

The correct final time in the investigated version of the Shampine-Gordon algorithm is computed by using a clever interval bisection routine as described in [20]. This guarantees that the evaluation of the last accepted step is at the final time and $\nabla t^1 = 0$.

4. APPLICATION TO A TECHNICAL SYSTEM

To allow comparisons with analytical results, we kept the example simple, but it was already possible to show the properties of the differentiated integration algorithms and various pitfalls that have to be considered. A more complicated example from robotics will be presented in this section to show that the presented effects also allow a correct handling of interesting real-world problems.

The robot in Figure 5 consists of seven bodies, has five degrees of freedom (i.e., it allows five independent motions), and is described by ten ODEs. It is described in detail in [2], where the sensitivity of the position of the end effector at the final time with respect to disturbances in several system parameters $\mathbf{p} = [F_{1z}, L, t_{end}, m_2, I_{3zz}]^T$ is investigated. F_{1z} is a driving force, L a geometrical length, t_{end} the final time, m_2 a mass, and I_{3zz} a component of the inertia tensor. For optimization purposes, additional criteria such as minimal energy consumption or minimal process time are interesting, but for clarity here we restrict ourselves to only one criterion. The results are verified by using the adjoint variable method (AVM) and very costly finite-difference approximations with adaptive order control [2].

The reference criterion and reference gradient obtained by using the adjoint variable method with integration error tolerances near machine accuracy is for the component $\partial\psi/\partial p_1$ as follows:

$$\psi = -4.136636, \quad \frac{\partial\psi}{\partial p_1} = 0.0186126. \tag{4.1}$$

Usually it is not required to compute the gradients to such high accuracy. If the relative and absolute error bounds for the Shampine-Gordon integration algorithm are chosen as relerr=abserr=10^{-8}, we get the following errors in the component $\partial\psi/\partial p_1$:

AVM: relerr $= 8.06 \cdot 10^{-7}$, abserr $= 1.50 \cdot 10^{-8}$,
AD+correction: relerr $= 2.31 \cdot 10^{-7}$, abserr $= 4.30 \cdot 10^{-8}$.

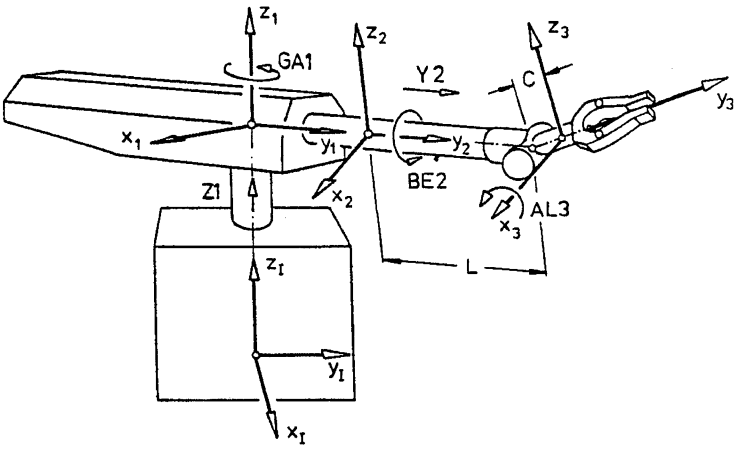

FIGURE 5. Robot

The errors in the other components are similar, and are omitted here. Both gradients are sufficiently accurate, and both methods can be used, for example, for the sensitivity analysis of multibody systems. Also, for all other components, correct results are computed with both methods.

In general, the computation of the adjoint variable gradients is more efficient, but they are based on a hand-coded, highly optimized algorithm whose implementation took man-years, while the AD-generated code is fairly simple to create and requires (including the result verification) much less time, at the expense of a less efficient execution.

We note, however, that, while usually even big codes can be run through ADIFOR within a few days, the time for the verification of the gradients can be much higher. Unless both the algorithm and its implementation are well understood, one should check the results carefully. This need for verification is not due to potential weaknesses of AD tools, but due to the fact that AD differentiates an algorithm without any knowledge of the mathematics that underlie the algorithm. This is both a strength, as programs of arbitrary size can be handled (ADIFOR has successfully differentiated codes of 120,000 lines in length, and produced the desired results), but also a potential weakness, as the discussions so far have shown. Thus, AD is no "silver bullet," but we believe that, at least from our experience in multibody optimization, it does substantially ease the effort required for derivative computation while delivering acceptable performance. Finite difference approximations are in our experience inacceptable due to their lack of efficiency and reliability. The other alternative, the development of specialized code for the gradient computation, is costly in terms of human effort, but can be justified when maximal efficiency is a major design goal, for example, in the development of the adjoint variable method. It is worth emphasizing though, that the implementation of the adjoint variable method is closely tied to a particular integration scheme. It is cumbersome to exchange the integration algorithm, whereas AD techniques allow integration schemes to be substituted quite easily.

5. Conclusions

From the investigations described above, we can summarize our conclusions in three groups:

- The numerical behavior of the criteria *and* the gradient computation must be studied carefully. It is not obvious, for example, that the stepsize control is determined only by the state variables required for the criteria computation; and, therefore, the errors introduced in the state variables for the gradients may be bigger than the prescribed error bounds for the state variables for the criteria. Similar issues arise in the context of applying AD to iterative solvers [10]. In our experience, this behavior is acceptable in many practical applications because for optimization purposes, for example, it is often sufficient to compute the gradients with lesser accuracy than the criteria.
- The formulation of the gradient equations for the actual computation performed (versus their counterpart in the world of continuous mathematics), such as (2.4), may be subtle. For example, the formulation of implicit, parameter-dependent final conditions instead of a fixed final time would introduce new dependencies of the criteria from the intermediate time steps. Also, time discretization and stepsize control are likely not the only influencing factors from numerics. Features such as variable-order polynomial interpolations and projections depend also on the input quantities, are assigned gradients by the AD tools, and therefore influence the finally computed gradients. Corrections of AD-computed gradients are required to arrive at the mathematically desired results, and the remarks given here for the adaptive time discretization thus may need to be extended to handle other auxiliary variables in an algorithm.
- Despite the fact that the application of plain AD often yields the right results, the inclusion of expert knowledge can highly improve the performance and numerical behavior. If, in our example, the differentiation of the stepsize control in the AD-generated code is switched off, we can compute correct gradients more efficiently. However, these modifications require a lot of knowledge about the problem and the gradient computation. Thus, even if AD tools provide annotation capabilities that allow a user to treat certain variables as constant with respect to differentiation, one still needs to be careful not to miss any dependencies.

Thus, while the work presented here allowed us to obtain the desired derivatives from an algorithm relevant for practical problems such as the Shampine-Gordon algorithm, we may definitely not conclude that the a posteriori correction of (2.8) is sufficient for all other integration algorithms as well. AD tools such as ADIFOR or ADIC [5] allow the differentiation of arbitrary complex codes, but for each of them, one must decide whether and which modifications or a posteriori corrections are required to obtain correct results. General rules are hardly possible, but we expect that the work presented here will cover a fair number of cases. Moreover, the work helps to sharpen the user's eyes for other possible sources of "errors" arising from the discrepancy between the derivatives of the integration algorithm and the derivatives of the solution that is being approximated by this algorithm.

Acknowledgments

We thank the anonymous referee for pointing out a major oversight in the initial draft.

References

[1] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, *Computational differentiation: Techniques, applications, and tools*, SIAM, Philadelphia, 1996. MR **97h:**65005

[2] Dieter Bestle and Peter Eberhard, *Analyzing and optimizing multibody systems*, Mechanical Structures and Machinery **20** (1992), no. 1, 67–92.

[3] Christian H. Bischof and Mohammad R. Haghighat, *On hierarchical differentiation*, Computational Differentiation: Techniques, Applications, and Tools (Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, eds.), SIAM, Philadelphia, 1996, pp. 83–94.

[4] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering **3** (1996), no. 3, 18–32.

[5] Christian Bischof, Lucas Roh, and Andrew Mauer, *ADIC — An extensible automatic differentiation tool for ANSI-C*, Software–Practice and Experience **27** (1997), no. 12, 1427–1456.

[6] John C. Butcher, *The numerical analysis of ordinary differential equations (Runge-Kutta and general linear methods)*, John Wiley and Sons, New York, 1987. MR **88d:**65002

[7] J. Cash and A. Karp, *A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides*, ACM Transactions on Mathematical Software **16** (1990), no. 3, 201–222. CMP 91:01

[8] Evin Cramer, Paul Frank, Gregory Shubin, John Dennis, and Robert Michael Lewis, *Problem formulations for multidisciplinary optimization*, SIAM J. Optimization **4** (1994), no. 4, 754–776. MR **95h:**90002

[9] Peter Eberhard, *Zur Mehrkriterienoptimierung von Mehrkörpersystemen*, VDI Fortschritt-Berichte **11** (1996), no. 227.

[10] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson, *Derivative convergence of iterative equation solvers*, Optimization Methods and Software **2** (1993), 321–355.

[11] Andreas Griewank and Shawn Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, Automatic Differentiation of Algorithms: Theory, Implementation, and Application (Andreas Griewank and George F. Corliss, eds.), SIAM, Philadelphia, 1991, pp. 126–135. MR **92i:**65223

[12] Andreas Griewank, *On automatic differentiation*, Mathematical Programming: Recent Developments and Applications (Amsterdam), Kluwer Academic Publishers, 1989, pp. 83–108. MR **92k:**65002

[13] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving ordinary differential equations I: Nonstiff problems*, Springer Verlag, New York, 1987. MR **87m:**65005

[14] E. Hairer and G. Wanner, *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, Springer Series in Computational Mathematics, vol. 14, Springer-Verlag, New York, 1991. MR **92a:**65016

[15] Alan Hindmarsh, *ODEPACK, a systematized collection of ODE solvers*, pp. 55–64, North-Holland, Amsterdam, 1983. CMP 16:15

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in Fortran: The art of scientific computing*, 2nd ed., Cambridge University Press, 1992. MR **93i:**65001a

[17] Louis B. Rall, *Automatic differentiation: Techniques and applications*, Lecture Notes in Computer Science, vol. 120, Springer Verlag, Berlin, 1981.

[18] A. Sandu, G. R. Carmichael, and F. A. Potra, *Sensitivity analysis for atmospheric chemistry models via automatic differentiation*, Atmospheric Environment **31** (1997), no. 3, 475–489.

[19] Granville Sewell, *The numerical solution of ordinary and partial differential equations*, Academic Press, San Diego, California, 1988. MR **89i:**65001

[20] L. Shampine and M. Gordon, *Computer solution of ordinary differential equations*, Freeman, San Francisco, 1975. MR **57:**18104

[21] J. Stoer and R. Bulirsch, *Introduction to numerical analysis*, Springer Verlag, New York, 1980. MR **83d:**65002

Institute B of Mechanics, University of Stuttgart, 70550 Stuttgart, Germany
E-mail address: pe@mechb.uni-stuttgart.de

Rheinisch-Westfälische Technische Hochschule Aachen, Seffenter Weg 23, D-52056 Aachen, Germany
E-mail address: bischof@rz.rwth-aachen.de