Mihai Alexe and Adrian Sandu

DENSERKS: A suite of Fortran sensitivity solvers using continuous, explicit Runge-Kutta schemes

Computer Science Department
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
Phone: (540)-231-2193
Fax: (540)-231-6075
Email: {mihai,asandu}@cs.vt.edu

DENSERKS: A suite of Fortran sensitivity solvers using continuous, explicit Runge-Kutta schemes

MIHAI ALEXE and ADRIAN SANDU
Virginia Polytechnic Institute and State University

DENSERKS is a Fortran sensitivity equation solver package designed for integrating models whose evolution can be described by ordinary differential equations (ODEs). A salient feature of DENSERKS is its support for both forward and adjoint sensitivity analyses, with built-in integrators for both first and second order continuous adjoint models. The software implements explicit Runge-Kutta methods with adaptive timestepping and high-order dense output schemes for the forward and the tangent linear model trajectory interpolation. Implementations of six Runge-Kutta methods are provided, with orders of accuracy ranging from two to eight. This makes DENSERKS suitable for a wide range of practical applications. The use of dense output, a novel approach in adjoint sensitivity analysis solvers, allows for a high-order cost-effective interpolation. This is a necessary feature when solving adjoints of nonlinear systems using highly accurate Runge-Kutta methods (order five and above). To minimize memory requirements and make long-time adjoint model integrations computationally efficient, DENSERKS implements a two-level checkpointing mechanism. The code is tested on a selection of problems illustrating first and second order sensitivity analysis with respect to initial model conditions. The resulting derivative information is also used in a gradient-based optimization algorithm to minimize cost functionals dependent on a given set of model parameters.

Categories and Subject Descriptors: G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms: Algorithms, Design

Additional Key Words and Phrases: ODEs, Runge-Kutta methods, dense output, sensitivity analysis, tangent linear models, adjoint models, quadrature equations

1. INTRODUCTION

Sensitivity analysis is a research area attracting considerable attention due to the wide range of applicability of its results. In short, the objective of sensitivity analysis is to obtain qualitative and quantitative information about the relationship between changes in the inputs or parameters of a given system and changes in the outputs of that respective system. Parameter identification [Navon 1997], chemical kinetics [Damian et al. 2002], data assimilation [LeDimet et al. 2002], optimal control [Griesse and Walther 2003], ocean and atmosphere dynamics [Adcroft et al. 2007; Sandu et al. 2005], and design optimization [Özyurt and Barton 2005b] are

Authors' address: Department of Computer Science, Virginia Polytechnic and State University, 2200 Kraft Dr, Blacksburg, VA, 24061, USA; email: malexe@cs.vt.edu, asandu@cs.vt.edu; phone: +1-540-231-5568; fax: +1-540-231-9218

several of the areas where sensitivity information proves to be extremely useful. We focus on systems whose time evolution can be modeled as initial-value problems, using ODEs. Starting from this formulation, we derive the equations used in *forward* and *adjoint* sensitivity analysis for quantifying the model output variation as a response to perturbations in the initial conditions or other system parameters. We designed `DENSERKS` to solve *continuous* forward, tangent linear and adjoint models. While the discrete adjoint sensitivity analysis approach based on explicit Runge-Kutta methods has been proven to be consistent with the continuous adjoint model formulation [Hager 2000; Walther 2007; Sandu 2006], and tools such as `TAMC` [Giering 1999], `TAF` [Giering and Kaminski 2003], and `TAPENADE` [Hascöet and Pascual 2004] considerably speed up the code generation for discrete tangent linear or adjoint models, the code generated by automatic differentiation (AD) is frequently sub-optimal and hand-coded modifications of the differentiated code are normally required in order to guarantee its corectness [Eberhard and Bischof 1999].

The continuous sensitivity analysis approach taken in `DENSERKS` requires the user to provide only the right-hand side functions for the model equations. Once the Fortran code corresponding to the forward model equations has been written, AD can (and most of the time should) be used to generate very efficient code for the computation of tangent linear and adjoint model right-hand side functions. These are then supplied to `DENSERKS` for the forward or backward time integrations. Further details on the use of AD with `DENSERKS` are given in the appendix.

1.1 Related software packages

A significant effort has been dedicated to creating efficient sensitivity solvers, and currently there are several high quality implementations available for public use. `SUNDIALS` [Hindmarsh et al. 2005] is a suite of ODE solvers featuring forward and first order adjoint sensitivity analysis capabilities. The `CVODES` solver [Serban and Hindmarsh 2003], part of `SUNDIALS`, is a sensitivity-enabled ODE solver. `CVODES` users can choose between backward differentiation schemes or Adams-Moulton methods for forward, tangent linear and adjoint model integrations. The forward model trajectory is recreated via cubic Hermite interpolation or variable-order polynomial interpolation [Hindmarsh and Serban 2006b].

Cao, Li and Petzold designed and implemented software for both the forward and adjoint sensitivity analysis of differential-algebraic equations (DAEs) with index up to two [Li and Petzold 1999; Cao et al. 2002]. Both their `DASPK` and `DASPKADJOINT` packages use variable-order backward differentiation formulas to solve the DAE sensitivity systems. Sandu and Miehe [2006] discuss the implementation of implicit Runge-Kutta and Rosenbrock methods and their discrete and continuous adjoints. Third order Hermite interpolation is used to approximate the original model solutions at the points required in the continuous adjoint model solvers. Their code is integrated into the Kinetic PreProcessor (`KPP`) software for solving chemical kinetics [Damian et al. 2002; Sandu et al. 2003; Daescu et al. 2003].

In this paper we introduce a new sensitivity solver package based on explicit Runge-Kutta methods. To the authors' knowledge, there is no publicly available sensitivity analysis software that makes use of both explicit Runge-Kutta methods and dense output schemes for forward, tangent linear and (continuous) adjoint model integrations. This paper fills the gap; moreover, our package also implements a second-order adjoint model solver based on a set of equations equivalent to those describing the directional second-order adjoint (dSOA) method [Özyurt and Barton 2005a].

1.2 Organization

The rest of this paper is organized as follows. Section 2 contains mathematical background on Runge-Kutta numerical methods and dense output schemes, as well as the analytical derivation of the tangent linear and adjoint models employed in sensitivity analysis. Section 3 provides details on the implementation of `DENSERKS`. Information about the availability of our software can be found in section 4. In section 5 we test our integrators on a set of selected problems and report the results. We conclude with a summary in section 6.

2. THEORETICAL BACKGROUND

2.1 Runge-Kutta methods

Consider a system modeled by the following initial-value problem, henceforth referred to as the *forward model*:

$$\begin{aligned} \dot{y} &= F\big(t, y\left(t,p\right), p\big), \quad t^0 \le t \le t^F \,, \\ y\left(t^0\right) &= y^0\left(p\right) \,, \end{aligned} \tag{1}$$

where $y(t,p) \in \mathbb{R}^{n_x}$ is the state vector, $p \in \mathbb{R}^{n_p}$ denotes a vector of system parameters, and $F : \mathbb{R}^{n_x+1} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$. We assume that (1) has a unique solution $y = y(t)$, and that F is twice continuously differentiable with respect to both y and p, for all $t^0 \le t \le t^F$.

Runge-Kutta schemes are a well-known class of numerical methods for solving initial value problems (1). This paper focuses on embedded explicit Runge-Kutta methods. An s-stage embedded Runge-Kutta integration scheme for (1) can be defined by its Butcher tableau [Hairer et al. 1993]:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \\ \hline & \hat{b}_1 & \hat{b}_2 & \cdots & \hat{b}_{s-1} & \hat{b}_s \end{array} \tag{2}$$

such that

$$\begin{aligned} y^{n+1} &= y^n + h \sum_{j=1}^{s} b_j k_j \\ \hat{y}^{n+1} &= y^n + h \sum_{j=1}^{s} \hat{b}_j k_j \,, \end{aligned} \tag{3}$$

where

$$k_j = F\left(t^n + hc_j,\, y^n + h\sum_{i=1}^{j-1} a_{ij}k_i,\, p\right)\,, \tag{4}$$

$t^{n+1} = t^n + h$ and $y^n \approx y(t^n)$ denotes the numerical solution of (1) at t^n. Both approximations y^{n+1} and $\hat{y}^{n+1}$ use the same function values (stages) k_j, but they have different orders of accuracy, depeding on the particular choice of method coefficients a_{ij}, b_j, $\hat{b}_j$ and c_j. The next-step solution y^{n+1} is used to continue the integration, whereas the second solution, $\hat{y}^{n+1}$ is used to estimate the local truncation error as

$$e^{n+1} = y^{n+1} - \hat{y}^{n+1}\,. \tag{5}$$

This error estimate is used to automatically control the integration time step size. Hairer et al. [1993] give further details on the error control mechanism for Runge-Kutta integrators (Section II.4).

2.2 Dense Output for Runge-Kutta methods

In many applications one needs the approximate solution at certain prescribed time points; it is usually inefficient to have the Runge-Kutta routine compute those approximations, since this would impose unnecessary constraints on the size of the integration time steps. This situation motivated the construction of *dense output* formulas [Hairer et al. 1993]. Given an approximate solution $y^n \approx y(t^n)$, the dense output formulas yield cheap numerical approximations for the solution $y(t^n+\theta h)$ for the entire interval $0 \le \theta \le 1$. For most Runge-Kutta schemes of higher order, one needs to append $s^* - s$ extra stages in order to accommodate dense output schemes. Thus, the performance penalty incurred by the use of dense output formulas is equal to at most the cost of a few extra function evaluations per time step. A dense output formula for (3) has the form

$$u(\theta) = y^n + h\sum_{j=1}^{s^*} \tilde{b}_j(\theta)k_j\,, \tag{6}$$

where $0 \le \theta \le 1$, k_j is defined in (4), and $\tilde{b}_j(\theta)$ are polynomials in θ determined such that

$$u(\theta) - y(t^n + h\theta) = \mathcal{O}\left(h^{q^*+1}\right)\,. \tag{7}$$

Following Hairer et al. [1993], consider an interval $\left[t^n, t^{n+1}\right]$ far away from the initial time t^0, and denote by $z(x)$ the local solution of (1) that passes through the point (t^n, y^n). Then the error of the dense output formula can be written as:

$$u(\theta) - y\left(t^n + h\theta\right) = \left[u(\theta) - z(t^n + h\theta)\right] + \left[z(t^n + h\theta) - y(t^n + h\theta)\right]\,. \tag{8}$$

The first term in the right hand side of (8) is the interpolation error and has magnitude $\mathcal{O}\left(h^{q^*+1}\right)$. The second term denotes the global error of the method and is of magnitude $\mathcal{O}\left(h^q\right)$. Therefore, to guarantee an order-q accurate dense output approximation, it is sufficient to require $q^* = q - 1$. For $q \le 4$, cubic Hermite interpolation is sufficiently accurate. However, for larger values of q, performing polynomial interpolation while preserving the number of stages becomes an increasingly cumbersome process, and the quality of the interpolated solution depends on the choice of interpolation points. Continuous Runge-Kutta schemes allow for an efficient interpolation, with only a modest increase in the computational cost coming from the $s^* - s$ additional stages incorporated in (3). Since the accuracy constraints on the dense output coefficients usually allow for one or more degrees of freedom, one selects the coefficients of the polynomials $\tilde{b}_j(\theta)$ such that a certain error norm or cost function is minimized (see, e.g., [Baker et al. 1996]). Various dense output methods for constructing high-order interpolants have been proposed, notably by Sharp and Verner [1998], based on the bootstrapping scheme of Verner [1993].

2.3 First order forward sensitivity analysis

One objective in sensitivity analysis is to compute the sensitivity of the forward model solution with respect to the n_p parameters over the prescribed time interval $\left[t^0, t^F\right]$. Let the sensitivity of the solution y with respect to the parameter p_i be denoted by τ_i:

$$\tau_i(t) \equiv \frac{\partial y}{\partial p_i}(t)\,. \tag{9}$$

Under our smoothness assumptions on the solution of (1), we can formally differentiate the ODE in (1) with respect to any parameter p_i [Hairer et al. 1993]. Then the *tangent linear model* (usually abbreviated as TLM) describes the time evolution of the sensitivities τ_i:

$$\begin{aligned}\dot{\tau}_i(t) &= F_y \tau_i(t) + F_{p_i}\,, \quad t^0 \le t \le t^F\,,\\ \tau_i(t^0) &= \frac{\partial y\left(t^0\right)}{\partial p_i} = \frac{\partial y^0}{\partial p_i}\,, \quad i = 1 \dots n_p\,,\end{aligned} \tag{10}$$

with

$$F_y = \left(\frac{\partial F_i}{\partial y_j}\left(t, y, p\right)\right)_{i,j=1\dots n_x}\,, \tag{11}$$

and

$$F_{p_i} = \frac{\partial F}{\partial p_i}\,. \tag{12}$$

2.3.1 *First order forward sensitivities with respect to the initial conditions.* A special case that often arises in practice is $p = y^0$ (this implies that $n_p = n_x$). In this case one computes the sensitivities of the solution $y(t^F)$ with respect to the initial conditions y^0 given in (1). The tangent linear model specializes to:

$$\begin{aligned}\dot{\tau}_i(t) &= F_y \tau_i(t)\,, \quad t^0 \le t \le t^F\,,\\ \tau_i(t^0) &= e_i\,,\ i = 1 \dots n_x\,,\end{aligned} \tag{13}$$

where $e_i \in \mathbb{R}^{n_x}$ is the i-th unit vector. Solving (13) we obtain

$$\tau_i(t^F) = \frac{\partial y^F}{\partial y_i^0}\,. \tag{14}$$

2.3.2 *First order forward sensitivities with respect to vector perturbations in the parameters.* It is also useful to analyze the following scenario: suppose we are looking for the sensitivity of the final model solution with respect to a change $\delta p \in \mathbb{R}^{n_p}$ in *all* the parameters p. Then we can solve a slightly modified version of (10):

$$\begin{aligned} \dot{\tau} &= F_y \tau + F_p \delta p \,, \quad t^0 \le t \le t^F \,, \\ \tau(t^0) &= \frac{\partial y^0}{\partial p} \delta p \equiv \delta y^0 \,, \end{aligned} \tag{15}$$

to get

$$\tau(t^F) = \frac{\partial y^F}{\partial p} \delta p \equiv \delta y^F \,. \tag{16}$$

This particular form of the tangent linear model plays an important role in the second order sensitivity analysis framework, described in detail in section 2.6.

2.4 First order adjoint sensitivity analysis

An alternative objective is to find the sensitivity of a cost functional G that depends on the original model state $y(t)$ and on the n_p parameters:

$$G(p) = \int_{t^0}^{t^F} g\big(t, y(t,p), p\big)\, dt \,, \tag{17}$$

where $g : \mathbb{R}^{n_x+1} \times \mathbb{R}^{n_p} \to \mathbb{R}$ is a given real-valued function.

The gradient $\frac{\partial G}{\partial p}$ can be obtained by using the Lagrange multiplier technique [Özyurt and Barton 2005a; Cao et al. 2002; Hindmarsh et al. 2005]. We introduce the Lagrange multipliers $\lambda \in \mathbb{R}^{n_x}$ and form the extended cost functional

$$\begin{aligned} \hat{G}(p) &= G(p) - \int_{t^0}^{t^F} \lambda^T \left(\dot{y} - F(t,y,p)\right) dt \\ &= \int_{t^0}^{t^F} g(t,y,p) - \lambda^T \big(\dot{y} - F(t,y,p)\big)\, dt \,. \end{aligned} \tag{18}$$

Under the model constraint (1), $\hat{G}(p) = G(p)$. It follows that the sensitivity of $\hat{G}$ with respect to the parameters p is

$$\frac{\partial \hat{G}}{\partial p} = \frac{\partial G}{\partial p} = \int_{t^0}^{t^F} (g_y y_p + g_p) - \int_{t^0}^{t^F} \lambda^T (-F_p - F_y y_p + \dot{y}_p)\, dt \,. \tag{19}$$

Integration by parts yields:

$$\begin{aligned} \frac{\partial G}{\partial p} &= \int_{t^0}^{t^F} (g_y y_p + g_p)\, dt - \int_{t^0}^{t^F} (-\lambda^T F_p - \lambda^T F_y y_p - \dot{\lambda}^T y_p)\, dt - (\lambda^T y_p)|_{t^0}^{t^F} \\ &= \int_{t^0}^{t^F} (g_p + \lambda^T F_p)\, dt - \int_{t^0}^{t^F} (-g_y - \lambda^T F_y - \dot{\lambda}^T) y_p\, dt - (\lambda^T y_p)|_{t^0}^{t^F} \,. \end{aligned} \tag{20}$$

To avoid the computation of the forward sensitivities $y_p(t)$ at $t > t^0$, one defines the *first order adjoint variable* $\lambda \in \mathbb{R}^{n_x}$ as the solution of the *first order adjoint* *model*, described by the following final-value problem:

$$\begin{aligned}\dot{\lambda} &= -F_y^T(t,y)\lambda - g_y^T\,, \quad t^F \geq t \geq t^0\,, \\ \lambda(t^F) &= 0\,. \end{aligned} \tag{21}$$

The gradient (20) now reads

$$\frac{\partial G}{\partial p} = \int_{t^0}^{t^F} (g_p + \lambda^T F_p)\, dt + \left(\lambda^T y_p\right)\big|_{t=t^0}\,. \tag{22}$$

and it becomes apparent that one can compute the entire gradient $\partial G/\partial p$ at the cost of a single first order adjoint model integration, plus an evaluation of (22). Evaluating the integral in (22) amounts to solving the *quadrature equations*

$$\begin{aligned}\dot{w} &= -F_p^T\lambda - g_p^T\,, \quad t^F \geq t \geq t^0\,, \\ w\left(t^F\right) &= 0\,, \end{aligned} \tag{23}$$

backward in time, to get

$$\frac{\partial G}{\partial p} = \left(w^T + \lambda^T y_p\right)\big|_{t=t^0}\,, \tag{24}$$

where $w \in \mathbb{R}^{n_p}$ are the *quadrature variables.* To avoid storing intermediate values for λ, it is generally advisable to integrate (21) and (23) simultaneously, e.g. as an extended system of $n_x + n_p$ ODEs, or on different computational grids.

2.4.1 *First order adjoint sensitivities for pointwise functionals.* We now consider a special case of (21). If we are interested in evaluating the gradient of a pointwise cost functional $G^F(p) = g\big(y\left(t^F, p\right), p\big)$, we can use the following relation [Cao et al. 2002]:

$$\frac{\partial G^F}{\partial p}\left(t^F, p\right) = \frac{\partial g}{\partial p}\left(t^F, p\right) = \frac{d}{dt^F}\frac{\partial G}{\partial p}(p)\,. \tag{25}$$

A straightforward application of the Leibniz differentiation rule yields:

$$\begin{aligned}\frac{d}{dt^F}\frac{\partial G}{\partial p} &= \left(\gamma^T y_p\right)\big|_{t=t^0} + g_p\left(t^F\right) + \left(\lambda^T F_p\right)\big|_{t=t^F} + \int_{t^0}^{t^F} \gamma^T F_p\, dt \\ &= \left(\gamma^T y_p\right)\big|_{t=t^0} + g_p\left(t^F\right) + \int_{t^0}^{t^F} \gamma^T F_p\, dt\,, \end{aligned} \tag{26}$$

where the last equation follows from the final condition on λ in (21) and

$$\gamma \equiv \frac{d\lambda}{dt^F} \tag{27}$$

denotes the solution of the adjoint problem

$$\begin{aligned}\dot{\gamma} &= -F_y^T\gamma\,, \quad t^F \geq t \geq t^0\,, \\ \gamma\left(t^F\right) &= g_y^T\left(t^F\right)\,, \end{aligned} \tag{28}$$

obtained by differentiating (21) with respect to t^F.

2.4.2 *First order adjoint sensitivities with respect to the initial conditions.* Integrating the ODE in (28) with final conditions $\lambda(t^F) = e_i$ yields the sensitivities of the i-th component of the forward model model state $y_i^F \equiv y_i(t^F)$ with respect to the model initial conditions y^0:

$$\lambda\left(t^0\right) = \frac{\partial y_i^F}{\partial y^0}\,. \tag{29}$$

2.5 Tangent linear versus adjoint models

By inspecting (10) and (21), one notes that each model is best suited for use in one of two complementary scenarios. Forward sensitivity analysis is preferable when the number of parameters is (considerably) smaller than the number of functionals whose gradients need to be evaluated. Conversely, adjoint models are more efficient when the sensitivity of one or a few cost functionals with respect to a large number of parameters is desired. One can assert that the adjoint approach is (relatively) insensitive to an increase in n_p as opposed to the forward sensitivity analysis method [Özyurt and Barton 2005a]. The dependence on n_p becomes important when $n_x \ll n_p$, and the cost of integrating the quadrature equations (23) dominates the sensitivity computations.

2.6 Second order adjoint sensitivity analysis

Problems in areas such as data assimilation [Sandu and Zhang 2007; Kalnay 2002] require the minimization of a given cost functional of type (17). Numerical optimization schemes like the various flavors of the Newton method [Nocedal and Wright 2006] make use of second-order information to accelerate convergence. In practice, the computation of the full Hessian of the given cost functional is avoided, due to the high computational complexity of such a task. To make this process computationally tractable, second order information is usually computed in the form of Hessian-vector products (e.g., in a Hessian-free Newton implementation). Such products can be obtained efficiently by using the *second order adjoint* framework. Second order adjoint models have been used successfully in dynamic optimization problems such as 4D-Var data assimilation [Wang et al. 1992; LeDimet et al. 2002; Sandu and Zhang 2007] to compute second order derivative information for various cost functionals with respect to parameters or model initial conditions. `DENSERKS` implements the directional second-order adjoint (*dSOA*) method [Özyurt and Barton 2005a; 2005b].

Our goal is to compute the Hessian-vector product $\left(\partial^2 G/\partial p^2\right)\cdot\delta p$ where $\delta p \in \mathbb{R}^{n_p}$ is a perturbation in the (time-independent) parameters. By differentiating (20) with respect to p (similar to the procedure presented in [Özyurt and Barton 2005a]), we arrive at the following *second order adjoint* model equation:

$$\begin{aligned} \dot\sigma + F_y^T\sigma \;&=\; -\left(F_{yy}\odot(y_p\delta p)\right)^T\lambda - \left(F_{yp}\odot\delta p\right)^T\lambda - g_{yy}\left(y_p\delta p\right) - g_{yp}\delta p\,, \\ \sigma\left(t^F\right) \;&=\; 0\;\;,\;\; t^F \ge t \ge t^0\,, \end{aligned} \tag{30}$$

where $\sigma \in \mathbb{R}^{n_x}$ is the *second order adjoint variable*,

$$F_{yy} = \left(\frac{\partial^2 F_i}{\partial y_j \partial y_k}\right)_{i,j,k=1\ldots n_x}\,, \tag{31}$$

and the $\odot$ symbol denotes the tensor product

$$F_{yy} \odot (y_p \delta p) \;=\; \left(\sum_{k=1}^{n_x} \left(F_{yy}\right)_{i,j,k} (y_p \delta p)_k \right)_{i,j} . \tag{32}$$

F_{yp}, F_{pp}, F_{py} are defined by similar formulas.

Using the second order adjoint model trajectory $\sigma(t)$, we can compute the Hessian-vector product for any given vector $\delta p \in \mathbb{R}^p$ as:

$$\begin{aligned} \frac{\partial^2 G}{\partial p^2}\delta p \;=\; & \int_{t^0}^{t^F} \left(g_{pp}\delta p + g_{py}(y_p\delta p) + F_p^T \sigma + \left(F_{pp} \odot \delta p\right)^T \lambda + \left(F_{py} \odot (y_p \delta p)\right)^T \lambda \right) dt \\ & + \left. \left(\left(y_{pp} \odot \delta p\right)^T \lambda + y_p^T \sigma \right) \right|_{t=t^0} . \end{aligned} \tag{33}$$

The corresponding quadrature equations for $v \in \mathbb{R}^{n_p}$

$$\begin{aligned} \dot{v} + F_p^T \sigma \;&=\; -\left(F_{pp} \odot \delta p\right)^T \lambda - \left(F_{py} \odot (y_p \delta p)\right)^T \lambda - g_{pp}\delta p - g_{py}(y_p \delta p) \\ v\left(t^F\right) \;&=\; 0 \,, \; t^F \geq t \geq t^0 \,, \end{aligned} \tag{34}$$

are integrated alongside the first and second order adjoint models (21), (30) to yield

$$\frac{\partial^2 G}{\partial p^2}\delta p = v\left(t^0\right) + \left. \left(\left(y_{pp} \odot \delta p\right)^T \lambda + y_p^T \sigma \right) \right|_{t=t^0} . \tag{35}$$

An additional tangent linear model integration is necessary to obtain the $y_p \delta p$ term [Cao et al. 2002]. This requires the solution of (15) with $\tau(t) \equiv y_p(t)\delta p$.

2.6.1 *Second order adjoint sensitivity of pointwise functionals.* As in the case of the first order adjoint, one can compute the second order directional derivative of g by using the relation

$$\frac{\partial^2 g}{\partial p^2}(t^F, p) = \frac{d}{dt^F} \frac{\partial^2 G}{\partial p^2}(p) . \tag{36}$$

Let

$$\mu \equiv \frac{\partial \sigma}{\partial t^F} . \tag{37}$$

By differentiating (30) with respect to t^F, we obtain the following formulation of the second order adjoint model

$$\begin{aligned} \dot{\mu} + F_y^T \mu \;&=\; -\left(F_{yp} \odot \delta p\right)^T \gamma - \left(F_{yy} \odot (y_p \delta p)\right)^T \gamma \;\;,\;\; t^F \geq t \geq t^0 \\ \mu(t^F) \;&=\; \left. \left(g_{yy}(y_p \delta p) + g_{yp}\delta p\right) \right|_{t^F} \;, \end{aligned} \tag{38}$$

where $\gamma(t)$ is the solution of (28). A similar differentiation of (33) gives

$$\frac{\partial^2 g}{\partial p^2}\delta p = \mu(t^0) + \left. \left(\left(y_{pp} \odot \delta p\right)^T \gamma + y_p^T \mu \right) \right|_{t=t^0} \tag{39}$$

2.6.2 *Second order adjoint sensitivities with respect to the initial conditions.* If $n_p = n_x$, $p = y^0$ and $\delta p = \delta y^0$, i.e., one is interested in computing second-order sensitivity information with respect to model initial conditions, the second order

Runge-Kutta method	**Order of accuracy**	**Error control order**	**Interpolation method**
RK2 (Fehlberg)	2	3	Hermite (3rd/5th order)
RK3	3	2	Hermite (3rd/5th order)
RK4 (“3/8-rule”)	4	3	Hermite (3rd/5th order)
RK5	5	4	Hermite (3rd/5th order), Dense output (4th order)
RK6 (DOPRI5)	6	5	Hermite (3rd/5th order), Dense output (6th order)
RK8 (DOPRI8)	8	6	Hermite (3rd/5th order), Dense output (7th order)

Table I. Explicit Runge-Kutta methods implemented by `DENSERKS`: RK2 - Fehlberg-type Runge-Kutta method of order 2(3); RK3 - 3rd order Runge-Kutta with second order error control; RK4 - 4th order Runge-Kutta method with 3rd order error control built on the classic “3/8 rule”; RK5 - DOPRI5(4); RK6 - Runge-Kutta pair built on top of RK6(5)9FM; RK8 - DOPRI8(6).

adjoint system (38) simplifies to:

$$\begin{aligned} \dot{\sigma} + F_y^T \sigma &= -\left(F_{yy} \odot \delta y\right)^T \lambda \ , \ \ t^F \geq t \geq t^0 \, , \\ \sigma(t^F) &= g_{yy} \delta y \left|_{t^F}\right. \ , \end{aligned} \tag{40}$$

where $\delta y = y_p \delta p$.

Solving (40) alongside (21) yields

$$\sigma(t^0) = \frac{\partial^2 G}{\left(\partial y^0\right)^2} \delta y^0 \, . \tag{41}$$

3. IMPLEMENTATION DETAILS

3.1 Runge-Kutta implementations

`DENSERKS` implements several explicit Runge-Kutta methods, listed in Table I. It is important to note that the user needs to employ the same Runge-Kutta method for both the forward and adjoint mode integrations when solving (21) and (30). This is motivated by the fact that the stages k_j from the forward or tangent linear model run are reused by the dense output mechanism for trajectory interpolation during the adjoint integration. Thus stages corresponding to a method of the same order are required if the adjoint solution is to be fully accurate (i.e. have the same accuracy as the Runge-Kutta method used in the adjoint model integration). The Butcher tableaus and the dense output coefficients corresponding to the Runge-Kutta methods in Table I can be found in [Hairer et al. 1993; Baker et al. 1996; Dormand and Prince 1980; Prince and Dormand 1981; Dormand et al. 1989]. The `DENSERKS` adjoint model integrators can use either cubic/quintic Hermite polynomial interpolation or dense output for forward trajectory recomputations. The interpolation options are dependent on the particular choice of Runge-Kutta method.

3.2 Checkpointing

Upon closer inspection of the first (21) and second order (30) adjoint models, one notices that they are terminal value problems whose solution depends on the forward model and tangent linear model trajectories, respectively. Since the Runge-Kutta integrators have a variable step, one needs to store the information generated during the forward integrations and reuse it to interpolate the forward trajectory during the adjoint model run at the required time points. Once the forward model integration data is available in memory, the dense output mechanism can be used for the interpolation. However, memory requirements for large-scale applications and long-time integration processes render infeasible any storage mechanisms based exclusively on random-access memory. To mitigate this problem, we employ a two-level *checkpointing* scheme, thus making use of available disk storage [Hindmarsh et al. 2005; Adcroft et al. 2007; Cao et al. 2002]. A file checkpoint written by the forward model integrator is defined as a triplet

$$\mathrm{CHK}^{\mathrm{fwd}} = (h, t, y) \tag{42}$$

that fully captures the integrator's state at a particular time point t. Here h denotes the time step that is taken to get from t to the next time point. This data is sufficient to allow a *hot restart* of the integration from time t, i.e., the new trajectory will be identical to the previously computed forward trajectory starting from t. A file checkpoint is stored on disk every N_d integration steps (including an extra checkpoint at t^0). In between any two checkpoints, memory-based tapes are used to store the integration data. The tape entries entry can be viewed as a tuple of the form:

$$\mathrm{MEM}^{\mathrm{fwd}} = (h, t, y, F, k_{2\ldots s^*})\,. \tag{43}$$

Note that we store both the right-hand side F and stage values k_j in memory, since they are required in the dense output scheme (6). However, they are not needed in the file checkpoints. At the end of the forward integration, N_c file checkpoints will have been written ($N_c \geq 1$ since a checkpoint is always written at $t = t^0$). At this point, the backward integrator is run and any necessary forward trajectory recomputations are automatically performed. Note that the data stored in the memory buffers between the last checkpoint and t^F is reused in the adjoint integration, not recomputed, for increased efficiency. Let us denote the computational cost of a full forward integration by π_{FWD} and that of a first order adjoint model integration by π_{ADJ}. Then we have that the cost of a full backward problem integration, π_{BKWD}, is bounded as

$$\pi_{\mathrm{FWD}} + \pi_{\mathrm{ADJ}} \leq \pi_{\mathrm{BKWD}} < 2 \times \pi_{\mathrm{FWD}} + \pi_{\mathrm{ADJ}}\,. \tag{44}$$

For a visual representation of a multilevel checkpointing scheme, the reader is kindly directed to [Adcroft et al. 2007; Hindmarsh et al. 2005].

The discussion above also applies (with minor modifications) to the second order adjoint model integration. We now need to store on disk quadruples that include the tangent linear model state at time t:

$$\mathrm{CHK}^{\mathrm{tlm}} = (h, t, y, \tau)\ . \tag{45}$$

Likewise, the memory tape entries are enlarged to accomodate TLM-related information:

$$\mathrm{MEM}^{\mathrm{tlm}} = (h, t, y, \tau, F, F_y\tau + F_p\delta p, k_{2\ldots s^*}, k^{\tau}_{2\ldots s^*})\ . \tag{46}$$

Note that the forward and the tangent linear models share the same time steps (h) and time moments (t) in (45–46), since DENSERKS integrates the two models simultaneously, as a system of $2 \times n_x$ ODEs.

The computational cost of a second order adjoint solve, π_{BKWD2}, can be similarly bounded as

$$\pi_{\mathrm{L}} \le \pi_{\mathrm{BKWD2}} < \pi_{\mathrm{H}} \,, \tag{47}$$

where

$$\begin{aligned} \pi_{\mathrm{L}} &= \pi_{\mathrm{FWD}} + \pi_{\mathrm{TLM}} + \pi_{\mathrm{ADJ}} + \pi_{\mathrm{SOA}} \\ \pi_{\mathrm{H}} &= 2 \times \pi_{\mathrm{FWD}} + 2 \times \pi_{\mathrm{TLM}} + \pi_{\mathrm{ADJ}} + \pi_{\mathrm{SOA}} \,, \end{aligned} \tag{48}$$

and π_{TLM} and π_{SOA} denote the computational cost incurred when integrating (13) and (30), respectively. When n_p additional quadrature equations need to be integrated, their associated cost has to be incorporated in (44) and (48).

3.3 Using automatic differentiation to obtain derivative information

From the equations that describe the first order and second order adjoint models (21,30), as well as from the gradient (22) and Hessian-vector product (33), it becomes clear that one needs an efficient way to compute Jacobian-vector, Jacobian-transpose times vector and Hessian-vector tensor products.

Assuming the right-hand side $F(t, y, p)$ of the original model (1) is twice continuously differentiable in both y and p, we can obtain first and second order derivative information via automatic differentiation. The forward mode of automatic differentiation can be used to yield $F_y \tau_i$ in (10). Likewise, applying the reverse mode of AD is an efficient method to generate code for the computation of $F_y^T \lambda$. In both cases the products are evaluated directly, without accumulation of the full Jacobian matrices. The "cheap gradient theorem" [Griewank 2000] asserts that both matrix-vector products are evaluated at a cost not exceeding five evaluations of $F(t, y, p)$. Moreover, AD yields derivative information that has machine precision accuracy: the result is not affected by the truncation errors present in any finite-difference approximation [Griewank 2000].

Second order derivative information can be obtained by running AD twice. The "forward over adjoint mode" (a forward mode differentiation of existing first order adjoint code) avoids the computation of the first order sensitivity matrix $y_p \in \mathbb{R}^{n_x \times n_p}$ and has proven to be very efficient from a numerical implementation perspective [Özyurt and Barton 2005a]. Further details on AD and its use with DENSERKS are provided in the appendix of this paper.

3.4 Code organization and usage

Table II lists the names of the model integrator subroutines along with their file location, and succinct explanations of their purpose and usage. Sample drivers for the test problems described in section 5 are also provided with DENSERKS. We note that backward time integration in the forward mode is not supported, nor is forward time integration in the adjoint mode. Thus, all integrators in Table II require that $t^0 \le t^F$. In addition, the following source files are part of DENSERKS:

(1) `erk_tapes.f90`, `erk_tlm_tapes.f90`: Implementations of the memory buffer and file checkpoint mechanisms (for temporary storage of the forward and tan-

gent linear model trajectories between two consecutive file checkpoints). Contain functions for writing and reading file checkpoints and storing/retrieving data from the memory buffers.

(2) `erk_utils.f90`: An implementation of several Level 1 and Level 2 `BLAS` utility functions used by the `DENSERKS` integrators. Alternatively, the user can simply forward these calls to an optimized `BLAS` implementation available for his or her computer architecture.

A sample call sequence for first order adjoint model integrations is the following (we assume that file checkpointing is enabled, otherwise the values of `Nc` and `Nd` are ignored):

```
!! size of forward and adjoint model state vectors
   integer nx
!! size of quadrature system state
   integer np
!! initial and final integration time
   double precision t0, tF
!! forward and adjoint model state vectors
   double precision y(nx), ady(nx)
!! quadrature system state vector
   double precision w(np)
!! write a file checkpoint every Nd time steps
   integer Nd
!! allocate the memory buffers and initialize the checkpoint files
   call rk_AllocateTapes(...)
!! initialize y, ady, w and other integrator parameters as needed
   y, ady, w = ...
!! integrate the forward model
   call RKINT(nx,y,...,t0,tF,Nd,Nc,...)
!! Nc (integer) = total number of file checkpoints written
!! integrate the first order adjoint model
   call RKINT_ADJDR(nx,ady,...,np,w,...,Nc,...)
!! free memory
   call rk_DeallocateTapes
```

A similar sequence of `DENSERKS` subroutine calls ensures a correct backward-time integration of a given second-order adjoint model. In the case of multiple integrations of the same first or second order adjoint system (with different final conditions), it is recommended to use `RKINT_ADJDR_M` and `RKINT_SOADR_M`, respectively. Use of these subroutines results in a significant performance improvement over the approach of making several `RKINT_ADJDR` or `RKINT_SOADR` calls, due to the reuse of forward and tangent linear model trajectory data over all adjoint model integrations. This lowers the cost of M first order adjoint model integrations, $\pi_{\mathrm{M\_BKWD}}$, from $M \times \pi_{\mathrm{BKWD}}$ in (44) to

$$\pi_{\mathrm{FWD}} + M \times \pi_{\mathrm{ADJ}} \le \pi_{\mathrm{M\_BKWD}} < 2 \times \pi_{\mathrm{FWD}} + M \times \pi_{\mathrm{ADJ}} \,. \tag{49}$$

Integrator	**Source file**	**Description**
`RKINT`	`erk.f90`	Forward model integrator. Writes forward model trajectory data to the memory buffers and file checkpoints, if the memory buffering and file checkpointing mechanisms are enabled. This routine is user-callable.
`RKINT_TLM`	`erk_tlm.f90`	Tangent linear model integrator. Simultaneously integrates the forward model (the resulting system has size $2 \times n_x$). Writes forward and tangent linear model trajectory data to the memory buffers and file checkpoints, if the memory buffering and file checkpointing mechanisms are enabled. This routine is user-callable.
`RKINT_ADJ`	`erk_adj.f90`	First order adjoint model integrator. The routine is used to integrate the first order adjoint model and the associated quadrature equations (for a total backward problem size of $n_x + n_p$) between two given consecutive checkpoints. `RKINT_ADJ` reads forward model trajectory data from the memory buffers, implicitly assuming they hold valid forward integration information. The user should *not* call this subroutine directly. Instead, all first order adjoint integrations should be done via calls to the `RKINT_ADJDR` wrapper subroutine.
`RKINT_SOA`	`erk_soa.f90`	Second order adjoint model integrator. Simultaneously integrates the first and second order adjoint model equations and the quadrature equations (the resulting system has size $2 \times n_x + n_p$). Reads data from the memory buffers only. The user should *not* call this subroutine directly. Instead, all first order adjoint integrations should be done via calls to the `RKINT_SOADR` wrapper subroutine.
`RKINT_ADJDR`	`erk_adjdr.f90`	Wrapper for `RKINT_ADJ`. Handles all checkpoint reads and calls `RKINT` to calculate the forward model trajectory, should such recomputations be required. Calls `RKINT_ADJ` for first order adjoint model integration between consecutive checkpoints. This routine is user-callable.
`RKINT_SOADR`	`erk_soadr.f90`	Wrapper for `RKINT_SOA`. Handles all checkpoint reads and calls `RKINT_TLM` to calculate the tangent linear model trajectory, should such recomputations be required. Calls `RKINT_SOA` for second order adjoint model integration between consecutive checkpoints. This routine is user-callable.
`RKINT_ADJDR_M`	`erk_adjdr_M.f90`	Wrapper for `RKINT_ADJ` used for multiple first order adjoint integrations that reuse the forward model trajectory data generated by `RKINT`. `RKINT_ADJDR_M` handles all checkpoint reads and calls `RKINT` to calculate the forward model trajectory, should such recomputations be required (in this case, at most one extra forward model integration is performed). Calls `RKINT_ADJ` for first order adjoint model integration between consecutive checkpoints. This subroutine is user-callable.
`RKINT_SOADR_M`	`erk_soadr_M.f90`	Wrapper for `RKINT_SOA` used for multiple second order adjoint model integrations that reuse the forward and tangent linear model trajectory data generated by `RKINT_TLM`. `RKINT_SOADR_M` handles all checkpoint reads and calls `RKINT_TLM` to calculate the forward and tangent linear model trajectories, should such recomputations be required (in this case, at most one extra forward and tangent linear model integration is performed). Calls `RKINT_SOA` for second order adjoint model integration between consecutive checkpoints. This subroutine is user-callable.

Table II. `DENSERKS` integrator subroutines.

A similar reduction in computational cost is noticed in the case of M second order adjoint model integrations with `RKINT_SOADR_M`. The corresponding cost bounds $\pi_{\rm L}$ and $\pi_{\rm H}$ in (48) become

$$\begin{aligned} \pi_{\rm LM} &= \pi_{\rm FWD} + \pi_{\rm TLM} + M \times \pi_{\rm ADJ} + M \times \pi_{\rm SOA} \\ \pi_{\rm HM} &= 2 \times \pi_{\rm FWD} + 2 \times \pi_{\rm TLM} + M \times \pi_{\rm ADJ} + M \times \pi_{\rm SOA}\,, \end{aligned} \tag{50}$$

thus

$$\pi_{\rm LM} \le \pi_{\rm M\_BKWD2} < \pi_{\rm HM}\,. \tag{51}$$

4. AVAILABILITY AND SYSTEM REQUIREMENTS

The source code is released under a BSD open source license and is freely available for download at `http://people.cs.vt.edu/~asandu/Software/DENSERKS`.

4.1 System requirements

To successfully compile and run `DENSERKS`, the user needs a Fortran 90 compliant compiler. The code has been tested on Unix and Linux systems with the following compilers: Intel Fortran compiler (`ifort`, Linux IA-32 version 9.1 and Unix IA-64 version 10.0), the Portland Group compiler (`pgf90`, Linux version 6.0-2), Lahey (`lf95` for 32-bit Linux) and `g95`[1]. Section 4.2 contains important information about building `DENSERKS` with Intel's `ifort` compiler.

4.2 Compiler optimization caveat

We have noticed that compilation of the `DENSERKS` source code using Intel's Fortran compiler `ifort` (Linux 32-bit version 9.1) with agressive optimizations enabled (`-O2` or `-O3`) results in an erroneous binary file: a spurious time step-related error is reported before the end of the final forward model integration time step. This issue can be mitigated by compiling with the `-mp` switch, which forces the compiler to maintain floating point precision (some arithmetic optimizations will be disabled). Enabling full optimizations with all other compilers used in our tests has resulted in valid binaries.

5. NUMERICAL EXPERIMENTS

All numerical experiments were performed using double precision floating-point arithmetic on a 3 Ghz Intel Pentium 4 workstation with 2 GB of memory running Fedora Core 6 Linux.

5.1 The Arenstorf orbit

The Arenstorf orbit system is usually given as a set of two second order ODEs [Hairer et al. 1993], and can be readily transformed into a first order initial-value problem:

$$\begin{aligned} \dot{y}_1 &= y_3 \\ \dot{y}_2 &= y_4 \end{aligned}$$

[1]Freely available under a GPL license at http://www.g95.org/

$$\begin{aligned}\dot{y}_3 &= y_1 + 2y_4 - \hat{\mu}\frac{y_1+\mu}{\left[(y_1+\mu)^2+y_2^2\right]^{3/2}} - \mu\frac{y_1-\hat{\mu}}{\left[(y_1-\hat{\mu})^2+y_2^2\right]^{3/2}} \\ \dot{y}_4 &= y_2 - 2y_3 - \hat{\mu}\frac{y_2}{\left[(y_1+\mu)^2+y_2^2\right]^{3/2}} - \mu\frac{y_2}{\left[(y_1-\hat{\mu})^2+y_2^2\right]^{3/2}}\,, \qquad (52)\end{aligned}$$

where

$$\begin{aligned}\mu &= 0.012277471 \qquad (53)\\ \hat{\mu} &= 1-\mu\,. \qquad (54)\end{aligned}$$

The initial conditions are:

$$\begin{aligned}y_1(t^0) &= 0.994\\ y_2(t^0) &= 0\\ y_3(t^0) &= 0\\ y_4(t^0) &\approx -2.0016 \qquad (55)\end{aligned}$$

We consider one-tenth of a full orbit period as our forward model integration window: $t^0 = 0$ and $t^F \approx 1.7065$. Figures 1 and 2 illustrate the absolute errors of the forward, tangent linear and first order adjoint model solvers using the DOPRI5(4) and DOPRI8(6) Runge-Kutta numerical schemes with 4th and 7th order dense output. As expected, the corresponding tangent linear and adjoint model solutions are 5th and 8th order accurate, respectively. This shows that the dense output strategy implemented in DENSERKS makes it possible to compute a high-quality adjoint solution that has the same order of accuracy as the underlying Runge-Kutta method. The reference solution was obtained by integrating the TLM model with very tight absolute and relative tolerances (ATOL = RTOL = 10^{-14}) using the DOPRI8(53) code written by Hairer et al. [1993]. One should note that, in the figures illustrating relative and absolute integration errors, there is no continuous dependence of the errors on the requested numerical accuracy. This is because both the forward and the adjoint integration algorithms are adaptive. The continuous graphs are only intended for better visualization.

5.2 The Van der Pol oscillator

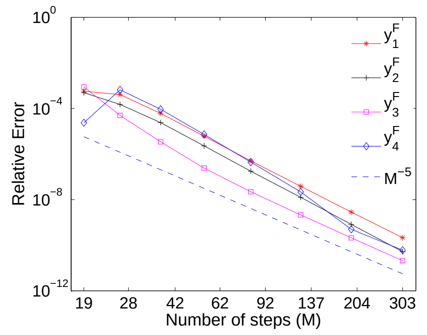
5.2.1 *First order sensitivity analysis.* We now consider the scaled version of the Van der Pol oscillator equation [Hairer and Wanner 1994]. The corresponding first-order ODE system reads:

$$\begin{aligned}\dot{y}_1 &= y_2\\ \dot{y}_2 &= \left(\left(1-y_1^2\right)y_2 - y_1\right)/\epsilon \qquad (56)\end{aligned}$$

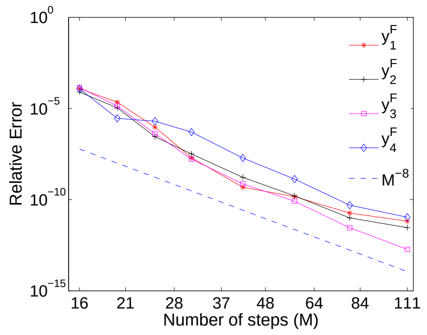
with $\epsilon = 10^{-2}$ and initial conditions

$$\begin{aligned}y_1(t^0) &= 2\\ y_2(t^0) &= 0 \qquad (57)\end{aligned}$$

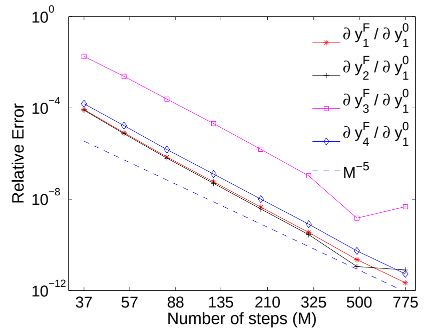
The time integration interval spans from $t^0 = 0$ to $t^F = 2$. Figure 3 illustrates the forward and adjoint model solutions, their order of accuracy, as well as the



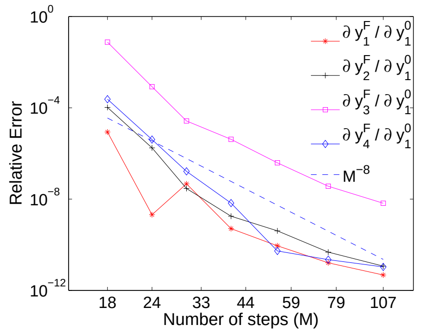
(a) Forward model: DOPRI5(4)



(b) Forward model: DOPRI8(6)



(c) Tangent linear model: DOPRI5(4)

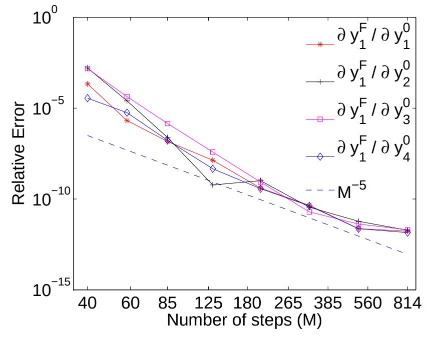


(d) Tangent linear model: DOPRI8(6)

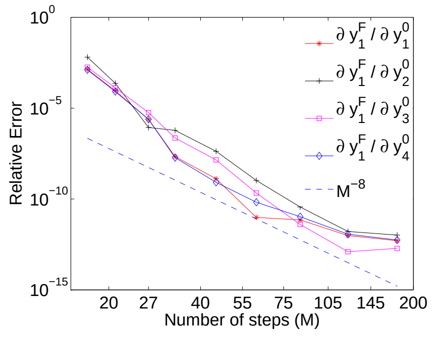
Fig. 1. Arenstorf orbit problem: Relative errors in the forward and tangent linear model solutions obtained using the DOPRI5(4) with 4th order dense output ((a),(c)) and DOPRI8(6) with 7th order dense output ((b),(d)). The integration interval length is equal to one tenth of a full orbit period: $t^0 = 0$, $t^F \approx 1.707$. As expected, both the forward ((a),(b)) and the tangent linear model solutions ((c),(d)) are fully accurate. In each TLM run we computed the first column ($j = 1$) of the Jacobian matrix $\left(\partial y_i^F / \partial y_j^0\right)_{i,j=1\ldots4}$.

time steps taken by the two integrators. Note that both the forward and adjoint integrators are adaptive: the time step size is adjusted to keep the numerical solution within the user-specified tolerance bounds. Step size control is performed independently during both the forward and the reverse integration. Figures 3 (e) and (f) assess how the forward integration errors affect the accuracy of the adjoint solution. It can be seen that for this particular problem, we obtain an eight order decrease in the error of the adjoint solution even when interpolating data corresponding to a less accurate forward trajectory. Thus, using the adaptive forward model integrator with looser tolerances still results in an accurate adjoint model solution.

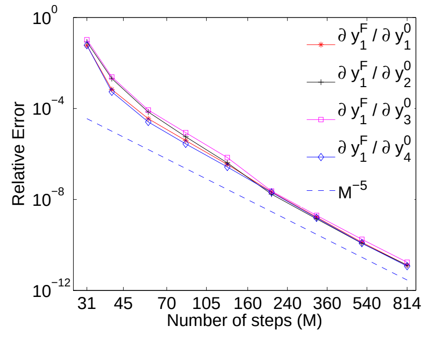
Let us consider ϵ as a model parameter and compute the sensitivity $\partial G/\partial\epsilon$ for a given cost function $G(y)$ as in (17). To this aim we need to integrate the quadrature equations (22) alongside the first order adjoint model. Figure 4 illustrates the gradient results in the case of $G = y(T)$ (where $T > t^0$ is a fixed time point).



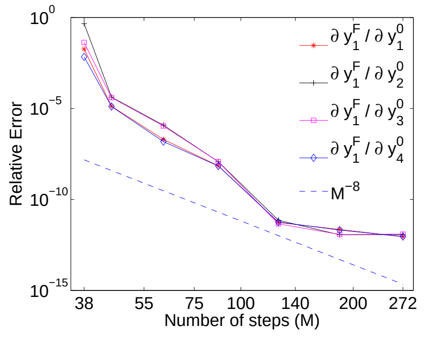
(a) Adjoint model: DOPRI5(4)



(b) Adjoint model: DOPRI8(6)



(c) Adjoint model: DOPRI5(4)
with variable forward tolerances

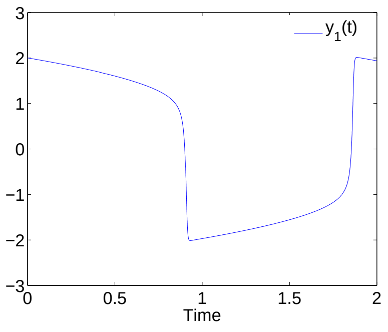


(d) Adjoint model: DOPRI8(6)
with variable forward tolerances

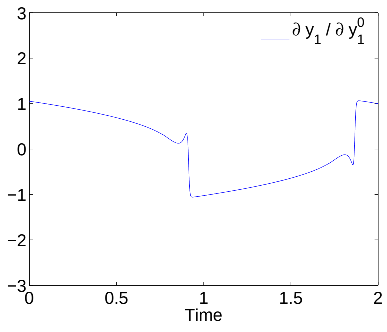
Fig. 2. Arenstorf orbit problem: Relative errors in the first order adjoint model solutions obtained using the DOPRI5(4) with 4th order dense output ((a),(c)) and DOPRI8(6) with 7th order dense output ((b),(d)). The integration interval length is equal to one tenth of a full orbit period: $t^0 = 0$, $t^F \approx 1.707$. As expected, the adjoint model solutions are fully accurate. The user can choose between two approaches: either run the adjoint model with variable tolerances, while keeping the forward integration tolerances constant ((a) and (c) illustrate this case, with $\mathrm{ATOL}^{\mathrm{fwd}} = \mathrm{RTOL}^{\mathrm{fwd}} = 10^{-12}$), or vary both the forward and adjoint model tolerances during several adjoint model test runs ((b) and (d) show the results of several adjoint model integrations where the prescribed absolute and relative forward and adjoint integrator tolerances are equal and vary between 10^{-4} and 10^{-12}). In each adjoint model run we computed the first row ($i = 1$) of the Jacobian matrix $\left(\partial y_i^F / \partial y_j^0\right)_{i,j=1\ldots4}$.

5.2.2 *Second order sensitivity analysis.* As a second-order sensitivity analysis example, we choose the following form of the Van der Pol system (adapted from [Özyurt and Barton 2005a]):

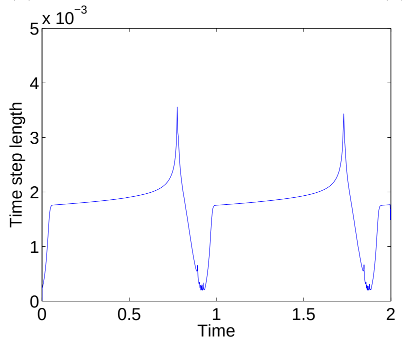
$$\begin{aligned}
\dot{y}_1 &= \left(1 - y_2^2\right) y_1 - y_2 + v(t,p) \\
\dot{y}_2 &= y_1 \\
\dot{y}_3 &= y_1^2 + y_2^2 + v^2(t,p)
\end{aligned} \tag{58}$$



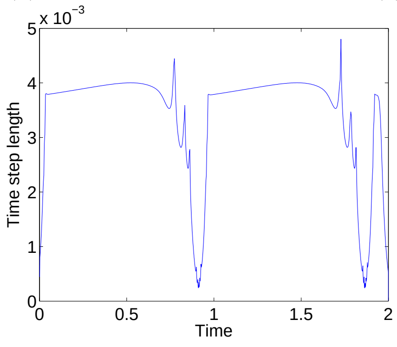
(a) Forward model solution: DOPRI8(6)



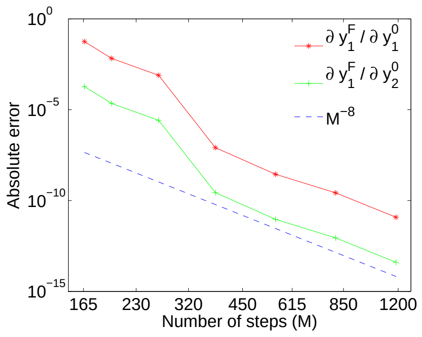
(b) Adjoint model solution: DOPRI8(6)



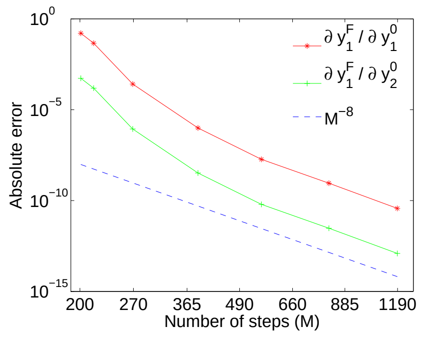
(c) Forward integration time steps



(d) Adjoint integration time steps



(e) Relative errors: DOPRI8(6) with fixed forward tolerances



(f) Relative errors: DOPRI8(6) with variable forward tolerances

Fig. 3. The Van der Pol oscillator (56): $t^0 = 0$, $t^F = 2$, $\epsilon = 10^{-2}$. Solutions obtained using DOPRI8(6) for the forward (a) and adjoint model (b). Here ATOL $= 10^{-12}$ and RTOL $= 10^{-10}$ for both model runs. The adaptive time steps taken during the two model integrations are plotted in (c) and (d). The bottom-two plots illustrate the absolute errors of the adjoint solution obtained using a fixed tolerance for the forward model run (ATOL$^{\text{fwd}}$ = RTOL$^{\text{fwd}} = 10^{-12}$ in (e)) and, alternatively, variable tolerance values for both the forward and adjoint integrators (in (f)).

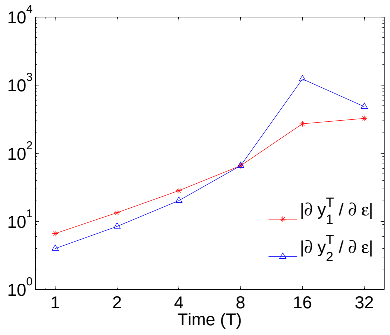


Fig. 4. The Van der Pol oscillator (56): Absolute value of the components of the gradient $\frac{\partial y}{\partial \epsilon} = \left[\frac{\partial y_1}{\partial \epsilon}\ \frac{\partial y_2}{\partial \epsilon}\right]^T$ at various time moments T.

with $t^0 = 0$, $t^F = 5$,

$$v(t,p) = \sum_{i=1}^{n_p - 1} t\, p_i\, p_{i+1} \tag{59}$$

and the initial conditions $y_1(t^0) = 0$, $y_2(t^0) = 1$, and $y_3(t^0) = 0$. The objective functional is chosen as

$$G(p) = y_3\left(t^F, p\right) \,, \tag{60}$$

and

$$\begin{aligned} p &= \left(\frac{1}{n_p}, \frac{1}{n_p}, \ldots, \frac{1}{n_p}\right) \,, \\ \delta p &= \left(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{n_p}\right) \,. \end{aligned} \tag{61}$$

The forward sensitivity system, the first and second order adjoint models and the quadrature equations are given in [Özyurt and Barton 2005a] for a general function $v(t,p)$. We approximate the Hessian-vector product

$$\left.\frac{\partial^2 g}{\partial p^2}\,\delta p\right|_{t^F,p} \tag{62}$$

using two distinct approaches. We first compute a finite difference approximation to (62):

$$\left.\frac{\partial^2 g}{\partial p^2}\,\delta p\right|_{t^F,p} \approx \frac{\nabla_p\, g\left(t^F, p + \epsilon\,\delta p\right) - \nabla_p\, g\left(t^F, p\right)}{\epsilon} \,, \tag{63}$$

and then we compare the results against those provided by the second order adjoint method (39). The relative computational cost of these two methods is illustrated in Table III for increasing values of n_p. We note that the advantage of the dSOA method is twofold. First, the computational cost of dSOA is inferior to that of the finite difference method in most of the test runs (as shown in the last column of Table III). Second, the accuracy of the Hessian-vector product approximation can

n_p	π_{FWD} [ms]	π_{BKWD2} [ms]	$\pi_{\text{BKWD2}}/\pi_{\text{FDIFF}}$
100	1	67	1.18
200	2	79	0.99
400	3	96	0.95
800	7	137	0.87
1600	11	245	0.82
3200	20	418	0.81
6400	41	794	0.77
12800	79	1659	0.78

Table III. Second order adjoint of the Van der Pol system (58): Computational cost of evaluating $\frac{d^2g}{dp^2}\delta p$ - the second order adjoint method (π_{BKWD2}) versus finite differences (π_{FDIFF}). DOPRI8 has been used for all model integrations.

easily be controlled via the user-chosen integration tolerances, whereas in the finite difference method (63) the accuracy depends on the particular choice of ϵ (and, at best, it is equal to the square root of the chosen error tolerance).

5.3 The convection-diffusion PDE

We consider the following initial boundary-value problem (adapted from [Hindmarsh and Serban 2006a]):

$$\begin{aligned} \frac{\partial y}{\partial t} &= p_1 \frac{\partial^2 y}{\partial x^2} + p_2 \frac{\partial y}{\partial x} \\ t^0 &= 0 \le t \le t^F = 1 \\ x_0 &= 0 \le x \le x_1 = 2\,, \end{aligned} \tag{64}$$

with initial and boundary conditions

$$\begin{aligned} y(t, x_0) &= y(t, x_1) = 0\,, \quad \forall\, t \in \left[t^0, t^F\right]\,, \\ y(t^0, x) &= y^0(x) = x(2-x)e^{2x}\,. \end{aligned} \tag{65}$$

Let us denote the solution of (64) with $p_1 = 1$ and $p_2 = 0.5$ by $y^{\text{ref}}(t,x)$. The objective function reads

$$G(t,p) = \int_{x_0}^{x_1} g(y)\, dx = \frac{1}{2}\int_{x_0}^{x_1} \left(y(t,x) - y^{\text{ref}}\right)^2\, dx\,. \tag{66}$$

We are interested in the computing the gradient

$$\nabla_p G(t^F) = \left[\frac{\partial G}{\partial p_1}\left(t^F\right) \quad \frac{\partial G}{\partial p_2}\left(t^F\right)\right]^T\,, \tag{67}$$

using the first order adjoint sensitivity analysis method described in (17) – (23). Additionally, we want to find the parameters for which the value of $G(t^F)$ is minimized, i.e. solve the optimization problem

$$p^* = \arg\min_p G\left(t^F, p\right)\,. \tag{68}$$

For this purpose we employ the `L-BFGS-B` optimization routine described in [Zhu et al. 1997]. Problem (68) has the obvious solution $p^* = p^{\mathrm{ref}}$, with $G\left(t^F, p^*\right) = 0$. The gradient (67) can be obtained as

$$\begin{aligned} \frac{\partial G}{\partial p_1}\left(t^F, p\right) &= \int_{t^0}^{t^F} \int_{x_0}^{x_1} \lambda \, \frac{\partial^2 y}{\partial x^2} \, dx \, dt \\ \frac{\partial G}{\partial p_2}\left(t^F, p\right) &= \int_{t^0}^{t^F} \int_{x_0}^{x_1} \lambda \, \frac{\partial y}{\partial x} \, dx \, dt \,, \end{aligned} \tag{69}$$

where $\lambda(t,x)$ is the solution of the adjoint PDE:

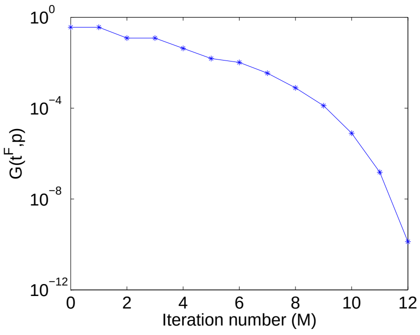
$$\begin{aligned} \frac{\partial \lambda}{\partial t} &= -p_1 \frac{\partial^2 \lambda}{\partial x^2} + p_2 \frac{\partial \lambda}{\partial x} \\ \lambda(t^F, x) &= \frac{\partial g}{\partial y}\left(t^F, x\right) = y\left(t^F, x\right) - y^{\mathrm{ref}}\left(t^F, x\right) \\ \lambda\left(t, x_0\right) &= \lambda\left(t, x_1\right) = 0 \,. \end{aligned} \tag{70}$$

We discretize the spatial derivatives in (64) and (70) using centered finite difference formulas on a uniform grid. Eliminating the homogeneous Dirichlet boundary conditions, it follows that (64) and (70) are each described by a system of n_x ODEs. To compute the gradient (67), we need to add two additional quadrature equations to the backward problem (one for each parameter p); thus, the adjoint system has size $n_x + 2$.

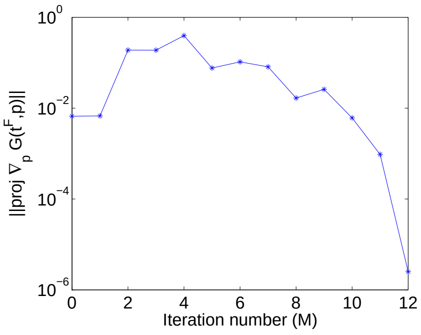
Figure 5 illustrates the decrease in the cost function $G(p)$ and its projected gradient norm during several `L-BFGS-B` iterations. The starting point is $(p_1^0, p_2^0) = (3, 3)$ and $n_x = 70$. After 12 iterations the parameters converge to the reference values $(p_1^{\mathrm{ref}}, p_2^{\mathrm{ref}}) = (1, 0.5)$.

6. SUMMARY

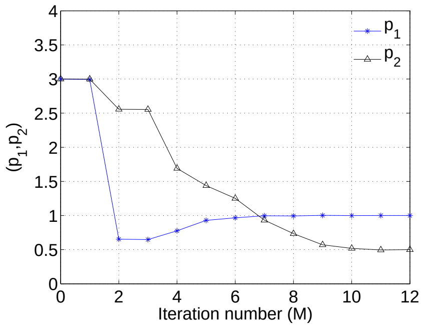
`DENSERKS` is a new addition to the range of publicly available software for performing sensitivity analysis of time-dependent models described by systems of ODEs. Its prominent features are: the implementation of several explicit Runge-Kutta integration schemes with adaptive time steps, the use of high order dense output schemes for forward model trajectory interpolation during the backward run, and the built-in support for second order adjoint sensitivity analysis. A storage strategy that combines file checkpointing and memory buffering of the forward model trajectories allows for an efficient integration of adjoint models for nonlinear problems. When performing multiple first or second order adjoint integrations but with different initial conditions and (or) computing sensitivities of multiple pointwise cost functionals for a given forward model, `DENSERKS` can reuse forward or tangent linear model integration data for enhanced efficiency. Thus up to two forward or tangent linear model integrations are required (this is a worst case bound when file checkpointing is enabled). The dense output mechanism allows for a cost-efficient interpolation, the overhead introduced by this approach being equal to at most a few function evaluations per (forward integration) time step. `DENSERKS` implements a selection of Runge-Kutta methods with order of accuracy from two up to eight. This makes the integrators suitable for a wide range of practical applications demanding various solution accuracies. The dense output schemes, themselves up to



(a) Cost function $G\left(t^F, p\right)$



(b) Norm of the projected gradient of $G\left(t^F, p\right)$



(c) Convergence of p towards $p^{\text{ref}} = (1, 0.5)$

Fig. 5. The convection-diffusion equation: Optimization results using `L-BFGS-B`. (a) The decrease in the cost function $G\left(t^F, p\right)$ and (b) the decrease in the projected norm of the gradient of the cost function versus `L-BFGS-B` iteration count (M). (c) The convergence of the parameters towards the reference values $\left(p_1^{\text{ref}}, p_2^{\text{ref}}\right) = (1, 0.5)$, where $G\left(t^F, p^{\text{ref}}\right) = 0$. The starting point is $\left(p_1^0, p_2^0\right) = (3, 3)$ and the size of the spatially discretized forward ODE system is $n_x = 70$.

7th order accurate, ensure full accuracy of the adjoint model solution, as illustrated in the numerical examples. Cubic or quintic Hermite interpolation is also available for use with the lower-order Runge-Kutta integrators. We hope that this range of capabilities will make `DENSERKS` useful for a large community of computational scientists and practitioners.

APPENDIX

In the following we illustrate the use of the well-known automatic differentiation tool `TAMC` for tangent linear and adjoint code generation. Detailed information on TAMC is given in the user's manual [Giering 1999]. Note that a similar approach is recommended when working with with other AD tools such as `TAF` [Giering and Kaminski 2003] or `TAPENADE` [Hascöet and Pascual 2004].

Following the approach of Griewank [2000], consider the (nonlinear) vector equation

$$b = F(a)\,, \tag{71}$$

with $a\,, b \in \mathbb{R}^{n_x}$ and F is assumed to be twice continuously differentiable in a. The forward mode of AD with a as input and b as output yields

$$\delta b = F'(a)\delta a\,, \tag{72}$$

whereas the reverse mode with the same input and output computes

$$\begin{aligned} \bar{a} &= \bar{a} + \bar{b}F'(a) \\ \bar{b} &= 0\,. \end{aligned} \tag{73}$$

Here the tangent linear variable and the adjoint variable corresponding to a variable x are denoted by δx and $\bar{x}$, respectively, and $F'(a) \equiv dF/da$. The costate (adjoint) variables are shaped like row vectors. Let $b \leftrightarrow \dot{y}$ and $a \leftrightarrow y$ in (71), with y and $\dot{y} = dy/dt$ defined in (1). One can then associate in (72)

$$\begin{aligned} \delta a &\leftrightarrow \tau_i \\ \delta b &\leftrightarrow \dot{\tau}_i \end{aligned} \tag{74}$$

where τ_i are the forward sensitivity variables defined in (9).

DENSERKS requires that the right-hand side of the forward model equation be implemented as a subroutine with the following parameter list:

```
   SUBROUTINE RHS (NX, T, Y, F)
    INTEGER ::  NX
    DOUBLE PRECISION ::  T
!! Numerical solution of the forward model at time T
    DOUBLE PRECISION ::  Y(NX)
!! RHS of the forward model at time T
    DOUBLE PRECISION ::  F(NX)
```

Note that, for this and for all the following examples, the user is free to choose any subroutine name. DENSERKS only requires the subroutine to have the specified parameter list.

It follows that running TAMC in the forward mode on RHS with Y as input and F as output can automatically generate the code for

```
   SUBROUTINE TLM_RHS (NX, T, Y, DY, DF)
!! Numerical solution of the tangent linear model at time T
    DOUBLE PRECISION ::  DY(NX)
!! RHS of the tangent linear model at time T
    DOUBLE PRECISION ::  DF(NX)
```

which is the subroutine that DENSERKS requires for tangent linear model right-hand side evaluations. and The terms F_{p_i} in (10), or $F_p\delta p$ in (15) can be generated by differentiating the code inside FEVAL with respect to the system parameters[2].

For the adjoint mode, we can identify in (73)

$$\bar{a} \leftrightarrow \dot{\lambda}^T \quad \text{and} \quad \bar{b} \leftrightarrow \lambda^T\,. \tag{75}$$

[2]One can add the parameter vector as a dummy argument to RHS and then differentiate.

Thus, setting $\bar{a}=0$ at the beginning of the adjoint subroutine and removing the code for the second assignment in (73) leads to

```
    SUBROUTINE ADJ_RHS (NX, T, Y, ADY, ADF)
!! Numerical solution of the first order adjoint model at time T
     DOUBLE PRECISION ::  ADY(NX)
!! RHS of the first order adjoint model at time T
     DOUBLE PRECISION ::  ADF(NX)
```

A similar approach can be used for building the right hand side of the quadrature system (23). Running TAMC in the adjoint mode and passing in the parameters p as the inputs and F as the output yields the product $(dF/dp)^T \cdot \lambda$.

For the second order adjoint, we consider the simpler case of sensitivity analysis with respect to initial conditions ($p = y^0$). Thus we need to generate the Hessian-vector product in the model equations (40). Forward mode differentiation of (73) results in code that computes

$$\dot{\bar{a}} = \dot{\bar{a}} + \dot{\bar{b}}F'(a) + \bar{b}F''(a)\dot{a}\,, \tag{76}$$

and we can identify using (40):

$$\begin{aligned} \dot{\bar{a}} &\leftrightarrow \dot{\sigma}^T \\ \dot{\bar{b}} &\leftrightarrow \sigma^T \\ \dot{a} &\leftrightarrow \delta y \\ \bar{b} &\leftrightarrow \lambda^T \\ a &\leftrightarrow y \end{aligned} \tag{77}$$

Starting from ADJ_RHS, we perform a forward differentiation with Y, ADY and ADF as inputs and ADF as output. After some straightforward code manipulation, we arrive at

```
    SUBROUTINE SOA_RHS (NX, NP, T, Y, DY, DP, ADY, AD2Y, AD2F)
!! Numerical solution of the second order adjoint model at time T
     DOUBLE PRECISION ::  AD2Y(NX)
!! RHS of the second order adjoint model at time T
     DOUBLE PRECISION ::  AD2F(NX)
```

The code generation for the other Hessian-vector products in (30), (34) and (38) can be done in an analogous manner. Similarly, the second order quadrature equations may easily be derived by forward differentiation of the first order quadrature code.

ACKNOWLEDGMENTS

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

REFERENCES

Adcroft, A., Campin, J.-M., Heimbach, P., Hill, C., and Marshall, J. 2007. *MIT General Circulation Model User's Manual.* MIT, Boston, MA, USA.

Baker, T. S., Dormand, J. R., Gilmore, J. P., and Prince, P. J. 1996. Continuous approximation with embedded Runge-Kutta methods. *Appl. Numer. Math. 22,* 1-3, 51–62.

Cao, Y., Li, S., and Petzold, L. 2002. Adjoint sensitivity analysis for differential-algebraic equations: Algorithms and software. *J. Comp. Appl. Math. 149,* 1, 171–191.

Cao, Y., Li, S., Petzold, L., and Serban, R. 2002. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM J. Sci. Comput. 24,* 3, 1076–1089.

Daescu, D. N., Sandu, A., and Carmichael, G. R. 2003. Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: II—numerical validation and applications. *Atmospheric Environment 37,* 36, 5097–5114.

Damian, V., Sandu, A., Damian, M., Potra, F., and Carmichael, G. R. 2002. The kinetic preprocessor KPP - a software environment for solving chemical kinetics. *Comput. Chem. Eng. 26*, 1567–1579.

Dormand, J. R., Lockyer, M. A., McGorrigan, N. E., and Prince, P. J. 1989. Global error estimation with Runge-Kutta triples. *Comput. Math. Appl. 18,* 9, 835–846.

Dormand, J. R. and Prince, P. J. 1980. A family of embedded Runge-Kutta formulae. *J. Comput. Appl. Math. 6,* 1, 19–26.

Eberhard, P. and Bischof, C. 1999. Automatic differentiation of numerical integration algorithms. *Math. Comput. 68,* 226, 717–731.

Giering, R. 1999. *Tangent linear and Adjoint Model Compiler, Users manual 1.4.*

Giering, R. and Kaminski, T. 2003. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *Proc. Appl. Math. Mech. 2,* 1, 54–57.

Griesse, R. and Walther, A. 2003. Parametric sensitivities for optimal control problems using automatic differentiation. *Optim. Control Appl. Meth.* 28, 297–314.

Griewank, A. 2000. *Evaluating derivatives: principles and techniques of algorithmic differentiation.* SIAM, Philadelphia, PA, USA.

Hager, W. W. 2000. Runge-Kutta methods in optimal control and the transformed adjoint system. *Numer. Math. 87,* 2, 247–282.

Hairer, E., Nørsett, S. P., and Wanner, G. 1993. *Solving Ordinary Differential Equations: Nonstiff Problems.* Computational Mathematics, vol. I. Springer-Verlag.

Hairer, E. and Wanner, G. 1994. *Solving Ordinary Differential Equations: Stiff and Differential-Algebraic Problems.* Computational Mathematics, vol. II. Springer-Verlag.

Hascöet, L. and Pascual, V. 2004. TAPENADE 2.1 user's guide. Tech. Rep. 0300, INRIA, Sophia Antipolis, France.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw. 31,* 3, 363–396.

Hindmarsh, A. C. and Serban, R. 2006a. *Example Programs for CVODES v.2.5.0.* Lawrence Livermore National Laboratory, Livermore, CA, USA.

Hindmarsh, A. C. and Serban, R. 2006b. *User documentation for CVODES v 2.5.0.* Lawrence Livermore National Laboratory, Livermore, CA, USA.

Kalnay, E. 2002. *Atmospheric modeling, data assimilation and predictability.* Cambridge University Press.

LeDimet, F. X., Navon, I. M., and Daescu, D. 2002. Second order information in data assimilation. *Mon. Weather Rev. 130,* 3, 629–648.

Li, S. and Petzold, L. 1999. Design of new DASPK for sensitivity analysis. Tech. Rep. TRCS99-28, University of California at Santa-Barbara, Santa Barbara, CA, USA.

Navon, I. M. 1997. Practical and theoretical aspects of adjoint parameter estimation and identifiability in meteorology and oceanography. *Dyn. Atmos. Oceans 27*, 55–79.

NOCEDAL, J. AND WRIGHT, S. J. 2006. *Numerical optimization*, 2nd ed. Springer Series in Operations Research. Springer-Verlag.

ÖZYURT, D. B. AND BARTON, P. I. 2005a. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. Sci. Comput. 26,* 5, 1725–1743.

ÖZYURT, D. B. AND BARTON, P. I. 2005b. Large-scale dynamic optimization using the directional second-order adjoint method. *Ind. Eng. Chem. Res. 44*, 1804–1811.

PRINCE, P. J. AND DORMAND, J. R. 1981. High order embedded Runge-Kutta formulae. *J. Comput. Appl. Math. 7*, 67–76.

SANDU, A. 2006. On the properties of Runge-Kutta discrete adjoints. In *International Conference on Computational Science (4)*. 550–557.

SANDU, A., DAESCU, D., AND CARMICHAEL, G. R. 2003. Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: Part I—theory and software tools. *Atm. Env. 37,* 36, 5083–5096.

SANDU, A., DAESCU, D., CARMICHAEL, G. R., AND CHAI, T. 2005. Adjoint sensitivity analysis of regional air quality models. *J. Comput. Phys. 204,* 1, 222–252.

SANDU, A. AND MIEHE, P. 2006. Forward, tangent linear, and adjoint Runge-Kutta methods in KPP–2.2 for efficient chemical kinetic simulations. Tech. Rep. TR-06-17, Virginia Tech, Blacksburg, VA, USA.

SANDU, A. AND ZHANG, L. 2007. Discrete second order adjoints in chemistry transport modeling: Computational aspects and applications. Tech. Rep. TR-07-27, Virginia Tech, Blacksburg, VA, USA.

SERBAN, R. AND HINDMARSH, A. C. 2003. CVODES: the sensitivity-enabled ODE solver in SUNDIALS. Tech. Rep. UCRL-JP-200037, Lawrence Livermore National Laboratory, Livermore, CA, USA.

SHARP, P. W. AND VERNER, J. H. 1998. Generation of high-order interpolants for explicit Runge-Kutta pairs. *ACM Trans. Math. Softw. 24,* 1, 13–29.

VERNER, J. H. 1993. Differentiable interpolants for high-order Runge-Kutta methods. *SIAM J. Numer. Anal. 30,* 5, 1446–1466.

WALTHER, A. 2007. Automatic differentiation of explicit Runge-Kutta methods for optimal control. *Comput. Optim. Appl. 36,* 1, 83–108.

WANG, Z., NAVON, I. M., LEDIMET, F. X., AND ZOU, X. 1992. The second order adjoint analysis: Theory and applications. *Meteorol. Atmos. Phys. 50,* 1-3, 3–20.

ZHU, C., BYRD, R. H., LU, P., AND NOCEDAL, J. 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw. 23,* 4, 550–560.

Received Month Year; revised Month Year; accepted Month Year