Technical Report TR-08-08
Computational Science Laboratory
Department of Computer Science
Virginia Tech

On the discrete adjoints of adaptive time stepping algorithms

Mihai Alexe Adrian Sandu

April 2008

On the discrete adjoints of adaptive time stepping algorithms ⋆

Mihai Alexe * and Adrian Sandu

Computational Science Laboratory,
Department of Computer Science,
Virginia Polytechnic Institute and State University,
2202 Kraft Drive, Blacksburg, VA, 24060, USA.

Abstract

We investigate the behavior of adaptive time stepping numerical algorithms under the reverse mode of automatic differentiation (AD). By differentiating the time step controller and the error estimator of the original algorithm, reverse mode AD generates spurious adjoint derivatives of the time steps. The resulting discrete adjoint models become inconsistent with the adjoint ODE, and yield incorrect derivatives. To regain consistency, one has to cancel out the contributions of the non-physical derivatives in the discrete adjoint model. We demonstrate that the discrete adjoint models of one-step, explicit adaptive algorithms, such as the Runge–Kutta schemes, can be made consistent with their continuous counterparts using simple code modifications. Furthermore, we extend the analysis to cover second order adjoint models derived through an extra forward-mode differentiation of the discrete adjoint code. Two numerical examples support the mathematical derivations.

Key words: Runge - Kutta methods, discrete adjoints, automatic differentiation, adaptive time stepping, ordinary differential equations (ODEs)

1. Introduction and motivation

Automatic differentiation (AD) [11] is a technique for augmenting computer programs with sensitivity analysis features, e.g., the ability to compute gradients or other type of derivative information. Currently, AD comes in two flavors. The implementation can either exploit programming language-specific features and implement AD via operator overloading (ADOL-C [13], ADMIT/ADMAT [5]), or rely on source-to-source transformations that generate tangent linear or adjoint codes from existing model implementations (ADI-C [3], ADIFOR [2], TAF [10], TAMC [8], TAPENADE [16]).

Any computer program may be viewed as an ordered sequence of simple statements (such as additions, multiplications, or simple function calls like exp()), for which derivative computations are trivial. AD tools break each routine into these simple building blocks, differentiate the basic statements one by one, and then put together the desired derivatives using the chain rule from differential calculus. The *forward mode* of AD allows one to compute derivatives of all program outputs with respect to a chosen input, by propagating

⋆ This work has been supported by the National Science Foundation through the award NSF CCF-0635194.

* Corresponding author. Phone: +1-540-231-6186. Fax: +1-540-231-6075.
Email addresses: malexe@cs.vt.edu (Mihai Alexe), sandu@cs.vt.edu (Adrian Sandu).
URLs: http://csl.cs.vt.edu (Mihai Alexe), http://people.cs.vt.edu/~asandu (Adrian Sandu).

gradient information from one program statement to the next. The resulting *tangent linear* model of the forward algorithm preserves the control flow of the original program. Conversely, the *reverse mode* of AD builds the *discrete adjoint* of the routine that is being differentiated, reversing the original control flow. This allows the computation of derivatives of a single program output with respect to several input variables, using one discrete adjoint call.

The behavior of numerical integration algorithms under automatic differentiation was first studied in the broader context of general iterative solvers [1,12]. It has been shown that derivatives of iterative solvers converge to the derivative of the original solution, under suitable assumptions [12]. Hager [14] investigated Runge–Kutta adjoints for optimal control. He gave the general formulation for the continuous adjoint of a Runge–Kutta scheme, and derived the constraints that need to be satisfied by the forward method coefficients such that the continuous adjoint will retain the order of accuracy of the forward Runge–Kutta method, up to methods of order four.

Discrete adjoints of numerical integration algorithms are attractive, because, unlike solvers for the continuous adjoint equations, they can be generated automatically using AD. Walther [25] proved that explicit Runge–Kutta methods of order $p \leq 4$ retain their order of accuracy under reverse mode AD. Sandu [20] showed that discrete adjoints of explicit and implicit Runge-Kutta methods of any order p, give a p-th order accurate solution to the continuous adjoint ODE. Hager, Sandu and Walther all assume that the time steps are kept fixed throughout the integration.

Adaptivity features induce additional complications when differentiating a numerical integration algorithm. This issue is of considerable importance, since all modern ODE integrators make use of time step controllers, error estimators, or numerical extrapolation to maximize efficiency and speed. Forward mode automatic differentiation of the adaptivity mechanism leads to spurious derivatives, as noted by Eberhard and Bischof [7]. These non-physical gradients introduce large errors in the tangent linear model trajectory, leading to incorrect numerical sensitivities. Analyzing explicit Runge–Kutta - like methods, Eberhard and Bischof proposed a code correction that restores the accuracy of the discrete tangent linear model solution.

In this paper, we investigate the behavior of adaptive ODE integration schemes under reverse-mode automatic differentiation. We show that the discrete adjoints of adaptive methods are inconsistent with the adjoint ODE due to the differentiation of the time step controller. Adjoining results in more complex code than forward mode differentiation, due to the reversal of the original control flow. We build the discrete adjoint of a general adaptive, one-step explicit integration method, and quantify the perturbation introduced by the adjoint time step gradients in the adjoint solution. The discrete adjoint solutions are shown to contain a $\mathcal{O}(1)$ error. Hence, the naive invocation of reverse mode AD yields incorrect gradients. This also holds for a p-th order Runge–Kutta method. One has to eliminate the perturbations induced by AD before one can trust the discrete adjoint solution. We propose two equivalent and easy to implement code corrections, that make the discrete adjoint consistent with its continuous counterpart. Moreover, we extend our analysis to second order adjoints of adaptive integrators, and show how to cancel the influence of the spurious derivatives. Finally, we present two numerical examples that support the mathematical derivations.

2. Forward and adjoint sensitivity analysis

We will consider a dynamical system modeled by an initial-value problem (IVP) of the form

$$\begin{aligned} \dot{x} &= z\big(t, x(t,q), q\big) \ , \ t^0 \leq t \leq t^F \ , \\ x(t^0, q) &= x^0(q) \ , \end{aligned} \tag{2.1}$$

where $x \in \mathbb{R}^{n_x}$ denotes the system state, and $q \in \mathbb{R}^{n_q}$ is a set of system parameters. We assume that the IVP (2.1) is well-posed and $z(t,x,q) : \mathbb{R}^{1+n_x+n_p} \to \mathbb{R}^{n_x}$ is at least twice continuously differentiable in all arguments.

In a vast range of practical problems (arising e.g., in optimal control, mechanical engineering, parameter identification, or sensitivity analysis) we are given a cost functional whose value depends on the dynamical system trajectory:

$$\mathcal{J} = \int_{t^0}^{t^F} \gamma(t,x,q)\, dt\,, \tag{2.2}$$

We are then required to compute the sensitivities of $\mathcal{J}$ with respect to changes in a prescribed set of parameters q, or in the initial conditions x^0. It is known [23] that these derivatives can be obtained from the solutions $s_i(t) = \partial x(t)/\partial q_i$ of the tangent linear model (TLM):

$$\begin{aligned} \dot{s}_i &= \frac{\partial z}{\partial x} s_i + \frac{\partial z}{\partial q_i}\ ,\ t^0 \le t \le t^F\ , \\ s_i(t^0) &= \frac{\partial x^0}{\partial q_i}\ ,\ i = 1 \dots n_q\ , \end{aligned} \tag{2.3}$$

or by solving the adjoint final value problem

$$\begin{aligned} \dot{w} &= -\left(\frac{\partial z}{\partial x}\right)^T w - \left(\frac{\partial \gamma}{\partial x}\right)^T\ ,\ t^F \ge t \ge t^0\ , \\ w(t^F) &= 0\ . \end{aligned} \tag{2.4}$$

It can be shown [4,22,24] that

$$\frac{\partial \mathcal{J}}{\partial q_i} = \int_{t^0}^{t^F} \gamma_x s_i + \gamma_{q_i}\, dt\,, \tag{2.5}$$

where $S = \begin{bmatrix} s_1 & s_2 & \dots & s_{n_q} \end{bmatrix}$ is the sensitivity matrix, and

$$\frac{\partial \mathcal{J}}{\partial q} = \left(w^T S\right)\Big|_{t^0} + \int_{t^0}^{t^F} \gamma_q + w^T z_q\, dt\,. \tag{2.6}$$

Without loss of generality we consider only the case of computing sensitivities with respect to the initial conditions. Assuming time-invariant parameters, we can write (2.1 – 2.2) as an extended ODE system:

$$\begin{aligned} \begin{bmatrix} \dot{x}(t) \\ \dot{q}(t) \\ \dot{\theta}(t) \end{bmatrix} &= \begin{bmatrix} z(t,x,q) \\ 0 \\ \gamma(t,x,q) \end{bmatrix}\ ,\ t^0 \le t \le t^F\ , \\ \begin{bmatrix} x(t^0) \\ q(t^0) \\ \theta(t^0) \end{bmatrix} &= \begin{bmatrix} x^0 \\ q^0 \\ 0 \end{bmatrix}\ . \end{aligned} \tag{2.7}$$

It follows that

$$\mathcal{J} = \theta(t^F)\,. \tag{2.8}$$

We rewrite (2.7) as

$$\begin{aligned} \dot{y} &= F(t,y)\ ,\ t^0 \le t \le t^F\ , \\ y(t^0) &= y^0\,, \end{aligned} \tag{2.9}$$

where

$$y = \begin{bmatrix} x \\ q \\ \theta \end{bmatrix} \in \mathbb{R}^N\ ,\quad F = \begin{bmatrix} z \\ 0 \\ \gamma \end{bmatrix}\ . \tag{2.10}$$

The propagation of small perturbations δy^0 in the initial conditions of (2.9) is governed by the tangent linear model:

$$\begin{aligned} \delta\dot{y} &= \frac{\partial F}{\partial y}(t,y)\,\delta y \ , \quad t^0 \le t \le t^F \ , \\ \delta y(t^0) &= \frac{\partial y}{\partial y^0}(t^0)\,. \end{aligned} \tag{2.11}$$

A small perturbation δy^0 in the initial condition of (2.9) will cause a small change in $\mathcal{J}$:

$$\delta\mathcal{J} = \frac{\partial J}{\partial y^0}\,\delta y^0\,, \tag{2.12}$$

up to first order in δy^0. Thus, we will need N TLM integrations with N linearly independent perturbations to obtain the complete gradient vector. On the other hand, the adjoint of (2.9) can yield the gradient at the cost of a single backward-time integration:

$$\begin{aligned} \dot{\lambda} &= -\left(\frac{\partial F}{\partial y}(t,y)\right)^T \lambda \ , \quad t^F \ge t \ge t^0 \ , \\ \lambda(t^F) &= \left(\frac{\partial \mathcal{J}}{\partial y}\right)^T \big(y(t^F)\big) \ , \end{aligned} \tag{2.13}$$

Then:

$$\frac{\partial \mathcal{J}}{\partial y^0} = \lambda(t^0)\,. \tag{2.14}$$

Higher order derivatives [17] have proved to be useful in areas such as data assimilation and chemistry transport modeling [23,26]. The second order adjoint framework provides sensitivity information in the form of Hessian - vector products. The second order derivatives can be obtained from the final value problem [23]:

$$\begin{aligned} \dot{\sigma} &= -\left(\frac{\partial F}{\partial y}(t,y)\right)^T \sigma - \left(\frac{\partial^2 F}{\partial y^2}(t,y)\otimes\delta y(t)\right)^T \lambda \\ \sigma(t^F) &= \frac{\partial^2 \mathcal{J}}{\partial y^2}(y(t^F))\cdot\delta y(t^F)\,, \end{aligned} \tag{2.15}$$

where δy is the solution of the TLM (2.11). Note that the first and second order adjoint systems are coupled, and their solutions depend on the forward and tangent linear model trajectories. The operator " $\otimes$ " denotes the tensor product

$$\frac{\partial^2 F}{\partial y^2}\otimes\delta y = \left[\sum_{k=1}^{N}\left(\frac{\partial^2 F}{\partial y^2}\right)_{i,j,k}\cdot\delta y_k\right]_{1\le i,j\le N}\,. \tag{2.16}$$

One can then show that

$$\sigma(t^0) = \frac{\partial^2 \mathcal{J}}{\partial y^2}(y(t^0))\cdot\delta y^0\,. \tag{2.17}$$

For cost functionals such as (2.2) and large N, equations (2.11) and (2.13) show that it is considerably more efficient to compute the gradient $\partial\mathcal{J}/\partial y^0$ through the adjoint method, since only one adjoint model solve is required. One can follow two strategies to compute adjoint sensitivities numerically:

a. Use the *differentiate - then - discretize* approach, i.e., derive the adjoint ODE (2.13) analytically, and then solve it using a numerical method, e.g., a p-th order Runge-Kutta method [15]. Note that we can use AD to generate the right-hand side of the adjoint ODE, but the Runge-Kutta method itself is not differentiated [7]. Assuming the numerical scheme is stable, the algorithm will converge to a p-th order accurate approximation to the adjoint $\lambda(t)$. The main drawback of this approach is that it requires the analytical derivation of the adjoint model. This can be a difficult task, and cannot be performed in an automatic fashion.

b. The alternative is to *discretize - then - differentiate* the IVP (2.9). First, one discretizes the forward model equations, and then integrates this discrete model using an adaptive time stepping method. The next step is to build the *discrete adjoint of the numerical integrator* using reverse mode AD. This approach is obviously attractive since discrete adjoints can be generated in an automated fashion. On the other hand, it is not guaranteed that the discrete adjoint of a numerical method is a consistent approximation to the corresponding continuous adjoint ODE.
Assume that the forward integration algorithm converges to a p-th order accurate approximation to the exact solution $y(t^F)$. We investigate numerical methods that *retain accuracy under adjoining if the time step h is kept fixed*, i.e. the discrete adjoint solution λ^n satisfies

$$|\lambda^n - \lambda(t^n)| \sim \mathcal{O}\left(h^p\right) \ , \quad \forall\, n \,, \tag{2.18}$$

as $h \to 0$. This is true for Runge–Kutta methods [20,25], but not for multistep methods [21]. For adaptive algorithms, the time steps taken during the forward integration will depend on the forward solution y. Reverse mode AD will pick up this dependence and generate non-physical adjoint time step gradients. We will show that these spurious derivatives lead to incorrect discrete adjoints. To recover an accurate adjoint solution, one has to eliminate the AD artifacts from the discrete adjoint update, as discussed in sections 3 and 4.

3. A general framework for adaptive-step differentiation

One-step, explicit adaptive integration schemes for (2.9) can be written as

$$y^{n+1} = f\left(y^n, h^n, t^n\right) \tag{3.1a}$$
$$h^{n+1} = g\left(y^n, h^n, t^n\right) \tag{3.1b}$$
$$t^{n+1} = t^n + h^n \ , \qquad 0 \le n \le M-1 \,. \tag{3.1c}$$

Here M denotes the total number of integration steps, i.e. $t^M = t^F$. $f : \mathbb{R}^{N+2} \to \mathbb{R}^N$ is chosen according to some prescribed accuracy criteria, and $g : \mathbb{R}^{N+2} \to \mathbb{R}$ is a step size controller, used to compute the new step size h^{n+1} based on an estimate of the local error. The initial time step h^0 is constant. Superscripts are used to indicate discrete time moments, while subscripts denote partial derivatives, unless noted otherwise. Since we are now dealing with discrete models, the cost function $\mathcal{J}$ has to be approximated in terms of the numerical solution. We consider a discrete approximation of the form

$$\widehat{\mathcal{J}} = \widehat{\mathcal{J}}\left(y^M\right) \,. \tag{3.2}$$

We consider methods with the following properties. For a fixed time step $h^n = h$, the method

$$y^{n+1} = f(y^n, h, t^n) \ : \tag{3.3}$$

a. is asymptotically consistent with the adjoint ODE (2.13),
b. is accurate of order p, and
c. retains its order of accuracy under forward or reverse-mode differentiation, i.e., its tangent linear and adjoint formulations are p-th order accurate numerical schemes for the solution of the tangent linear and adjoint ODE, respectively.

Any p-th order Runge–Kutta method satisfies these assumptions [20].

The discrete adjoint of an integration scheme with above properties reads [23]:

$$\lambda^n = \left(\frac{\partial f}{\partial y}\left(y^n, h^n, t^n\right)\right)^T \lambda^{n+1} \ , \quad M-1 \ge n \ge 0 \,,$$
$$\lambda^M = \frac{\partial \widehat{\mathcal{J}}}{\partial y^M} \,. \tag{3.4}$$

We will show that the reverse mode of AD applied to (3.1a – 3.1c) does *not* yield (3.4). Instead, the AD engine introduces spurious derivatives of the time step controller into the discrete adjoint solution. Our correction eliminates the AD artifacts.

Consider the integrator state vector for (3.1a – 3.1c):

$$\boldsymbol{v}^n = \begin{bmatrix} y^n \\ h^n \\ t^n \end{bmatrix} . \tag{3.5}$$

The derivation of the tangent linear model of (3.1a – 3.1c) amounts to constructing the local Jacobian matrix [9] for the state evolution

$$\boldsymbol{v}^n \to \boldsymbol{v}^{n+1} . \tag{3.6}$$

Thus, the TLM of (3.1a) – (3.1c) has the following form (see the appendix for the convention used to represent function derivatives):

$$\begin{bmatrix} \delta y^{n+1} \\ \delta h^{n+1} \\ \delta t^{n+1} \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial y} & \frac{\partial f}{\partial h} & \frac{\partial f}{\partial t} \\ \frac{\partial g}{\partial y} & \frac{\partial g}{\partial h} & \frac{\partial g}{\partial t} \\ \mathbf{0}_{1\times N} & 1 & 1 \end{bmatrix} \begin{bmatrix} \delta y^n \\ \delta h^n \\ \delta t^n \end{bmatrix} , \quad 0 \le n \le M-1 . \tag{3.7}$$

All partial derivatives are evaluated at (y^n, h^n, t^n). The adjoint model is built by transposing the Jacobian of (3.6). Denote the adjoint state vector at step n by

$$\bar{\boldsymbol{\lambda}}^n = \begin{bmatrix} \lambda^n \\ \mu^n \\ \nu^n \end{bmatrix} . \tag{3.8}$$

The n-th step in the discrete adjoint of (3.1a – 3.1c) can be written as

$$\begin{bmatrix} \lambda^n \\ \mu^n \\ \nu^n \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial f}{\partial y}\right)^T & \left(\frac{\partial g}{\partial y}\right)^T & \mathbf{0}_{N\times 1} \\ \left(\frac{\partial f}{\partial h}\right)^T & \frac{\partial g}{\partial h} & 1 \\ \left(\frac{\partial f}{\partial t}\right)^T & \frac{\partial g}{\partial t} & 1 \end{bmatrix} \begin{bmatrix} \lambda^{n+1} \\ \mu^{n+1} \\ \nu^{n+1} \end{bmatrix} . \tag{3.9}$$

This leads to:

$$\lambda^n = \left(\frac{\partial f}{\partial y}\right)^T \lambda^{n+1} + \left(\frac{\partial g}{\partial y}\right)^T \mu^{n+1} \tag{3.10a}$$

$$\mu^n = \left(\frac{\partial f}{\partial h}\right)^T \lambda^{n+1} + \frac{\partial g}{\partial h}\,\mu^{n+1} + \nu^{n+1} \tag{3.10b}$$

$$\nu^n = \left(\frac{\partial f}{\partial t}\right)^T \lambda^{n+1} + \frac{\partial g}{\partial t}\,\mu^{n+1} + \nu^{n+1} , \quad M-1 \ge n \ge 0 . \tag{3.10c}$$

The last term in the adjoint update (3.10a) is a side-effect of AD, generated by the differentiation of the time step controller mechanism (3.1b). μ is a spurious adjoint derivative of the time step h. These perturbations can generate $\mathcal{O}(1)$ errors in the discrete adjoint solution at t^0:

$$\lambda^0 = \lambda(t^0) + \mathcal{O}(1) . \tag{3.11}$$

In section 4 we prove that this happens with Runge-Kutta methods. The numerical experiments in section 6 confirm this conclusion (see Figure 1 and Figure 2).

We present two approaches to eliminating the spurious derivatives. One can:

(i) zero out the non-physical adjoint derivatives:

$$\mu^{n+1} \leftarrow 0\,, \tag{3.12}$$

before the update (3.10a), or

(ii) implement a correction of the form:

$$\lambda^n \leftarrow \lambda^n - \left(\frac{\partial g}{\partial y}\right) \mu^{n+1}\,. \tag{3.13}$$

Suppose that (3.1b) is implemented in FORTRAN as

```
subroutine g (t,y,h,hNew)
```

with `h` $= h^n$ and `hNew` $= h^{n+1}$. A source-to-source AD tool such as TAMC [8] or TAPENADE [16] (with `t` $= t^n$, `y` $= y^n$, and `h` $= h^n$ chosen as independent variables, and `hNew` $= h^{n+1}$ the dependent variable) will generate:

```
subroutine adg (t,y,h,adh,ady,adt,adhNew) ,
```

with `adh` $= \mu^n$ and `adhNew` $= \mu^{n+1}$. Since the value of `adhNew` is lost after the call to `adg()`, we need to save it into a temporary variable and reuse it in (3.13):

```
tmp ← adhNew .
```

We now post-process the adjoint trajectory λ^n right after the call to `adg()` inserted by AD by calling

```
call adg (t,y,h,0,ady,0,-tmp) .
```

This call implements (3.13). Since an evaluation of g is computationally inexpensive relative to a call to f, and the additional cost of adjoining is not greater than that of a (small) constant number of forward model evaluations [11], the overhead of (3.13) is small compared to the overall cost of an adjoint Runge–Kutta step.

4. Explicit adaptive Runge–Kutta methods

We now focus on explicit Runge–Kutta methods. Let $k_1, k_2 \ldots k_r \in \mathbb{R}^N$ be the Runge–Kutta stages, which are now part of the discrete integrator state. Also, $k \equiv [k_1 \;\; k_2 \;\; \ldots \;\; k_r] \in \mathbb{R}^{N\times r}$. As in the previous section, we will identify the integrator state variables at t^n by the superscript n.

The Runge–Kutta method can be written as follows:

$$k = \tau\,(y^n, h^n, t^n) \tag{4.1a}$$

$$y^{n+1} = y^n + h^n \sum_{m=1}^{r} b_m\, k_m \tag{4.1b}$$

$$h^{n+1} = h^n \cdot \tilde{g}\,(y^n, t^n, k) \tag{4.1c}$$

$$t^{n+1} = t^n + h^n\;, \qquad 0 \le n \le M-1\,. \tag{4.1d}$$

Here $b \in \mathbb{R}^r$ denote the Runge–Kutta weights. We assume that k^{n+1} are the stage values after the update (4.1d), and $k^M = 0$. The time step controller (4.1c) has a form that is widely used in practice (see (6.1) for an example). Therefore, the forward integrator state is described by the following vector:

$$\boldsymbol{v}^n = \begin{bmatrix} k_1^n \\ \vdots \\ k_m^n \\ y^n \\ h^n \\ t^n \end{bmatrix} . \tag{4.2}$$

The Runge–Kutta discrete adjoint consistent with the adjoint ODE (2.13) is computed using the chain rule of differentiation:

$$\begin{aligned} \lambda^n &= \left(\boldsymbol{I}_N + h^n \sum_{m=1}^{r} b_m \frac{\partial k_m}{\partial y}\left(t^n, h^n, y^n\right) \right)^T \lambda^{n+1} , \quad M-1 \geq n \geq 1 , \\ \lambda^M &= \frac{\partial \widehat{\mathcal{J}}}{\partial y^M} . \end{aligned} \tag{4.3}$$

Next, we will show that the discrete adjoint of (4.1a) – (4.1d) given by AD is *different* from (4.3).

Since both forward-mode differentiation and adjoining can be performed line by line [11], we can first build the TLM for (4.1a) and then for (4.1b) – (4.1d). This mimics the behavior of AD, since it necessary to account for the dependency arising between (4.1a) and (4.1b) through the stage values k. Hence, the n-th step of the TLM reads:

$$\begin{bmatrix} \delta k_1^{n+1/2} \\ \vdots \\ \delta k_r^{n+1/2} \\ \delta y^{n+1/2} \\ \delta h^{n+1/2} \\ \delta t^{n+1/2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0}_N & \dots & \boldsymbol{0}_N & \dfrac{\partial k_1}{\partial y} & \dfrac{\partial k_1}{\partial h} & \dfrac{\partial k_1}{\partial t} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{0}_N & \dots & \boldsymbol{0}_N & \dfrac{\partial k_r}{\partial y} & \dfrac{\partial k_r}{\partial h} & \dfrac{\partial k_r}{\partial t} \\ \boldsymbol{0}_N & \dots & \boldsymbol{0}_N & \boldsymbol{I}_N & \boldsymbol{0}_{N\times 1} & \boldsymbol{0}_{N\times 1} \\ \boldsymbol{0}_{1\times N} & \dots & \boldsymbol{0}_{1\times N} & \boldsymbol{0}_{1\times N} & 1 & 0 \\ \boldsymbol{0}_{1\times N} & \dots & \boldsymbol{0}_{1\times N} & \boldsymbol{0}_{1\times N} & 0 & 1 \end{bmatrix} \begin{bmatrix} \delta k_1^n \\ \vdots \\ \delta k_r^n \\ \delta y^n \\ \delta h^n \\ \delta t^n \end{bmatrix} \tag{4.4}$$

$$\begin{bmatrix} \delta k_1^{n+1} \\ \vdots \\ \delta k_r^{n+1} \\ \delta y^{n+1} \\ \delta h^{n+1} \\ \delta t^{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_N & \dots & \boldsymbol{0}_N & \boldsymbol{0}_N & \boldsymbol{0}_{N\times 1} & \boldsymbol{0}_{N\times 1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{0}_N & \dots & \boldsymbol{I}_N & \boldsymbol{0}_N & \boldsymbol{0}_{N\times 1} & \boldsymbol{0}_{N\times 1} \\ h^n b_1 \boldsymbol{I}_N & \dots & h^n b_r \boldsymbol{I}_N & \boldsymbol{I}_N & \displaystyle\sum_m b_m k_m & \boldsymbol{0}_{N\times 1} \\ h^n \dfrac{\partial \tilde{g}}{\partial k_1} & \dots & h^n \dfrac{\partial \tilde{g}}{\partial k_r} & h^n \dfrac{\partial \tilde{g}}{\partial y} & \tilde{g} & h^n \dfrac{\partial \tilde{g}}{\partial t} \\ \boldsymbol{0}_{1\times N} & \dots & \boldsymbol{0}_{1\times N} & \boldsymbol{0}_{1\times N} & 1 & 1 \end{bmatrix} \begin{bmatrix} \delta k_1^{n+1/2} \\ \vdots \\ \delta k_r^{n+1/2} \\ \delta y^{n+1/2} \\ \delta h^{n+1/2} \\ \delta t^{n+1/2} \end{bmatrix} . \tag{4.5}$$

Here $\boldsymbol{I}_N$ denotes the N-by-N identity matrix, $\boldsymbol{0}_N$ stands for the zero N-by-N matrix, and the index $n+1/2$ indicates an intermediate step. We note that all entries of the above Jacobian matrices are evaluated at (t^n, y^n, h^n). The adjoint variables form the costate vector:

$$\bar{\boldsymbol{\lambda}}^n = \begin{bmatrix} \psi_1^n \\ \vdots \\ \psi_r^n \\ \lambda^n \\ \mu^n \\ \nu^n \end{bmatrix} . \tag{4.6}$$

The discrete adjoint follows immediately by transposing the Jacobian matrices and reversing the direction of integration:

$$\begin{aligned}
\psi_m^{n+1/2} &= h^n b_m \lambda^{n+1} + h^n \left(\frac{\partial \tilde{g}}{\partial k_m}\right)^T \mu^{n+1} \ , \ m = 1 \dots r \\
\lambda^{n+1/2} &= \lambda^{n+1} + h^n \left(\frac{\partial \tilde{g}}{\partial y}\right)^T \mu^{n+1} \\
\mu^{n+1/2} &= \mu^{n+1} + \tilde{g}\mu^{n+1} + \sum_{m=1}^{r} b_m k_m^T \lambda^{n+1} \\
\nu^{n+1/2} &= h^n \frac{\partial \tilde{g}}{\partial t} \mu^{n+1} + \nu^{n+1}
\end{aligned} \tag{4.7}$$

$$\begin{aligned}
\psi^n &= \psi^{n+1/2} \\
\lambda^n &= \sum_{m=1}^{r} \left(\frac{\partial k_m}{\partial y}\right)^T \psi_m^{n+1/2} + \lambda^{n+1/2} \\
\mu^n &= \sum_{m=1}^{r} \left(\frac{\partial k_m}{\partial h}\right)^T \psi_m^{n+1/2} + \mu^{n+1/2} \\
\nu^n &= \sum_{m=1}^{r} \left(\frac{\partial k_m}{\partial t}\right)^T \psi_m^{n+1/2} + \nu^{n+1/2} \ , \qquad M - 1 \geq n \geq 0 \, .
\end{aligned} \tag{4.8}$$

Thus AD computes the following discrete adjoint update:

$$\begin{aligned}
\lambda^n &= \left(\boldsymbol{I}_N + h^n \sum_{m=1}^{r} b_m \left(\frac{\partial k_m}{\partial y}\right)^T \right) \lambda^{n+1} \\
&\quad + h^n \mu^{n+1} \left(\sum_{m=1}^{r} b_m \left(\frac{\partial k_m}{\partial y}\right)^T \left(\frac{\partial \tilde{g}}{\partial k_m}\right)^T + \left(\frac{\partial \tilde{g}}{\partial y}\right)^T \right) \\
&= \left(\boldsymbol{I}_N + h^n \sum_{m=1}^{r} b_m \left(\frac{\partial k_m}{\partial y}\right)^T \right) \lambda^{n+1} + \mathcal{O}\left(h^n\right) .
\end{aligned} \tag{4.9}$$

All partial derivatives are evaluated at (y^n, h^n, t^n). The update (4.9) is clearly different from (4.3). The $\mathcal{O}\left(h^n\right)$ perturbations introduced at each step in the discrete adjoint λ^n add up to a $\mathcal{O}\left(1\right)$ perturbation in the adjoint solution λ^0:

$$\left|\lambda^0 - \lambda(t^0)\right| = \mathcal{O}(1) \, . \tag{4.10}$$

To eliminate all perturbations, one can zero out the adjoint derivative μ^{n+1}, as in (3.12). Alternatively, we can insert

$$\psi_m^{n+1/2} = \psi_m^{n+1/2} - h^n \left(\frac{\partial \tilde{g}}{\partial k_m} \right)^T \mu^{n+1} \;,\; m = 1 \ldots r$$
$$\lambda^{n+1/2} = \lambda^{n+1/2} - h^n \left(\frac{\partial \tilde{g}}{\partial y} \right)^T \mu^{n+1} \,. \tag{4.11}$$

after (4.7). The FORTRAN implementation of (4.11) is based on a second call of `adg` (the adjoint of (4.1c)), and is very similar to the one described in the previous section. Both correction strategies will result in a discrete adjoint solution that is a p-th order accurate approximation to the true adjoint $\lambda(t^0)$. Section 6 shows the divergent AD adjoint, as well as the convergence of the corrected solution, for a 5-th order Runge-Kutta method.

5. Discrete second order adjoints

It is known that second order adjoints can be computed either by two reverse-mode differentiations (adjoint-over-adjoint) or, more efficiently, through a forward mode differentiation of the original model's adjoint (forward-over-adjoint) [11,18]. Thus, it is natural to ask if we can couple the adjoint corrections (3.13) or (4.11) with the tangent linear code modifications described in [7]. We will show that this approach leads to accurate discrete second order adjoints of adaptive numerical integrators.

We will work within the general framework described in section 3. The second order discrete adjoint system allows one to obtain second order derivative information for the cost function (3.2), in the form of Hessian-vector products

$$\frac{\partial^2 \widehat{\mathcal{J}}}{(\partial y^0)^2} \cdot \delta y^0 \,, \tag{5.1}$$

with $\delta y^0 \in \mathbb{R}^N$ an arbitrary vector. We henceforth mark all discrete second-order adjoint variables by an upper dot. Thus, the second order adjoint state vector at t^n has the following structure (compare with 3.8):

$$\dot{\boldsymbol{\lambda}} = \begin{bmatrix} \dot{\lambda} \\ \dot{\mu} \\ \dot{\nu} \end{bmatrix} \,. \tag{5.2}$$

The discrete second order adjoint consistent with the ODE (2.15) reads:

$$\dot{\lambda}^n = \left(\left. \frac{\partial f}{\partial y} \right|_{(y^n, h^n, t^n)} \right)^T \dot{\lambda}^{n+1} + \left(\left. \frac{\partial^2 f}{\partial y^2} \right|_{(y^n, h^n, t^n)} \otimes \delta y^n \right)^T \lambda^{n+1} \quad , \quad M - 1 \geq n \geq 0$$
$$\dot{\lambda}^M = \frac{\partial^2 \widehat{\mathcal{J}}}{(\partial y^M)^2} \cdot \delta y^M \,. \tag{5.3}$$

In (5.3) λ^n satisfies the first order adjoint equation (3.4), and δy^n is solution of the discrete TLM model:

$$\delta y^n = \frac{\partial f}{\partial y} \, \delta y^{n-1} \,, \qquad 1 \leq n \leq M \,. \tag{5.4}$$

Note that the initial tangent linear model state is set to δy^0. Solving (5.3) yields [23]:

$$\dot{\lambda}^0 = \frac{\partial^2 \widehat{\mathcal{J}}}{(\partial y^0)^2} \cdot \delta y^0 \,. \tag{5.5}$$

We now investigate the second order adjoint model of (3.1a – 3.1c) given by two successive invocations of AD. For efficiency [18], we take the forward-over-adjoint path, i.e., we differentiate (3.10a) – (3.10c) in the direction

$$\delta \boldsymbol{v}^0 = \begin{bmatrix} \delta y^0 \\ \delta h^0 \\ \delta t^0 \end{bmatrix}^T . \tag{5.6}$$

Hence, the second order adjoint model generated by AD has the following structure:

$$\begin{aligned}
\dot{\lambda}^n \;=\; & \left(\frac{\partial^2 f}{\partial y^2} \otimes \delta y^n\right)^T \lambda^{n+1} + \left(\frac{\partial f}{\partial y}\right)^T \dot{\lambda}^{n+1} \\
& + \delta h^n \left(\frac{\partial^2 f}{\partial y\, \partial h}\right)^T \lambda^{n+1} + \delta t^n \left(\frac{\partial^2 f}{\partial y\, \partial t}\right)^T \lambda^{n+1} + \left(\frac{\partial g}{\partial y}\right)^T \dot{\mu}^{n+1} \\
& + \mu^{n+1} \left(\frac{\partial^2 g}{\partial y^2}\right)^T \delta y^n + \mu^{n+1} \left(\frac{\partial^2 g}{\partial y\, \partial h}\right)^T \delta h^n + \mu^{n+1} \left(\frac{\partial^2 g}{\partial y\, \partial t}\right)^T \delta t^n
\end{aligned} \tag{5.7a}$$

$$\begin{aligned}
\dot{\mu}^n \;=\; & (\delta y^n)^T \left(\frac{\partial^2 f}{\partial h\, \partial y}\right)^T \lambda^{n+1} + \delta h^n \left(\frac{\partial^2 f}{\partial h^2}\right)^T \lambda^{n+1} + \delta t^n \left(\frac{\partial^2 f}{\partial h\, \partial t}\right)^T \lambda^{n+1} \\
& + \mu^{n+1} \frac{\partial^2 g}{\partial h\, \partial y}\, (\delta y^n)^T + \delta h^n \frac{\partial^2 g}{\partial h^2}\, \mu^{n+1} + \delta t^n \frac{\partial^2 g}{\partial h\, \partial t}\, \mu^{n+1} \\
& + \left(\frac{\partial f}{\partial h}\right)^T \dot{\lambda}^{n+1} + \frac{\partial g}{\partial h}\, \dot{\mu}^{n+1} + \dot{\nu}^{n+1}
\end{aligned} \tag{5.7b}$$

$$\begin{aligned}
\dot{\nu}^n \;=\; & (\delta y^n)^T \left(\frac{\partial^2 f}{\partial t\, \partial y}\right)^T \lambda^{n+1} + \delta h^n \left(\frac{\partial^2 f}{\partial t\, \partial h}\right)^T \lambda^{n+1} + \delta t^n \left(\frac{\partial^2 f}{\partial t^2}\right)^T \lambda^{n+1} + \\
& + (\delta y^n)^T \frac{\partial^2 g}{\partial t\, \partial y}\, \mu^{n+1} + \delta h^n \frac{\partial^2 g}{\partial t\, \partial h}\, \mu^{n+1} + \delta t^n \frac{\partial^2 g}{\partial t^2}\, \mu^{n+1} \\
& + \frac{\partial f}{\partial t}\, \dot{\lambda}^{n+1} + \frac{\partial g}{\partial t}\, \dot{\mu}^{n+1} + \dot{\nu}^{n+1}\,, \qquad M-1 \geq n \geq 0\,.
\end{aligned} \tag{5.7c}$$

The update (5.7) is not identical to (5.3): several spurious terms are added at each time step. Also, note that δy^n gets updated by the AD-generated TLM model (3.7)

$$\delta y^n = \frac{\partial f}{\partial y}\delta y^{n-1} + \frac{\partial f}{\partial h}\delta h^{n-1} + \frac{\partial f}{\partial t}\delta t^{n-1}\,, \qquad 1 \leq n \leq M\,, \tag{5.8}$$

instead of (2.11). This mismatch is another source of errors in the second order discrete adjoint solution $\dot{\lambda}^n$, since the second order adjoint computations need to make use of an accurate tangent linear model trajectory.

To cancel out all the AD-induced perturbations in the discrete second order adjoint, one should:

(i) restore the TLM solution δy^n to the value given by (5.4). This can be done by setting

$$\begin{aligned}
\delta h^n &\leftarrow 0 \\
\delta t^n &\leftarrow 0
\end{aligned} \tag{5.9}$$

before the update (5.7). Alternatively, one can apply the post-processing strategy described in [7] at each time step. Either one of these approaches will give an accurate TLM solution. Note that (5.9) implies that the forward differentiation of (3.10a) – (3.10c) is now performed in the direction of $\delta v^0 = \left[\left(\delta y^0\right)^T \;\; 0 \;\; 0\right]^T$.

(ii) zero out the non-physical first and second order adjoint derivatives of the time step:

$$\begin{aligned}
\mu^{n+1} &\leftarrow 0 \\
\dot{\mu}^{n+1} &\leftarrow 0\,.
\end{aligned} \tag{5.10}$$

After (5.9) and (5.10), the second order adjoint trajectory is restored to the value given by (5.3). For completeness, we remark that an approach equivalent to (5.10) is to implement (3.13), and then post-process [7] the second order adjoint solution.

It is important to note that any assignments to the TLM or first order adjoint variables, such as (5.9) or (5.10), should be introduced *after* the second differentiation. Any assignments that happen before the second order adjoint code generation may influence the behavior of the AD engine and result in unusable code.

6. Numerical experiments

All numerical tests were performed on an Intel Pentium 4 Workstation running Fedora Core Linux, with the Runge–Kutta routines coded in double precision Fortran 90.

We used the 5th order DOPRI5(4) Runge–Kutta method [6] with variable time - stepping in all numerical experiments. DOPRI5(4) uses the stage values to compute two solution approximations of different accuracies at every time step. The first of these two numerical solutions is used to advance the integrator state, and the other (less accurate) solution serves to control the local error and the time step size. This approach significantly lowers the cost of step rejections. The time step controller is based on the following formula [15]:

$$h^{n+1} = h^n \cdot \min\left\{5\,,\, \max\left\{0.2\,,\, 0.9\,\|e^n\|^{-1/5}\right\}\right\}. \tag{6.1}$$

Here $\|e^n\|$ is an estimate for the weighted norm of the local error at step n. This estimate is computed based on the relative ($rtol$) and absolute ($atol$) integration tolerances:

$$\|e^n\| = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{y^n_{err}(i)}{atol(i) + rtol(i)\cdot|y^n(i)|}\right)^2}\,, \tag{6.2}$$

with y_{err} approximating the local error of the Runge–Kutta method.

The reference solutions were obtained by integration of the analytical models, using MATLAB's `ode45`, with $atol = 10^{-13}, rtol = 10^{-12}$.

6.1. *First order adjoint sensitivity analysis*

We first investigate the discrete adjoint of the Prothero - Robinson IVP [19]:

$$\begin{aligned} \dot{y} &= \gamma\,(y - \phi(t)) + \dot{\phi}(t)\,, \\ t^0 &= 0 \le t \le t^F = 2\,, \\ y(t^0) &= [0.5 \quad 0.5]^T\,, \end{aligned} \tag{6.3}$$

with $\gamma = -5$, and

$$\phi(t) = [\sin t \quad \cos t]^T\,. \tag{6.4}$$

We choose:

$$\mathcal{J}(y) = y_1(t^F)\,. \tag{6.5}$$

Hence, the continuous adjoint of (6.3) reads:

$$\begin{aligned} \dot{\lambda} &= \gamma\lambda \quad,\quad t^F \ge t \ge t^0\,, \\ \lambda(t^F) &= [1 \quad 0]^T\,. \end{aligned} \tag{6.6}$$

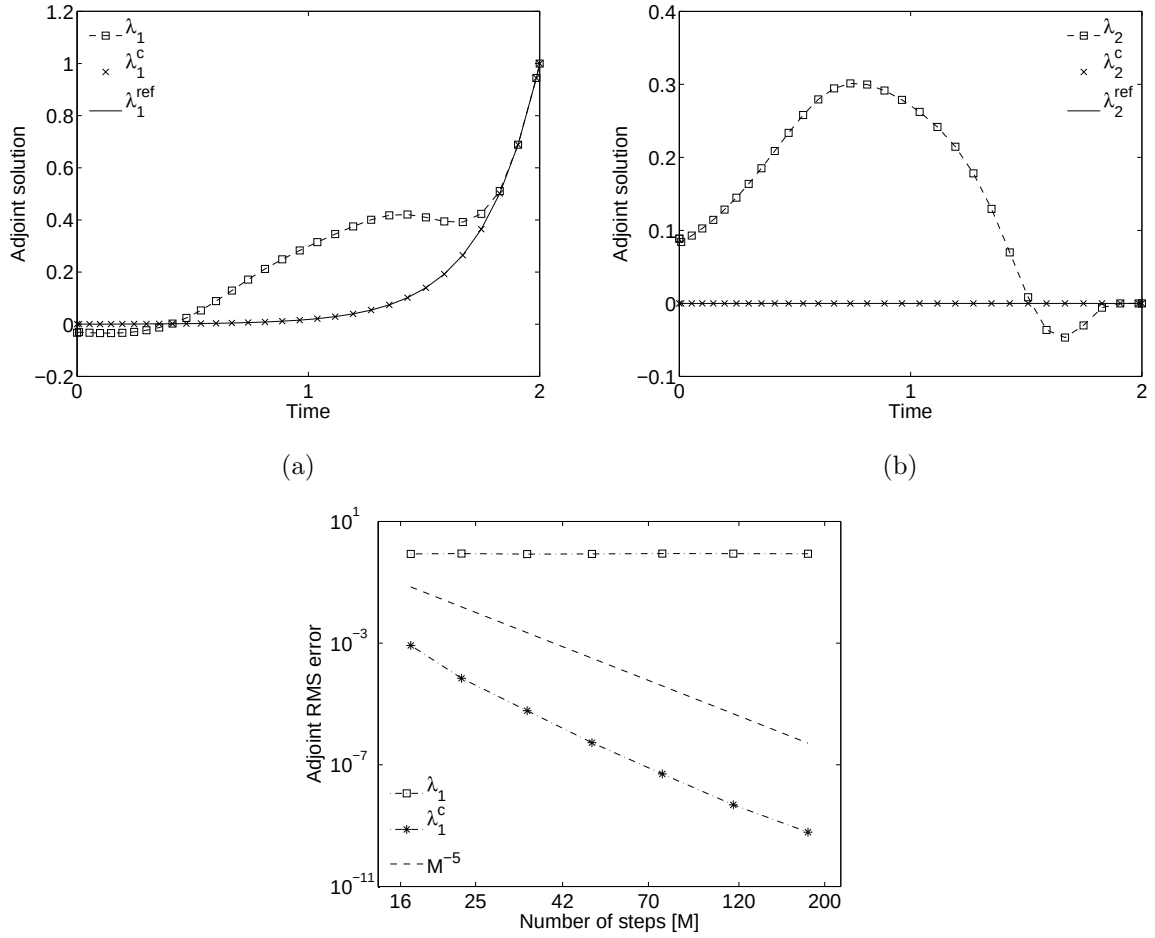

Fig. 1. (a–b) Discrete adjoint trajectories ($atol = rtol = 10^{-7}$), and (c) RMS errors for the system (6.3). The AD adjoints λ_1, λ_2, and the corrected solutions λ_1^c, λ_2^c were computed with DOPRI5(4). The reference solution λ^{ref} was obtained through a backward time integration of the continuous adjoint (6.6), using MATLAB's `ode45` function. It is clear from (c) that the AD discrete adjoint of the DOPRI method is inconsistent with the continuous system (6.6). After the correction, the adjoint solution and the reference adjoint trajectory are visually indistinguishable. As seen in (c), post-processing restores the discrete adjoint solution to full accuracy.

We differentiate our Runge–Kutta implementation using the reverse mode of TAMC. Figure 1 shows the two discrete adjoint solution components and their root-mean-square (RMS) errors, both before (λ_1, λ_2) and after (λ_1^c, λ_2^c) the adjoint correction (3.13) is applied. The RMS errors are computed using the formula:

$$\epsilon_{\mathrm{RMS}} = \sqrt{\frac{1}{M}\sum_{n=0}^{M}\left(\frac{|\lambda^{\mathrm{ref}}(t^n) - \lambda^n|}{\max\left\{|\lambda^{\mathrm{ref}}(t^n)|, |\lambda^n|, tol\right\}}\right)^2}, \tag{6.7}$$

with $tol \approx 10^{-12}$ (to avoid divisions by zero). One can see in 1(a–b) that black-box invocation of AD results in very large errors in both components of the discrete adjoint trajectory. This is consistent with the behavior predicted by the mathematical derivations (4.9). All accuracy of the adjoint solution is lost. After the correction is applied, the accuracy of the discrete adjoint solution matches that of the underlying Runge–Kutta method, as Figure 1(c) illustrates.

Note that the two adjoint corrections (3.12) and (3.13) yield the same adjoint solutions (up to machine roundoff). Figure 1 only shows the numerical results obtained with (3.12).

6.2. *Second order adjoint sensitivity analysis*

For second order sensitivity analysis, we introduce a nonlinear term in the right-hand side of (6.3):

$$\begin{aligned} \dot{y}_1 &= \gamma\,(y_1 - \sin t) + y_2^3 \cos t\,, \\ \dot{y}_2 &= \gamma\,(y_2 - \cos t) - y_1^3 \sin t\,, \\ t^0 = 0 \le t \le t^F &= 2\,, \\ y(t^0) &= [0.5 \quad 0.5]^T\,. \end{aligned} \tag{6.8}$$

Let the cost function $\mathcal{J}$ be defined by (6.5), and

$$\delta y^0 = [1 \quad 0]^T \tag{6.9}$$

in (5.1). Then, the first-order adjoint system for (6.8) has the form

$$\begin{aligned} \dot{\lambda}_1 &= -\gamma\lambda_1 + 3y_1^2\lambda_2 \sin t \\ \dot{\lambda}_2 &= -3y_2^2\lambda_1 \cos t - \gamma\lambda_2 \\ \lambda(t^F) &= [1 \quad 0]^T\,, \end{aligned} \tag{6.10}$$

whereas the second order adjoint model reads:

$$\begin{aligned} \dot{\sigma}_1 &= -\gamma\sigma_1 + 3y_1^2\sigma_2 \sin t + 6y_1\delta y_1\lambda_2 \sin t \\ \dot{\sigma}_2 &= -3y_2^2\sigma_1 \cos t - \gamma\sigma_2 - 6y_2\delta y_2\lambda_1 \cos t \\ \sigma_1(t^F) = \sigma_2(t^F) &= 0\,. \end{aligned} \tag{6.11}$$

Here $\sigma(t)$ denotes the second order adjoint variable, and δy is the solution of the tangent linear model:

$$\begin{aligned} \delta\dot{y}_1 &= \gamma\delta y_1 + 3y_2^2\delta y_2 \cos t \\ \delta\dot{y}_2 &= -3y_1^2\delta y_1 \sin t + \gamma\delta y_2 \\ \delta y(t^0) &= \delta y^0\,. \end{aligned} \tag{6.12}$$

We build the second order adjoint of our DOPRI5(4) implementation through forward over reverse differentiation. The results are shown in Figure 2. As expected from (5.7), the second order adjoint of the DOPRI method is inconsistent with its continuous counterpart (6.11). The reason for this is twofold: the errors in the second order adjoint variables are generated both by the perturbations present in the first order adjoint solution λ (see 4.9), and the spurious derivatives generated during the second (forward) differentiation [7]. Figure 2(a–b) contrasts the discrete solutions before – λ_1,λ_2 – and after – λ_1^c,λ_2^c – the post-processing (5.9 – 5.10), for $atol = rtol = 10^{-7}$. The corrected solution is visually indistinguishable from the reference λ^{ref}, whereas the AD-computed adjoint is several orders of magnitude away from the true solution. Finally, Figure 2 shows the order of accuracy of the post-processed discrete adjoint, which matches that of the DOPRI pair used in the forward model integration.

Figure 2 only shows the solutions obtained after zeroing out all spurious forward and adjoint time step derivatives. Applying (4.11) gives virtually identical results.

7. Conclusions

In this paper, we investigate the behavior of adaptive numerical integration algorithms under the reverse mode of automatic differentiation. To maximize efficiency and reduce computation time, such algorithms

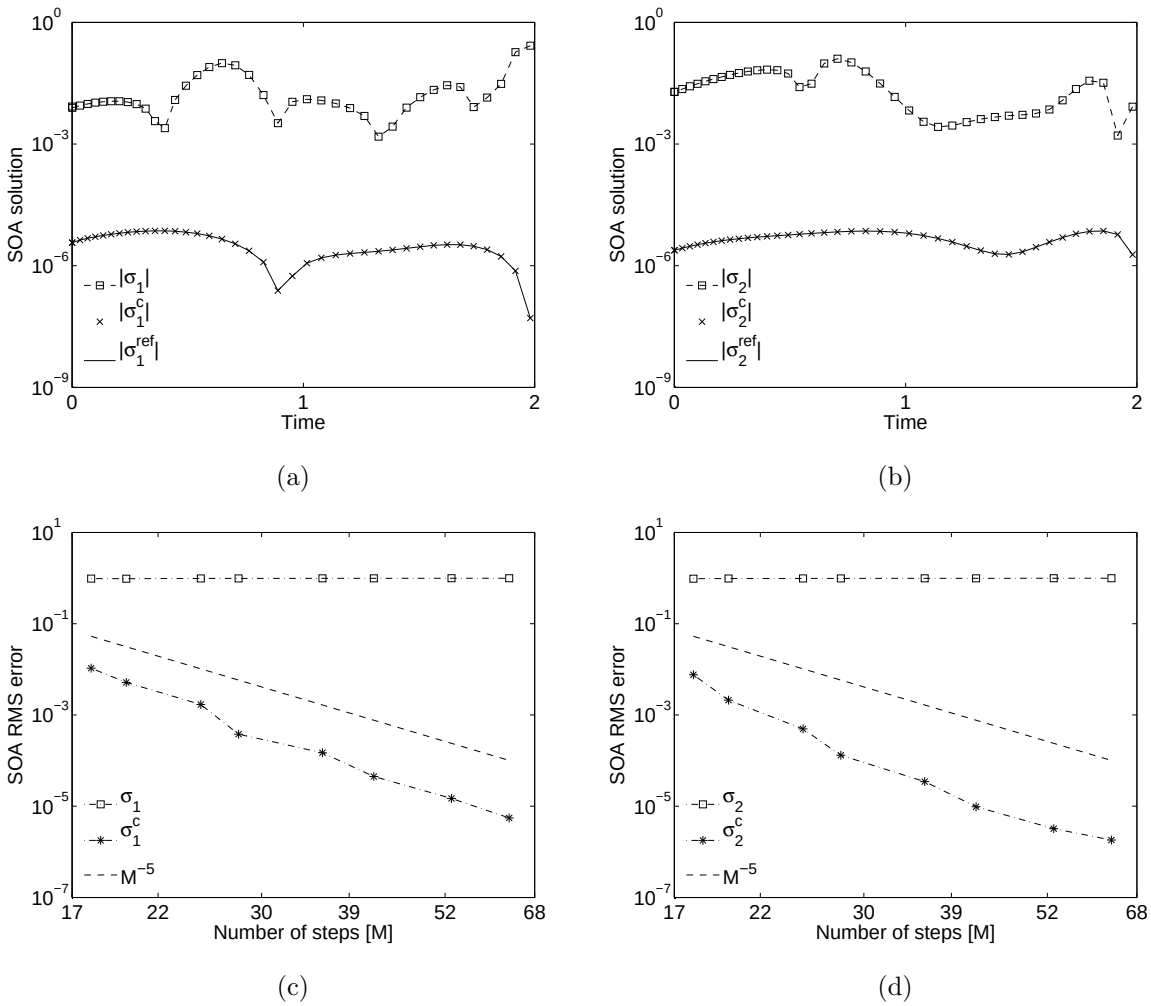

Fig. 2. Discrete second order adjoints (a–b) and RMS errors (c–d) for the IVP (6.8). The AD discrete second order adjoint $\sigma_{1,2}$ differs from the reference trajectory σ^{ref} by several orders of magnitude. However, the corrected solution $\sigma^c_{1,2}$ is visually indistinguishable from σ^{ref} ($atol = rtol = 10^{-7}$). Also, the decrease in the RMS errors of the corrected trajectory (c–d) confirms that canceling the spurious adjoint derivatives yields a 5th order accurate second order adjoint solution.

rely on time step controllers and local error estimators to keep the solution accuracy within user-specified bounds. Since the time steps taken by the forward method now depend on the model solution, the AD mechanism will pick up these dependencies and generate spurious adjoint time and time step derivatives. These non-physical gradients influence the discrete adjoint trajectory at every time step. The perturbations add up and generate a $\mathcal{O}(1)$ error in the final solution. Thus, running the discrete adjoint code as-is will give incorrect sensitivities. Analyzing the adjoint of a general one-step (explicit) adaptive integration scheme, reveals that simple code modifications result in a discrete adjoint solution that has the same order of accuracy as the underlying numerical method. We also look at second-order adjoints of adaptive integrators, obtained through forward-over-adjoint differentiation. The analysis shows that it is possible to obtain accurate second order derivative information through straightforward post-processing of the second order adjoint code. We give two examples that use Dormand and Prince's DOPRI5(4) Runge–Kutta pair. The numerical results fully support the mathematical derivations. The AD-generated Runge–Kutta adjoints are inconsistent with the continuous adjoint ODEs. However, once the adjoint corrections are applied, the first and second order discrete adjoint solutions have the same order of accuracy as the underlying Runge–Kutta algorithm.

APPENDIX: Derivative notation

We employ the following convention when representing function derivatives. Given a vector function of several variables

$$f : \mathbb{R}^m \to \mathbb{R}^p$$
$$f(v_1, v_2, \ldots, v_m) = \begin{pmatrix} f_1(v_1, v_2, \ldots, v_m) \\ f_2(v_1, v_2, \ldots, v_m) \\ \vdots \\ f_p(v_1, v_2, \ldots, v_m) \end{pmatrix}, \tag{7.1}$$

the Jacobian of f reads:

$$\frac{\partial f}{\partial v} = \begin{pmatrix} \frac{\partial f_1}{\partial v_1} & \cdots & \frac{\partial f_1}{\partial v_m} \\ \vdots & \cdots & \vdots \\ \frac{\partial f_p}{\partial v_1} & \cdots & \frac{\partial f_p}{\partial v_m} \end{pmatrix}. \tag{7.2}$$

The Hessian of f is a symmetric 3-tensor containing the second order derivatives of f with respect to all arguments:

$$H_{i,j,k}(v) = \frac{\partial^2 f_i}{\partial v_j \, \partial v_k}, \; 1 \le i \le p, \; 1 \le j, k \le m. \tag{7.3}$$

Two special cases occur frequently in practice. The derivative of a univariate function f ($m = 1$) is a column vector:

$$\frac{\partial f}{\partial v} = \begin{pmatrix} \frac{\partial f_1}{\partial v_1} \\ \vdots \\ \frac{\partial f_p}{\partial v_1} \end{pmatrix}. \tag{7.4}$$

Similarly, if $p = 1$, then we define the gradient of the scalar function f as the row vector

$$\frac{\partial f}{\partial v} = \begin{pmatrix} \frac{\partial f_1}{\partial v_1} & \cdots & \frac{\partial f_1}{\partial v_m} \end{pmatrix}. \tag{7.5}$$

References

[1] T. Beck, Automatic differentiation of iterative processes, in: ICCAM'92: Proceedings of the fifth international conference on Computational and applied mathematics, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 1994.

[2] C. H. Bischof, A. Carle, P. M. Khademi, A. Mauer, The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, Tech. Rep. MCS–P481–1194, Argonne, IL, USA (1994).

[3] C. H. Bischof, L. Roh, A. J. Mauer-Oats, ADIC: an extensible automatic differentiation tool for ANSI-C, Software: Practice & Experience 27 (12) (1997) 1427–1456.

[4] Y. Cao, S. Li, L. Petzold, R. Serban, Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution, SIAM Journal on Scientific Computing 24 (3) (2002) 1076–1089.

[5] T. F. Coleman, A. Verma, ADMIT-1: automatic differentiation and MATLAB interface toolbox, ACM Transactions on Mathematical Software 26 (1) (2000) 150–175.

[6] J. R. Dormand, P. J. Prince, A family of embedded Runge-Kutta formulae, Journal of Computational and Applied Mathematics 6 (1) (1980) 19–26.

[7] P. Eberhard, C. Bischof, Automatic differentiation of numerical integration algorithms, Mathematics of Computation 68 (226) (1999) 717–731.
[8] R. Giering, Tangent linear and Adjoint Model Compiler, Users manual 1.4 (1999).
[9] R. Giering, T. Kaminski, Recipes for adjoint code construction, ACM Transactions on Mathematical Software 24 (4) (1998) 437–474.
[10] R. Giering, T. Kaminski, Applying TAF to generate efficient derivative code of Fortran 77-95 programs, Proceedings in Applied Mathematics and Mechanics 2 (1) (2003) 54–57.
[11] A. Griewank, Evaluating derivatives: principles and techniques of algorithmic differentiation, SIAM, Philadelphia, PA, USA, 2000.
[12] A. Griewank, C. H. Bischof, G. F. Corliss, A. Carle, K. Williamson, Derivative convergence for iterative equation solvers, Optimization Methods and Software 2 (1993) 321–355.
[13] A. Griewank, D. Juedes, J. Utke, Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++, ACM Transactions on Mathematical Software 22 (2) (1996) 131–167.
[14] W. W. Hager, Runge-Kutta methods in optimal control and the transformed adjoint system, Numerische Mathematik 87 (2) (2000) 247–282.
[15] E. Hairer, S. P. Nørsett, G. Wanner, Solving Ordinary Differential Equations: Nonstiff Problems, vol. I of Computational Mathematics, Springer-Verlag, 1993.
[16] L. Hascöet, V. Pascual, TAPENADE 2.1 User's guide, Tech. Rep. 0300, INRIA, Sophia Antipolis, France (2004).
[17] F. X. LeDimet, I. M. Navon, D. Daescu, Second order information in data assimilation, Monthly Weather Review 130 (3) (2002) 629–648.
[18] D. B. Özyurt, P. I. Barton, Cheap second order directional derivatives of stiff ODE embedded functionals, SIAM Journal on Scientific Computing 26 (5) (2005) 1725–1743.
[19] A. Prothero, A. Robinson, On the stability and accuracy of one-step methods for solving stiff systems of ordinary differential equations, Mathematics of Computation 28 (125) (1974) 145–162.
[20] A. Sandu, On the properties of runge-kutta discrete adjoints, in: International Conference on Computational Science (4), 2006.
[21] A. Sandu, On consistency properties of discrete adjoint linear multistep methods, Tech. Rep. TR-07-40, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA (2007).
[22] A. Sandu, D. Daescu, G. R. Carmichael, Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: Part I – theory and software tools, Atmospheric Environment 37 (36) (2003) 5083–5096.
[23] A. Sandu, L. Zhang, Discrete second order adjoints in atmospheric chemical transport modeling, Journal of Computational Physics (In print).
[24] R. Serban, A. C. Hindmarsh, CVODES: the sensitivity-enabled ODE solver in SUNDIALS, Tech. Rep. UCRL-JP-200037, Lawrence Livermore National Laboratory, Livermore, CA, USA (2003).
[25] A. Walther, Automatic differentiation of explicit Runge-Kutta methods for optimal control, Computational Optimization and Applications 36 (1) (2007) 83–108.
[26] Z. Wang, I. M. Navon, F. X. LeDimet, X. Zou, The second order adjoint analysis: Theory and applications, Meteorology and Atmospheric Physics 50 (1-3) (1992) 3–20.