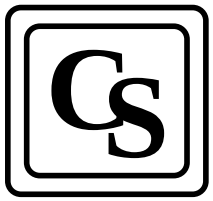

COMPUTER SCIENCE TECHNICAL REPORT

Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: I – Theory and Software Tools

A. Sandu, D. Dăescu,
and G.R. Carmichael

CSTR-02-01
April 2002

Michigan Technological University
1400 Townsend Drive, Houghton, MI 49931

Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: I – Theory and Software Tools

Adrian Sandu*, Dacian N. Daescu†, and Gregory R. Carmichael‡

* Department of Computer Science, Michigan Technological University, Houghton, MI 49931 (asandu@mtu.edu).

† Institute for Mathematics and its Applications, University of Minnesota, 400 Lind Hall 207 Church Street S.E. Minneapolis, MN 55455 (daescu@ima.umn.edu).

‡ Center for Global and Regional Environmental Research, The University of Iowa, Iowa City, IA 52242(gcarmich@icaen.uiowa.edu).

Abstract

The analysis of comprehensive chemical reactions mechanisms, parameter estimation techniques, and variational chemical data assimilation applications require the development of efficient sensitivity methods for chemical kinetics systems. The new release (KPP-1.2) of the Kinetic PreProcessor KPP contains software tools that facilitate direct and adjoint sensitivity analysis. The direct decoupled method, build using BDF formulas, has been the method of choice for direct sensitivity studies. In this work we extend the direct decoupled approach to Rosenbrock stiff integration methods. The need for Jacobian derivatives prevented Rosenbrock methods to be used extensively in direct sensitivity calculations; however, the new automatic differentiation and symbolic differentiation technologies make the computation of these derivatives feasible. The adjoint modeling is presented as an efficient tool to evaluate the sensitivity of a scalar response function with respect to the initial conditions and model parameters. In addition, sensitivity with respect to time dependent model parameters may be obtained through a single backward integration of the adjoint model. KPP software may be used to completely generate the continuous and discrete adjoint models taking full advantage of the sparsity of the chemical mechanism. Flexible direct-decoupled and adjoint sensitivity code implementations are achieved with minimal user intervention. In the companion paper [6] we present an extensive set of numerical experiments that validate the KPP software tools for several direct/adjoint sensitivity applications, and demonstrate the efficiency of KPP-generated sensitivity code implementations.

Keywords: Chemical kinetics, sensitivity analysis, direct decoupled method, adjoint model.

1 Introduction

The mathematical formulation of the chemical reactions mechanisms is given by a coupled system of stiff nonlinear differential equations

$$\frac{dy}{dt} = f(t, y; p) \ , \qquad y\left(t^0\right) = y^0 \ , \qquad t^0 \leq t \leq t^{\mathrm{F}} \ . \tag{1}$$

The solution $y(t) \in \Re^n$ represents the time evolution of the concentrations of the species considered in the chemical mechanism starting from the initial configuration y^0. Throughout this work vectors will be represented in column format and an upper script $(\cdot)^T$ will denote the transposition operator. The rate of change in the concentrations y is determined by the nonlinear production/loss function $f = (f_1, \ldots, f_n)^T$, which depends on a vector of parameters $p \in \Re^m$. In practice the vector p may represent reaction rate parameters, initial state of the model ($y^0 = p$), and/or additional source/sink terms (e.g. emissions rates). We assume that problem (1) has an unique solution $y = y(t, p)$ once the model parameters are specified. Comprehensive atmospheric reaction mechanisms take into consideration as many as 100 chemical species involved in hundreds of chemical reactions (see e.g. SAPRC-99, [4]), such that to efficiently integrate the system (1) fast and reliable numerical methods must be implemented [1, 20, 29, 30]. In addition, the model parameters are often obtained from experimental data and their accuracy is hard to estimate. The development and validation of chemical reactions mechanisms require a systematic sensitivity analysis to evaluate the effects of parameter variations on the model solution.

The sensitivities $S_\ell(t) \in \Re^n$ are defined as the derivatives of the solution with respect to the parameters

$$S_\ell(t) = \frac{\partial y(t)}{\partial p_\ell} \ , \qquad 1 \leq \ell \leq m \ . \tag{2}$$

A large sensitivity value $S_\ell(t)$ shows that the parameter p_ℓ plays an essential role in determining the model forecast $y(t)$, therefore one problem of interest is to evaluate the sensitivities $S(t)$ for $t^0 \leq t \leq t^{\mathrm{F}}$. Some practical applications (e.g. data assimilation) require the sensitivity of a scalar response function $g = g\left(y(t^{\mathrm{F}})\right)$ with respect to the model parameters

$$\frac{\partial g}{\partial p} = \left(S^T \frac{\partial g}{\partial y} \right) \Bigg|_{t=t^{\mathrm{F}}} \ . \tag{3}$$

Depending on the problem at hand, an appropriate method for sensitivity evaluation must be selected. The most popular efficient techniques for sensitivity studies are given by the direct-decoupled and adjoint sensitivity methods.

Given the multitude of applications, the continuous development of new reaction mechanisms and the frequent modifications of the existing ones, there is a need for software tools that facilitate the sensitivity analysis of general chemical kinetic mechanisms. The Kinetic PreProcessor KPP [7] has been successfully used in the forward integration of the chemical kinetics systems [29, 30, 33]. The new release (KPP-1.2) presented in this paper implements a comprehensive set of software tools for direct and adjoint sensitivity analysis. Given a chemical mechanism described by a list of chemical reactions, KPP generates a flexible code for the model, its forward integration and direct-decoupled/adjoint sensitivity analysis. The KPP generated code takes full advantage of the sparsity of the chemical mechanism and various numerical methods may be included with minimal user intervention.

The paper is organized as follows: a review of the direct decoupled sensitivity analysis and extensions to Runge-Kutta and Rosenbrock stiff integration methods are presented in Section 2. In Section 3 we review

the continuous and discrete adjoint sensitivity methods and address practical issues of the adjoint code implementation for chemical kinetics systems. Further insight on the computational complexity of various sensitivity methods as well as on the method selection is provided in Section 4. The Kinetic PreProcessor tools that facilitate the implementation of the sensitivity methods are presented in Section 5. Tutorial examples for building the direct-decoupled code and the adjoint code with KPP are presented in Sections 6 and 7, respectively. In Section 8 we outline the new numerical methods for sensitivity calculations available in the KPP numerical library. A summary of the results and concluding remarks are presented in Section 9.

2 Direct Sensitivity Analysis

For the direct analysis we consider the parameters p to be constant, i.e. they do not change in time. By differentiating (1) with respect to the parameters one obtains the sensitivity equations (variational equations)

$$\frac{dS_\ell}{dt} = J(t,y;p)\, S_\ell + f_{p_\ell}(t,y;p)\ , \quad S_\ell(t^0) = \frac{\partial y^0}{\partial p_\ell}\ , \qquad 1 \le \ell \le m \tag{4}$$

where J is the Jacobian matrix of the derivative function

$$J(t,y;p) = \frac{\partial\left[f_1(t,y;p),\cdots,f_n(t,y;p)\right]}{\partial\left[y_1,\cdots,y_n\right]}\ , \tag{5}$$

and f_{p_ℓ} are the derivative function partial derivatives with respect to the parameters

$$f_{p_\ell}(t,y;p) = \frac{\partial\left[f_1(t,y;p),\cdots,f_n(t,y;p)\right]}{\partial p_\ell}\ ,\ 1 \le \ell \le m\ . \tag{6}$$

The variational equations (4) are linear. The direct method solves simultaneously the model equation (1) together with the variational equations (4) to obtain both concentrations and sensitivities. The combined system (1)–(4) has the Jacobian

$$\frac{\partial\left[f, JS_1 + f_{p_1},\cdots,JS_m + f_{p_m}\right]}{\partial\left[y,S_1,\cdots,S_m\right]} = \begin{pmatrix} J & 0 & \cdots & 0 \\ \left(J\,S_1\right)_y + J_{p_1} & J & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ \left(J\,S_m\right)_y + J_{p_m} & 0 & \cdots & J \end{pmatrix}\ , \tag{7}$$

where the component matrices are

$$\left(J\,S_\ell\right)_y = \frac{\partial\left(J(t,y;p)\cdot S_\ell\right)}{\partial y}\ , \qquad J_{p_\ell} = \frac{\partial J(t,y;p)}{\partial p_\ell} = \frac{\partial f_{p_\ell}(t,y;p)}{\partial y}\ .$$

The eigenvalues of the combined Jacobian (7) are the eigenvalues of J (the Jacobian of the model equations), with different multiplicities; therefore if the model (1) is stiff the sensitivity equations (4) are also stiff. To maintain stability, an implicit time stepping method is needed. Implicit methods solve systems of the form

$$(I - h\gamma J)\, x = b\ ,$$

where $(I - h\gamma J)$ is called the prediction matrix, h is the stepsize and γ a parameter determined by the method. In the naive approach one would have to solve linear algebraic systems of dimensions $m(n+1) \times m(n+1)$ corresponding to (7). The direct decoupled method [9] exploits the special structure of the combined Jacobian (7); specifically, one only computes the LU factorization of the $n \times n$ model prediction matrix

$$I - h\gamma J = P^T \cdot L \cdot U\ .$$

Then the $m(n+1) \times m(n+1)$ prediction matrix for the Jacobian (7)

$$\begin{pmatrix} I - h\gamma J & 0 & \cdots & 0 \\ -h\gamma \left[(J\,S_1)_y + J_{p_1} \right] & I - h\gamma J & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ -h\gamma \left[(J\,S_m)_y + J_{p_m} \right] & 0 & \cdots & I - h\gamma J \end{pmatrix}$$

has the LU factorization

$$\begin{pmatrix} P^T \cdot L & 0 & \cdots & 0 \\ -h\gamma \left[(J\,S_1)_y + J_{p_1} \right] U^{-1} & P^T \cdot L & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ -h\gamma \left[(J\,S_m)_y + J_{p_m} \right] U^{-1} & 0 & \cdots & P^T \cdot L \end{pmatrix} \cdot \begin{pmatrix} U & 0 & \cdots & 0 \\ 0 & U & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & U \end{pmatrix} . \tag{8}$$

2.1 Remarks

In many practical applications sensitivities of $y(t)$ with respect to the initial values $y(t^0)$ are desired. Clearly, we can consider the initial values to be parameters $p = y(t^0)$; the sensitivity equations (4) hold with $f_{p_\ell} = 0$ and $S_\ell(t^0) = e_\ell$ (the ℓ-th vector of the canonical basis of $\mathcal{R}^n$).

Conversely, when the model parameters are constant in time, $p = p^0 = const$ the problem of evaluating the sensitivities with respect to the parameters can be reduced to computing sensitivities with respect to initial conditions. This is done by augmenting the state vector with the parameters $Y(t) = \left(y^T(t), p^T(t)\right)^T$, and the model system (1) with the parameter equations

$$\frac{dY}{dt} = \begin{pmatrix} dy/dt \\ dp/dt \end{pmatrix} = \begin{pmatrix} f(t,y;p) \\ 0 \end{pmatrix} , \quad Y(t^0) = \begin{pmatrix} y(t^0) \\ p(t^0) \end{pmatrix} = \begin{pmatrix} y^0 \\ p^0 \end{pmatrix} , \quad t^0 \le t \le t^{\mathrm{F}} . \tag{9}$$

The Jacobian matrix is then

$$J(Y) = \begin{pmatrix} J(t,y;p) & f_p(t,y;p) \\ 0_{(m,n)} & 0_{(m,m)} \end{pmatrix} . \tag{10}$$

The direct decoupled method was developed and is traditionally presented in the context of BDF time-stepping schemes [9, 22, 23]. In what follows we review this approach, then we extend the direct decoupled philosophy to Runge-Kutta and Rosenbrock integrators.

2.2 Direct-Decoupled Backward Differentiation Formulas

The model (4) is approximated by the k-step, order k linear multistep formula

$$y^{n+1} = \sum_{i=0}^{k-1} \alpha_i y^{n-i} + h\beta f(t^{n+1}, y^{n+1}; p) , \quad h = t^{n+1} - t^n. \tag{11}$$

The formula coefficients α_i, β are determined such that the method has order k of consistency. Relation (11) is a nonlinear system of equations which implicitly defines y^{n+1}, and which must be solved by a Newton-type iterative scheme. Typically the solution $y^n_{\{m+1\}}$ at iteration $m+1$ is computed as

$$[I - h\beta J(t^n, y^n; p)] \left(y^{n+1}_{\{m+1\}} - y^{n+1}_{\{m\}} \right) = \sum_{i=0}^{k-1} \alpha_i y^{n-i} + h\beta f(t^{n+1}, y^{n+1}_{\{m\}}; p) - y^{n+1}_{\{m\}} , \tag{12}$$

which requires one factorization of $I - h\beta J$ and one backsubstitution per iteration.

Discretization of the Continuous Sensitivity Equation. In this approach [9, 22, 23] one discretizes the continuous sensitivity equation (4) with the same BDF method used for discretizing the model (11)

$$S_\ell^{n+1} = \sum_{i=0}^{k-1} \alpha_i S_\ell^{n-i} + h\beta J(t^{n+1}, y^{n+1}; p)\, S_\ell^{n+1} + f_{p_\ell}(t^{n+1}, y^{n+1}; p) \ , \qquad 1 \le \ell \le m \tag{13}$$

Note that the system (13) is linear, therefore it admits a non-iterative solution

$$\left[I - h\beta J(t^{n+1}, y^{n+1}; p)\right] S_\ell^{n+1} = \sum_{i=0}^{k-1} \alpha_i S_\ell^{n-i} + f_{p_\ell}(t^{n+1}, y^{n+1}; p) \ , \qquad 1 \le \ell \le m \ . \tag{14}$$

The Direct Decoupled Method solves (12) first for the new-time solution y^{n+1}. The matrix $I - h\beta J(t^{n+1}, y^{n+1}; p)$ is computed and factorized; and the systems (14) are solved for $\ell = 1$ through m. Note that all systems (14) use the same matrix factorization, and this factorization is also reused in (12) for computing y^{n+2} during the next time step. Therefore the solution together with m sensitivities are obtained at the cost of a single matrix factorization per time step.

Discrete Sensitivity Equation. In this approach one considers directly the sensitivities of the numerical solution. A discrete equation involving these entities is obtained by taking the derivative of equation (11) with respect to p_ℓ ; it is easy to see that the process leads again to equation (13). Therefore, the numerical solutions of the the variational equations S_ℓ^n are also the sensitivities of the numerical solution dy^n/dp_ℓ.

2.3 Direct-Decoupled Runge-Kutta Methods

A general s-stage Runge-Kutta method is defined as [18, Section II.1]

$$\begin{aligned} y^{n+1} &= y^n + h \sum_{i=1}^{s} b_i k_i^0 \ , \qquad h = t^{n+1} - t^n \ , \\ k_i^0 &= f\left(t^n + c_i h,\, y^n + h \sum_{j=1}^{s} a_{ij} k_j^0 ; p \right) \ , \end{aligned} \tag{15}$$

where the coefficients a_{ij}, b_i and c_i are prescribed for the desired accuracy and stability properties. The stage derivative values k_i^0 are defined implicitly, and require solving a (set of) nonlinear system(s). Newton-type methods solve coupled linear systems of dimension (at most) $n \times s$.

Discretization of the Continuous Sensitivity Equation. Application of this method to the sensitivity equation (4) gives

$$\begin{aligned} S_\ell^{n+1} &= S_\ell^n + h \sum_{i=1}^{s} b_i k_i^\ell \quad \text{for} \quad 1 \le \ell \le m \ , \\ k_i^\ell &= J\left(t^n + c_i h,\, y^n + h \sum_{j=1}^{s} a_{ij} k_j^0 ; p \right) \left(S_\ell^n + h \sum_{j=1}^{s} a_{ij} k_j^\ell \right) \\ &\quad + f_{p_\ell}\left(t^n + c_i h,\, y^n + h \sum_{j=1}^{s} a_{ij} k_j^0 ;\, p \right) \ . \end{aligned} \tag{16}$$

The system (16) is linear and does not require an iterative procedure.

Discrete Sensitivity Equation. Clearly (16) can be obtained by differentiating (15) with respect to p_ℓ and setting $k_i^\ell = dk_i^0/dp_\ell$, therefore the Runge-Kutta numerical solutions of the sensitivity equations are equal to the sensitivities of the Runge-Kutta numerical solution of the model equation.

Computational Considerations. One first solves (15) for concentrations, which gives all stage derivative vectors $k_1^0, \cdots, k_s^0$. Next one solves (16) for sensitivities; at each stage k_i^ℓ is the solution of a linear system with the matrix

$$P_i = I - h\, a_{ii}\, J\left(t^n + c_i h,\, y^n + h\sum_{j=1}^{s} a_{ij} k_j^0; p\right) \ .$$

Clearly, each stage i requires the factorization of a different matrix P_i (even if all a_{ii} are the same); this factorization is then shared by all sensitivities (all ℓ's). Due to the repeated LU factorizations the standard Runge-Kutta methods do not seem suitable for direct sensitivity calculations.

2.4 Direct-Decoupled Rosenbrock Methods

An s-stage Rosenbrock method [19, Section IV.7] computes the next-step solution by the formulas

$$\begin{aligned} y^{n+1} &= y^n + \sum_{i=1}^{s} b_i k_i^0 \qquad h = t^{n+1} - t^n \ , \\ \left[I - h\gamma J(t^n, y^n; p)\right]\, k_i^0 &= hf\left(t^n + \alpha_i h,\, y^n + \sum_{j=1}^{i-1} \alpha_{ij} k_j^0; p\right) + hJ\left(t^n, y^n; p\right) \sum_{j=1}^{i-1} \gamma_{ij} k_j^0 \\ &\quad + h^2 \gamma_i f_t\left(t^n, y^n; p\right) \ , \end{aligned} \tag{17}$$

where s is the number of stages, $\alpha_i = \sum_j \alpha_{ij}$ and $\gamma_i = \sum_j \gamma_{ij}$. The formula coefficients (b_i, α_{ij} and γ_{ij}) give the order of consistency and the stability properties. For implementation purposes it is advantageous to use the alternative formulation [19, Section IV.7]

$$y^{n+1} = y^n + \sum_{i=1}^{s} m_i k_i^0 \ ,$$

$$\left[\frac{1}{h\gamma} I - J(t^n, y^n; p)\right]\, k_i^0 = f\left(t^n + \alpha_i h,\, y^n + \sum_{j=1}^{i-1} a_{ij} k_j^0; p\right) \tag{18}$$

$$+ \sum_{j=1}^{i-1} \frac{c_{ij}}{h} k_j^0 + h\gamma_i f_t\left(t^n, y^n; p\right) \ . \tag{19}$$

Discretization of the Continuous Sensitivity Equation. To apply the method (18) to the combined sensitivity equations (1)–(4) we need to make explicit use of the combined Jacobian (7). One step of the method reads

$$y^{n+1} = y^n + \sum_{i=1}^{s} m_i k_i^0, \qquad S_\ell^{n+1} = S_\ell^n + \sum_{i=1}^{s} m_i k_i^\ell \quad \text{for} \quad 1 \le \ell \le m \ , \tag{20}$$

$$\left[\frac{1}{h\gamma} I - J(t^n, y^n; p)\right]\, k_i^0 = f\left(t^n + \alpha_i h,\, y^n + \sum_{j=1}^{i-1} a_{ij} k_j^0; p\right) + \sum_{j=1}^{i-1} \frac{c_{ij}}{h} k_j^0 + h\gamma_i f_t\left(t^n, y^n; p\right) \ ,$$

$$\left[\frac{1}{h\gamma}I - J(t^n, y^n; p)\right]\ k_i^\ell = J\left(t^n + \alpha_i h,\, y^n + \sum_{j=1}^{i-1} a_{ij}k_j^0; p\right)\ \left(S_\ell^n + \sum_{j=1}^{i-1} a_{ij}k_j^\ell\right)$$
$$+ f_{p_\ell}\left(t^n + \alpha_i h,\, y^n + \sum_{j=1}^{i-1} a_{ij}k_j^0; p\right) + \sum_{j=1}^{i-1}\frac{c_{ij}}{h}k_j^\ell$$
$$+ \left(\frac{\partial J}{\partial p_\ell}(t^n, y^n; p)\right)\ k_i^0 + \left(\frac{\partial J}{\partial y}(t^n, y^n; p) \times S_\ell^n\right)\ k_i^0$$
$$+ h\gamma_i J_t\,(t^n, y^n; p)\, S_\ell^n + h\gamma_i f_{p_\ell, t}\,(t^n, y^n; p\,)$$

The method requires a single $n \times n$ LU decomposition per step to obtain both the concentrations and the sensitivities.

Discrete Sensitivity Equation. We note that the derivative of the method (18) with respect to p_ℓ leads also to the equation (20). Consequently, the sensitivities of the numerical solution coincide with the numerical solutions of the sensitivity equations.

Computational Considerations. Formula (20) requires the evaluation of the Hessian, i.e. the derivatives of the Jacobian with respect to y, as well as the Jacobian derivatives with respect to the parameters; these entities are 3-tensors. In addition, an extra Jacobian evaluation and one Jacobian-vector multiplication is needed at each stage. The need for Jacobian derivatives prevented Rosenbrock methods to be used extensively in direct sensitivity calculations. However, the new automatic differentiation and symbolic differentiation technologies make the computation of these derivatives feasible.

To avoid computing the derivatives of the Jacobian we can use a W-method [19, Section IV.7], and approximate the Jacobian (7) by $\mathrm{diag}(J, \cdots, J)$. This method gives consistent approximations of the sensitivities of the continuous solution, but these are now different than the sensitivities of the numerical solution.

3 Adjoint sensitivity analysis

The adjoint method provides an efficient alternative to the direct decoupled method for evaluating the sensitivity of a scalar response function with respect to the initial conditions and model parameters. Mathematical foundations of the adjoint sensitivity for nonlinear dynamical systems and various classes of response functionals are presented by Cacuci [2, 3]. The construction of the adjoint operators associated with linear and nonlinear dynamics and applications to atmospheric modeling are described in detail by Marchuk et al. [24, 25]. Menut et al. [21] and Vautard et al. [26] use the adjoint modeling for sensitivity studies in atmospheric chemistry. A review of the adjoint method applied to four-dimensional variational atmospheric chemistry data assimilation is presented in the recent work of Wang et al. [34]. In this section we review the continuous and discrete adjoint sensitivity methods and focus on the computational and implementation issues for chemical kinetics systems.

We will refer to the dynamical model (1) as a forward (direct) model and we assume that the solution $y = y(t, u)$ of the forward model is uniquely determined once the complete vector of model parameters $u = \left((y^0)^T, p^T\right)^T \in \mathcal{R}^{n+m}$ is specified. For a given scalar response function

$$g = g\big(y(t^{\mathrm{F}}, u)\big) \tag{21}$$

we are interested to evaluate the sensitivities

$$\nabla_u g = \left(\frac{\partial g}{\partial y_1^0}, \ldots, \frac{\partial g}{\partial y_n^0}, \frac{\partial g}{\partial p_1}, \ldots, \frac{\partial g}{\partial p_m} \right)^T \in \mathcal{R}^{n+m} \ . \tag{22}$$

In the adjoint sensitivity analysis one must distinguish between the continuous and the discrete adjoint modeling, see Sirkes [31]. While in general the derivation of the continuous adjoint model is presented, often in practice it is the discrete adjoint model that is implemented. This distinction is of particular importance in the context of stiff chemical reactions systems that require sophisticated numerical integrators.

3.1 Continuous adjoint sensitivity

The continuous adjoint model is obtained from the forward model using the linearization technique. To first order approximation, a perturbation in the input parameters $\delta u = \left((\delta y^0)^T, (\delta p)^T \right)^T$ leads to a perturbation in the response functional

$$\delta g = g(u + \delta u) - g(u) = \langle \nabla_u g(u), \delta u \rangle_{n+m} = \langle \nabla_y g, \delta y \rangle_n |_{t=t^F} \tag{23}$$

where $\langle \cdot, \cdot \rangle_n$ denotes the scalar product in $\mathcal{R}^n$. The time evolution of the perturbation $\delta y(t)$ is obtained by solving the tangent linear model problem

$$\frac{d\delta y}{dt} = J(t, y; p)\delta y + f_p(t, y; p)\delta p \ , \quad t^0 \le t \le t^{\mathrm{F}} \ , \tag{24}$$

$$\delta y(t^0) = \delta y^0 \ . \tag{25}$$

We introduce the adjoint variable $\lambda(t) \in \mathcal{R}^n$ (to be precisely defined later), take the scalar product of (24) with λ and integrate on $[t^0, t^{\mathrm{F}}]$ to obtain:

$$\int_{t^0}^{t^{\mathrm{F}}} \langle \lambda, \frac{d\delta y}{dt} \rangle_n dt = \int_{t^0}^{t^{\mathrm{F}}} \langle \lambda, J(t, y; p)\delta y + f_p(t, y; p)\delta p \rangle_n dt \ . \tag{26}$$

Integrating by parts the left side of (26), using matrix transposition on the right side, and rearranging the terms give the equivalent formulation

$$\langle \lambda, \delta y \rangle_n |_{t^0}^{t^{\mathrm{F}}} = \int_{t^0}^{t^{\mathrm{F}}} \langle \frac{d\lambda}{dt} + J^T(t, y; p)\lambda, \delta y \rangle_n + \langle f_p^T(t, y; p)\lambda, \delta p \rangle_m dt \ . \tag{27}$$

Therefore, if λ is defined as the solution of the adjoint problem

$$\frac{d\lambda}{dt} = -J^T(t, y; p)\lambda \tag{28}$$

$$\lambda(t^{\mathrm{F}}) = \nabla_y g(y(t^{\mathrm{F}})) \tag{29}$$

the perturbation in the response functional can be expressed from (23), (25) and (27) as

$$\delta g = \langle \lambda(t^0), \delta y^0 \rangle_n + \int_{t^0}^{t^{\mathrm{F}}} \langle f_p^T(t, y; p)\lambda, \delta p \rangle_m dt \ . \tag{30}$$

Consequently the sensitivity values are given by

$$\nabla_{y^0} g = \lambda(t^0) \ , \tag{31}$$

$$\nabla_p g = \int_{t^0}^{t^{\mathrm{F}}} f_p^T(t, y; p)\lambda dt \ . \tag{32}$$

The backward integration of the adjoint problem (28)-(29) provides all the intermediate values $\lambda(t), t^0 \leq t \leq t^{\rm F}$ that are needed to evaluate the integral in the right side of (32). An equivalent expression of $\nabla_p g$ is obtained by introducing a new adjoint variable $\nu(t) \in \mathcal{R}^m$ defined as the solution of the problem

$$\begin{aligned} \frac{d\nu}{dt} &= -f_p^T(t,y;p)\lambda & (33)\\ \nu(t^{\rm F}) &= 0 & (34) \end{aligned}$$

such that

$$\nu(t^0) = \int_{t^0}^{t^{\rm F}} f_p^T(t,y;p)\lambda dt \tag{35}$$

The sensitivities $\nabla_{y^0} g = \lambda(t^0)$ and $\nabla_p g = \nu(t^0)$ are then obtained by a backward integration of the coupled adjoint system (28)-(33) with the values at time $t^{\rm F}$ given by (29)-(34).

3.2 Some useful remarks

In this section we present some remarks that will prove to be useful later in our presentation when we discuss the discrete adjoint sensitivity and the implementation of the adjoint model.

Remark 1. The adjoint model (28)-(29) must be integrated backward in time from $t^{\rm F}$ to t^0 to evaluate $\lambda(t^0)$. Since the concept of backward integration may be less intuitive, we note that the adjoint problem may be formulated as a classical (forward time marching) initial value problem by introducing the change of variables

$$\tau = t^{\rm F} + t^0 - t; \;\; \overline{\lambda}(\tau) = \lambda(t) \tag{36}$$

Then, as t goes backward from $t^{\rm F}$ to t^0, τ goes forward from t^0 to $t^{\rm F}$ and

$$\overline{\lambda}(t^0) = \lambda(t^{\rm F}); \;\; \overline{\lambda}(t^{\rm F}) = \lambda(t^0) \tag{37}$$

After substituting in (28)-(29) and taking into account that $d\tau = -dt$, we obtain the equivalent formulation of the adjoint model

$$\begin{aligned} \frac{d\overline{\lambda}(\tau)}{d\tau} &= J^T\left(t^{\rm F} + t^0 - \tau, y(t^{\rm F} + t^0 - \tau); p\right)\, \overline{\lambda}(\tau)\ , \quad t^0 \leq \tau \leq t^{\rm F}\ , & (38)\\ \overline{\lambda}(t^0) &= \nabla_y g\big(y(t^{\rm F})\big) & (39) \end{aligned}$$

Remark 2. Modeling chemical kinetic systems requires the specification of time dependent model parameters. For example, photolytic reaction rates are determined by the solar radiation and thermal reactions rates depend on the temperature, therefore the reaction rates are implicitly a function of time. The adjoint modeling may be used to evaluate the sensitivity of the response functional with respect to time dependent model parameters. The values of $\lambda(t)$ represent the sensitivity of the response functional with respect to the model state at time t,

$$\nabla_{y(t)} g = \lambda(t)\ , \quad t^0 \leq t \leq t^{\rm F}\ . \tag{40}$$

Similarly, if the parameters are time-dependent $p = p(t)$, corresponding to a perturbation $\delta p(t)$ we obtain from (30) the time dependent sensitivity values

$$\nabla_{p(t)} g = f_p^T(t,y;p(t))\,\lambda(t)\ . \tag{41}$$

Remark 3. When the model parameters are constant in time, $p = p^0 = const$ the problem of evaluating the sensitivities $\nabla_p g$ may be reduced to the problem of the sensitivity with respect to the initial values of the augmented system (9). Considering the augmented Jacobian (10), the adjoint system

$$\frac{d}{dt}\begin{pmatrix} \lambda \\ \nu \end{pmatrix} = -\begin{pmatrix} J^T(t,y;p) & 0_{(n,m)} \\ f_p^T(t,y;p) & 0_{(m,m)} \end{pmatrix}\begin{pmatrix} \lambda \\ \nu \end{pmatrix} \tag{42}$$

is equivalent to (28)-(33).

Remark 4. In many applications the response functional depends on the state vector value over the integration time interval and it is expressed as

$$g = \int_{t^0}^{t^{\mathrm{F}}} \hat{g}\big(y(t,u)\big)\,dt \tag{43}$$

The problem of evaluating the sensitivities $\nabla_u g$ may be reduced to a standard sensitivity problem (21)-(22) by augmenting the state vector y with a new component y_{n+1} whose time evolution is governed by the equations

$$\frac{dy_{n+1}}{dt} = \hat{g}\big(y(t,u)\big) \tag{44}$$

$$y_{n+1}(t^0) = 0 \tag{45}$$

Therefore,

$$y_{n+1}(t^{\mathrm{F}},u) = \int_{t^0}^{t^{\mathrm{F}}} \hat{g}\big(y(t,u)\big)\,dt \tag{46}$$

and the adjoint sensitivity is applied to the augmented system (1)-(44), with the state vector $Y = (y^T, y_{n+1})^T$ and the response functional $g = y_{n+1}(t^{\mathrm{F}},u)$.

3.3 Discrete adjoint sensitivity

The discretization of the system (1) with a selected numerical method results in the discrete forward model. This is used to construct the discrete adjoint model as we now explain.

The discrete forward model obtains a numerical approximation $y^N \approx y(t^{\mathrm{F}})$ through a sequence of N intermediate states

$$y^{i+1} = F^i(y^i;p)\ , \quad i = 0,\ldots,N-1\ , \tag{47}$$

where F^i represents a one-step numerical integration formula which advances the solution from t^i to t^{i+1}. This establishes an explicit relationship between the evaluated response functional $g(y^N)$ and the model parameters.

To present a compact, yet explicit derivation of the discrete adjoint sensitivity formulae, we use Remark 3 and consider an augmented state vector $Y(t) = (y^T(t), p^T(t))^T$ where $p(t)$ is the solution of the problem (9). We rewrite the discrete equations (47) as

$$y^{i+1} = F^i(Y^i)\ , \quad i = 0,\ldots,N-1\ , \tag{48}$$

and attach the parameters equations

$$p^0 = p\ , \quad p^{i+1} = p^i\ , \quad i = 0,\ldots,N-1\ , \tag{49}$$

which must be satisfied by any consistent numerical method applied to (9). The sensitivity with respect to the initial conditions $Y^0 = \left((y^0)^T, (p^0)^T\right)^T$ is given by

$$\begin{pmatrix} \nabla_{y^0} g(y^N) \\ \nabla_p g(y^N) \end{pmatrix} = \nabla_{Y^0} g(y^N) = \left(\frac{\partial Y^N}{\partial Y^0}\right)^T \begin{pmatrix} \nabla_y g(y^N) \\ 0_m \end{pmatrix} . \tag{50}$$

A successive application of the chain rule followed by transposition gives

$$\left(\frac{\partial Y^N}{\partial Y^0}\right)^T = \left(\frac{\partial Y^1}{\partial Y^0}\right)^T \left(\frac{\partial Y^2}{\partial Y^1}\right)^T \cdots \left(\frac{\partial Y^{N-1}}{\partial Y^{N-2}}\right)^T \left(\frac{\partial Y^N}{\partial Y^{N-1}}\right)^T , \tag{51}$$

and by differentiating equations (48)-(49) we obtain

$$\left(\frac{\partial Y^{i+1}}{\partial Y^i}\right) = \begin{pmatrix} F_y^i(y^i, p^i) & F_p^i(y^i, p^i) \\ 0_{(m,n)} & I_{(m,m)} \end{pmatrix} , \quad i = 0, \ldots, N-1 . \tag{52}$$

Therefore, if we define the adjoint variables at $t^N = t^{\mathrm{F}}$

$$\lambda^N = \nabla_y g(y^N) , \tag{53}$$

$$\nu^N = 0_m , \tag{54}$$

and evaluate the adjoint variables at $t^i, i = N-1, \ldots, 1, 0$ using the recursive relations

$$\begin{pmatrix} \lambda^i \\ \nu^i \end{pmatrix} = \left(\frac{\partial Y^{i+1}}{\partial Y^i}\right)^T \begin{pmatrix} \lambda^{i+1} \\ \nu^{i+1} \end{pmatrix} = \begin{pmatrix} F_y^i(y^i, p^i)^T \lambda^{i+1} \\ F_p^i(y^i, p^i)^T \lambda^{i+1} + \nu^{i+1} \end{pmatrix} , \tag{55}$$

we obtain from (50)-(55) the sensitivities

$$\nabla_{y^0} g(y^N) = \lambda^0 , \tag{56}$$

$$\nabla_p g(y^N) = \nu^0 . \tag{57}$$

3.4 Practical issues of the adjoint code implementation

When chemical transformations are considered in an atmospheric model, the complexity of the implementation and the computational cost of the adjoint model are greatly increased. Two aspects must be emphasized: the non-linearity and the stiffness introduced in the model by the chemical reactions. The adjoint method relies on the linearization of the forward model and the non-linearity introduced by the chemistry may have a direct impact on the time interval length and the qualitative aspects of the adjoint sensitivity analysis. Since for non-linear problems the adjoint equations depend on the forward trajectory (obtained by direct integration of the model), in order to perform the adjoint computations the forward trajectory must be available in reverse order. Consequently a large amount of memory must be allocated for storing the state during the forward run.

As noticed in the previous sections in practice two strategies may be used to implement the adjoint code and we now discuss specific aspects of these strategies.

The first approach is to derive the continuous adjoint model associated with the continuous forward model dynamics, then to integrate the adjoint model with the numerical method of choice. In this approach the complexity of the numerical method used during the forward integration does not interfere with the adjoint computations and the user has the choice to select the backward integration method. Since the Jacobian matrices J and J^T have the same eigenvalues, the continuous adjoint formulation (38)-(39) presented in

Remark 1 shows that while during the forward integration one has to solve a *stiff nonlinear* system of ordinary differential equations (ODEs), during the adjoint integration a *stiff linear* system of ODEs must be solved. Therefore, highly stable implicit methods may be implemented at a reduced computational cost as no iterations are needed solving solving the adjoint.

Several additional issues must be considered in the continuous adjoint approach. In variational data assimilation applications requiring the minimization of a cost functional, the agreement between the computed gradient and the computed cost function is given by the accuracy of the numerical methods used for both forward and backward integration. This may impair the performance of a minimization algorithm since the provided gradient is not exact relative to the evaluated cost functional. From this point of view, the continuous adjoint model approach appears to be more suitable for adjoint sensitivity studies where no minimization process is involved. The computational errors of the forward run affect the adjoint computation. Therefore, solving the adjoint system with a more accurate numerical method than the one used in the forward integration may not be of benefit. When variable step size integration with error control is performed, the selected adjoint step size is in general distinct from the step size selected during the forward integration; consequently additional forward recomputations may be required.

The second method is the discrete adjoint approach, where the explicit dependence of the state vector trajectory on the input parameters is obtained by the numerical integration of the forward model. Generating the adjoint code from the discrete forward model has the advantage that the computed gradient is exact relative to the computed cost function. This approach appears to be suitable for the variational data assimilation where a minimization process must be performed since the minimization routine will receive the exact gradient of the evaluated cost function.

When the discrete adjoint modeling is used for sensitivity studies, stability and accuracy issues must be addressed [31]. Even when a variable step size forward integration with error control is performed, the accuracy of the computed sensitivity is hard to evaluate. Additional issues are related with the difficulty to generate the adjoint code if sophisticated numerical methods are used. For complex models and numerical methods hand generated codes are tedious to write and often subject to errors. Automatic differentiation tools may facilitate the adjoint code generation, but they must be used with caution and the correctness of the automatic generated adjoint code must be carefully verified. Taking full advantage of the sparsity of forward model during the adjoint computation may be hard to achieve. The computational cost of the adjoint code may be increased by the additional overhead related with the storage of the intermediate stages or variables inside the numerical method. The state of the art solvers available for differential equations may be written in a form which is not optimal for the adjoint code generation such that often one has to rewrite them in a form suitable for the adjoint compilers.

There is no general rule to decide which method should be used for the adjoint code implementation. The performance of the adjoint code is often problem dependent and a selection can be made only after an extensive analysis and testing for the particular problem to be solved have been performed. For atmospheric chemistry data assimilation and sensitivity studies a discrete adjoint model was used by Fisher and Lary [14] for a Bulirsch-Stoer integration scheme (Stoer and Bulirsch [32]) and by Elbern et al. [10] for a quasi-steady-state-approximation (QSSA) method. A continuous adjoint chemistry model with a fourth-order Rosenbrock solver (Hairer and Wanner [19]) for the forward/backward integration was successfully applied to 4D-Var chemical data assimilation by Errera and Fonteyn [13] in a hybrid adjoint approach for a transport-chemistry model (discrete adjoint for the transport integration, continuous adjoint for the chemistry integration).

4 Forward versus Reverse

For an in depth analysis of the algorithmic differentiation in forward and reverse mode, including trajectory storage strategies, we will refer to Griewank [17], Giering [16], and Rostaing [27]. To provide some insight on the complexity of the implementation of the forward and adjoint sensitivity methods, in this section we consider a numerical algorithm which takes as input a vector $y^0 = (y_1^0, \ldots, y_p^0)^T \in \mathcal{R}^p$ and returns the output $y = (y_1, \ldots, y_m)^T \in \mathcal{R}^m$ through a sequence of intermediate steps $y^0 \to y^1 \to \ldots \to y^k \to y$. If we assume that each of the vectors y^i, $1 \leq i \leq k$, has dimension n (extension to the general case n_i is straightforward) and denote by $f : \mathcal{R}^p \to \mathcal{R}^m$ the function $y^0 \to y$ and by f^i, $0 \leq i \leq k$, the functions defined by the relations $y^i \to y^{i+1}$ with the convention $y = y^{k+1}$, then $f^0 : \mathcal{R}^p \to \mathcal{R}^n$, $f^i : \mathcal{R}^n \to \mathcal{R}^n$ for $1 \leq i \leq k-1$, and $f^k : \mathcal{R}^n \to \mathcal{R}^m$. We can express f as the composition

$$y = f(y^0) = \left(f^k \circ f^{k-1} \circ \ldots \circ f^0\right)\ (y^0) \tag{58}$$

We are interested in the sensitivity of the output $y = f(y^0)$ with respect to the input y^0, which is given by the Jacobian matrix of dimension (m, p)

$$F = (F_{ij})_{1 \leq i \leq m,\ 1 \leq j \leq p} = \left(\frac{\partial y_i}{\partial y_j^0}\right)_{1 \leq i \leq m,\ 1 \leq j \leq p} . \tag{59}$$

We assume that all functions introduced above are differentiable and for each $0 \leq i \leq k$ and denote by F^i the Jacobian matrix associated with f^i. Then F^0 is a (n, p)-dimensional matrix, F^i, $1 \leq i \leq k-1$, are matrices of dimension (n, n), and F^k is a (m, n)-dimensional matrix. Differentiating relation (58) and applying the chain rule gives

$$F(y^0) = F^k(y^k) \cdot F^{k-1}(y^{k-1}) \cdot \ldots \cdot F^1(y^1) \cdot F^0(y^0) \tag{60}$$

where $\cdot$ stands for the matrix product operation.

Two techniques can be used to evaluate the product in the right hand side of (60). The first one is the forward mode where the product is evaluated in the same order as the intermediate states are computed, such that starting from the right we first multiply $F^1(y^1) \cdot F^0(y^0)$, then the result is multiplied by $F^2(y^2)$, and so on. The second technique is the reverse (backward) mode where the product is evaluated from the left, starting with the last intermediate states and computing $F^k(y^k) \cdot F^{k-1}(y^{k-1})$, then multiplying the result by $F^{k-2}(y^{k-2})$, and so on. To perform this computations we must assume the the values $y^k, y^{k-1}, \ldots$ have been previously stored and are readily available. Next we evaluate the number of floating point operations (for simplicity multiplications only) required by the forward and reverse mode computations.

Forward mode: first multiplication $F^1 \cdot F^0$ is between a (n, n) matrix and a (n, p) matrix resulting in a (n, p) matrix. This requires $n^2 \times p$ multiplications. The same is true for the next $k-2$ matrix multiplications. The last multiplication is between a (m, n) matrix and a (n, p) matrix which requires $m \times n \times p$ multiplications. The complexity of the forward mode computation is then

$$fwdmode = (k-1) \times n^2 \times p + m \times n \times p \tag{61}$$

Reverse mode: first multiplication $F^k \cdot F^{k-1}$ is between a (m, n) matrix and a (n, n) matrix resulting in a (m, n) matrix. This requires $n^2 \times m$ multiplications. The same is true for the next $k-2$ matrix multiplications. The last multiplication is between a (m, n) matrix and a (n, p) matrix which requires $m \times n \times p$ multiplications. The complexity of the reverse mode computation is then

$$revmode = (k-1) \times n^2 \times m + m \times n \times p \tag{62}$$

¿From (61) and (62) it follows

$$revmode < fwdmode \Leftrightarrow m < p \tag{63}$$

Therefore, the adjoint method is more efficient than the forward method in the case when the number of output parameters is smaller than the number of input parameters. In particular, if $p = n$ and $m = 1$ (e.g. the sensitivities of a scalar cost function are computed with respect to the initial state as the set of control parameters) the complexities of the forward and reverse modes are

$$fwdmode = (k-1) \times n^3 + n^2 \ , \quad revmode = k \times n^2 \ . \tag{64}$$

For a large n the benefits of the adjoint method are clear.

Remark. Significant insight on the role played by the length k of the forward trajectory on the cost of the adjoint implementation may be gained from this example. The first aspect to consider is the additional storage requirements which is of order $k \times n$ and implies a linear dependence on the trajectory length. The second aspect is related to the computational cost which depends linearly on k and quadratically on n such that, for example, the effect of increasing the trajectory length by a factor of 100 may be comparable with the effect of increasing the number of parameters by a factor of 10. For air pollution models, when explicit methods are used to integrate the stiff chemical reaction systems the step size taken may be as small as a few seconds. Even for a relatively small time integration window (say few hours) k may be of order 10^3 or greater such that the overhead introduced by the trajectory length is significant.

5 The Kinetic PreProcessor Tools

In this section we present the KPP software tools that are useful in derivative computations. A detailed discussion of the basic KPP capabilities can be found in our previous work [7, 8]. Here we focus on the new features introduced in the release 1.2 that allow an efficient sensitivity analysis of chemical kinetic systems.

5.1 Mass Action Kinetics

KPP builds simulation code for chemical systems with chemical reactions $r_1, \cdots, r_R$ involving chemical species $y_1, \cdots, y_n$. The chemical concentrations change in time according to the law of mass action kinetics, which states that each chemical reaction r_j

$$r_j) \quad \sum_{i=1}^{n} s_{i,j}^{-} y_i \xrightarrow{k_j} \sum_{i=1}^{n} s_{i,j}^{+} y_i \ , \quad 1 \le j \le R \ , \tag{65}$$

progresses at a reaction velocity ω_j proportional with the concentrations of the reactants

$$\omega_j(t,y) = k_j(t) p_j(y) \text{ (molecules per time unit)} \ , \quad p_j(y) = \prod_{i=1}^{n} y_i^{s_{i,j}^{-}} \ .$$

The proportionality constants k_j are called rate coefficients. The concentration of species y_i changes at a rate given by the cumulative effect of all chemical reactions,

$$\frac{d}{dt} y_i = \sum_{j=1}^{R} \left(s_{i,j}^{+} - s_{i,j}^{-} \right) \omega_j(t,y) \ , \quad i = 1, \cdots, n \ . \tag{66}$$

With the stoichiometric matrix

$$S = \left(s^+_{i,j} - s^-_{i,j}\right)_{1 \le i \le n,\ 1 \le j \le R} \ ,$$

the evolution equations (66) can be rewritten as

$$\begin{aligned} \frac{d}{dt} y &= S \cdot \mathrm{diag}\left[k_1(t) \cdots k_R(t)\right] \cdot p(y) && (67) \\ &= S \cdot \omega\left(t, y\right) && (68) \\ &= f\left(t, y\right) \ , && (69) \end{aligned}$$

where $p = \left[p_1 \cdots p_R\right]^T$ is the vector of reactant products and $\omega = \left[\omega_1 \cdots \omega_R\right]^T$ the vector of reaction velocities. KPP builds simulation code by symbolically forming the vector of reaction velocities ω and computing explicitly the product of the stoichiometric matrix with this vector.

5.2 An Example

We consider an example from stratospheric chemistry. This is a very simple Chapman-like mechanism, and we use it with the purpose of illustrating the KPP capabilities. However the software tools are general and can be applied to virtually any kinetic mechanism.

The species that interact are described below in KPP syntax. Some of the are "variable", meaning that their concentrations change according to the law of mass action kinetics; and some are "fixed", with the concentrations determined by physical and not chemical factors. For each species its atomic composition is given (unless the user chooses to ignore it). Comments are enclosed between curly brackets.

```
#INCLUDE atoms
#DEFVAR
  O   = O;          { Oxygen atomic ground state }
  O1D = O;          { Oxygen atomic excited state }
  O3  = O + O + O;  { Ozone }
  NO  = N + O;      { Nitric oxide }
  NO2 = N + O + O;  { Nitrogen dioxide }
#DEFFIX
  M   = ignore;     { Generic atmospheric molecule }
  O2  = O + O;      { Molecular oxygen }
```

The chemical kinetic mechanism described in the KPP language is shown below. Each reaction is followed by its rate coefficient. SUN is the normalized sunlight intensity, equal to one at noon and zero at midnight.

```
#EQUATIONS { Small Stratospheric Mechanism }
  {1.}  O2  + hv = 2O       : 2.643E-10 * SUN*SUN*SUN;
  {2.}  O   + O2 = O3       : 8.018E-17;
  {3.}  O3  + hv = O   + O2 : 6.120E-04 * SUN;
  {4.}  O   + O3 = 2O2      : 1.576E-15;
  {5.}  O3  + hv = O1D + O2 : 1.070E-03 * SUN*SUN;
  {6.}  O1D + M  = O   + M  : 7.110E-11;
  {7.}  O1D + O3 = 2O2      : 1.200E-10;
  {8.}  NO  + O3 = NO2 + O2 : 6.062E-15;
```

```
{9.}  NO2 + O   = NO   + O2 : 1.069E-11;
{10.} NO2 + hv  = NO   + O  : 1.289E-02 * SUN;
```

5.3 The derivative function

KPP orders the variable species such that the sparsity pattern of the Jacobian is maintained after an LU decomposition. For our example there are five variable species (NVAR=5) ordered as

```
I_O1D = 1,  I_O = 2,  I_O3 = 3,  I_NO = 4,  I_NO2 = 5,
```

and two fixed species (NFIX=2)

```
I_M = 1,  I_O2 = 2.
```

Note that KPP also considers radical species, but these are not present in this example (NRAD=0).

The resulting ODE system has dimension 5. The concentrations of fixed species are parameters in the derivative function. KPP computes the vector A of reaction rates and from this the vector A_VAR of component time derivatives. Below is the Fortran code for the derivative function as generated by KPP. The arguments V, R, and F stand for the concentrations of variable, radical and fixed species; and RCT is the vector of rate coefficients.

```
      SUBROUTINE FunVar ( V, R, F, RCT, A_VAR )
      INCLUDE 'small.h'
      REAL*8 V(NVAR), R(NRAD), F(NFIX), A_VAR(NVAR)
      REAL*8 RCT(NREACT), A(NREACT)
C Computation of equation rates
      A(1) = RCT(1)*F(2)
      A(2) = 8.018e-17*V(2)*F(2)
      A(3) = RCT(3)*V(3)
      A(4) = 1.576e-15*V(2)*V(3)
      A(5) = RCT(5)*V(3)
      A(6) = 7.11e-11*V(1)*F(1)
      A(7) = 1.2e-10*V(1)*V(3)
      A(8) = 6.062e-15*V(3)*V(4)
      A(9) = 1.069e-11*V(2)*V(5)
      A(10) = RCT(10)*V(5)
C Aggregate function
      A_VAR(1) = A(5)-A(6)-A(7)
      A_VAR(2) = 2*A(1)-A(2)+A(3)-A(4)+A(6)-A(9)+A(10)
      A_VAR(3) = A(2)-A(3)-A(4)-A(5)-A(7)-A(8)
      A_VAR(4) = -A(8)+A(9)+A(10)
      A_VAR(5) = A(8)-A(9)-A(10)
      END
```

5.4 The Jacobian

The Jacobian of the derivative function (5) is also automatically constructed by KPP; the KPP command uses several options

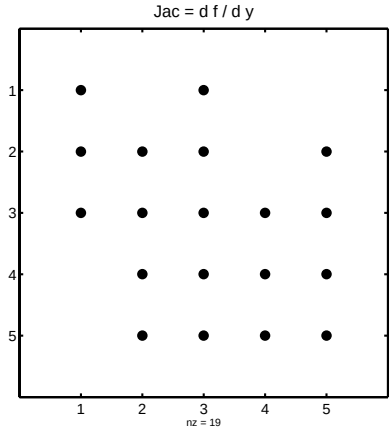

Figure 1: The sparsity pattern of the Jacobian for the small stratospheric system.

```
#JACOBIAN [ OFF | ON | SPARSE ]
```

The option OFF inhibits the generation of the Jacobian subroutine; the option ON generates the Jacobian as a full, square (`NVAR`×`NVAR`) matrix, while the option SPARSE generates the Jacobian in the sparse format. For our small stratospheric example the sparsity pattern of the Jacobian is shown in Figure 1. For this example the Jacobian is almost full (19 out of 25 entries are nonzero). However, accounting for sparsity is important for large chemical systems, where only a small fraction of the entries in the Jacobian are nonzero (typically 10% or less). By accounting for the sparsity one can obtain very efficient linear algebra solvers, etc.

The default sparse representation format is compressed on rows. KPP accounts for the fill-in due to the LU decomposition; the total number of nonzeros N_Z reflects that. JAC_{SP} stores the N_Z nonzero elements of the Jacobian in row order; each row i starts at position $CROW(i)$, and $CROW(N+1) = N_Z + 1$. The column position of element k $(1 \leq k \leq N_Z)$ is $ICOL(k)$, and the position of the diagonal element i is $DIAG(i)$. For the small stratospheric example the structure of Figure 1 leads to the following Jacobian sparse data structure

```
LU_ICOL_V = [ 1,3,1,2,3,5,1,2,3,4,5,2,3,4,5,2,3,4,5 ]
LU_CROW_V = [ 1,3,7,12,16,20 ]
LU_DIAG_V = [ 1,4,9,14,19,20 ]
```

To numerically solve for the chemical concentrations one must employ an implicit timestepping technique, as the system is usually stiff. Implicit integrators solve systems of the form

$$(I - h\,\gamma\,J)\; x = b$$

where h is the step size, γ a method-dependent parameter, J the Jacobian, and $I - h\gamma J$ the "prediction" matrix, whose sparsity structure is given by the sparsity structure of J. KPP generates the following sparse linear algebra subroutines.

```
SUBROUTINE KppDecomp(N,P,IER)
```

performs an in-place, non-pivoting, sparse LU decomposition of the prediction matrix P. Since the sparsity structure accounts for fill-in, all elements of the full LU decomposition are actually stored. The output argument IER returns a value that is nonzero if singularity is detected.

```
SUBROUTINE KppSolve(P,X)
```

uses the in-place LU factorization P as computed by KppDecomp; it performs sparse backward and forward substitutions; at input X contains the system right-hand-side vector, and at output it contains the solution. Similarly, the subroutine

```
SUBROUTINE KppSolveTR ( P, b, X )
```

solves the linear system $P^T X = b$ with the transposed coefficient matrix, and uses the same LU factorization as KppSolve. The sparse linear algebra subroutines KppDecomp and KppSolve are extremely efficient, as shown in [28].

Two other KPP-generated subroutines are useful for direct and adjoint sensitivity analysis.

```
SUBROUTINE JacVar_SP_Vec ( JVS, U, V )
```

computes the Jacobian times vector product ($V \longleftarrow JVS \cdot U$). The Jacobian JVS is in sparse format and a sparse multiplication with no indirect addressing is performed. Similarly, the subroutine

```
SUBROUTINE JacVarTR_SP_Vec ( JVS, U, V )
```

computes the sparse Jacobian transposed times vector product without indirect addressing ($V \longleftarrow JVS^T \cdot U$).

5.5 The Stoichiometric Formulation

KPP can generate the elements of the derivative function and the Jacobian in the stoichiometric formulation (67); this means that the product between the stoichiometric matrix, rate coefficients, and reactant products is not explicitly performed. The option that controls the code generation is

```
#STOICMAT [ OFF | ON ]
```

The `ON` value of the switch instructs KPP to generate code for the stoichiometric matrix, the vector of reactant products in each reaction, and the partial derivative of the time derivative function with respect to rate coefficients. These elements are discussed below.

The stoichiometric matrix is usually very sparse; for our example the matrix has 22 nonzero entries out of 50 entries. The total number of nonzero entries in the stoichiometric matrix is a constant

```
PARAMETER ( NSTOICM = 22 )
```

KPP produces the stoichiometric matrix in sparse, column-compressed format. Elements are stored in columnwise order in the one-dimensional vector of values `STOICM(1:NSTOICM)`; their row indices are stored in `IROW_STOICM(1:NSTOICM)`; the vector `CCOL_STOICM(1:NVAR+1)` contains pointers to the start of each column; for example column j starts in the sparse vector at position `CCOL_STOICM`(j) and ends at `CCOL_STOICM`($j+1$) $-$ 1. The last value `CCOL_STOICM(NVAR+1)=NSTOICM+1` is not necessary but simplifies the future handling of sparse data structures. For our example we have

```
STOICM = [      2.,-1.,1.,1.,1.,-1.,-1.,1.,-1.,-1.,1.,-1.,
               -1.,-1.,-1.,1.,-1.,1.,-1.,1.,1.,-1. ]
IROW_STOICM = [ 2,2,3,2,3,2,3,1,3,1,2,1,3,3,4,5,2,4,5,2,4,5 ]
CCOL_STOICM = [ 1,2,4,6,8,10,12,14,17,20,23 ]
```

The following subroutine computes the reactant products for each reaction; i.e. `A_RPROD` is the vector $p(y)$ in the formulation (67).

```
SUBROUTINE  ReactantProd ( V, R, F, A_RPROD )
```

In the stoichiometric formulation (67) the Jacobian is

$$J\,(t,y) = \frac{\partial f(t,y)}{\partial y} = S \cdot \mathrm{diag}\,[k_1(t) \cdots k_R(t)] \cdot \frac{\partial p(y)}{\partial y} = S \cdot \mathrm{diag}\,[k_1(t) \cdots k_R(t)] \cdot J^{\mathrm{RP}}(y)\ . \tag{70}$$

The following subroutine computes the Jacobian of reactant products vector; i.e. `JV_RPROD` is the matrix $J^{\mathrm{RP}} = \partial p(y)/\partial y$ above.

```
SUBROUTINE  JacVarReactantProd ( V, R, F, JV_RPROD )
```

The matrix `JV_RPROD` is of course sparse and is computed and stored in row compressed sparse format. The number of nonzeros is stored in the parameter `NJVRP`, the column indices in the vector `ICOL_JVRP` and the beginning of each row in `CROW_JVRP`; for our example

```
PARAMETER ( NJVRP = 13 )
CROW_JVRP = [1,1,2,3,5,6,7,9,11,13,14]
ICOL_JVRP = [2,3,2,3,3,1,1,3,3,4,2,5,4]
```

5.6 The Derivatives with respect to Reaction Coefficients

The stoichiometric formulation allows a direct computation of the derivatives with respect to rate coefficients. ¿From (67) one sees that the partial derivative of the time derivative function with respect to a reaction coefficient is given by the corresponding column in the stoichiometric matrix times the corresponding entry in the vector of reactant products

$$f\,(t,y) = S \cdot \mathrm{diag}\,[k_1(t) \cdots k_R(t)] \cdot p(y) \quad \Rightarrow \quad \frac{\partial f_{1:\mathtt{NVAR}}}{\partial k_j} = S_{1:\mathtt{NVAR},j} \cdot p_j(y)\ .$$

The following subroutine computes the partial derivative of the time derivative function with respect to a set of reaction coefficients; this is more efficient than computing each derivative separately.

```
SUBROUTINE  dFunVar_dRcoeff( V, R, F, NCOEFF, JCOEFF, DFDR )
```

A total of `NCOEFF` derivatives are taken with respect to reaction coefficients `JCOEFF(1)` through `JCOEFF(NCOEFF)`. `JCOEFF` therefore is a vector of integers containing the indices of reaction coefficients with respect to which we differentiate. The subroutine returns the `NVAR`×`NCOEFF` matrix `DFDR`; each column of this matrix contains the derivative of the function with respect to one rate coefficient. Specifically,

$$\mathtt{DFDR}_{1:\mathtt{NVAR},j} = \frac{\partial f_{1:\mathtt{NVAR}}}{\partial k_{\mathtt{JCOEFF(j)}}}\ , \quad 1 \le j \le \mathtt{NCOEFF}\ .$$

The partial derivative of the Jacobian with respect to the rate coefficient k_j can be obtained from (70) as the external product of column j of the stoichiometric matrix with row j of J^{RP}

$$\frac{\partial J_{1:\mathtt{NVAR},1:\mathtt{NVAR}}}{\partial k_j} = S_{1:\mathtt{NVAR},j} \cdot J^{\mathrm{RP}}_{j,1:\mathtt{NVAR}}\ .$$

In practice one needs the product of this Jacobian partial derivative with a user vector, i.e.

$$\frac{\partial J_{1:\mathtt{NVAR},1:\mathtt{NVAR}}}{\partial k_j} \cdot U_{1:\mathtt{NVAR}} = S_{1:\mathtt{NVAR},j} \cdot \left(J^{\mathrm{RP}}_{j,1:\mathtt{NVAR}} \cdot U_{1:\mathtt{NVAR}}\right)\ .$$

This is computed by the KPP-generated subroutine

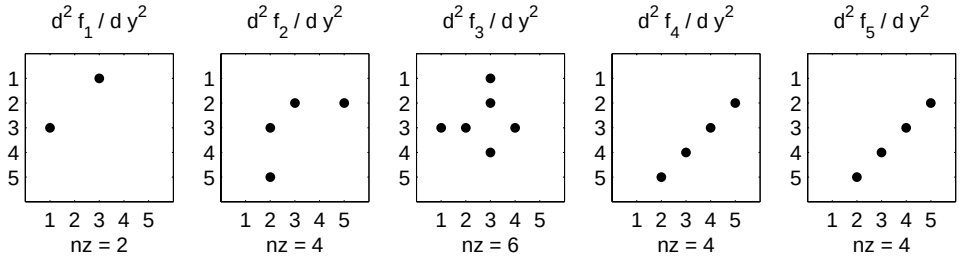

Figure 2: The Hessian of the small stratospheric system.

```
SUBROUTINE  dJacVar_dRcoeff( V, R, F, U, NCOEFF, JCOEFF, DJDR )
```

`U` is the user-supplied vector. A total of `NCOEFF` derivatives are taken with respect to reaction coefficients `JCOEFF(1)` through `JCOEFF(NCOEFF)`. The subroutine returns the `NVAR`×`NCOEFF` matrix `DJDR`; each column of this matrix contains the derivative of the Jacobian with respect to one rate coefficient times the user vector. Specifically,

$$\texttt{DJDR}_{\texttt{1:NVAR},j} = \frac{\partial J_{\texttt{1:NVAR},\texttt{1:NVAR}}}{\partial k_{\texttt{JCOEFF(j)}}} \cdot U_{\texttt{1:NVAR}} \ , \quad 1 \leq j \leq \texttt{NCOEFF} \ .$$

5.7 The Hessian

The Hessian contains second order derivatives of the time derivative functions. More exactly, the Hessian is a 3-tensor such that

$$H_{i,j,k}\left(t,y;p\right) = \frac{\partial^2 f_i\left(t,y_1,\cdots,y_n;p_1,\cdots,p_m\right)}{\partial y_j\,\partial y_k} \ , \qquad 1 \leq i,j,k \leq n \ . \tag{71}$$

For each component i there is a Hessian matrix $H_{i,:,:}$; Figure 2 depicts the Hessians of the time derivative functions for all components of the small stratospheric system. Since the time derivative function is smooth these Hessian matrices are symmetric

$$H_{i,j,k}\left(t,y;p\right) = \frac{\partial^2 f_i\left(t,y;p\right)}{\partial y_j\,\partial y_k} = \frac{\partial^2 f_i\left(t,y;p\right)}{\partial y_k\,\partial y_j} = H_{i,k,j}\left(t,y;p\right) \ . \tag{72}$$

An alternative way to look at the Hessian is to consider the 3-tensor as the derivative of the Jacobian (5) with respect to individual species concentrations

$$H_{i,j,k} = \frac{\partial J_{i,j}\left(t,y_1,\cdots,y_n;p\right)}{\partial y_k} \ , \qquad J_{i,j} = \frac{df_i\left(t,y;p\right)}{dy_j} \ , \qquad 1 \leq i,j,k \leq n \ . \tag{73}$$

The Hessian of the small stratospheric example is represented in Figure 3 as the derivative of the Jacobian

Clearly, the Hessian is a very sparse tensor. KPP computes the number of nonzero Hessian entries (and saves this in the variable NHESS). The Hessian itself is represented in coordinate sparse format; the real vector HESS holds the values, and the integer vectors $IHESS\_I$, $IHESS\_J$, $IHESS\_K$, the indices of nonzero entries

```
HESS    = [ 1, 2, ..., NHESS ]
IHESS_I = [ 1, 2, ..., NHESS ]
IHESS_J = [ 1, 2, ..., NHESS ]
IHESS_K = [ 1, 2, ..., NHESS ]
```

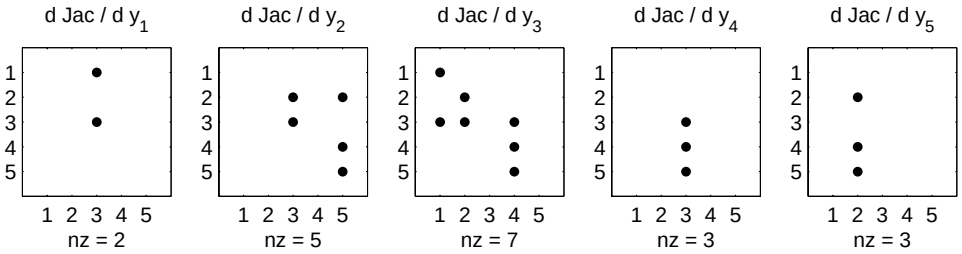

Figure 3: The Hessian of the small stratospheric system seen as derivatives of the Jacobian.

such that the nonzero Hessian entries are stored as

$$H_{i,j,k} = HESS(m)\ , \quad i = IHESS\_I(m)\ , \quad j = IHESS\_J(m)\ , \quad k = IHESS\_K(m)\ ,$$
$$\text{for} \quad H_{i,j,k} \neq 0 \quad \text{and} \quad 1 \leq m \leq NHESS\ .$$

The sparsity coordinate vectors $IHESS\_\{I, J, K\}$ are computed by KPP and initialized statically; these vectors are constant as the sparsity pattern of the Hessian do not change during the computation.

Note that due to the symmetry relation (72) it is enough to store only the upper part of each component's Hessian matrix; the sparse structures for the Hessian store only those entries $H_{i,j,k}$ for which $j \leq k$.

The subroutine

```
SUBROUTINE HessVar ( V, R, F, HESS )
```

Computes the Hessian (in coordinate sparse format) given the concentrations of the variable species V. Note that only the entries $H_{i,j,k}$ with $j \leq k$ are computed and stored.

The subroutine

```
SUBROUTINE HessVar_Vec ( HESS, U1, U2, HU )
```

computes the Hessian times user vectors product

$$HU \longleftarrow (HESS \times U2) \cdot U1$$

which is a vector result. The vector HU can be regarded as the derivative of Jacobian-vector product times vector

$$HU \longleftarrow \frac{\partial \left(J(y) \cdot U1 \right)}{\partial y} \cdot U2\ .$$

Similarly, the subroutine

```
SUBROUTINE HessVarTR_Vec ( HESS, U1, U2, HTU )
```

computes the Hessian transposed times user vectors product (which equals the derivative of Jacobian transposed-vector product times vector)

$$HTU \longleftarrow (HESS \times U2)^T \cdot U1 = \frac{\partial \left(J^T(y) \cdot U1 \right)}{\partial y} \cdot U2\ .$$

6 Building Direct-Decoupled Code with KPP

In this section we present a tutorial example for the direct-decoupled code implementation in the KPP framework. For simplicity, we consider the problem of evaluating the sensitivities $S_\ell(t) \in \mathcal{R}^n$ of the model state (concentrations) at time moments t, $t^0 \leq t < t^{\rm F}$, with respect to the concentration of a species ℓ at a previous time instance t^0

$$S_\ell(t) = \frac{\partial [y_1(t), \cdots, y_n(t)]}{\partial y_\ell(t^0)} \ , \quad \ell = 1, \ldots, n \ . \tag{74}$$

Thet forward numerical integration, $y^i \rightarrow y^{i+1}$, is performed using the first order linearly implicit Euler method [18]

$$\left(J(t^i, y^i) - \frac{1}{h} I \right) \left(y^i - y^{i+1} \right) = f\left(t^i, y^i \right) \ , \quad i = 0, \ldots, N-1 \ , \tag{75}$$

with a constant step size h; the final time is reached for $t^N = Nh = t^{\rm F}$. The implementation of one forward integration step using KPP generated routines is

```
      SUBROUTINE LEULER(n,y,h,t)
      ....
C -- compute the function, Jacobian
      CALL FunVar    (n,t,y,fval)
      CALL JacVar_SP (n,t,y,jac)
C -- compute J-(1/h)*I and factorize it
      jac(lu_diag_v(1:n)) = jac(lu_diag_v(1:n)) -1.0d0/h
      CALL KppDecomp (n, jac, ier)
C -- solve:  [y^{i}-y^{i+1}] = ( J - (1/h)*I )**(-1) * fval
      CALL KppSolve (jac, fval)
C -- update next step solution: y^{i+1} = y^{i} - [y^{i}-y^{i+1}]
      y(1:n) = y(1:n) - fval(1:n)
      END
```

To calculate sensitivities with respect to initial values we follow the formulas (20) in the one stage, autonomous form. The Hessian is denoted by $H = \partial J / \partial y$.

$$\begin{aligned}
y^{i+1} = y^i - k_0 \ , \qquad & S_\ell^{i+1} = S_\ell^i - k_\ell \quad \text{for} \quad 1 \leq \ell \leq m \ , \\
\left(J(t^i, y^i) - \frac{1}{h} I \right) \ k_0 \ &= \ f(t^i, y^i) \\
\left(J(t^i, y^i) - \frac{1}{h} I \right) \ k_\ell \ &= \ J\left(t^i, \, y^i \right) \ S_\ell^i + \left(H(t^i, y^i) \times S_\ell^i \right) \ k_i^0 \quad \text{for} \quad 1 \leq \ell \leq m \ .
\end{aligned}$$

For an implementation of the direct-decoupled method we consider a state vector $y(n + nm)$ for both the concentrations and the sensitivities; the concentrations are stored in $y(1, \cdots, n)$ and the sensitivities $S_\ell(1, \cdots, n)$ in $y(\ell n + 1, \cdots, \ell n + n)$ for all $1 \leq \ell \leq m$. The direct decoupled step is implemented as follows.

```
       SUBROUTINE DDM_LEULER(n,m,y,h,t)
       ....
C -- compute the function, Jacobian, Hessian
       CALL Fun       ( n, T, y, fval )
       CALL JacVar_SP ( n, T, y, jac  )
```

```
      CALL HessVar   ( n, T, y, hess )
C -- compute J-(1/h)*I and factorize it
      ajac(1:lu_nonzero_v) = jac(1:lu_nonzero_v)
      ajac(lu_diag_v(1:n)) = ajac(lu_diag_v(1:n)) - 1.0d0/H
      CALL KppDecomp (n, ajac, ier)
C -- for the concentrations: compute [fval(1:n) = y^{i}-y^{i+1}]
      CALL KppSolve (ajac, fval) ! for the concentrations
C -- for each sensitivity coefficient l:
C --     compute [fval(l*n+1:(l+1)*n) = y^{i}-y^{i+1}]
      DO l=1,m
         CALL JacVar_SP_Vec( jac,y(l*n+1),fval(l*n+1) )
         CALL HessVar_Vec  ( hess,y(l*n+1),fval(1),hval   )
         fval(l*n+1:(l+1)*n) = fval(l*n+1:(l+1)*n) + hval(1:n)
         CALL KppSolve (ajac, fval(l*n+1))
      END DO
C -- update next step solution: y^{i+1} = y^{i} - [y^{i}-y^{i+1}]
      y(1:n*(m+1)) = y(1:n*(m+1)) - fval(1:n*(m+1))
      T = T + H
      END
```

Note that an implementation of the direct decoupled code for computing sensitivities with respect to rate coefficients can be easily obtained following the same pattern and using the KPP-generated subroutines for the function and Jacobian derivatives with respect to the rate coefficients.

7 Building Adjoint Code with KPP

First applications of the KPP software tools to the adjoint code generation for chemical kinetics systems were presented by Daescu et al. [5] who reported a superior performance over the adjoint code generated with the general purpose adjoint compiler TAMC (Giering [15]) for a two-stage Rosenbrock method. In this section we present a tutorial example for the continuous and discrete adjoint code implementation in the KPP framework. For simplicity, we consider the problem of evaluating the sensitivities $S(t) \in \mathcal{R}^n$ of the concentration of the j^{th} chemical species in the model at a fixed instant in time $t^{\mathrm{F}} > t^0$ with respect to the model state at previous instants in time, $t^0 \leq t < t^{\mathrm{F}}$

$$S(t) = \left(\frac{\partial y_j(t^{\mathrm{F}})}{\partial y_1(t)}, \cdots, \frac{\partial y_j(t^{\mathrm{F}})}{\partial y_n(t)} \right)^T \quad , \tag{76}$$

with the linearly implicit Euler method (75).

7.1 Continuous adjoint implementation

The continuous adjoint system (28)-(29) is initialized with $\lambda^N = \lambda(t^{\mathrm{F}}) = e_j$, where e_j is the j^{th} vector of the canonical base in $\mathcal{R}^n$. One step of the backward integration, $\lambda^{i+1} \rightarrow \lambda^i$, using the linearly implicit Euler method is written (use (28) and (75) with $h \leftarrow -h$)

$$\left(J^T(t^{i+1}, y^{i+1}) - \frac{1}{h} I \right) (\lambda^{i+1} - \lambda^i) = J^T(t^{i+1}, y^{i+1}) \lambda^{i+1} \tag{77}$$

and after rearranging we obtain

$$\left(J(t^{i+1},y^{i+1})-\frac{1}{h}I\right)^T\lambda^i=-\frac{1}{h}\lambda^{i+1} \tag{78}$$

As we noticed in the previous section, since the adjoint problem is linear, fully implicit methods may be implemented at the same computational cost as linearly implicit methods. If the backward integration is performed with the implicit Euler method then

$$\left(J(t^{i},y^{i})-\frac{1}{h}I\right)^T\lambda^i=-\frac{1}{h}\lambda^{i+1} \tag{79}$$

with the only difference from (77) being that the Jacobian matrix in (79) is now evaluated at (t^i,y^i). The implementation of the continuous backward integration step using the KPP generated routines is outlined below. On input $ady=\lambda^{i+1}$, on output $ady=\lambda^i$. For the linearly implicit Euler method we input the time and state variables $t=t^{i+1},y=y^{i+1}$, whereas for the implicit Euler method the input is $t=t^i,y=y^i$.

```
      SUBROUTINE CAD_LEULER(n,y,ady,h,t)
      ....
C -- compute J-(1/h)*I and factorize it
      CALL JacVar_SP (n,t,y,jac)
      jac(lu_diag_v(1:n)) = jac(lu_diag_v(1:n)) -1.0d0/h
      CALL KppDecomp (n, jac, ier)
C -- compute:  ( J-(1/h)*I )**(-1) *  ( -(1/h)*ady )
      f(1:n) = (-1.0d0/h)*ady(1:n)
      CALL KppSolveTR(jac,f,ady)
      END
```

Remark 5. An analysis of the forward (75) and backward (78) integrations shows that the forward integration requires an additional function evaluation. Since cpu(KppSolve) $\approx$ cpu(KppSolveTR), it follows that per integration step, the continuous adjoint model integration using the (linearly) implicit Euler method is more economical than the forward model integration. However, for higher order methods the continuous adjoint integration is in general more expensive per step than the forward integration with the same numerical method. An adjoint function evaluation requires the product $J^T\lambda$ which is in general more expensive than evaluating f. In addition, forward recomputations may be required during the adjoint integration.

7.2 Discrete adjoint implementation

The discrete adjoint code is implemented according to the formulae (48) and (55). Differentiating (75) with respect to y_i we obtain

$$\frac{\partial}{\partial y^i}[J^i(\overline{y}^i-\overline{y}^{i+1})]+\left(J^i-\frac{1}{h}I\right)\left(I-\frac{\partial y^{i+1}}{\partial y^i}\right)=J^i \tag{80}$$

where $J^i=J(t^i,y^i)$ and the notation $\overline{y}^{i+1},\overline{y}^i$ is used to express the fact that these terms are treated as constants (given by their numerical value) during the differentiation of the first term in (80). From (80) we obtain explicitly

$$\frac{\partial y^{i+1}}{\partial y^i}=I-\left(J^i-\frac{1}{h}I\right)^{-1}\left(J^i+\frac{\partial}{\partial y^i}[J^i(\overline{y}^{i+1}-\overline{y}^i)]\right) \tag{81}$$

such that from (55) and (81) it results

$$\lambda^i = \left(\frac{\partial y^{i+1}}{\partial y^i}\right)^T \lambda^{i+1} = \lambda^{i+1} - \left(J^i + \frac{\partial}{\partial y^i}[J^i(\overline{y}^{i+1} - \overline{y}^i)]\right)^T \left(J^i - \frac{1}{h}I\right)^{-T} \lambda^{i+1} \tag{82}$$

Equation (82) represents the discrete adjoint integration step associated with the linearly implicit Euler forward integration. After introducing two new variables k and z

$$k = \overline{y}^{i+1} - \overline{y}^i \tag{83}$$

$$\left(J^i - \frac{1}{h}I\right)^T z = \lambda^{i+1} \tag{84}$$

equation (82) becomes

$$\lambda^i = \lambda^{i+1} - (J^i)^T z - \left(\frac{\partial}{\partial y^i}[J^i k]\right)^T z \tag{85}$$

The careful reader will notice that there is no need to evaluate the product $(J^i)^T z$ in the right side of (85). From (84) we have

$$(J^i)^T z = \lambda^{i+1} + \frac{1}{h}z \tag{86}$$

such that (85) may be simplified to

$$\lambda^i = -\frac{1}{h}z - \left(\frac{\partial}{\partial y^i}[J^i k]\right)^T z \tag{87}$$

To evaluate the right hand side term in (87), one issue which must be addressed is the presence of the second order derivatives and we emphasize that in general the matrix $\partial[J^i k]/\partial y^i$ is not symmetric. However, for any vector $z \in \mathcal{R}^n$ the matrix $\partial((J^i)^T z)/\partial y^i$ is symmetric since

$$\frac{\partial((J^i)^T z)}{\partial y^i} = \sum_{\ell=1}^{n} H_\ell^i z_\ell \ , \quad H_\ell^i = \left.\frac{\partial^2 f_\ell}{\partial y^2}\right|_{t=t^i, y=y^i} \ , \tag{88}$$

where H_ℓ^i is the Hessian matrix of the ℓ-th component of the rate function $f_\ell : \mathcal{R}^n \to \mathcal{R}$ evaluated at y^i. Therefore the following property holds

$$\left(\frac{\partial}{\partial y^i}[J^i k]\right)^T z = \left(\frac{\partial}{\partial y^i}\left[(J^i)^T z\right]\right) k \ , \quad \forall z, k \in \mathcal{R}^n \ . \tag{89}$$

A proof of the properties (88) and (89) is presented by Daescu et al. (2000). The right hand side term in equation (89) is now efficiently evaluated through a call to the KPP generated routine HessVarTR_Vec taking full advantage of the Hessian sparsity. Using (83)-(89), the KPP implementation of the discrete adjoint step (82) is

```
      SUBROUTINE DAD_LEULER(n,y,ynext,ady,h,t)
      ....
C -- compute J-(1/h)*I and factorize it
      CALL JacVar_SP (n,t,y,jac)
      jac(lu_diag_v(1:n)) = jac(lu_diag_v(1:n)) -1.0d0/h
      CALL KppDecomp (n, jac, ier)
C -- compute k
      k(1:n) = ynext(1:n)-y(1:n)
```

```
C -- compute z
      CALL KppSolveTR    (jac,ady,z)
C -- compute Hessian and (hess^T x z)*k
      CALL HessVar       ( y, hess )
      CALL HessVarTR_Vec ( hess, z, k, ady1 )
C -- update adjoints
      ady(1:n) = -z(1:n)/h  - ady1(1:n)
      END
```

On input $ady = \lambda^{i+1}, y = y^i, ynext = y^{i+1}, t = t^i$, on output $ady = \lambda^i$.

Remark 6. If the chemical mechanism involves only reactions up to second order, then the Hessian matrices H_l in (88) are state independent. Since in general the reaction rates are periodically updated (e.g. every 10 min.), there is no need to evaluate the Hessian matrix (CALL HessVar) at each integration step. Instead, by periodically updating the Hessian matrix significant computational savings may be achieved.

Remark 7. By comparison of (82) with (78)-(79) it can be seen that discrete adjoint model is a more demanding computational process and its efficient implementation is not a trivial task. For this reason, the use of discrete adjoints in atmospheric chemistry applications has been limited to explicit or low order linearly implicit numerical methods (Fisher and Lary [14], Elbern et al. [10, 11, 12], Daescu et al. [5]).

Remark 8. For linear dynamics $J = J(t)$, the expression involving second order derivatives in (87) is identically zero, such that the discrete adjoint equation simplifies to

$$\lambda^i = -\frac{1}{h} z = \left(J^i - \frac{1}{h} I \right)^{-T} (-\frac{1}{h} \lambda^{i+1}) \tag{90}$$

which is equivalent to (79). Therefore, the discrete adjoint model induced by the forward integration with the linearly implicit Euler method is equivalent with the backward integration of the continuous adjoint model using the implicit Euler method.

8 The KPP Numerical Library

The KPP numerical library is extended with a set of numerical integrators and drivers for direct decoupled and for adjoint sensitivity computations.

Several Rosenbrock methods are implemented for direct decoupled sensitivity analysis, namely Ros1, Ros2, Ros3, Rodas3, and Ros4. The implementations distinguish between sensitivities with respect to initial values and sensitivities with respect to parameters for efficiency. In addition, the off-the-shelf BDF direct-decoupled integrator Odessa [22] is available in the library; this modified version of Odessa uses the KPP sparse linear algebra routines.

Drivers are present for the general computation of sensitivities with respect to all initial values (general_ddm_ic) and with respect to several (user-defined) rate coefficients (general_ddm_rc). Note that KPP produces the building blocks for the simulation and also for the sensitivity calculations; it also provides application programming templates. Some minimal programming may be required from the users in order to construct their own application from the KPP building blocks.

The continuous adjoint model can be easily constructed using KPP-generated routines and is integrated with any user selected numerical method. The discrete adjoint models associated with the Ros1, Ros2, and Rodas3 integrators are provided for variable step size integration. Drivers for adjoint sensitivity and data assimilation applications are also included.

9 Conclusions and Future Work

The analysis of comprehensive chemical reactions mechanisms, parameter estimation techniques, and variational chemical data assimilation applications require the development of efficient sensitivity methods for chemical kinetics systems. Popular methods for chemical sensitivity analysis include the direct decoupled method and automatic differentiation.

In this paper we review the theory of the direct and the adjoint methods for sensitivity analysis in the context of chemical kinetic simulations. The direct method integrates the model and its sensitivity equations simultaneously and can efficiently evaluate the sensitivities of all concentrations with respect to one initial condition or model parameter. An efficient numerical implementation is the direct decoupled method, traditionally formulated using BDF formulas. We extended the direct decoupled approach to Runge-Kutta and Rosenbrock stiff integration methods. The adjoint method integrates the adjoint of the tangent linear model backwards in time and can efficiently evaluate the sensitivities of a scalar response function with respect to the initial conditions and model parameters. Sensitivities with respect to time dependent model parameters may be obtained through a single backward integration of the adjoint model.

The Kinetic PreProcessor KPP developed by the authors is a symbolic engine that translates a given chemical mechanism into Fortran or C kinetic simulation code. Efficiency is obtained by carefully exploiting the sparsity structure of the Jacobian. A comprehensive suite of stiff numerical integrators is also provided.

The second part of this paper presents the comprehensive set of software tools for sensitivity analysis that were developed and implemented in the new release of the Kinetic PreProcessor (KPP-1.2). KPP was extended to symbolically generate code for the stoichiometric formulation of kinetic systems; for a direct sparse multiplication of Jacobian transposed times vector; for the solution of linear systems involving the transposed Jacobian; for the derivatives of rate function and its Jacobian with respect to reaction coefficients; for the Hessian, i.e. the derivatives of the Jacobian with respect to the concentrations; and for the sparse tensor product of Hessian with user defined vectors. The Hessian and the derivatives with respect to rate coefficients are sparse entities; KPP analyzes their sparsity and produces simulation code together with efficient sparse data structures.

The use of these software tools to build sensitivity simulation code is outlined, following the theoretical overview of direct and adjoint sensitivity analysis. Direct-decoupled code is constructed in a straightforward way; BDF and Runge-Kutta direct decoupled integrators, as well as specific drivers are included in the KPP numerical library. The need for Jacobian derivatives prevented Rosenbrock methods to be used extensively in direct sensitivity calculations; however, the proposed symbolic differentiation technology makes the computation of these derivatives feasible. The continuous and discrete adjoint models are completely generated by the KPP software taking full advantage of the sparsity of the chemical mechanism. Flexible direct-decoupled and adjoint sensitivity code implementations are achieved, and various numerical integration methods can be employed with minimal user intervention.

In the companion paper [6] we present an extensive set of numerical experiments and demonstrate the efficiency of the KPP software as a tool for direct/adjoint sensitivity applications. These applications include direct decoupled and adjoint sensitivity calculations with respect to initial conditions and rate coefficients; time-dependent sensitivity analysis; variational data assimilation and parameters estimation.

Acknowledgements

The work of A. Sandu was supported in part by NSF CAREER award ACI-0093139.

References

[1] G.D. Byrne and A.M. Dean. The numerical solution of some chemical kinetics models with VODE and CHEMKIN II. *Computers Chem.*, 17:297–302, 1993.

[2] D. G. Cacuci. Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. *J. Math. Phys.*, 22:2794–2802, 1981.

[3] D. G. Cacuci. Sensitivity theory for nonlinear systems. II. Extensions to additional classes of responses. *J. Math. Phys.*, 22:2803–2812, 1981.

[4] W.P.L. Carter. Documentation of the SAPRC-99 Chemical Mechanism for VOC Reactivity Assessment. Technical Report No. 92-329, and 95-308, Final Report to California Air Resources Board Contract, May 2000.

[5] D. Daescu, G.R. Carmichael, and A. Sandu. Adjoint implementation of Rosenbrock methods applied to variational data assimilation problems. *Journal of Computational Physics*, 165(2):496–510, 2000.

[6] D. Daescu, A. Sandu, and G.R. Carmichael. Direct and Adjoint Sensitivity Analysis of Chemical Kinetic Systems with KPP: II – Numerical validation and Application. *Atmospheric Environment*, submitted, 2002.

[7] V. Damian-Iordache, A. Sandu, M. Damian-Iordache, G. R. Carmichael, and F. A. Potra. KPP - A symbolic preprocessor for chemistry kinetics - User's guide. Technical report, The University of Iowa, Iowa City, IA 52246, 1995.

[8] V. Damian-Iordache, A. Sandu, M. Damian-Iordache, G. R. Carmichael, and F. A. Potra. The Kinetic Preprocessor KPP - A Software Environment for Solving Chemical Kinetics. submitted to Computers and Chemical Engineering, 2001.

[9] A. M. Dunker. The decoupled direct method for calculating sensitivity coefficients in chemical kinetics. *Journal of Chemical Physics*, 81:2385, 1984.

[10] H. Elbern, H. Schmidt, and A. Ebel. Variational data assimilation for tropospheric chemistry modeling. *Journal of Geophysical Research*, 102(D13):15,967–15,985, 1997.

[11] H. Elbern and H. Schmidt. A four-dimensional variational chemistry data assimilation for Eulerian chemistry transport model. *Journal of Geophysical Research*, 104(D15):18,583–18,598, 1999.

[12] H. Elbern, H. Schmidt, and A. Ebel. Implementation of a parallel 4D-variational chemistry data-assimilation scheme. *Environmental management and health*, 10/4:236–244, 1999.

[13] Q. Errera and D. Fonteyn. Four-dimensional variational chemical assimilation of crista stratospheric measurements. *J. Geophys. Res.*, 106-D11:12,253–12,265, 2001.

[14] M. Fisher and D. J. Lary. Lagrangian four-dimensional variational data assimilation of chemical species. *Q.J.R. Meteorol. Soc.*, 121:1681–1704, 1995.

[15] R. Giering. Tangent linear and adjoint model compiler - users manual. Technical report, Max-Planck-Institut fur Meteorologie, 1997.

[16] R. Giering and T. Kaminski. Recipes for adjoint code construction. *Recipes for Adjoint Code Construction*, 1998.

[17] A. Griewank. *Evaluating derivatives: Principles and Techniques of Algorithmic Differentiation.* Frontiers in Applied Mathematics 19, SIAM, Philadelphia, 2000.

[18] E. Hairer, S.P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems.* Springer-Verlag, Berlin, 1993.

[19] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems.* Springer-Verlag, Berlin, 1991.

[20] M.Z. Jacobson and R.P. Turco. SMVGEAR: a sparse-matrix, vectorized Gear code for atmospheric models. *Atmospheric Environment*, 17:273–284, 1994.

[21] Menut L., Vautard R., Beekmann M., , and Honor C. Sensitivity of photochemical pollution using the adjoint of a simplified chemistry-transport model. *Journal of Geophysical Research - Atmospheres*, 105-D12(15):15,379–15,402, 2000.

[22] J.R. Leis and M.A. Kramer. ODESSA - An Ordinary Differential Equation Solver with Explicit Simultaneous Sensitivity Analysis. *ACM Transactions on Mathematical Software*, 14(1):61, 1986.

[23] M. Caracotsios and W. E. Stewart. Sensitivity analysis of initial value problems with mixed ODEs and algebraic equations. *COMPCE*, 9:359–365, 1985.

[24] G.I. Marchuk. *Adjoint Equations and Analysis of Complex Systems.* Kluwer Academic Publishers, 1995.

[25] G.I. Marchuk, Agoshkov, and P.V. I.V., Shutyaev. *Adjoint Equations and Perturbation Algorithms in Nonlinear Problems.* CRC Press, 1996.

[26] Vautard R., M.Beekmann, and L. Menut. Applications of adjoint modeling in atmospheric chemistry: sensitivity and inverse modeling. *Environmental Modeling and Software*, 15:703–709, 2000.

[27] N. Rostaing, Dalmas, and A. S., Galligo. Automatic differentiation in odyssée. *Tellus*, 45:558–568, 1993.

[28] , A. Sandu, F.A. Potra, V. Damian and G.R. Carmichael. Efficient implementation of fully implicit methods for atmospheric chemistry. *Journal of Computational Physics*, 129:101–110, 1996.

[29] A. Sandu, M. van Loon, F.A. Potra, G.R. Carmichael, and J. G. Verwer. Benchmarking stiff ODE solvers for atmospheric chemistry equations I - Implicit vs. Explicit. *Atmospheric Environment*, 31:3151–3166, 1997.

[30] A. Sandu, J. G. Blom, E. Spee, J. G. Verwer, F.A. Potra, and G.R. Carmichael. Benchmarking stiff ODE solvers for atmospheric chemistry equations II - Rosenbrock Solvers. *Atmospheric Environment*, 31:3459–3472, 1997.

[31] Z. Sirkes and E. Tziperman. Finite difference of adjoint or adjoint of finite difference? *Mon. Weather Rev.*, 49:5–40, 1997.

[32] J. Stoer and R. Burlich. *Introduction to Numerical Analysis.* Springer Verlag, New York, 1980.

[33] 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*, 20:1456–1480, 1999.

[34] K.Y. Wang, D.J. Lary, Shallcross, D.E., Hall aS.M., and Pyle J.A. A review on the use of the adjoint method in four-dimensional atmospheric-chemistry data assimilation. *Q.J.R. Meteorol. Soc.*, 127(576 (Part B)):2181–2204, 2001.