FATODE: A LIBRARY FOR FORWARD, ADJOINT, AND TANGENT LINEAR INTEGRATION OF ODES*

HONG ZHANG† AND ADRIAN SANDU‡

Abstract. FATODE is a FORTRAN library for the integration of ordinary differential equations with direct and adjoint sensitivity analysis capabilities. The paper describes the capabilities, implementation, code organization, and usage of this package. FATODE implements four families of methods – explicit Runge-Kutta for nonstiff problems and fully implicit Runge-Kutta, singly diagonally implicit Runge-Kutta, and Rosenbrock for stiff problems. Each family contains several methods with different orders of accuracy; users can add new methods by simply providing their coefficients. For each family the forward, adjoint, and tangent linear models are implemented. General purpose solvers for dense and sparse linear algebra are used; users can easily incorporate problem-tailored linear algebra routines. The performance of the package is demonstrated on several test problems. To the best of our knowledge FATODE is the first publicly available general purpose package that offers forward and adjoint sensitivity analysis capabilities in the context of Runge Kutta methods. A wide range of applications are expected to benefit from its use; examples include parameter estimation, data assimilation, optimal control, and uncertainty quantification.

Key words. Runge Kutta methods, tangent linear model, adjoint model, sensitivity analysis.

AMS subject classifications. 97N80, 65L99, 49Q12

1. Introduction.

Many dynamical systems in science and engineering are modeled by ordinary differential equations (ODEs)

$$y' = f\left(t, y; p\right) , \quad t_0 \leq t \leq t_f , \quad y(t_0) = y_0 . \tag{1.1}$$

Here $y(t) \in \mathbb{R}^d$ is the solution vector, y_0 the initial condition, and $p \in \mathbb{R}^m$ a vector of model parameters. Stiffness results from the existence of multiple dynamical scales, with the fastest characteristic times being much smaller than the time scales of interest in the simulation. It is well known that the numerical solution of stiff systems requires unconditionally stable discretizations which allow time steps that are not bounded by the fastest time scales in the system [17]. Here we assume that the system parameters p are independent of time. In the context of the ODE system (1.1), sensitivity analysis yields derivatives of the solution with respect to the initial conditions or system parameters, as follows

$$S_\ell(t) = \frac{\partial y(t)}{\partial p_\ell}, \quad 1 \leq \ell \leq m. \tag{1.2}$$

Two main approaches are available for computing the sensitivities (1.2). The direct (or tangent linear) method is efficient when the number of parameters is smaller than the dimension of the system ($m \ll d$), while the adjoint method is efficient when the number of parameters is larger than the dimension of the system ($m \gg d$). Furthermore, two distinct approaches can be taken for defining adjoint sensitivities; the continuous adjoint (differentiate then discretize) and the discrete adjoint (discretize then differentiate) approaches typically lead to different computational results.

*This work is supported by the National Science Foundation through the awards NSF DMS–0915047, NSF CCF–0916493, NSF OCI–0904397.

†Computational Science Laboratory, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 (zhang@vt.edu)

‡Computational Science Laboratory, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 (sandu@cs.vt.edu)

Sensitivity analysis is an essential ingredient for uncertainty quantification, parameter estimation, optimization, optimal control, and construction of reduced order models. Only a few available software packages for the solution of ODEs have the capability to compute sensitivities. One of the earlier packages is Odessa [19], which performs direct sensitivity analysis. A modern package is CVODES within SUNDIALS [26] from Lawrence Livermore National Laboratory. CVODES is able to compute direct and continuous adjoint sensitivities. Both Odessa and CVODES are based on backward differentiation formulas (BDF). A software based on explicit Runge-Kutta discretizations is DENSERKS [2] which implements continuous adjoint sensitivity analysis for non-stiff ODEs. The Kinetic PreProcessor KPP [23, 1] is a widely used tool for the simulation of chemical kinetics, and incorporates Runge-Kutta and Rosenbrock solvers that are endowed with tangent linear and discrete adjoint sensitivity analysis capabilities.

In this paper we present a library of explicit/implicit Runge-Kutta and Rosenbrock solvers for the simulation of nonstiff and stiff ODEs. The library, called FATODE, performs forward simulations, and sensitivity analysis via the discrete adjoint and the tangent linear methods. FATODE is based on the KPP library of solvers. While the KPP implementation is specifically aimed at chemical kinetic systems, the FATODE implementation is general and suitable for a wide range of applications.

The paper is organized as follows. Section 2 reviews several numerical integration algorithms. Section 3 discusses the structure and implementation of FATODE. The code usage in applications is summarized in Section 4. An example is shown in Section 5, and conclusions are drawn in Section 6.

2. Numerical integration methods in FATODE. Explicit Runge-Kutta methods [16] are well suited for solving nonstiff systems of ODEs. Implicit methods are preferred for solving stiff systems due to their better numerical stability properties. FATODE implements four families of methods: explicit Runge-Kutta, Rosenbrock, fully implicit Runge-Kutta, and singly diagonally implicit Runge-Kutta, as well as their tangent linear models and discrete adjoint models. The implicit methods have been implemented in implemented in KPP [23, 1] and have proved to be very efficient for solving many stiff chemical problems including CBM-IV [11], SAPRC [5] and NASA HSRP/AESA.

Runge-Kutta methods. A general s-stage Runge-Kutta method reads [16]

$$T_i = t_n + c_j\, h\,, \quad Y_i = y_n + h \sum_{j=1}^{s} a_{i,j}\, f\left(T_j\,,\, Y_j\right)\,, \quad i = 1, \ldots, s\,, \tag{2.1a}$$

$$y_{n+1} = y_n + h \sum_{j=1}^{s} b_j\, f\left(T_j\,,\, Y_j\right)\,. \tag{2.1b}$$

where the coefficients

$$\boldsymbol{A} = \left[a_{i,j}\right]_{1\le i,j\le s}\,, \quad \boldsymbol{b} = \left[b_i\right]_{1\le i\le s}\,, \quad \boldsymbol{c} = \left[c_i\right]_{1\le i\le s} = \boldsymbol{A}\cdot\boldsymbol{1}_{(s,1)}\,, \tag{2.2}$$

define the method and determine its accuracy and stability properties. Explicit Runge-Kutta methods are characterized by the coefficients $a_{i,j} = 0$ for all i and $j \ge i$. Singly diagonal implicit Runge-Kutta methods are defined by (2.1) with the coefficients $a_{i,i} = \gamma$ and $a_{i,j} = 0$ for all i and $j > i$. Fully implicit methods have three stages and require a coupled solution of all of them. Detailed information on available Runge Kutta methods is given in Table 2.1.

Rosenbrock methods. An s-stage Rosenbrock method [16] is given by the formulas

$$T_i = t_n + \alpha_i h\,, \quad Y_i = y_n + \sum_{j=1}^{i-1} \alpha_{i,j}\, k_j\,, \tag{2.3a}$$

$$k_i = h\, f\,(T_i, Y_i) + h\boldsymbol{f_y}(t_n, y_n) \cdot \sum_{j=1}^{i} \gamma_{i,j} k_j + \gamma_i h^2 f_t(t_n, y_n), \quad i = 1, \ldots, s\,, \tag{2.3b}$$

$$y_{n+1} = y_n + \sum_{j=1}^{s} b_j k_j\,, \tag{2.3c}$$

where particular methods are defined by their coefficients

$$\boldsymbol{\alpha} = [\alpha_{i,j}]_{1\le i,j\le s}\,, \quad \boldsymbol{b} = [b_i]_{1\le i\le s}\,, \quad \boldsymbol{\gamma} = [\gamma_{i,j}]_{1\le i,j\le s}\,, \tag{2.4}$$

and

$$\alpha_i = \sum_{j=1}^{s} \alpha_{i,j}\,, \quad \gamma_i = \sum_{j=1}^{s} \gamma_{i,j}\,; \quad \gamma_{i,i} = \gamma\,, \quad \alpha_{i,i} = 0\,; \quad \alpha_{i,j} = \gamma_{i,j} = 0\,,\ \forall\, i > j\,.$$

We have $\gamma_{i,i} = \gamma$ for all i for computational efficiency. Here $\boldsymbol{f_y} = \partial f/\partial y$ represents the Jacobian of the ODE function, as discussed in Appendix A. We will denote matrices and tensors by bold symbols, and vectors and scalar by regular symbols. Rosenbrock methods are attractive because of their outstanding stability properties and conservation of the linear invariants of the system. They typically outperform backward differentiation formulas such as those implemented in SMVGEAR [10] for medium accuracy solutions.

Detailed information regarding these methods is given in Table 2.1.

3. Implementation. This section summarizes several implementation aspects of FATODE. We start with the the forward integrators, and continue with the implementation of the tangent linear models and discrete adjoint models. The derivation of the sensitivity models can be found in earlier literature [24, 6]. We also describe linear algebra olvers and error control schemes in FATODE, which are important factors in determining the efficiency of the code.

3.1. Forward model integration.

Explicit Runge Kutta methods. The implementation of explicit methods is based on (2.1). The stage vectors are computed in succession using

$$Y_1 = y_n\,; \quad Y_i = y_n + h \sum_{j=1}^{i-1} a_{i,j}\, f(T_j, Y_j)\,, \quad i = 2, \ldots, s\,. \tag{3.1}$$

Matrices for solving implicit methods. The implementations of implicit methods use the following matrices

$$\mathbf{R}_n\,(\gamma, t, y) = \boldsymbol{I}_{(d,d)} - h\,\gamma\,\boldsymbol{f_y}(t\,,\, y)\,, \tag{3.2a}$$

$$\widetilde{\mathbf{R}}_n\,(\gamma, t, y) = \frac{1}{h\,\gamma}\boldsymbol{I}_{(d,d)} - \boldsymbol{f_y}(t\,,\, y)\,, \tag{3.2b}$$

TABLE 2.1

Time stepping methods implemented in FATODE. *(ERK, FIRK, SDIRK and ROS stand for Explicit Runge-Kutta, Fully Implicit Runge-Kutta, Singly Diagonally Implicit Runge-Kutta and Rosenbrock, respectively.)*

Family	Method	Stages	Order	Stability properties
ERK	RK2(3) [16]	3	2	conditionally stable
	RK3(2) [16]	4	3	conditionally stable
	RK4(3) [16]	5	4	conditionally stable
	DOPRI-5 [16]	7	5	conditionally stable
	Verner [16]	8	6	conditionally stable
	DOPRI-853 [16]	12	8	conditionally stable
FIRK	Radau-1A [17]	3	5	Stiffly-accurate
	Radau-2A [17]	3	5	Stiffly-accurate
	Lobatto-3C [17]	3	4	Stiffly-accurate
	Gauss [17]	3	6	Stiffly-accurate
SDIRK	Sdirk-2a	2	2	L-stable
	Sdirk-2b	2	2	L-stable
	Sdirk-3a	3	2	Stiffly-accurate
	Sdirk-4a [17]	5	4	L-stable
	Sdirk-4b [17]	5	4	L-stable
ROS	Ros-2 [28]	2	2	L-stable
	Ros-3 [25]	3	3	L-stable
	Rodas-3 [25]	4	3	Stiffly-stable
	Ros-4 [17]	4	4	L-stable
	Rodas-4 [17]	6	4	Stiffly-stable

and

$$\widehat{\mathbf{R}}_n = \begin{bmatrix} 1 - h\,a_{1,1}\,\boldsymbol{f_y}(T_1, Y_1) & \cdots & -h\,a_{1,s}\,\boldsymbol{f_y}(T_s, Y_s) \\ \vdots & \ddots & \vdots \\ -h\,a_{s,1}\,\boldsymbol{f_y}(T_1, Y_1) & \cdots & 1 - h\,a_{s,s}\,\boldsymbol{f_y}(T_s, Y_s) \end{bmatrix} \in \mathbb{R}^{ds \times ds}\,. \tag{3.2c}$$

Replacing each $\boldsymbol{f_y}(T_i, Y_i)$ in (3.2c) by $\boldsymbol{f_y}(t_n, t_n)$ leads to the approximation:

$$\overline{\mathbf{R}}_n = \boldsymbol{I}_{(ds,ds)} - h\,\boldsymbol{A} \otimes \boldsymbol{f_y}(t_n\,,\,y_n) \approx \widehat{\mathbf{R}}_n\,, \tag{3.2d}$$

where $\otimes$ is the matrix Kronecker product [17].

Implicit Runge-Kutta methods. To reduce the influence of round-off errors, we apply the transformation $z_i = Y_i - y_n$ [17] in the formulas (2.1) to obtain

$$T_i = t_n + c_i\,h\,, \quad z_i = h\sum_{j=1}^{s} a_{i,j}\,f(T_j\,,\,y_n + z_j)\,, \quad i = 1, \ldots, s\,, \tag{3.3a}$$

$$y_{n+1} = y_n + \sum_{i=1}^{s} d_i\,z_i\,. \tag{3.3b}$$

The new coefficients are

$$\boldsymbol{d} = [d_i]_{1 \le i \le s}\,, \quad \boldsymbol{d}^T = \boldsymbol{b}^T \cdot \boldsymbol{A}^{-1}\,. \tag{3.4}$$

Singly diagonally implicit Runge-Kutta methods. The stage equations (3.3a) read

$$z_i = h \sum_{j=1}^{i-1} a_{i,j}\, f(T_j\,,\, y_n + z_j) + h\,\gamma\, f(T_i\,,\, y_n + z_i)\,. \tag{3.5}$$

The nonlinear systems of equations (3.5) are solved in succession for each stage $i = 1, \ldots, s$ by simplified Newton iterations of the form

$$\begin{aligned} \mathbf{R}_n\,(\gamma, t_n, y_n) \cdot \Delta z_i^{[k]} &= z_i^{[k]} - h \sum_{j=1}^{i-1} a_{i,j}\, f(T_j\,,\, y_n + z_j) \\ z_i^{[k+1]} &= z_i^{[k]} - \Delta z_i^{[k]}\,, \quad k = 0, 1, \cdots\,. \end{aligned} \tag{3.6}$$

The same matrix is shared for all iterations and all stages, so that only one LU decomposition of $\mathbf{R}_n$ is performed in each time step.

Fully implicit Runge-Kutta methods. Fully implicit Runge-Kutta methods require the solution of the $ds \times ds$ nonlinear system (3.3a) [24]. With the compact notation

$$Z = \left[z_1^T \cdots z_s^T\right]^T\,, \quad F(Z) = \left[f^T(T_1, y_n + z_1) \cdots f^T(T_s, y_n + z_s)\right]^T\,, \tag{3.7}$$

where $Z, F(Z) \in \mathbb{R}^{ds}$, the nonlinear system (3.3a) can be written as

$$Z = \left(\boldsymbol{A} \otimes \boldsymbol{I}_{(d,d)}\right) \cdot F(Z)\,. \tag{3.8}$$

The system (3.8) is solved by simplified Newton iterations [17],

$$\begin{aligned} \overline{\mathbf{R}}_n \cdot \Delta Z^{[k]} &= Z^{[k]} - (h\,\boldsymbol{A} \otimes \boldsymbol{I}_{(d,d)}) \cdot F(Z^{[k]}) \\ Z^{[k+1]} &= Z^{[k]} - \Delta Z^{[k]}\,, \quad k = 0, 1, \cdots\,. \end{aligned} \tag{3.9}$$

Note that only the Jacobian at the beginning of the time step is used in Newton's iterations. Following [17], our implementation of the fully implicit s-stage Runge-Kutta method uses a transformation of the system (3.9) to a complex form such that the costly ds-dimensional real LU decomposition is replaced by d-dimensional LU decompositions of matrices of the form $\mathbf{R}\,(\lambda_i, t_n, y_n)$, where λ_i are the eigenvalues of $\mathbf{A}$. For the 3-stage methods implemented in FATODE the coefficient matrices $\mathbf{A}$ have one real and two complex conjugate eigenvalues, which leads to solving one real and one complex d-dimensional systems.

Rosenbrock methods. For implementation purpose, we use the alternative formulation [16] of the formula (2.3)

$$T_i = t_n + \alpha_i h\,, \quad Y_i = y_n + \sum_{j=1}^{i-1} \alpha_{i,j}\, k_j\,, \tag{3.10a}$$

$$\widetilde{\mathbf{R}}_n\,(\gamma, t_n, y_n) \cdot k_i = f\,(T_i, Y_i) + \sum_{j=1}^{i-1} \frac{c_{i,j}}{h}\, k_j + h\,\gamma_i\, f_t\,(t_n, y_n)\,, \tag{3.10b}$$

$$y_{n+1} = y_n + \sum_{i=1}^{s} m_i\, k_i\,, \tag{3.10c}$$

where

$$[a_{i,j}]_{1 \le i,j \le s} = \boldsymbol{\alpha} \cdot \boldsymbol{\gamma}^{-1}\,, \quad [c_{i,j}]_{1 \le i,j \le s} = \mathrm{diag}(\boldsymbol{\gamma}^{-1}) - \boldsymbol{\gamma}^{-1}\,, \quad [m_i]_{1 \le i \le s} = \boldsymbol{\gamma}^{-T} \cdot \boldsymbol{b}\,.$$

At each stage (3.10b) the solution of a linear system of dimension $d \times d$ is required. The same matrix $\widetilde{\mathbf{R}}_n$ is shared by all the stages and one LU decomposition per step is required.

3.2. Tangent linear model integration. Small changes δy_0 in the initial conditions result in small perturbations $\delta y(t)$ of the solution of ODE system (1.1). Let $\dot{y} = \delta y / \|\delta y_0\|$ be the directions of solution change. These directions propagate forward in time according to the *tangent linear ODE*:

$$\dot{y}' = \boldsymbol{f_y}(t\,,\, y) \cdot \dot{y}\,, \quad t_0 \le t \le t_f\,, \quad \dot{y}(t_0) = \dot{y}_0\,, \quad \dot{y}(t) \in \mathbb{R}^d\,. \tag{3.11}$$

The sensitivity equations (3.11) are solved forward in time together with original ODE system (1.1). Tangent linear models are derived for direct sensitivity analysis with each of the families of methods in FATODE. Highly efficient implementations are obtained by re-using the LU decompositions from the forward solution on the sensitivity equations [9].

Tangent linear Runge Kutta methods. A tangent linear Runge-Kutta (2.1) method reads

$$Y_i = y_n + h \sum_{j=1}^{s} a_{i,j} f(T_j, Y_j), \quad \dot{Y}_i = \dot{y}_n + h \sum_{j=1}^{s} a_{i,j}\, \boldsymbol{f_y}(T_i\,,\, Y_i) \cdot \dot{Y}_i, \tag{3.12a}$$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i f(T_i, Y_i), \quad \dot{y}_{n+1} = \dot{y}_n + h \sum_{i=1}^{s} b_i\, \boldsymbol{f_y}(T_i\,,\, Y_i) \cdot \dot{Y}_i\,. \tag{3.12b}$$

Similar to the implementation of implicit forward integrators, we introduce the sensitivity stage variables $\dot{z}_i = \dot{Y}_i - \dot{y}_n$ and the sensitivity part becomes

$$\dot{z}_i - h \sum_{j=1}^{s} a_{i,j}\, \boldsymbol{f_y}(T_j\,,\, Y_j) \cdot \dot{z}_j = h \sum_{j=1}^{s} a_{i,j}\, \boldsymbol{f_y}(T_j\,,\, Y_j) \cdot \dot{y}_n\,, \quad i = 1, \cdots, s\,, \tag{3.13a}$$

$$\dot{y}_{n+1} = \dot{y}_n + \sum_{i=1}^{s} d_i\, \dot{z}_i\,. \tag{3.13b}$$

Using the compact notation (3.7) and the matrix (3.2c) the stage equations (3.13a) can be written as

$$\widehat{\mathbf{R}}_n \cdot \dot{Z} = \left(\boldsymbol{I}_{(sd,sd)} - \widehat{\mathbf{R}}_n \right) \cdot (\mathbf{1}_s \otimes \dot{y}_n)\,. \tag{3.14}$$

Explicit RK methods. For ERK methods the equations (3.12a) are solved successively for each stage $i = 1, \cdots, s$, using

$$\dot{Y}_1 = \dot{y}_n\,; \quad \dot{Y}_i = \dot{y}_n + h \sum_{j=1}^{i-1} a_{i,j}\, \boldsymbol{f_y}(T_j\,,\, Y_j) \cdot \dot{Y}_j\,, \quad i = 2, \cdots, s\,.$$

Singly diagonally implicit RK methods. For SDIRK methods the system (3.13a) reduces to s independent d-dimensional linear systems that are solved successively for each stage $i = 1, \cdots, s$

$$\mathbf{R}_n\,(\gamma, T_i, Y_i) \cdot \dot{z}_i = h \sum_{j=1}^{i-1} a_{i,j}\, \boldsymbol{f_y}(T_j\,,\, Y_j) \cdot (\dot{y}_n + \dot{z}_j) + h\,\gamma\, \boldsymbol{f_y}(T_i\,,\, Y_i) \cdot \dot{y}_n\,.$$

FATODE allows users to choose to solve the linear system (3.15) directly at the expense of an additional LU decomposition of the matrix $\mathbf{R}_n(\gamma, T_i, Y_i)$ per stage, or to apply simplified Newton iterations of the form

$$\begin{aligned} \mathbf{R}_n(\gamma, t_n, y_n) \cdot \Delta \dot{z}_i^{[m]} &= \dot{z}_i^{[m]} - h \sum_{j=1}^{i} a_{i,j} \, \boldsymbol{f_y}(T_j \, , \, Y_j) \cdot (\dot{y}_n + \dot{z}_j^{[m]}) \\ \dot{z}_i^{[m+1]} &= \dot{z}_i^{[m]} - \Delta \dot{z}_i^{[m]} \, , \quad m = 0, 1, \cdots . \end{aligned} \tag{3.15}$$

The LU decomposition of the matrix $\mathbf{R}_n(\gamma, t_n, y_n)$ is also necessary in forward integration. So the equations (3.15) re-use the LU decomposition which is available after the equations (3.6) are calculated in each step.

Fully implicit RK methods. For fully implicit Runge-Kutta methods two options are available for solving the system (3.14). One is to construct the $ds \times ds$ linear system (3.14) explicitly and solve it directly by factorizing the matrix $\widehat{\mathbf{R}}_n$.

The other is to apply simplified Newton iterations of the form

$$\begin{aligned} \overline{\mathbf{R}}_n \cdot \Delta \dot{Z}^{[m]} &= \widehat{\mathbf{R}}_n \cdot \left(\mathbf{1}_s \otimes \dot{y}_n + \dot{Z}^{[m]} \right) - \mathbf{1}_s \otimes \dot{y}_n \\ \dot{Z}^{[m+1]} &= \dot{Z}^{[m]} - \Delta \dot{Z}^{[m]} \, , \quad m = 0, 1, \cdots . \end{aligned} \tag{3.16}$$

The matrix $\overline{\mathbf{R}}_n$ of the resulting $ds \times ds$ linear system is available from the forward solution process, i.e., the calculations of the equations (3.9). The real and complex LU decompositions can be reused. According to our experience, the second option is usually more efficient than the first one for large systems.

Rosenbrock methods. The tangent linear Rosenbrock method consists of the formula formula (3.10) plus the sensitivity part, which is obtained by differentiating the formula (3.10). In each step we solve the combined set of equations

$$\widetilde{\mathbf{R}}(h\gamma, t_n, y_n) \cdot k_i = f(T_i, Y_i) + \sum_{j=1}^{i-1} \frac{c_{i,j}}{h} \, k_j + h\gamma_i k_i f_t(t_n, y_n) \, , \tag{3.17a}$$

$$\begin{aligned} \widetilde{\mathbf{R}}(h\gamma, t_n, y_n) \cdot \dot{k}_i = \boldsymbol{f_y}(T_i, Y_i) \cdot \left(\dot{y}_n + \sum_{j=1}^{i-1} a_{i,j} \dot{k}_j \right) + \sum_{j=1}^{i-1} \frac{c_{i,j}}{h} \, \dot{k}_j \\ + (\dot{y}_n \cdot \boldsymbol{f_{y,y}}(t_n, \, y_n)) \cdot k_i + h \, \gamma_i \, f_{y,t}(t_n, \, y_n) \cdot \dot{y}_n \, , \end{aligned} \tag{3.17b}$$

$$y_{n+1} = y_n + \sum_{i=1}^{s} m_i \, k_i, \tag{3.17c}$$

$$\dot{y}_{n+1} = \dot{y}_n + \sum_{i=1}^{s} m_i \, \dot{k}_i \, . \tag{3.17d}$$

The stage vectors $\dot{k}_i$ are obtained in succession by solving a sequence of linear systems, all of which re-use the LU decomposition of $\widetilde{\mathbf{R}}(h\gamma, t_n, y_n)$ performed in (3.10). Formula (3.17b) involves the Hessian tensor $\boldsymbol{f_{y,y}}(t_n, \, y_n)$. In practice, an analytical Hessian tensor is difficult to obtain, and its evaluation is costly in both CPU time and memory storage. Note that the above equation only needs the product $(\dot{y}_n \cdot \boldsymbol{f_{y,y}}(t_n, \, y_n)) \cdot k_i$. Such terms can be obtained efficiently using automatic differentiation [4, 13] twice, in forward over reverse mode.

3.3. Adjoint model integration. Adjoint sensitivity analysis provides an efficient alternative to the direct method when gradients of a relatively few derived functionals with respect to many model parameters required. The continuous (differentiate then discretize) and the discrete (discretize-then-differentiate) adjoint approaches lead, in general, to different computational results [27]. The continuous adjoint approach requires interpolation to obtain intermediate state variables at the times required by the the backward integration, which brings additional computational effort. The discrete adjoint approach follows exactly the same sequence of time steps as the forward integration, but in reverse order.

FATODE implements discrete adjoints of all the methods. Such discrete adjoints have good theoretical properties, in the sense that they are consistent discretizations of the adjoint ODE [21, 22]. For efficiency our implementation distinguishes between sensitivities with respect to initial conditions and sensitivities with respect to parameters. We first discuss sensitivities with respect to initial conditions.

The goal is to evaluate the sensitivities a scalar function of interest

$$\Psi = g(y(t_F)) \tag{3.18}$$

with respect to the initial conditions. The discrete adjoint model equations are obtained directly from the discrete forward model equations

$$y_{n+1} = \Phi^n(y_n), \quad n = 0, \ldots, N-1 \tag{3.19}$$

where Φ^n represents the one-step numerical integration formula which advances the solution from t_n to t_{n+1}.

The discrete adjoint model equations propagates the adjoint variables λ_n backwards in time

$$\lambda_N = \boldsymbol{g}_{\boldsymbol{y}}^T(y_N)\,; \quad \lambda_n = \boldsymbol{\Phi}_{\boldsymbol{y}}^n(y_n)^T \cdot \lambda_{n+1}, \quad n = N-1, \ldots, 1. \tag{3.20}$$

The adjoint solution at the initial time represents the sensitivities

$$(\partial\Psi/\partial y_0)^T = \lambda_0. \tag{3.21}$$

For details on derivation see [23, 21].

The discrete adjoint Runge-Kutta method [15] solving the discrete adjoint equations (3.20) reads

$$u_i = h\, \boldsymbol{f}_{\boldsymbol{y}}^T\left(T_i, Y_i\right) \cdot \left(b_i \lambda_{n+1} + \sum_{j=1}^{s} a_{j,i} u_j \right), \quad i = s, \ldots, 1\,, \tag{3.22a}$$

$$\lambda_n = \lambda_{n+1} + \sum_{j=1}^{s} u_j\,. \tag{3.22b}$$

The stage equations (3.22a) form a $ds \times ds$ linear system involving the transpose of matrix (3.2c):

$$\begin{aligned} &U = \left[u_1^T \cdots u_s^T\right]^T, \\ &\widehat{\mathbf{R}}_n^T \cdot U = h\, \left[b_1 \lambda_{n+1}^T\, \boldsymbol{f}_{\boldsymbol{y}}\left(T_1, Y_1\right) \cdots b_s \lambda_{n+1}^T\, \boldsymbol{f}_{\boldsymbol{y}}\left(T_s, Y_s\right)\right]^T. \end{aligned} \tag{3.23}$$

Explicit RK methods. The stage equations (3.22a) are solved in succession for stages s down to 1:

$$
\begin{aligned}
& u_s = h\, b_s\, \boldsymbol{f}_{\boldsymbol{y}}^T(T_s,\, Y_s)\, \lambda^{n+1}\,, \\
& u_i = h\, \boldsymbol{f}_{\boldsymbol{y}}^T(T_i,\, Y_i) \cdot \left(b_i\, \lambda^{n+1} + \sum_{j=i+1}^{s} a_{j,i}\, u_j \right)\,, \quad i = s-1, \ldots, 1\,.
\end{aligned}
\tag{3.24}
$$

Each stage i requires the computation of the Jacobian $\boldsymbol{f}_{\boldsymbol{y}}(T_i, Y_i)$, forming the vector $b_i\, \lambda^{n+1} + \sum_{j=i+1}^{s} a_{j,i} u_j$ from previously computed stages $u_s \ldots u_{i+1}$, and performing one Jacobian vector product.

Singly diagonally implicit Runge-Kutta methods. For SDIRK methods the s stages of the system (3.22a) are solved successively from the last stage to the first. Each stage requires the solution of a different linear system:

$$
\mathbf{R}_n(\gamma, T_i, Y_i) \cdot u_i = h\, \boldsymbol{f}_{\boldsymbol{y}}^T(T_i, Y_i) \cdot \left(b_i\, \lambda^{n+1} + \sum_{j=i+1}^{s} a_{j,i} u_j \right)\,, \quad i = s, \ldots, 1\,.
\tag{3.25}
$$

FATODE offers two options: to form and solve one linear system (3.25) per stage, or to employ simplified Newton iterations of the form (3.15) and re-use the LU decomposition of $\mathbf{R}_n(\gamma, t_n, y_n)$ for all stages.

Fully implicit Runge-Kutta methods. For the fully implicit Runge-Kutta methods the $ds \times ds$ system (3.23) is fully coupled. FATODE offers two approaches to solve it. The first is to build and solve directly (3.23) via a $ds \times ds$ LU decomposition of $\widehat{\mathbf{R}}_n$. The second approach uses simplified Newton iterations of the form (3.14), where $\widehat{\mathbf{R}}_n$ is replaced by $\overline{\mathbf{R}}_n$ in (3.23), and the transformation to real and complex systems is performed. The real and complex LU factorizations associated with the matrix $\overline{\mathbf{R}}_n$ are re-used in all iterations. These factorizations are computed during the forward solution, and in principle they can be checkpointed. The tradeoff between the size of the LU factorizations to store and the time needed to recompute them will determine the best strategy.

Rosenbrock methods. The discrete Rosenbrock adjoint [1] is

$$
\widetilde{\mathbf{R}}^T\,(h\gamma, t_n, y_n) \cdot u_i = m_i\, \lambda_{n+1} + \sum_{j=i+1}^{s} \left(a_{j,i}\, v_j + \frac{c_{j,i}}{h}\, u_j \right)\,,
\tag{3.26a}
$$

$$
v_i = \boldsymbol{f}_{\boldsymbol{y}}^T\,(T_i\,,\, Y_i) \cdot u_i\,, \quad i = s, \ldots, 1\,,
\tag{3.26b}
$$

$$
\lambda_n = \lambda_{n+1} + \sum_{i=1}^{s} (u_i \cdot \boldsymbol{f}_{\boldsymbol{y},\boldsymbol{y}}(t_n, y_n)) \cdot k_i + h \boldsymbol{f}_{\boldsymbol{y},\boldsymbol{t}}^T(t_n, y_n) \cdot \sum_{i=1}^{s} \gamma_i u_i + \sum_{i=1}^{s} v_i\,.
\tag{3.26c}
$$

The linear system (3.26a) can be solved directly at each stage. Users have to supply a routine for calculating the term $(u_i \cdot \boldsymbol{f}_{\boldsymbol{y},\boldsymbol{y}}(t_n,\, y_n)) \cdot k_i$, whose meaning is explained in Appendix A. Automatic differentiation tools like TAMC [13] provide considerable help: the product between the Hessian transposed times vector can be obtained by two consecutive runs of TAMC in forward mode.

Checkpointing. The adjoint model requires two runs. First the forward code performs a regular integration of the ODE system. At each step the time t_n, the step size h, the state vector y_n, and the intermediate stage vectors z_i or k_i are all saved in checkpoints. Next, the discrete adjoint code is run backward in time and

traces the same sequence of steps, but in reverse order. At each step, the data in the checkpoint storage is retrieved and used to construct the adjoint system. The results of LU factorizations can, in principle, be checkpointed and reused in the adjoint calculations. This is not done in our implementation, due to the following reasons. The memory and input/output costs for storing LU factors can be extremely large when the system is large or when many steps are taken. Next, FATODE is designed such that the details of the linear solver are transparent to the main integrator. This transparency cannot be preserved when one builds and manages a stack for data structures that are specific to particular linear solvers.

3.4. Adjoint sensitivities with respect to a vector of parameters. We now consider the case where the adjoint sensitivity is computed with respect to a time-independent vector of parameters $p \in \mathbb{R}^m$ which appears in the right hand side of (1.1). The quantity of interest is a scalar derived function in the general form

$$\Psi = g\left(y(t_F), p\right) + \int_{t_0}^{t_F} r\left(t, y(t), p\right)\, dt\,. \tag{3.27}$$

To account for the evolution of the parameters we add the formal equations for the parameter evolution $p' = 0$. To compute the cost function (3.27) we add the quadrature variables $q \in \mathbb{R}$ whose evolution is defined by $q(t_0) = 0$ and $q' = r\,(y, p)$. We have that $\Psi = g\left(y(t_F), p\right) + q(t_F)$. The equation (1.1) becomes

$$\begin{bmatrix} y \\ p \\ q \end{bmatrix}' = \begin{bmatrix} f\,(t, y, p) \\ 0 \\ r\,(t, y, p) \end{bmatrix}, \quad t_0 \le t \le t_f\,; \quad \begin{bmatrix} y(t_0) \\ p(t_0) \\ q(t_0) \end{bmatrix} = \begin{bmatrix} y_0 \\ p \\ 0 \end{bmatrix}. \tag{3.28}$$

As shown in Appendix C the numerical solution of (3.28) provides the discrete y_n and q_n. The discrete adjoint model equations calculate the adjoint variables λ_n and μ_n backward in time, such that

$$\lambda_N = \boldsymbol{g}_{\boldsymbol{y}}^T(y_N, p)\,, \quad \mu_N = \boldsymbol{g}_{\boldsymbol{p}}^T(y_N, p)\,; \quad \lambda_0 = \left(\partial\Psi/\partial y_0\right)^T\,, \quad \mu_0 = \left(\partial\Psi/\partial p\right)^T\,. \tag{3.29}$$

For details on derivation see [21] and Appendix C .

Runge-Kutta methods. The derivation of sensitivities with respect to parameters for Runge Kutta methods is detailed in Appendix D. Consider the Runge-Kutta method (2.1) applied to the extended ODE system (3.28)

$$Y_i = y_n + h \sum_{j=1}^{s} a_{i,j}\, f\left(T_j\,,\, Y_j\,,\, p\right)\,, \quad i = 1, \ldots, s\,, \tag{3.30a}$$

$$y_{n+1} = y_n + h \sum_{j=1}^{s} b_j\, f\left(T_j, Y_j, p\right)\,, \tag{3.30b}$$

$$q_{n+1} = q_n + h \sum_{j=1}^{s} b_j\, r\left(T_j, Y_j, p\right)\,. \tag{3.30c}$$

Note that, since r does not depend on q, there is no need to compute the stage values for the quadrature variable.

As shown in Appendix D the discrete adjoint Runge-Kutta method (D.2) reads

$$u_i = h\, \boldsymbol{f}_{\boldsymbol{y}}^T(T_i, Y_i, p) \cdot \left(b_i\, \lambda_{n+1} + \sum_{j=1}^{s} a_{j,i}\, u_j \right) + h\, b_i\, \boldsymbol{r}_{\boldsymbol{y}}^T(T_i, Y_i, p)\,, \tag{3.31a}$$

$$v_i = h\, \boldsymbol{f_p}^T(T_i, Y_i, p) \cdot \left(b_i\, \lambda_{n+1} + \sum_{j=1}^{s} a_{j,i}\, u_j \right) + h\, b_i\, \boldsymbol{r_p}^T(T_i, Y_i, p)\,, \quad i = s \dots 1\,, \tag{3.31b}$$

$$\lambda_n = \lambda_{n+1} + \sum_{j=1}^{s} u_j\,, \tag{3.31c}$$

$$\mu_n = \mu_{n+1} + \sum_{j=1}^{s} v_j\,. \tag{3.31d}$$

The stages u_i are obtained by solving the system in a similar way as solving system (3.24) and the implementation differs between SDIRK and fully implicit 3-stage Runge-Kutta method. Then v_i can be readily obtained from the right-hand side calculation.

Rosenbrock methods. Applying the Rosenbrock method (3.10) to the extended system (3.28) gives the following formula for evaluating the quadrature term in the cost functional (3.27):

$$\ell_i = h\gamma \left(r(T_i, Y_i) + \sum_{j=1}^{i-1} \frac{c_{i,j}}{h} \ell_j + h\gamma_i r_t + r_y \cdot k_i \right)\,, \tag{3.32a}$$

$$q_{n+1} = q_n + \sum_{i=1}^{s} m_i \ell_i\,. \tag{3.32b}$$

Equation (3.32) is evaluated simultaneously with the ODE integration. The derivation of the adjoint Rosenbrock method for computing sensitivities with respect to parameters is given in Appendix E.

$$\begin{aligned}
&\widetilde{\mathbf{R}}_n^T(\gamma, t_n, y_n)\, u_i = \widehat{u}_i\, r_y^T(t_n, y_n, p) + m_i\, \lambda_{n+1} + \sum_{j=i+1}^{s} \left(a_{j,i}\, v_j + \frac{c_{j,i}}{h}\, u_j \right) \\
&v_i = h\, \boldsymbol{f_y}^{\boldsymbol{T}}(T_i, Y_i, p) \cdot u_i + h\, \widehat{u}_i\, r_y^T(T_i, Y_i, p) \\
&\overline{v}_i = h\, \boldsymbol{f_p}^{\boldsymbol{T}}(T_i, Y_i, p) \cdot u_i + h\, \widehat{u}_i\, r_p^T(T_i, Y_i, p) \\
&\lambda_n = \lambda_{n+1} + \sum_{i=1}^{s} \Big((u_i \cdot \boldsymbol{f_{y,y}}) \cdot k_i + (\widehat{u}_i \cdot r_{y,y}) \cdot k_i \Big) \\
&\qquad + h\, \boldsymbol{f}_{\boldsymbol{y,t}}^T(t_n, y_n, p) \cdot \sum_{i=1}^{s} \gamma_i u_i + h\, r_{y,t}^T(t_n, y_n, p) \cdot \sum_{i=1}^{s} \gamma_i \overline{u}_i + \sum_{i=1}^{s} v_i \\
&\mu_n = \mu_{n+1} + \sum_{i=1}^{s} \Big((u_i \cdot \boldsymbol{f_{p,y}}) \cdot k_i + (\widehat{u}_i \cdot r_{p,y}) \cdot k_i \Big) \\
&\qquad + h\, \boldsymbol{f}_{\boldsymbol{p,t}}^{\boldsymbol{T}}(t_n, y_n, p) \cdot \sum_{i=1}^{s} \gamma_i u_i + h\, r_{p,t}^T(t_n, y_n, p) \cdot \sum_{i=1}^{s} \gamma_i \overline{u}_i + \sum_{i=1}^{s} \overline{v}_i\,.
\end{aligned}$$

where

$$\widehat{\boldsymbol{u}} = \boldsymbol{\gamma} \cdot \mathrm{diag}(\boldsymbol{\gamma}^{-1}) \cdot \boldsymbol{\gamma}^{-T} \cdot \boldsymbol{b}\,.$$

3.5. Linear solvers. The most computationally intensive part in solving large-scale ODE systems by implicit methods is the solution of linear systems at each step. Linear systems arise from the simplified Newton iterations applied to solve the nonlinear systems in case of fully implicit Runge-Kutta methods and SDIRK methods. For Rosenbrock methods, linear systems appear directly in the formula (2.3). In general, all implicit time stepping methods in FATODE require the solution of linear systems with matrices $\mathbf{R}$ or $\widetilde{\mathbf{R}}$ defined in (3.2). These matrices inherit the sparsity structure of the system Jacobian.

Best efficiency is achieved when taking advantage of the problem-specific characteristics. Consequently, FATODE was designed to allow users to provide their own linear solvers and sparse data structures. The linear system can be solved by either direct methods or iterative methods. Direct methods perform LU factorizations which are reused for efficiency. We have incorporated three direct methods in current version of FATODE. For dense systems calls to LAPACK [3] routines are provided. For large sparse systems, substantial memory and execution time benefits can be gained by calling the direct sparse solvers UMFPACK [7] and SUPERLU [8] and representing the sparse matrices in a compressed column format. Interfaces to both these codes are provided. Iterative methods can be more efficient than direct methods for solving extremely large linear systems regardless of the reuse of LU decompositions. The choice between direct and iterative methods depends on the specific properties of the problems. FATODE does not provide iterative solvers; these need to be supplied by the user.

3.6. Step size control. Variable step size control is routinely adopted by general ODE solvers to control numerical errors and maximize efficiency. FATODE's forward integrators use estimates of the truncation error to decide whether the step is accepted or rejected and to compute the next step size. The maximum number of integration steps before unsuccessful return can be specified by the user.

Two different error estimation options are provided for fully implicit Runge-Kutta methods. One is the classical error estimation [17] which uses an embedded third order method based on an additional explicit stage. The embedded solution is

$$\hat{y}_{n+1} = y_n + h\left(\hat{b}_0\, f(t_n, y_n) + \sum_{i=1}^{s} \hat{b}_i\, f(t_n + c_i\, h, y_n + z_i)\right) \tag{3.33}$$

where the increment vectors z_i are already computed in previous stages. The second estimator uses two additional stages: an explicit stage at the beginning of the time step and another SDIRK stage which re-uses the LU decomposition from the solution of the main integrator. The SDIRK stage reads

$$\hat{y}_{n+1} = y_n + h\left(\hat{b}_0\, f(t_n, y_n) + \sum_{i=1}^{s} \hat{b}_i\, f(t_n + c_i h, y_n + z_i) + \gamma\, f(t_n + h, y_n + z_{s+1})\right).$$

The coefficients are chosen such that the order of consistency of $\hat{y}_{n+1}$ is $\hat{p} = p - 1$, where p is the order of y_{n+1}. The difference vector $Est = \hat{y}_{n+1} - y_{n+1}$ is used as a local error estimator. Based on our experience, the classical error estimator is inaccurate for low relative tolerances, while the SDIRK error estimator yields very good results and keeps the accuracy of the solution below the specified tolerance. For the ERK, SDIRK, and Rosenbrock methods the classical error estimators based on embedded solutions are implemented; they proved to work well in practice.

The local error test is performed as follows. Let $Tol_k = atol_k + rtol_k \cdot |y_{n+1,k}|$, where *atol* and *rtol* are the absolute and relative error tolerance specified by users and $|y_{n+1,k}|$ is the absolute value of the k-th component of y_{n+1}. The relative and absolute error tolerances can be either vector or scalar (in which case the same tolerance values are used for all components k). The local error test takes the form

$$Err = \sqrt{\frac{1}{d}\sum_{k=1}^{d}\left(\frac{Est_k}{Tol_k}\right)^2} < 1\,. \tag{3.34}$$

If the test passes the step size is accepted, otherwise it rejected and the step is recomputed. The new step size, for both accepted and rejected cases, is given by the formula

$$h_{\text{new}} = h_{\text{old}} \cdot \min\left(\mathcal{F}_{\max}, \max\left(\mathcal{F}_{\min}, \mathcal{F}_{\text{safe}} \cdot Err^{-1/(\hat{p}+1)}\right)\right)\,,$$

where $\mathcal{F}_{\max}$ is the upper bound on step increase factor, $\mathcal{F}_{\min}$ the lower bound on step decrease factor, and $\mathcal{F}_{\text{safe}}$ is a safety factor. The default values of these factors depend on the specific method. All of them can be changed by the user in the parameter settings. If the step size is rejected at the first step, the step increase factor $\mathcal{F}_{\max}$ is set to 1. and the step size is reduced by a factor of 10. Furthermore, the step size can be constrained by minimum ($h_{\min}$) and maximum ($h_{\max}$) values. The starting step size h_{start} can be specified by the user.

For the tangent linear model integration FATODE provides two options for controlling errors in the sensitivities. The first option is to use only the forward error estimates for step size control. The second option is to estimate the truncation errors for both the forward solution and the tangent linear model solution. The solution error is taken as the maximum between the forward truncation error and the truncation error of any column of the sensitivities. This solution error is then used to control the step size.

The discrete adjoint model integration traces the same sequence of steps as the forward integration, in reverse order. Therefore, the choice of the step sizes is completely determined during the forward integration, and the accuracy of the adjoint solution will depend on the error control performed during the forward run.

3.7. Code organization. FATODE implements four types of methods: explicit, fully implicit, and singly diagonally implicit Runge-Kutta methods, and Rosenbrock methods. For each family of methods, a module is given for the main integrator, a module for linear system solver interface, and a set of modules for generic linear system solvers. They form the basic structure shown in Figure 3.1.

The main integration module provides the basic time stepping framework, and is independent of the linear system solver. The forward integrator calls the user-supplied right-hand side function and Jacobian and accesses the linear system solver, in order to compute the ODE solution. The tangent linear integrator and the adjoint integrator require users to also specify the parameters of interest as additional inputs. The tangent linear integrator, by default, considers the initial conditions of the ODE system as the parameters of interest and computes the sensitivity of the ODE solution with respect to them. The adjoint integrator, by default, computes the sensitivity of an objective function Ψ with respect to the initial conditions. The function and its derivatives are supplied by the user; function derivatives are used to define the adjoint initial conditions, see (3.20) and (C.3). FATODE also implements sensitivities

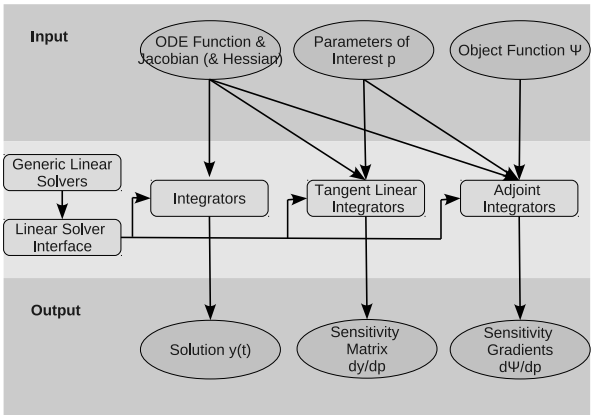


FIG. 3.1. *Overall structure of* FATODE

of a general cost function (3.27) with respect to parameters other than the initial conditions (e.g., reaction coefficients in a chemical kinetic ODE system). This version of adjoint code is based on the techniques described in Section 3.4.

The linear system solver module provides interfaces to generic linear solvers, which can be called seamlessly by the routines in the integrator modules. Specifically, we provide interfaces to the following four generic routines: *LS_Init*, *LS_Decomp*, *LS_Solve*, and *LS_Free*. *LS_Init* deals with initialization and memory allocation required by the specific linear solver. *LS_Decomp* performs the LU decomposition, and *LU_Solve* solves the triangular systems by substitution. *LS_Free* frees the memory allocated and clears the objects created during the initialization stage. The main integrator makes calls to these functions without having to consider the implementation details of these routines; many linear algebra packages can be used without having to modify the time stepping code. The user can choose one of the linear solvers provided (LAPACK, UMFPACK, SUPERLU), or can add a new linear solver by providing their own implementation and add it to this module. The linear system solver module also contains several routines related to computing the Jacobian and its transpose, and taking the product of the Jacobian (or its transpose) with a vector vector. The implementation of these operations depends critically on the data structures used to represent the Jacobian. For example, Jacobian matrix could be stored in either dense format or in one of the many available sparse formats, and each representation leads to a totally different implementation. For computational efficiency users should to take advantage of the appropriate data structure for the Jacobian matrix when they add a new linear solver. All the required code modifications are done within this single module.

Different families of methods require solving different types of linear systems. Fully implicit Runge-Kutta methods involve real and complex linear systems of dimension $d \times d$, or real linear systems of dimension $ds \times ds$. Singly diagonally implicit Runge-Kutta and Rosenbrock methods deal with only real-valued linear systems of dimension $d \times d$. Jacobian related operations also vary greatly between the three families of implicit methods. To manage this complexity we provide individual linear solver modules for each of the implicit time stepping families in FATODE.

The linear system modules in FATODE currently include three linear solvers: LAPACK [3], UMFPACK [7] and SUPERLU [8]. Each requires its own data format, e.g., a

full matrix is used with the LAPACK version, while a compressed column sparse matrix is needed with UMFPACK or SUPERLU versions. The existing linear solvers can be used as templates for users who wish to add their own. A new solver requires the user to provide the four basic solution routines, as well as the Jacobian related routines if necessary.

4. Code usage. This section discusses the usage of FATODE for forward ODE integration, direct sensitivity analysis via tangent linear models, and computing sensitivities of a cost function with respect to initial conditions and specified parameters via adjoint models. The user interface is very similar to classic ODE solvers such as LSODE, VODE, RADAU5. A uniform interface is provided for all four integration families in FATODE. This interface allows users to control nearly every aspect of the solution process, such as method selection, step size adjustment, error estimation, convergence of simplified Newton iterations, and so on.

4.1. ODE integration. The call to the integrator routine is as follows:

```
CALL INTEGRATE(
     TIN,TOUT,NVAR,NNZERO,VAR,RTOL,ATOL,FUN,ICNTRL,
     RCNTRL,ISTATUS,RSTATUS,IERR)
```

All arguments except ISTATUS, RSTATUS, IERR are input arguments. ICNTRL, RCNTRL, ISTATUS, RSTATUS, IERR are optional. The arguments have the following meaning:

TIN	start time
TOUT	end time
NVAR	the number of ordinary differential equations to be solved
NNZERO	number of non-zero elements in the Jacobian matrix
VAR	a vector of length NVAR containing the initial values of the dependent variables. Upon return, VAR is the numerical solution at the last step.
RTOL	relative error tolerance
ATOL	absolute error tolerance
FUN	the name of the user-supplied subroutine that computes the f in the ODE
ICNTRL	an optional integer-valued array containing input parameters
RCNTRL	an optional real-valued array containing input parameters
ISTATUS	an optional integer-valued array containing output statistics
RSTATUS	an optional real-valued array containing output statistics
IERR	job status upon return

The value of variable NNZERO is only needed in the case that a sparse linear solver is used. ICNTRL and RCNTRL provide a wide range of options for users to tune the behavior of the integrator. The first four elements in ICNTRL and the first seven elements in RCNTRL are general options which apply to all families. ICNTRL(1) indicates whether the system is autonomous. ICNTRL(2) indicates whether ATOL and RTOL are vectors or scalars. ICNTRL(3) allows the user to select a particular method within each family. ICNTRL(4) specifies maximum number of integration steps before unsuccessful return. RCNTRL(1-7) control the behavior of the integration step size adjustment process.

Some switches are specific to certain families of methods. Simplified Newton iterations are used in fully implicit and singly diagonally implicit Runge-Kutta methods.

For these two families, users can specify the maximum number of Newton iterations, stopping criterion, bounds on step decrease, increase, and on step rejection. Default values are assigned to these parameters after careful experimenting and using [16]. Additional options regarding step size control and error estimation are provided for the fully implicit Runge-Kutta family. In this family two step-size strategies are implemented: the classical approach and the modified predictive controller [14]. There are also two strategies implemented for error estimation: the classical error estimation [16] using the embedded third order method based on an additional explicit stage, and an error estimator based on an additional SDIRK stage which re-uses the real LU decomposition from the main method.

4.2. Tangent linear methods and forward sensitivity calculations. The call to the tangent linear model integrator is as follows:

```
CALL INTEGRATE_TLM(
     NVAR,NTLM,NNZERO,Y,Y_TLM,TIN,TOUT,ATOL_TLM,RTOL_TLM,
     ATOL,RTOL,FUN,ICNTRL,RCNTRL,ISTATUS,RSTATUS,IERR)
```

There are four more arguments compared to forward integrator: NTLM specifies the number of sensitivity coefficients to be computed, Y_TLM contains sensitivities of Y with respect to the specified coefficients, and ATOL_TLM and RTOL_TLM are used to calculate error estimates for sensitivity coefficients if the switch ICNTRL(9) is set to 1. All the options of the forward code also apply to TLM code. Several TLM-specific options are as follows:

- use forward error estimation only, or control the truncation errors for both the forward and the sensitivity solutions; and
- control the convergence of TLM simplified Newton iterations (3.15), (3.16), for fully implicit Runge-Kutta and SDIRK families.

The TLM integrators generate sensitivity results along with the numerical solution of forward ODE system. When sensitivities of $y(t)$ with respect to fixed model parameters are desired the augmenting technique described in Section 3.4 can be applied. Note that the argument list for the Rosenbrock method includes the additional function HESS_VEC for calculating the Hessian times vector term in (3.17b).

4.3. Adjoint methods and discrete adjoint sensitivity calculations. The call to the adjoint model integrator is as follows:

```
CALL INTEGRATE_ADJ(
     NVAR,NP,NADJ,NNZERO,Y,Lambda,Mu,TIN,TOUT,ATOL_adj,
     RTOL_adj,ATOL,RTOL,FUN,JAC,ADJINIT,HESSTR_VEC,JACP,
     DRDY,DRDP,HESSTR_VEC_F_PY,HESSTR_VEC_R_PY,HESSTR_VEC_R,
     ICNTRL_U,RCNTRL_U,ISTATUS_U,RSTATUS_U,Q,QFUN)
```

The integer arguments NADJ and NP specify the number of cost functionals and the number of system parameters for which adjoints are evaluated simultaneously. Lambda (of dimension NVAR $\times$ NADJ) and Mu (optional variable of dimension NP $\times$ NADJ) contain the sensitivities of the cost functional(s) with respect to the initial conditions and system parameters respectively. ADJINIT is a routine provided by users to initialize the adjoint variables; it is called after the forward run ends, and before the backward run starts. The optional variable Q represents the quadrature term in the cost functional and the optional routine QFUN computes the integrand for the quadrature term. JACP, DRDY, DRDP are optional routines to calculate the derivatives $\boldsymbol{f_p}$, $\boldsymbol{r_y}$, and $\boldsymbol{r_p}$ discussed in Section 3.4.

Several additional optional arguments (with names beginning with HESS), perform Hessian related operations and are only available in the interface for the Rosenbrock methods. The behavior of the program is controlled by the user through these optional arguments.

- Mu and JACP are required when the sensitivities with respect to system parameters are desired.
- Q and QFUN are required when the cost functional(s) contain(s) a quadrature term. Upon completion Q stores the value of the quadrature term at the final step.
- DRDY (DRDP) are required when the cost functional(s) contain(s) a quadrature term and the sensitivities with respect to initial conditions (or system parameters, respectively) are desired.

The integrator performs a forward run from t_0 (specified by TIN) to t_F (specified by TOUT) followed by a backward, adjoint run from t_F to t_0. ATOL and RTOL define the tolerances for the forward run while ATOL_ADJ and RTOL_ADJ for the backward run. Note that ATOL_ADJ and RTOL_ADJ control only the convergence of the iterations, not the time steps since the backward run follows exactly the same sequence of time steps as the forward run in reverse order.

If the simplified Newton iterations for solving the adjoint stage variables do not converge, the code switches automatically to a direct solution method (3.23) – at the expense of additional LU decompositions. The other options for adjoint methods are the same as with the forward code.

5. Numerical experiments. In this section we illustrate the capabilities of FATODE and evaluate the performance of each family of methods. We compare FATODE with the well established code CVODE and CVODES within SUNDIALS [18] for both the ODE solution, and direct and adjoint sensitivity analysis. In addition, we assess the efficiency of sparse linear solvers incorporated in FATODE by comparing their performance with LAPACK full linear solvers. All the experiments are performed on a 8-core (1.86Ghz) Intel Xeon workstation running Fedora 14 (x86_64) Linux. PGI FORTRAN Version 7.2-5 is used for compilation, with level O3 optimization. Unless stated otherwise, the default settings for parameters are used in all FATODE calls.

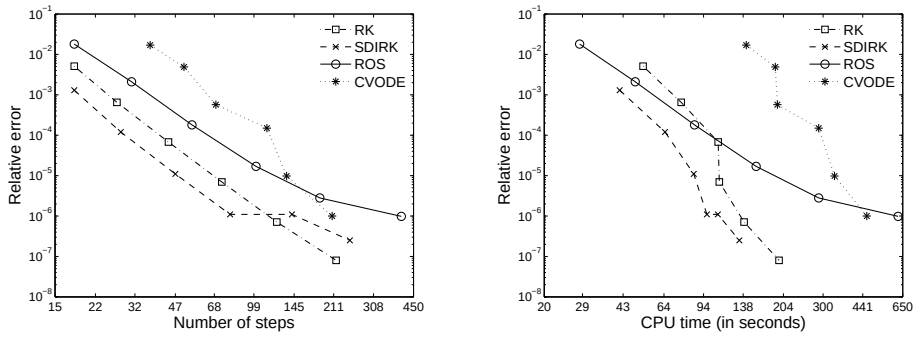
We consider the two-dimensional Saint-Venant system of shallow water equations

$$
\begin{aligned}
&\frac{\partial}{\partial t} h + \frac{\partial}{\partial x} (uh) + \frac{\partial}{\partial y} (vh) = 0, \\
&\frac{\partial}{\partial t} (uh) + \frac{\partial}{\partial x} (u^2 + \frac{1}{2} g h^2) + \frac{\partial}{\partial y} (uvh) = 0, \\
&\frac{\partial}{\partial t} (vh) + \frac{\partial}{\partial t} (uvh) + \frac{\partial}{\partial y} (v^2 h + \frac{1}{2} g h^2) = 0,
\end{aligned}
\tag{5.1}
$$

on the spatial domain $\Omega = [-3, 3]^2$, where $u(t, x, y)$, $v(t, x, y)$ are the components of the velocity field, $h(t, x, y)$ is the fluid layer thickness, and g denotes the standard value of the gravitational acceleration.

The shallow water equations (5.2) are converted to a semi-discrete form using third order upwind finite differences. The spatial domain is covered by a grid of size 40×40, resulting in an ODE system of dimension $40 \times 40 \times 3 = 4800$ which is solved by FATODE.

5.1. ODE solution. CVODE [18] is a general-purpose ODE solver which uses the Adams-Moulton method for non-stiff ODE systems and the BDF method for



(a) Relative error vs number of steps

(b) Relative error vs CPU time

FIG. 5.1. *Work-precision diagrams for the forward integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.11]$ *using stiff integrators. Comparison is made between FIRK, SDIRK, and Rosenbrock methods in* FATODE *with the BDF method in* CVODE. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $10^{-2}, \ldots, 10^{-7}$.

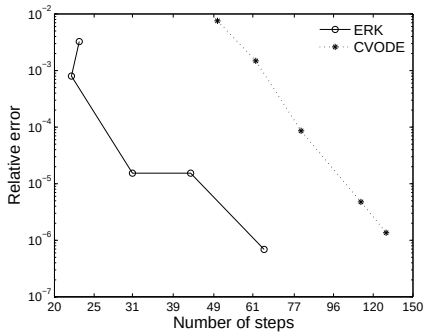
stiff ODE systems. Both methods are implemented in a variable-order variable-step form. In our test, we choose the option of the BDF method for the comparison with implicit methods in FATODE and the option of the Adams-Moulton method for the comparison with explicit Runge-Kutta methods in FATODE. From each family of implicit integrators in FATODE we select a representative fourth-order method for the tests: Lobatto3C (fully implicit Runge-Kutta), Rodas4 (Rosenbrock), and Sdirk-4a (singly diagonally implicit Runge-Kutta). The Gustafsson predictive error controller is used for all integrators. An additional SDIRK stage was used in the error estimator in the fully implicit Runge-Kutta integrator. For the explicit Runge-Kutta family we select the fifth-order seven-stage method DOPRI5.

Since solution of linear systems dominates the computational cost of implicit integration, especially for large-scale ODE systems, it is necessary to use the same linear solvers for both CVODE and FATODE for a fair performance comparison. In our comparison experiments we use the direct linear solver (DGETRF and DGETRS) from LAPACK. In all cases, the full Jacobian is supplied.

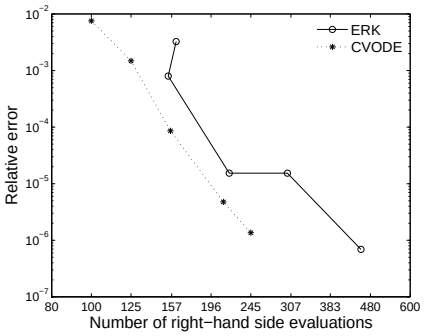
We vary both absolute and relative error tolerances from 10^{-2} to 10^{-7} to obtain solutions of different levels of accuracy (the absolute and relative error tolerances are equal to each other). A reference solution is obtained by using LSODE [20], a well known but relatively slow ODE solver, with a very tight relative and absolute tolerances of 10^{-9}. The relative error is defined in the $\mathbb{L}^2$ norm:

$$err = \frac{\|y_{t_F} - y_{\text{ref}}\|_2}{\|y_{\text{ref}}\|_2} \tag{5.2}$$

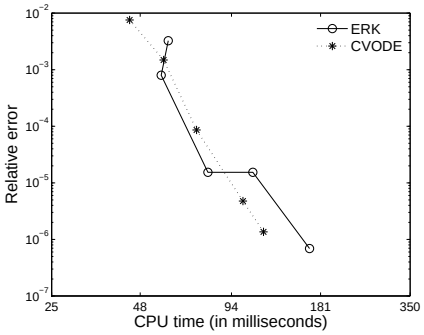
where y_{t_F} is the numerical solution at the final step and y_{ref} is the reference solution at t_F. The work-precision diagrams for the stiff solvers are shown in Figure 5.1. The results indicate that singly diagonally implicit and fully implicit Runge-Kutta methods implemented in FATODE outperform the BDF method implemented in CVODE, requiring considerably less time steps to reach a desired accuracy. The Rosenbrock method is also more efficient than CVODE for accuracy levels below 10^{-6} . The work-precision diagrams for the non-stiff solvers are shown in Figure 5.2. For the same level of accuracy, the explicit Runge-Kutta method in FATODE takes fewer steps than the



(a) Relative error vs number of steps



(b) relative error vs number of right-hand side evaluations



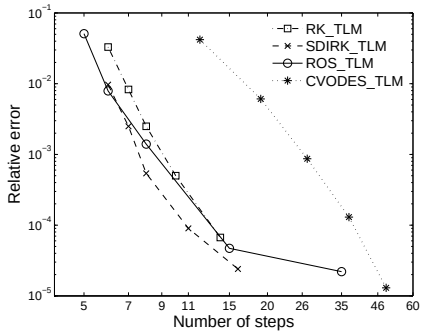
(c) Relative error vs CPU time

FIG. 5.2. *Computational statistics for the forward integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.11]$ *using nonstiff integrators. Comparison is made between the explicit RK method in* FATODE *with the Adams-Moulton method in* CVODE. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $0.3 \times 10^{-2}, \ldots, 0.3 \times 10^{-7}$.

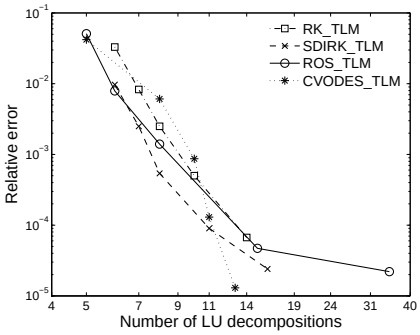
Adams-Moulton method in CVODE, but the cost per step is higher. The performance is similar in terms of CPU time.

5.2. Direct sensitivity analysis. We now calculate the sensitivities of all solution components at the final time with respect to the initial value of the first solution component $\partial y_i(t_F)/\partial y_1(t_0)$, $i = 1 \ldots n$ using tangent linear model integration. The FATODE results are compared against those obtained with CVODES [26], an extension of CVODE capable to perform sensitivity analysis. The LAPACK linear solvers are used in both CVODES and FATODE.

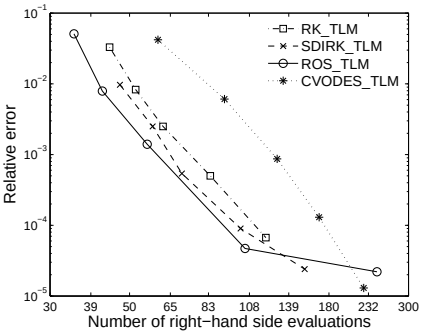
Tangent linear model results with stiff integrators are shown in Figure 5.3 and 5.4). The three implicit methods in FATODE and CVODES require nearly the same number of LU decompositions for relative errors below 10^{-4}. The performance of the SDIRK and Rosenbrock methods is comparable to that of the BDF method in CVODES. The fully implicit Runge-Kutta method is nearly three times more costly because it requires either solving a large real-valued system or solving a complex-valued system. It is also observed that FATODE saves a considerable number of steps and right-hand side evaluations. Tangent linear model results with non-stiff integrators are shown in Figure 5.4. A finite difference approximation to the product of Jacobian matrix and



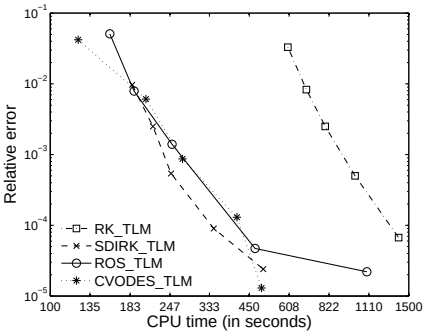
(a) Relative error vs number of steps



(b) Relative error vs number of LU decompositions



(c) Relative error vs number of right-hand side evaluations

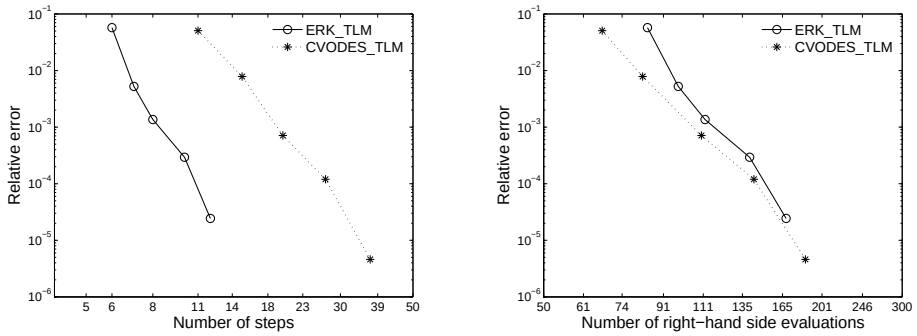


(d) Relative error vs CPU time

FIG. 5.3. *Computational statistics for the tangent linear model integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.011]$ *using stiff integrators. Comparison is made between implicit methods in* FATODE *with the BDF method in* CVODES. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $10^{-2}, \ldots, 10^{-6}$. *The tangent linear model computes the sensitivity* $\partial y_i(t_F)/\partial y_1(t_0),\ i = 1 \ldots d$.

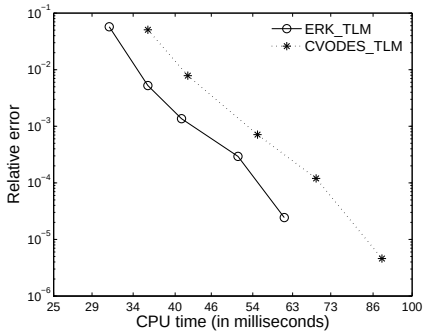
sensitivity vector is used with both the explicit Runge-Kutta method in FATODE and the Adams-Moulton method in CVODES for a fair comparison. The performance of the two codes is similar in terms of both CPU time and number of right-hand side evaluations.

5.3. Adjoint sensitivity analysis. We calculate the sensitivities of the first solution component at the final time with respect to all initial values $\partial y_1(t_F)/\partial y_i(t_0)$, $i = 1 \ldots d$ using adjoint model integration. Work-precision diagrams for the implicit solvers are shown in Figure 5.5. All implicit methods in FATODE outperforms the BDF method in CVODES in terms of work-precision ratio, with the highest efficiency being achieved by the Rosenbrock method. Work-precision diagrams for the explicit solvers are shown in Figure 5.6. The explicit Runge-Kutta method in FATODE runs slightly slower than the Adams-Moulton method in CVODES, though the former takes fewer steps. This is due to the fact that the explicit Runge-Kutta method in FATODE requires more Jacobian evaluations and Jacobian-vector products during the adjoint backward run.



(a) Relative error vs number of steps

(b) relative error vs number of right-hand side evaluations

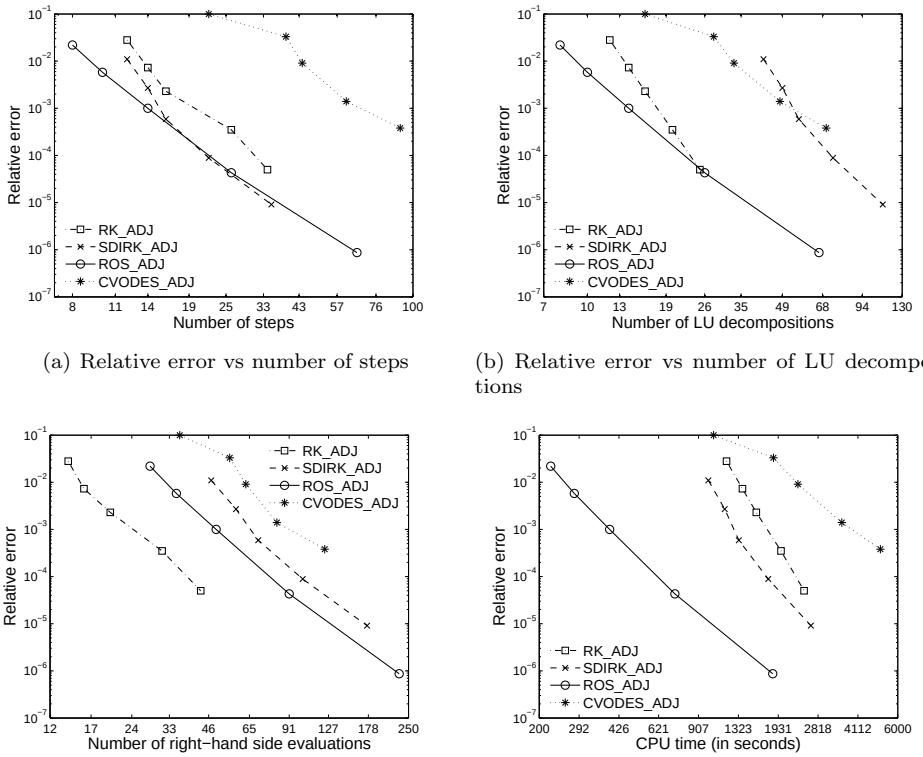


(c) Relative error vs CPU time

FIG. 5.4. *Computational statistics for the tangent linear model integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.011]$ *using nonstiff integrators. Comparison is made between the explicit RK method in* FATODE *with the Adams-Moulton method in* CVODES. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $10^{-2}, \ldots, 10^{-6}$. *The tangent linear model computes the sensitivity* $\partial y_i(t_F)/\partial y_1(t_0),\ i = 1 \ldots d$. *In* FATODE, *we choose the option to use only the forward error for step size control.*

5.4. Efficiency of sparse linear solvers. In our test, the Jacobian matrix is of dimension 4800, but only 100,800 elements are non-zeros; about 99.5625% of the entries of this sparse matrix are zeros. The incorporation of sparse linear solvers leads to significant computational savings and allows for very efficient forward, tangent linear, and adjoint model integration. Table 5.1 shows the compute times for all integrators in FATODE using different linear solvers. The tolerances are set as $ATOL = RTOL = 10^{-6}$ and time interval is from $t_0 = 0$ to $t_F = 0.011$. As expected, FATODE benefits significantly from sparse linear solvers. UMFPACK is typically a little faster than SUPERLU, both of them give essentially the same results as the full algebra linear solver does.

6. Conclusions. This paper presents the FATODE library for the integration of stiff ODE systems, and for performing direct and discrete adjoint sensitivity analysis. FATODE generalizes the KPP numerical library, and implements fully implicit Runge-Kutta, singly diagonally implicit Runge-Kutta, and Rosenbrock methods. While KPP is specifically designed to solve chemical kinetic systems, FATODE offers a general purpose implementation that makes it potentially useful for a wide range of applications.



(a) Relative error vs number of steps

(b) Relative error vs number of LU decompositions

(c) Relative error vs number of right-hand side evaluations

(d) Relative error vs CPU time

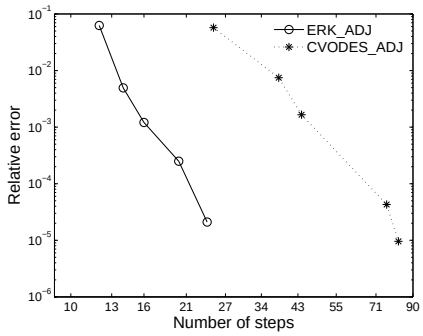
FIG. 5.5. *Computational statistics for the adjoint integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.011]$ *using stiff integrators. Comparison is made between implicit methods in* FATODE *with the BDF method in* CVODES. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $10^{-2}, \ldots, 10^{-6}$. *The adjoint model computes the sensitivity* $\partial y_1(t_F)/\partial y_i(t_0),\ i = 1 \ldots d$.

Its capabilities of direct and adjoint sensitivity analysis will enable users to tackle important problems such as uncertainty quantification, parameter estimation, and data assimilation within these application domains. In addition, FATODE contains stand-alone generic linear system solvers to deal with different types of systems. The decoupling between main integrators and linear solvers makes it feasible and convenient for users to provide their own implementations of linear solvers.

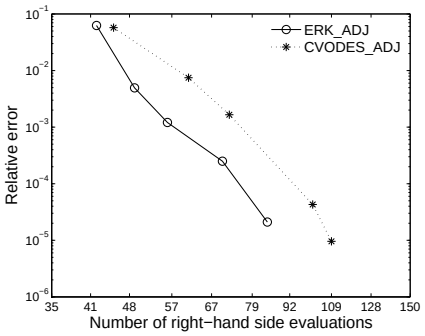
Code availability. The code is available from the web page `http://people.cs.vt.edu/~asandu/Software/FATODE/index.html`. The code has been tested under the following compilers: Portland group's pgf90, Lahey's lf95, Sun's sunf90, gfortran, g95, Absoft.

REFERENCES

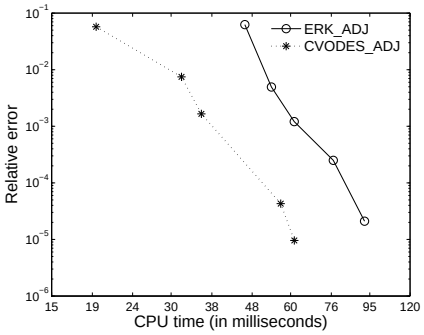
[1] A. SANDU AND D. DAESCU AND G.R. CARMICHAEL, *Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: II – Numerical validation and applications*, Atmospheric Environment, 37 (2003), pp. 5097–5114.



(a) Relative error vs number of steps



(b) relative error vs number of right-hand side evaluations



(c) Relative error vs CPU time

FIG. 5.6. *Computational statistics for the adjoint integration of the shallow water equations (5.2) over the time interval* $[t_0, t_F] \approx [0, 0.011]$ *using nonstiff integrators. Comparison is made between the explicit RK method in* FATODE *with the Adams-Moulton method in* CVODES. *Different points on the plots correspond to different absolute and relative tolerances levels in the range* $10^{-2}, \ldots, 10^{-6}$. *The adjoint model computes the sensitivity* $\partial y_1(t_F)/\partial y_i(t_0),\ i = 1 \ldots d$.

[2] M. ALEXE AND A. SANDU, *Forward and adjoint sensitivity analysis with continuous explicit Runge-Kutta schemes*, Applied Mathematics and Computation, 208 (2009), pp. 328–334.

[3] E. ANDERSON, Z. BAI, C. BISCHOF, L. S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, S. HAMMARLING, A. GREENBAUM, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' guide (third ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[4] H. M. BCKER, G. F. CORLISS, P. D. HOVLAND, U. NAUMANN, AND B. NORRIS, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering, Springer, New York, 2005.

[5] W. P. L. CARTER, *A detailed mechanism for the gas-phase atmospheric reactions of organic compounds*, Atmospheric Environment, 24 (1990), p. 481518.

[6] D. DAESCU, G. R. CARMICHAEL, AND A. SANDU, *Adjoint implementation of Rosenbrock methods applied to variational data assimilation problems*, Journal of Computational Physics, 165 (2000), pp. 496–510.

[7] TIMOTHY A. DAVIS, *Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, 30 (2004), pp. 196–199.

[8] J. W. DEMMEL, S. C. EISENSTAT, J. R. GILBERT, X. S. LI, AND J. W. H. LIU, *A supernodal approach to sparse partial pivoting*, SIAM Journal on Matrix Analysis and Applications, 20 (1999), pp. 720–755.

[9] A. M. DUNKER, *The decoupled direct method for calculating sensitivity coefficients in chemical kinetics*, 81 (1984), pp. 2385–2393.

[10] P. ELLER, K. SINGH, A. SANDU, K. BOWMAN, D. K. HENZE, AND M. LEE, *Implementation*

TABLE 5.1
Overall compute time (in seconds) using different linear solvers

	Full algebra	Sparse algebra	
Solver	LAPACK	UMFPACK	SUPERLU
ROS	286.4	18.7	26.8
ROS_TLM	1093.2	300.4	321.5
ROS_ADJ	1832.6	363.5	399.9
RK	857.7	25.6	44.9
RK_TLM	1390.0	92.4	126.9
RK_ADJ	2467.9	149.9	206.6
SDIRK	175.3	9.9	15.4
SDIRK_TLM	499.9	122.0	135.8
SDIRK_ADJ	2631.5	174.2	238.0

and evaluation of an array of chemical solvers in the Global Chemical Transport Model GEOS-Chem, Geoscientific Model Development, 2 (2009), p. 8996.

[11] M. W. GERY, G. Z. WHITTEN, J. P. KILLUS, AND M. C. DODGE, *A photochemical kinetics mechanism for urban and regional scale computer modeling*, Journal of Geophysical Research, 94 (1989), p. 1292512956.

[12] R. GIERING, *Tangent linear and adjoint model compiler, users manual 1.4*, (1999).

[13] R. GIERING AND T. KAMINSKI, *Recipes for adjoint code construction*, ACM Transactions on Mathematical Software, 24 (1998), pp. 437–474.

[14] K. GUSTAFSSON, *Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods*, ACM Transactions on Mathematical Software, 20 (1994), pp. 496–517.

[15] W. HAGER, *Runge Kutta methods in optimal control and the transformed adjoint system*, Numerische Mathematik, 87 (2000), pp. 247–282.

[16] E. HAIRER, S.P. NORSETT, AND G. WANNER, *Solving Ordinary Differential Equations I. Non-stiff Problems*, Springer-Verlag, Berlin, 1993.

[17] E. HAIRER AND G. WANNER, *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, Springer Series in Computational Mathematics, Berlin, 1991.

[18] A. C. HINDMARSH, P. N. BROWN, K. E. GRANT, S. L. LEE, R. SERBAN, D. E. SHUMAKER, AND C. S. WOODWARD, *SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers*, ACM Transactions on Mathematical Software, 31 (2005), pp. 363–396.

[19] J.R. LEIS AND M.A. KRAMER, *ODESSA - an ordinary differential equation solver with explicit simultaneous sensitivity analysis*, ACM Transactions on Mathematical Software, 14 (1986), pp. 61–75.

[20] K. RADHAKRISHNAN AND A. C. HINDMARSH, *Description and use of lsode, the livermore solver for ordinary differential equations*, Tech. Report UCRL-ID-113855, Lawrence Livermore National Laboratory, 1993.

[21] A. SANDU, *On the properties of Runge Kutta discrete adjoints*, in ICCS 2006, IV,LNCS 3994, Berlin Heidelberg, 2006, Springer-Verlag, pp. 550–557.

[22] ———, *Solution of inverse ODE problems using discrete adjoints*, Large Scale Inverse Problems and Quantification of Uncertainty, (2010), pp. 345–364.

[23] A. SANDU, D. DAESCU, AND G.R. CARMICHAEL, *Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: I – Theory and Software Tools*, Atmospheric Environment, 37 (2003), pp. 5083–5096.

[24] A. SANDU AND P. MIEHE, *Forward, Tangent Linear,and Adjoint Runge Kutta Methods in KPP–2.2 for Efficient Chemical Kinetic Simulations*, International Journal of Computer Mathematics, (2008).

[25] A. SANDU, J.G. VERWER, J.G. BLOM, E.J. SPEE, G.R. CARMICHAEL, AND F.A. POTRA, *Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock methods*, Atmospheric Environment, 31 (1997), pp. 3459–3472.

[26] R. SERBAN AND A.C. HINDMARSH, *CVODES, the sensitivity-enabled ODE solver in SUNDIALS*, Tech. Report UCRL-PROC-210300, Lawrence Livermore National Laboratory, 2003.

[27] Z. SIRKES AND E. TZIPERMAN, *Finite difference of adjoint or adjoint of finite difference*, Monthly Weather Review, 125 (1997), pp. 3373–3378.

[28] J. VERWER, E.J. SPEE, J. G. BLOM, AND W. HUNSDORFER, *A second order Rosenbrock method*

applied to photochemical dispersion problems, SIAM Journal on Scientific Computing, 20 (1999), pp. 1456–1480.

Appendix A. Derivatives.

Consider the following derivatives of $f(y)$

$$\boldsymbol{f_y} = \left(\frac{\partial f_i}{\partial y_j}\right)_{1\le i,j\le d}, \quad \boldsymbol{f_p} = \left(\frac{\partial f_i}{\partial p_j}\right)_{1\le i\le d, 1\le j\le m}, \quad \boldsymbol{f_{y,y}} = \left(\frac{\partial^2 f_i}{\partial y_j\,\partial y_k}\right)_{1\le i,j,k\le d}.$$

For $u, v \in \mathbb{R}^d$ we have that

$$(u\cdot \boldsymbol{f_{y,y}})\cdot v = \left(\sum_{j,k=1}^{n} u_j \frac{\partial^2 f_j}{\partial y_i\,\partial y_k} v_k\right)_{1\le i\le d},$$

$$(\boldsymbol{f_{y,y}}\cdot u)\cdot v = (\boldsymbol{f_{yy}}\cdot v)\cdot u = \boldsymbol{f_{y,y}}(u,v) = \left(\sum_{j,k=1}^{n} \frac{\partial^2 f_i}{\partial y_j\,\partial y_k} u_j\, v_k\right)_{1\le i\le d}.$$

The derivatives of Jacobian-vector products are

$$\left(\frac{d}{dy}\left(\boldsymbol{f_y}\cdot u\right)\right)\cdot v = (\boldsymbol{f_{y,y}}\cdot u)\cdot v\,, \quad \left(\frac{d}{dy}\left(\boldsymbol{f_y}^T\cdot u\right)\right)\cdot v = (u\cdot \boldsymbol{f_{y,y}})\cdot v\,.$$

For the Hessian (extended) transpose times vector term we have that

$$\begin{aligned}
\left(\begin{bmatrix} u^y \\ u^p \\ u^z \end{bmatrix}\cdot \tilde{H}^T\right)\cdot \begin{bmatrix} v^y \\ v^p \\ v^z \end{bmatrix} &= \frac{d}{d(y,p,z)}\left(\begin{bmatrix} \boldsymbol{f}_y^T & \mathbf{0} & r_y^T \\ \boldsymbol{f}_p^T & \mathbf{0} & r_p^T \\ 0 & 0 & 0 \end{bmatrix}\cdot \begin{bmatrix} u^y \\ u^p \\ u^z \end{bmatrix}\right)\cdot \begin{bmatrix} v^y \\ v^p \\ v^z \end{bmatrix} \\
&= \frac{d}{d(y,p,z)}\left(\begin{bmatrix} f_y^T\cdot u^y + r_y^T\cdot u^z \\ f_p^T\cdot u^y + r_p^T\cdot u^z \\ \mathbf{0} \end{bmatrix}\right)\cdot \begin{bmatrix} v^y \\ v^p \\ v^z \end{bmatrix} \\
&= \begin{bmatrix} (u^y\cdot f_{y,y})\cdot v^y + (u^y\cdot f_{y,p})\cdot v^p + (u^z\cdot r_{y,y})\cdot v^y + (u^z\cdot r_{y,p})\cdot v^p \\ (u^y\cdot f_{p,y})\cdot v^y + (u^y\cdot f_{p,p})\cdot v^p + (u^z\cdot r_{p,y})\cdot v^y + (u^z\cdot r_{p,p})\cdot v^p \\ \mathbf{0} \end{bmatrix}
\end{aligned}$$

Appendix B. Automatic differentiation.

In the following we illustrate the use of the well-known automatic differentiation tool TAMC for code generation. Detailed information on TAMC is given in the users manual [12].

Consider a given subroutine with the following parameter list:

```
subroutine fun(n, t, y, f)
! dimension of state vector y
integer :: n
double precision :: t
! y is the numerical solution at time t and p is the
! right hand side function at time t
double precision :: y(n), f(n)
```

The forward mode of TAMC can generate derivative code to compute the sensitivity of the dependent variable f with respect to the independent variable y by:

```
./stamc -reply <your_email> -toplevel fun -input y
   -output f -forward -pure -Jacobian <m> <source file name>
```
The generated code has the following parameter list:
```
      subroutine g_fun(n, t, y, g_y, g_f)
      ! dimension of state vector y
      integer :: n
      double precision :: t
      ! y is the numerical solution at time t and p is the
      ! right hand side function at time t
      double precision :: y(n), p(n)
      double precision :: g_y(n,m), g_f(n,m)
```
If you specify the number after option '-Jacobian' with the number of dependent variables and initialize the input variable g_f with identity matrix, full Jacobian will be obtained and stored in g_y. Otherwise, the number after option '-Jacobian' defines the number of columns of a seed matrix S, denoted by m.

$$\frac{\partial f}{\partial y} \cdot S_{[n \times m]} \tag{B.1}$$

The backward mode of TAMC can generate derivative code to compute the sensitivity of the dependent variable with respect to the many independent variables by:
```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -reverse -pure -Jacobian <m> <source file name>
```
The generated code has the following parameter list:
```
      subroutine adfun(n, t, y, ady, adf)
      ! dimension of state vector y
      integer :: n
      double precision :: t
      ! y is the numerical solution at time t and p is the
      ! right hand side function at time t
      double precision :: y(n), p(n)
      double precision :: ady(m,n), adf(m,n)
```
If you specify the number after option '-Jacobian' with the number of dependent variables and initialize the input variable *adf* with identity matrix, full Jacobian will be obtained and stored in *ady*. Otherwise, the number after option '-Jacobian' defines the number of rows of a seed matrix S, denoted by m.

$$S_{[m \times n]} \cdot \frac{\partial f}{\partial y} \tag{B.2}$$

Note that if you use the option '-Jacobian 1', product of Jacobian times a vector $\frac{\partial f}{\partial y} \cdot v$ is provided by the forward mode while product of Jacobian transpose times a vector $\left(\frac{\partial f}{\partial y}\right)^T \cdot v$ is provided by the backward mode.

To obtain the code calculating the Hessian times vector term $(u \cdot H) \cdot v$, we run TAMC in forward over reverse mode
```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -reverse -pure -Jacobian 1 <source file name>
./stamc -reply <your_email> -toplevel fun -input y
      -output g_y -forward -pure -Jacobian 1 <source file name>
```

The generated code has the following parameter list:

```
subroutine g_adfun(n, t, y, adf, g_y, g_ady)
! dimension of state vector y
integer :: n
double precision :: t
! y is the numerical solution at time t and p is the
! right hand side function at time t
double precision :: y(n), p(n)
double precision :: g_y(1,n), adf(1,n),g_ady(1,n)
```

where the variable *adf* corresponds to the vector v and g_y correspond to the vector u.

Similarly the Hessian transpose times vector term $(u \cdot H^T) \cdot v$ can be obtained by two consecutive forward runs:

```
./stamc -reply <your_email> -toplevel fun -input y
      -output f -forward -pure -Jacobian 1 <source file name>
./stamc -reply <your_email> -toplevel fun -input y
      -output g_y -forward -pure -Jacobian 1 <source file name>
```

The generated code has the following parameter list:

```
subroutine g_g_fun(n, t, y, g_y, g_f, g_g_y)
! dimension of state vector y
integer :: n
double precision :: t
! y is the numerical solution at time t and p is the
! right hand side function at time t
double precision :: y(n), p(n)
double precision :: g_y(1,n), g_f(1,n),g_g_y(1,n)
```

where the variable *g_f* corresponds to the vector v and *g_y* correspond to the vector u.

Appendix C. Discrete sensitivities with respect to parameters. A numerical solution of (3.28) provides the discrete forward model equations:

$$\begin{aligned} &y_{n+1} = \Phi^n\left(y_n, p_n\right), \\ &p_{n+1} = p_n, \\ &q_{n+1} = q_n + \Omega^n\left(y_n, p_n\right), \quad n = 0, \ldots, N-1, \end{aligned} \tag{C.1}$$

Replacing $p_n = p$ for all n leads to (C.2)

$$y_{n+1} = \Phi^n\left(y_n, p\right); \quad q_{n+1} = q_n + \Omega^n\left(y_n, p\right), \quad n = 0, \ldots, N-1, \tag{C.2}$$

We define the following co-state variables at $t_N = t_F$

$$\begin{bmatrix} \lambda_N \\ \mu_N \\ \theta_N \end{bmatrix} = \left(\frac{d\Psi}{d[y_N, p_N, q_N]}\right)^T = \begin{bmatrix} g_y^T(y_N, p) \\ g_p^T(y_N, p) \\ 1 \end{bmatrix} \tag{C.3}$$

and calculate the adjoint variables in time according to the discrete adjoint model equations

$$\begin{bmatrix} \lambda_i \\ \mu_i \\ \theta_i \end{bmatrix} = \begin{bmatrix} \Phi_y^i(y_i, p) & \Phi_p^i(y_i, p) & 0 \\ 0 & \boldsymbol{I} & 0 \\ \Omega_y^i(y_i, p) & \Omega_p^i(y_i, p) & \boldsymbol{I} \end{bmatrix}^T \begin{bmatrix} \lambda_{i+1} \\ \mu_{i+1} \\ \theta_{i+1} \end{bmatrix}, \quad i = N-1, \ldots, 1.$$

or

$$\begin{bmatrix} \lambda_i \\ \mu_i \\ \theta_i \end{bmatrix} = \begin{bmatrix} \Phi_y^i(y_i,p) & 0 & \Omega_y^i(y_i,p) \\ \Phi_p^i(y_i,p) & \boldsymbol{I} & \Omega_p^i(y_i,p) \\ 0 & 0 & \Omega_z^i \end{bmatrix} \begin{bmatrix} \lambda_{i+1} \\ \mu_{i+1} \\ \theta_{i+1} \end{bmatrix}, \quad i = N-1,\dots,1.$$

or, equivalently,

$$\begin{aligned} \lambda_n &= \left(\Phi_y^n(y_n,p_n)\right)^T \cdot \lambda_{n+1} + \left(\Omega_y^n(y_n,p_n)\right)^T \cdot \theta_{n+1}\,, \\ \mu_n &= \mu_{n+1} + \left(\Phi_p^n(y_n,p)\right)^T \cdot \lambda_{n+1} + \left(\Omega_p^n(y_n,p)\right)^T \cdot \theta_{n+1}\,, \\ \theta_n &= \Omega_q^n(y_n,p,q_n)\,\theta_{n+1}\,. \end{aligned}$$

and finally

$$\begin{aligned} \lambda_n &= \left(\Phi_y^n(y_n,p)\right)^T \lambda_{n+1} + \left(\Omega_y^n(y_n,p_n)\right)^T \theta_{n+1}\,, \\ \mu_n &= \mu_{n+1} + \left(\Phi_p^n(y_n,p_n)\right)^T \lambda_{n+1} + \left(\Omega_p^n(y_n,p_n)\right)^T \theta_{n+1}\,, \\ \theta_n &= \theta_{n+1}\,. \end{aligned} \tag{C.4}$$

Replacing in (C.4) leads to equations (C.5)

$$\begin{aligned} &\lambda_N = g_y^T(y_N,p)\,, \quad \mu_N = g_p^T(y_N,p)\,, \\ &\lambda_n = \left(\Phi_y^n(y_n,p)\right)^T \cdot \lambda_{n+1} + \left(\Omega_y^n(y_n,p)\right)^T\,, \\ &\mu_n = \mu_{n+1} + \left(\Phi_p^n(y_n,p)\right)^T \cdot \lambda_{n+1} + \left(\Omega_p^n(y_n,p)\right)^T\,, \quad n = N-1,\dots,0\,. \end{aligned} \tag{C.5}$$

The following sensitivities are obtained

$$\left(\frac{\partial \Psi}{\partial y_0}\right)^T = \lambda_0\,, \quad \left(\frac{\partial \Psi}{\partial p}\right)^T = \mu_0\,. \tag{C.6}$$

For details on derivation see [21].

Appendix D. Discrete sensitivities with respect to parameters: Runge-Kutta methods.

The extended ODE (3.28) has the following extended Jacobian

$$\boldsymbol{J}(t,y,p) = \begin{bmatrix} \boldsymbol{f_y}(t,y,p) & \boldsymbol{f_p}(t,y,p) & \mathbf{0}_{(d,1)} \\ \mathbf{0}_{(m,d)} & \mathbf{0}_{(m,m)} & \mathbf{0}_{(m,1)} \\ r_y(t,y,p) & r_p(t,y,p) & 0_{(1,1)} \end{bmatrix}. \tag{D.1}$$

$$\boldsymbol{J}^T(t,y,p) = \begin{bmatrix} \boldsymbol{f}_y^T(t,y,p) & \mathbf{0}_{(d,m)} & r_y^T(t,y,p) \\ \boldsymbol{f}_p^T(t,y,p) & \mathbf{0}_{(m,m)} & r_p^T(t,y,p) \\ \mathbf{0}_{(1,d)} & \mathbf{0}_{(1,m)} & 0_{(1,1)} \end{bmatrix}.$$

Using the extended co-state vector and the extended Jacobian (D.1) into the discrete Runge-Kutta adjoint (3.22) leads to

$$\begin{aligned} \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} &= h \begin{bmatrix} \boldsymbol{f}_y^T(t,y,p) & \mathbf{0}_{(d,m)} & r_y^T(t,y,p) \\ \boldsymbol{f}_p^T(t,y,p) & \mathbf{0}_{(m,m)} & r_p^T(t,y,p) \\ \mathbf{0}_{(1,d)} & \mathbf{0}_{(1,m)} & 0_{(1,1)} \end{bmatrix} \\ &\quad \cdot \left(b_i \begin{bmatrix} \lambda_{n+1} \\ \mu_{n+1} \\ \theta_{n+1} \end{bmatrix} + \sum_{j=1}^{s} a_{j,i} \begin{bmatrix} u_j \\ v_j \\ w_j \end{bmatrix} \right), \quad i = s,\dots,1\,, \end{aligned} \tag{D.2a}$$

$$\begin{bmatrix} \lambda_n \\ \mu_n \\ \theta_n \end{bmatrix} = \begin{bmatrix} \lambda_{n+1} \\ \mu_{n+1} \\ \theta_{n+1} \end{bmatrix} + \sum_{j=1}^{s} \begin{bmatrix} u_j \\ v_j \\ w_j \end{bmatrix}. \tag{D.2b}$$

From the last equation of (D.2b) we see that $w_i = 0$, $\forall\, i$, and therefore $\theta_n = \theta_{n+1} = \cdots = \theta_N = 1$. The discrete adjoint Runge-Kutta method (D.2) can be rewritten as (3.31).

Appendix E. Discrete sensitivities with respect to parameters: Rosenbrock methods.

Extending the discrete Rosenbrock adjoint (3.26) in the same way as we did for Runge Kutta methods yields

$$
\begin{aligned}
&\begin{bmatrix} \frac{\boldsymbol{I}_{(d,d)}}{h\gamma_{i,i}} - \boldsymbol{f_y^T}(t_n,y_n,p) & 0 & -r_y^T(t_n,y_n,p) \\ -\boldsymbol{f_p^T}(t_n,y_n,p) & \frac{\boldsymbol{I}_{(d,d)}}{h\gamma_{i,i}} & -r_p^T(t_n,y_n,p) \\ 0 & 0 & \frac{\boldsymbol{I}_{(d,d)}}{h\gamma_{i,i}} \end{bmatrix} \cdot \begin{bmatrix} u_i \\ \overline{u}_i \\ \widehat{u}_i \end{bmatrix} = m_i \begin{bmatrix} \lambda_{n+1} \\ \mu_{n+1} \\ \theta_{n+1} \end{bmatrix} \\
&\qquad + \sum_{j=i+1}^{s} \left(a_{j,i} \begin{bmatrix} v_j \\ \overline{v}_j \\ \widehat{v}_j \end{bmatrix} + \frac{c_{j,i}}{h} \begin{bmatrix} u_j \\ \overline{u}_j \\ \widehat{u}_j \end{bmatrix} \right), \\
&\begin{bmatrix} v_i \\ \overline{v}_i \\ \widehat{v}_i \end{bmatrix} = h \begin{bmatrix} \boldsymbol{f_y^T}(T_i,Y_i,p) & 0 & r_y^T(T_i,Y_i,p) \\ \boldsymbol{f_p^T}(T_i,Y_i,p) & 0 & r_p^T(T_i,Y_i,p) \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_i \\ \overline{u}_i \\ \widehat{u}_i \end{bmatrix}, \quad i = s, s-1, \cdots, 1\,, \\
&\begin{bmatrix} \lambda_n \\ \mu_n \\ \theta_n \end{bmatrix} = \begin{bmatrix} \lambda_{n+1} \\ \mu_{n+1} \\ \theta_{n+1} \end{bmatrix} + \sum_{i=1}^{s} \left(\begin{bmatrix} u_i \\ \overline{u}_i \\ \widehat{u}_i \end{bmatrix} \cdot \tilde{\boldsymbol{f}}_{\boldsymbol{y},\boldsymbol{y}}^{\,T} \right) \cdot \begin{bmatrix} k_i \\ \overline{k}_i \\ \widehat{k}_i \end{bmatrix} \\
&\qquad + h \begin{bmatrix} \boldsymbol{f_{y,t}^T}(t_n,y_n,p) & 0 & r_{y,t}^T(t_n,y_n,p) \\ \boldsymbol{f_{p,t}^T}(t_n,y_n,p) & 0 & r_{p,t}^T(t_n,y_n,p) \\ 0 & 0 & 0 \end{bmatrix} \cdot \sum_{i=1}^{s} \gamma_i \begin{bmatrix} u_i \\ \overline{u}_i \\ \widehat{u}_i \end{bmatrix} + \sum_{i=1}^{s} \begin{bmatrix} v_i \\ \overline{v}_i \\ \widehat{v}_i \end{bmatrix}.
\end{aligned}
$$

Using the derivative notation in Appendix A, this equation can be written component by component as follows:

$$
\begin{aligned}
&\frac{1}{h\gamma_{i,i}}\,\widehat{u}_i = m_i\,\theta_{n+1} + \sum_{j=i+1}^{s} \left(a_{j,i}\,\widehat{v}_j + \frac{c_{j,i}}{h}\,\widehat{u}_j \right), \\
&\left(\frac{\boldsymbol{I}_{(d,d)}}{h\gamma_{i,i}} - \boldsymbol{f_y^T}(t_n,y_n,p) \right) u_i = r_y^T(t,y,p)\,\widehat{u}_i + m_i\,\lambda_{n+1} + \sum_{j=i+1}^{s} \left(a_{j,i}\,v_j + \frac{c_{j,i}}{h}\,u_j \right), \\
&\frac{1}{h\gamma_{i,i}}\,\overline{u}_i = \boldsymbol{f_p^T}(t_n,y_n,p)u_i + r_p^T(t_n,y_n,p)\widehat{u}_i + m_i\,\mu_{n+1} + \sum_{j=i+1}^{s} \left(a_{j,i}\,\overline{v}_j + \frac{c_{j,i}}{h}\,\overline{u}_j \right), \\
&v_i = h\,\boldsymbol{f_y^T}(T_i,Y_i,p)\,u_i + h\,r_y^T(T_i,Y_i,p)\,\widehat{u}_i\,, \\
&\overline{v}_i = h\,\boldsymbol{f_p^T}(T_i,Y_i,p)\,u_i + h\,r_p^T(T_i,Y_i,p)\,\widehat{u}_i\,, \\
&\widehat{v}_i = 0\,, \\
&\lambda_n = \lambda_{n+1} + \sum_{i=1}^{s} \left((u_i \cdot f_{y,y}) \cdot k_i + (u_i \cdot f_{y,p}) \cdot \overline{k}_i + (\widehat{u}_i \cdot r_{y,y}) \cdot k_i + (\widehat{u}_i \cdot r_{y,p}) \cdot \overline{k}_i \right), \\
&\qquad + h\,\boldsymbol{f_{y,t}^T}(t,y,p) \cdot \sum_{i=1}^{s} \gamma_i u_i + h\,r_{y,t}^T(t,y,p) \cdot \sum_{i=1}^{s} \gamma_i \overline{u}_i + \sum_{i=1}^{s} v_i\,, \\
&\mu_n = \mu_{n+1} + \sum_{i=1}^{s} \left((u_i \cdot f_{p,y}) \cdot k_i + (u_i \cdot f_{p,p}) \cdot \overline{k}_i + (\widehat{u}_i \cdot r_{p,y}) \cdot k_i + (\widehat{u}_i \cdot r_{p,p}) \cdot \overline{k}_i \right),
\end{aligned}
$$

$$+h\,\boldsymbol{f}_{\boldsymbol{p},\boldsymbol{t}}^{\boldsymbol{T}}(t,y,p)\cdot\sum_{i=1}^{s}\gamma_i u_i + h\,r_{p,t}^{T}(t,y,p)\cdot\sum_{i=1}^{s}\gamma_i\overline{u}_i + \sum_{i=1}^{s}\overline{v}_i\,,$$

$$\theta_n = \theta_{n+1} + \sum_{i=1}^{s}\widehat{v}_i\,.$$

This can be further simplified. Note that $\theta_n = 1$ for all n and that the numbers $\widehat{u}_i$ can be computed by the simple recurrence

$$\frac{1}{h\gamma_{i,i}}\,\widehat{u}_i = m_i + \sum_{j=i+1}^{s}\Big(\frac{c_{j,i}}{h}\,\widehat{u}_j\Big)\;;$$

$$\widehat{u}_s = h\,\gamma_{ss}\,m_s\;;\quad \widehat{u}_i = h\,\gamma_{i,i}\,m_i + \gamma_{i,i}\sum_{j=i+1}^{s}c_{j,i}\,\widehat{u}_j\,,\quad i = s-1,\cdots,1\,.$$

The vector combined by k_i, $\overline{k}_i$, and $\widehat{k}_i$ is obtained from the solution of the extended form of forward integration (3.10), where the second equation

$$\text{(E.1)}\qquad \begin{bmatrix} y_{n+1}\\ p_{n+1}\\ q_{n+1}\end{bmatrix} = \begin{bmatrix} y_n\\ p_n\\ q_n\end{bmatrix} + \sum_{i=1}^{s} m_i \begin{bmatrix} k_i\\ \overline{k}_i\\ \widehat{k}_i\end{bmatrix}$$

indicates $\overline{k}_i = 0$ for all time steps since $p = 0$ as assumption. And the quantity $\overline{u}$ is not needed to update the adjoint variables λ and μ so that the third equation can be omitted.

Thus we obtain:

$$\Big(\frac{\boldsymbol{I}_{(d,d)}}{h\gamma_{i,i}} - \boldsymbol{f}_{\boldsymbol{y}}^{\boldsymbol{T}}(t_n,y_n,p)\Big)\,u_i = \widehat{u}_i\,r_y^T(t_n,y_n,p) + m_i\,\lambda_{n+1} + \sum_{j=i+1}^{s}\Big(a_{j,i}\,v_j + \frac{c_{j,i}}{h}\,u_j\Big)$$

$$v_i = h\,\boldsymbol{f}_{\boldsymbol{y}}^{\boldsymbol{T}}(T_i,Y_i,p)\cdot u_i + h\,\widehat{u}_i\,r_y^T(T_i,Y_i,p)$$

$$\overline{v}_i = h\,\boldsymbol{f}_{\boldsymbol{p}}^{\boldsymbol{T}}(T_i,Y_i,p)\cdot u_i + h\,\widehat{u}_i\,r_p^T(T_i,Y_i,p)$$

$$\lambda_n = \lambda_{n+1} + \sum_{i=1}^{s}\Big((u_i\cdot\boldsymbol{f}_{\boldsymbol{y},\boldsymbol{y}})\cdot k_i + (\widehat{u}_i\cdot r_{y,y})\cdot k_i\Big)$$
$$+h\,\boldsymbol{f}_{\boldsymbol{y},\boldsymbol{t}}^{T}(t_n,y_n,p)\cdot\sum_{i=1}^{s}\gamma_i u_i + h\,r_{y,t}^{T}(t_n,y_n,p)\cdot\sum_{i=1}^{s}\gamma_i\overline{u}_i + \sum_{i=1}^{s}v_i$$

$$\mu_n = \mu_{n+1} + \sum_{i=1}^{s}\Big((u_i\cdot\boldsymbol{f}_{\boldsymbol{p},\boldsymbol{y}})\cdot k_i + (\widehat{u}_i\cdot r_{p,y})\cdot k_i\Big)$$
$$+h\,\boldsymbol{f}_{\boldsymbol{p},\boldsymbol{t}}^{\boldsymbol{T}}(t_n,y_n,p)\cdot\sum_{i=1}^{s}\gamma_i u_i + h\,r_{p,t}^{T}(t_n,y_n,p)\cdot\sum_{i=1}^{s}\gamma_i\overline{u}_i + \sum_{i=1}^{s}\overline{v}_i\,.$$