Atmos. Chem. Phys., 6, 187–195, 2006
www.atmos-chem-phys.org/acp/6/187/
SRef-ID: 1680-7324/acp/2006-6-187
European Geosciences Union

Atmospheric Chemistry and Physics

Technical note: Simulating chemical systems in Fortran90 and Matlab with the Kinetic PreProcessor KPP-2.1

A. Sandu[1] **and R. Sander**[2]

[1]Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24060, USA
[2]Air Chemistry Department, Max-Planck Institute for Chemistry, Mainz, Germany

Received: 18 July 2005 – Published in Atmos. Chem. Phys. Discuss.: 13 September 2005
Revised: 30 November 2005 – Accepted: 15 September 2005 – Published: 26 January 2006

Abstract. This paper presents the new version 2.1 of the Kinetic PreProcessor (KPP). Taking a set of chemical reactions and their rate coefficients as input, KPP generates Fortran90, Fortran77, Matlab, or C code for the temporal integration of the kinetic system. Efficiency is obtained by carefully exploiting the sparsity structures of the Jacobian and of the Hessian. A comprehensive suite of stiff numerical integrators is also provided. Moreover, KPP can be used to generate the tangent linear model, as well as the continuous and discrete adjoint models of the chemical system.

1 Introduction

Next to laboratory studies and field work, computer modeling is one of the main methods to study atmospheric chemistry. The simulation and analysis of comprehensive chemical reaction mechanisms, parameter estimation techniques, and variational chemical data assimilation applications require the development of efficient tools for the computational simulation of chemical kinetics systems. From a numerical point of view, atmospheric chemistry is challenging due to the coexistence of very stable (e.g., CH_4) and very reactive (e.g., $O(^1D)$) species. Several software packages have been developed to integrate these stiff sets of ordinary differential equations (ODEs), e.g., Facsimile (Curtis and Sweetenham, 1987), AutoChem (http://gest.umbc.edu/AutoChem), Spack (Djouad et al., 2003), Chemkin (http://www.reactiondesign.com/products/open/chemkin.html), Odepack (http://www.llnl.gov/CASC/odepack/), and KPP (Damian et al., 1995, 2002). KPP is currently being used by many academic, research, and industry groups in several countries (e.g. von Glasow et al., 2002; von Kuhlmann et al., 2003; Trentmann et al., 2003; Tang et al., 2003; Sander et al., 2005). The well-established Master Chemical Mechanism (MCM, http://mcm.leeds.ac.uk/MCM/) has also recently been modified to add the option of producing output in KPP syntax. In the present paper we focus on the new features introduced in the release 2.1 of KPP. These features allow an efficient simulation of chemical kinetic systems in Fortran90 and Matlab. Fortran90 is the programming language of choice for the vast majority of scientific applications. Matlab (http://www.mathworks.com/products/matlab/) provides a high-level programming environment for algorithm development, numerical computations, and data analysis and visualization. The Matlab code produced by KPP allows a rapid implementation and analysis of a specific chemical mechanism. KPP-2.1 is distributed under the provisions of the GNU public license (http://www.gnu.org/copyleft/gpl.html) and can be obtained on the web at http://people.cs.vt.edu/~asandu/Software/Kpp. It is also available in the electronic supplement to this paper at http://www.atmos-chem-phys.org/acp/6/187/acp-6-187-sp.zip.

The paper is organized as follows. Section 2 describes the input information necessary for a simulation, and Sect. 3 presents the output produced by KPP. Aspects of the simulation code generated in Fortran90, Fortran77, C, and Matlab are discussed in Sect. 4. Several applications are presented in Sect. 5. The presentation focuses on the main aspects of modeling but, in the interest of space, omits a number of important (but previously described) features. For a full description of KPP, inculding the installation procedure, the reader should consult the user manual in the electronic supplement.

2 Input for KPP

To create a chemistry model, KPP needs as input a chemical mechanism, a numerical integrator, and a driver. Each of these components can either be chosen from the KPP library

Correspondence to: A. Sandu
(sandu@cs.vt.edu)

or provided by the user. The KPP input files (with suffix .kpp) specify the model, the target language, the precision, the integrator and the driver, etc. The file name (without the suffix .kpp) serves as the default root name for the simulation. In this paper we will refer to this name as "ROOT". To specify a KPP model, write a ROOT.kpp file with the following lines:

```
#MODEL       small_strato
#LANGUAGE    Fortran90
#DOUBLE      ON
#INTEGRATOR  rosenbrock
#DRIVER      general
#JACOBIAN    SPARSE_LU_ROW
#HESSIAN     ON
#STOICMAT    ON
```

We now explain these elements with the help of an example.

2.1 The chemical mechanism

The KPP software is general and can be applied to any kinetic mechanism. Here, we revisit the Chapman-like stratospheric chemistry mechanism from Sandu et al. (2003) to illustrate the KPP capabilities.

$$O_2 \xrightarrow{h\nu} 2O \quad \text{(R1)}$$
$$O + O_2 \rightarrow O_3 \quad \text{(R2)}$$
$$O_3 \xrightarrow{h\nu} O + O_2 \quad \text{(R3)}$$
$$O + O_3 \rightarrow 2O_2 \quad \text{(R4)}$$
$$O_3 \xrightarrow{h\nu} O(^1D) + O_2 \quad \text{(R5)}$$
$$O(^1D) + M \rightarrow O + M \quad \text{(R6)}$$
$$O(^1D) + O_3 \rightarrow 2O_2 \quad \text{(R7)}$$
$$NO + O_3 \rightarrow NO_2 + O_2 \quad \text{(R8)}$$
$$NO_2 + O \rightarrow NO + O_2 \quad \text{(R9)}$$
$$NO_2 \xrightarrow{h\nu} NO + O \quad \text{(R10)}$$

The #MODEL command selects a kinetic mechanism which consists of a species file (with suffix .spc) and an equation file (with suffix .eqn). The species file lists all the species in the model. Some of them are variable (defined with #DEFVAR), meaning that their concentrations change according to the law of mass action kinetics. Others are fixed (defined with #DEFFIX), with the concentrations determined by physical and not chemical factors. For each species its atomic composition is given (unless the user chooses to explicitly ignore it).

```
#INCLUDE atoms
#DEFVAR
  O   = O;
  O1D = O;
  O3  = O + O + O;
  NO  = N + O;
  NO2 = N + O + O;
#DEFFIX
  M   = ignore;
  O2  = O + O;
```

The chemical kinetic mechanism is specified in the KPP language in the equation file:

```
#EQUATIONS { Stratospheric Mechanism }
<R1>  O2  + hv = 2O       : 2.6e-10*SUN;
<R2>  O   + O2 = O3       : 8.0e-17;
<R3>  O3  + hv = O   + O2 : 6.1e-04*SUN;
<R4>  O   + O3 = 2O2      : 1.5e-15;
<R5>  O3  + hv = O1D + O2 : 1.0e-03*SUN;
<R6>  O1D + M  = O   + M  : 7.1e-11;
<R7>  O1D + O3 = 2O2      : 1.2e-10;
<R8>  NO  + O3 = NO2 + O2 : 6.0e-15;
<R9>  NO2 + O  = NO  + O2 : 1.0e-11;
<R10> NO2 + hv = NO  + O  : 1.2e-02*SUN;
```

Here, each line starts with an (optional) equation tag in angle brackets. Reactions are described as "the sum of reactants equals the sum of products" and are followed by their rate coefficients. SUN is the normalized sunlight intensity, equal to one at noon and zero at night. The value of SUN can either be calculated with the KPP-generated subroutine Update_SUN or provided by the user.

2.2 The target language and data types

The target language Fortran90 (i.e. the language of the code generated by KPP) is selected with the command:

```
#LANGUAGE Fortran90
```

Other options are Fortran77, C, and Matlab. The capability to generate Fortran90 and Matlab code are new features of KPP-2.1, and this is what we focus on in the current paper.

The data type of the generated model is set to double precision with the command:

```
#DOUBLE ON
```

2.3 The numerical integrator

The #INTEGRATOR command specifies a numerical solver from the templates provided by KPP or implemented by the user. More exactly, it points to an integrator definition file. This file is written in the KPP language and contains a reference to the integrator source code file, together with specific options required by the selected integrator.

Each integrator may require specific KPP-generated functions (e.g., the production/destruction function in aggregate or in split form, and the Jacobian in full or in sparse format, etc.) These options are selected through appropriate parameters given in the integrator definition file. Integrator-specific parameters that can be fine tuned for better performance are also included in the integrator definition file.

KPP implements several Rosenbrock methods: Ros-1 and Ros-2 (Verwer et al., 1999), Ros-3 (Sandu et al., 1997), Rodas-3 (Sandu et al., 1997), Ros-4 (Hairer and Wanner, 1991), and Rodas-4 (Hairer and Wanner, 1991). For details on Rosenbrock methods and their implementation, consult section IV.7 of Hairer and Wanner (1991).

The KPP numerical library also contains implementations of several off-the-shelf, highly popular stiff numerical solvers:

– Radau5:
This Runge Kutta method of order 5 based on Radau-IIA quadrature (Hairer and Wanner, 1991) is stiffly accurate. The KPP implementation follows the original implementation of Hairer and Wanner (1991), and the original Fortran 77 implementation has been translated to Fortran 90 for incorporation into the KPP library. While Radau5 is relatively expensive (when compared to the Rosenbrock methods), it is more robust and is useful to obtain accurate reference solutions.

– SDIRK4:
This is an L-stable, singly-diagonally-implicit Runge Kutta method of order 4. SDIRK4 originates from Hairer and Wanner (1991), and the original Fortran 77 implementation has been translated to Fortran 90 for incorporation into the KPP library.

– SEULEX:
This variable order stiff extrapolation code is able to produce highly accurate solutions. The KPP implementation follows the one in Hairer and Wanner (1991), and the original Fortran 77 implementation has been translated to Fortran 90 for incorporation into the KPP library.

– LSODE, LSODES:
The Livermore ODE solver (Radhakrishnan and Hindmarsh, 1993) implements backward differentiation formula (BDF) methods for stiff problems. LSODE has been translated from Fortran 77 to Fortran 90 for incorporation into the KPP library. LSODES (Radhakrishnan and Hindmarsh, 1993), the sparse version of the Livermore ODE solver LSODE, is modified to interface directly with the KPP generated code.

– VODE:
This solver (Brown et al., 1989) uses a different formulation of backward differentiation formulas. The version of VODE present in the KPP library uses directly the KPP sparse linear algebra routines.

– ODESSA:
The BDF-based direct-decoupled sensitivity integrator ODESSA (Leis and Kramer, 1986) has been modified to use the KPP sparse linear algebra routines.

All methods in the KPP library have excellent stability properties for stiff systems. The computational work per time step depends on the type of method (BDF, Runge Kutta, or Rosenbrock). For Rosenbrock methods it increases with the number of stages. The order of accuracy typically increases with the number of stages as well. For example, Ros-2 uses only two stages and is an order 2 method while Ros-4 has 4 stages and is an order 4 method. Despite their added work, higher order methods are typically more efficient when higher accuracy is desired. The KPP library offers different methods that can have relative advantages depending on the level of desired solution accuracy. Regarding stability, all the Rosenbrock methods are L-stable, and therefore are well suited for stiff problems. For these methods the embedded formulas are also L-stable (or strongly A-stable) which makes the step size controller work well for stiff problems. Radau5 is a fully implicit Runge Kutta method with 3 stages and order 5. Each step is relatively expensive since it requires one real and one complex LU decomposition. However, Radau5 is very robust and can provide solutions of high accuracy. It is recommended that Radau5 is used to obtain reference solutions, against which one can check the correctness and the accuracy of other methods. A lighter (less expensive) Runge Kutta method is SDIRK4, a diagonally implicit formula of order 4, which only requires one real LU decomposition per step (SDIRK4 has about the same cost as a Rosenbrock method, but it performs some additional back substitutions required in the solution of nonlinear systems at each step). While SDIRK4 does not have the strong nonlinear stability properties of Radau5, it is L-stable and is also well suited for solving stiff chemical kinetics.

The implementations use the KPP sparse linear algebra routines by default. However, full linear algebra (using LAPACK routines) is also implemented. To switch from sparse to full linear algebra the user only needs to define the preprocessor variable (`-DFULL_ALGEBRA=1`) during compilation.

2.4 The driver

The so-called driver is the main program. It is responsible for calling the integrator routine, reading the data from files and writing the results. Existing drivers differ from one another by their input and output data file format, and by the auxiliary files created for interfacing with visualization tools. For large 3-D atmospheric chemistry simulations, the 3-D code will replace the default drivers and call the KPP-generated routines directly. In our current implementation of KPP into a 3-D model (Sander et al., 2005), the chemical mechanism is integrated separately for each grid cell. The interaction between the grid cells (advection, diffusion, convection ...) is calculated outside of KPP, i.e. the operator splitting approach is used.

2.5 The substitution preprocessor

Templates are inserted into the simulation code after being run by the substitution preprocessor. This preprocessor replaces placeholders in the template files with their particular values in the model at hand. For example, `KPP_ROOT` is replaced by the model (ROOT) name, `KPP_REAL` by the selected real data type, and `KPP_NSPEC` and `KPP_NREACT` by the numbers of species and reactions, respectively.

2.6 The inlined code

In order to offer maximum flexibility, KPP allows the user to include pieces of code in the kinetic description file. Inlined code begins on a new line with `#INLINE` and the inline type. Next, one or more lines of code follow, written in Fortran90, Fortran77, C, or Matlab, as specified by the inline type. The inlined code ends with `#ENDINLINE`. The code is inserted into the KPP output at a position which is also determined by inline type.

3 Output of KPP

KPP generates code for the temporal integration of chemical systems. This code consists of a set of global variables, plus several functions and subroutines. The code distinguishes between variable and fixed species. The ordinary differential equations are produced for the time evolution of the variable species. Fixed species enter the chemical reactions, but their concentrations are not modified. For example, the atmospheric oxygen O_2 is reactive, however its concentration is in practice not influenced by chemical reactions.

3.1 Parameters

KPP defines a complete set of simulation parameters which are global and can be accessed by all functions. Important simulation parameters are the total number of species (`NSPEC`=7), the number of variable (`NVAR`=5) and fixed (`NFIX`=2) species, the number of chemical reactions (`NREACT`=10), the number of nonzero entries in the Jacobian (`NONZERO`=18) and in the sparse Hessian (`NHESS`=10), etc. The explicit values given here refer to our example from Sect. 2.1. KPP orders the variable species such that the sparsity pattern of the Jacobian is maintained after an LU decomposition and defines their indices explicitly. For our example:

```
ind_O1D=1, ind_O=2, ind_O3=3, ind_NO=4,
ind_NO2=5, ind_M=6, ind_O2=7
```

3.2 Global data

KPP defines a set of global variables that can be accessed by all routines. This set includes `C(NSPEC)`, the array of concentrations of all species. `C` contains variable (`VAR(NVAR)`) and fixed (`FIX(NFIX)`) species. Rate coefficients are stored in `RCONST(NREACT)`, the current integration time is `TIME`, absolute and relative integration tolerances are `ATOL(NSPEC)` and `RTOL(NSPEC)`, etc.

3.3 The chemical ODE function

The chemical ODE system for our example is:

$$
\begin{aligned}
\frac{\mathrm{d}[\mathrm{O}(^1\mathrm{D})]}{\mathrm{d}t} &= k_5\,[\mathrm{O_3}] - k_6\,[\mathrm{O}(^1\mathrm{D})]\,[\mathrm{M}] - k_7\,[\mathrm{O}(^1\mathrm{D})]\,[\mathrm{O_3}] \\
\frac{\mathrm{d}[\mathrm{O}]}{\mathrm{d}t} &= 2\,k_1\,[\mathrm{O_2}] - k_2\,[\mathrm{O}]\,[\mathrm{O_2}] + k_3\,[\mathrm{O_3}] \\
&\quad - k_4\,[\mathrm{O}]\,[\mathrm{O_3}] + k_6\,[\mathrm{O}(^1\mathrm{D})]\,[\mathrm{M}] \\
&\quad - k_9\,[\mathrm{O}]\,[\mathrm{NO_2}] + k_{10}\,[\mathrm{NO_2}] \\
\frac{\mathrm{d}[\mathrm{O_3}]}{\mathrm{d}t} &= k_2\,[\mathrm{O}]\,[\mathrm{O_2}] - k_3\,[\mathrm{O_3}] - k_4\,[\mathrm{O}]\,[\mathrm{O_3}] - k_5\,[\mathrm{O_3}] \\
&\quad - k_7\,[\mathrm{O}(^1\mathrm{D})]\,[\mathrm{O_3}] - k_8\,[\mathrm{O_3}]\,[\mathrm{NO}] \\
\frac{\mathrm{d}[\mathrm{NO}]}{\mathrm{d}t} &= -k_8\,[\mathrm{O_3}]\,[\mathrm{NO}] + k_9\,[\mathrm{O}]\,[\mathrm{NO_2}] + k_{10}\,[\mathrm{NO_2}] \\
\frac{\mathrm{d}[\mathrm{NO_2}]}{\mathrm{d}t} &= k_8\,[\mathrm{O_3}]\,[\mathrm{NO}] - k_9\,[\mathrm{O}]\,[\mathrm{NO_2}] - k_{10}\,[\mathrm{NO_2}]
\end{aligned}
$$

where square brackets denote concentrations of the species. The chemical ODE system has dimension `NVAR`. The concentrations of fixed species are parameters in the derivative function. KPP computes the vector `A` of reaction rates and the vector `Vdot` of variable species time derivatives. The input arguments `V` and `F` are the concentrations of variable and fixed species, respectively. `RCT` contains the rate coefficients. Below is the Fortran90 code generated by KPP for the ODE function of our example stratospheric system. It shows how the KPP-generated code for `Vdot` corresponds to the ODE system shown above, using the indicies from Sect. 3.1:

```
SUBROUTINE Fun (V, F, RCT, Vdot )
   USE small_Precision
   USE small_Params
   REAL(kind=DP) ::  V(NVAR), &
         F(NFIX), RCT(NREACT), &
         Vdot(NVAR), A(NREACT) &
! Computation of equation rates
   A(1) = RCT(1)*F(2)
   A(2) = RCT(2)*V(2)*F(2)
   A(3) = RCT(3)*V(3)
   A(4) = RCT(4)*V(2)*V(3)
   A(5) = RCT(5)*V(3)
   A(6) = RCT(6)*V(1)*F(1)
   A(7) = RCT(7)*V(1)*V(3)
   A(8) = RCT(8)*V(3)*V(4)
   A(9) = RCT(9)*V(2)*V(5)
   A(10) = RCT(10)*V(5)
! Aggregate function
   Vdot(1) = A(5)-A(6)-A(7)
   Vdot(2) = 2*A(1)-A(2)+A(3) &
             -A(4)+A(6)-A(9)+A(10)
```

```
   Vdot(3) = A(2)-A(3)-A(4)-A(5) &
             -A(7)-A(8)
   Vdot(4) = -A(8)+A(9)+A(10)
   Vdot(5) = A(8)-A(9)-A(10)
END SUBROUTINE Fun
```

3.4 The chemical ODE Jacobian

The Jacobian matrix for our example contains 18 non-zero elements:

$\mathbf{J}(1,1) = -k_6\,[\mathrm{M}] - k_7\,[\mathrm{O_3}]$
$\mathbf{J}(1,3) = k_5 - k_7\,[\mathrm{O(^1D)}]$
$\mathbf{J}(2,1) = k_6\,[\mathrm{M}]$
$\mathbf{J}(2,2) = -k_2\,[\mathrm{O_2}] - k_4\,[\mathrm{O_3}] - k_9\,[\mathrm{NO_2}]$
$\mathbf{J}(2,3) = k_3 - k_4\,[\mathrm{O}]$
$\mathbf{J}(2,5) = -k_9\,[\mathrm{O}] + k_{10}$
$\mathbf{J}(3,1) = -k_7\,[\mathrm{O_3}]$
$\mathbf{J}(3,2) = k_2\,[\mathrm{O_2}] - k_4\,[\mathrm{O_3}]$
$\mathbf{J}(3,3) = -k_3 - k_4\,[\mathrm{O}] - k_5 - k_7\,[\mathrm{O(^1D)}] - k_8\,[\mathrm{NO}]$
$\mathbf{J}(3,4) = -k_8\,[\mathrm{O_3}]$
$\mathbf{J}(4,2) = k_9\,[\mathrm{NO_2}]$
$\mathbf{J}(4,3) = -k_8\,[\mathrm{NO}]$
$\mathbf{J}(4,4) = -k_8\,[\mathrm{O_3}]$
$\mathbf{J}(4,5) = k_9\,[\mathrm{O}] + k_{10}$
$\mathbf{J}(5,2) = -k_9\,[\mathrm{NO_2}]$
$\mathbf{J}(5,3) = k_8\,[\mathrm{NO}]$
$\mathbf{J}(5,4) = k_8\,[\mathrm{O_3}]$
$\mathbf{J}(5,5) = -k_9\,[\mathrm{O}] - k_{10}$

It defines how the temporal change of each chemical species depends on all other species. For example, $\mathbf{J}(5,2)$ shows that NO_2 (species number 5) is affected by O (species number 2) via reaction number R9.

The command `#JACOBIAN` controls the generation of the Jacobian routine. The option `OFF` inhibits the construction of the Jacobian. The option `FULL` generates the subroutine `Jac(V,F,RCT,JVS)` which produces a square (`NVAR`×`NVAR`) Jacobian.

The options `SPARSE_ROW` and `SPARSE_LU_ROW` generate the routine `Jac_SP(V,F,RCT,JVS)` which produces the Jacobian in sparse (compressed on rows) format. With the option `SPARSE_LU_ROW`, KPP extends the number of nonzeros to account for the fill-in due to the LU decomposition. The vector `JVS` contains the `LU_NONZERO` elements of the Jacobian in row order. Each row `I` starts at position `LU_CROW(I)`, and `LU_CROW(NVAR+1)=LU_NONZERO+1`. The location of the `I`-th diagonal element is `LU_DIAG(I)`. The sparse element `JVS(K)` is the Jacobian entry in row `LU_IROW(K)` and column `LU_ICOL(K)`. The sparsity coordinate vectors are computed by KPP and initialized statically. These vectors are constant as the sparsity pattern of the Hessian does not change during the computation.

The routines `Jac_SP_Vec(JVS,U,V)` and `JacTR_SP_Vec(JVS,U,V)` perform sparse multiplication of `JVS` (or its transpose) with a user-supplied vector `U` without any indirect addressing.

3.5 Sparse linear algebra

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

$$\mathbf{P}\,\boldsymbol{x} = (\mathbf{I} - h\gamma\mathbf{J})\,\boldsymbol{x} = \boldsymbol{b} \tag{1}$$

where the matrix $\mathbf{P}$=$\mathbf{I}$−$h\gamma\mathbf{J}$ is refered to as the "prediction matrix". $\mathbf{I}$ the identity matrix, h the integration time step, γ a scalar parameter depending on the method, and $\mathbf{J}$ the system Jacobian. The vector $\boldsymbol{b}$ is the system right hand side and the solution $\boldsymbol{x}$ typically represents an increment to update the solution.

The chemical Jacobians are typically sparse, i.e. only a relatively small number of entries are nonzero. The sparsity structure of $\mathbf{P}$ is given by the sparsity structure of the Jacobian, and is produced by KPP with account for the fill-in. By carefully exploiting the sparsity structure, the cost of solving the linear system can be considerably reduced.

KPP orders the variable species such that the sparsity pattern of the Jacobian is maintained after an LU decomposition. KPP defines a complete set of simulation parameters, including the numbers of variable and fixed species, the number of chemical reactions, the number of nonzero entries in the sparse Jacobian and in the sparse Hessian, etc.

KPP generates the routine `KppDecomp(P,IER)` which performs an in-place, non-pivoting, sparse LU decomposition of the matrix $\mathbf{P}$. Since the sparsity structure accounts for fill-in, all elements of the full LU decomposition are actually stored. The output argument IER returns a nonzero value if singularity is detected.

The routines `KppSolve(P,b,x)` and `KppSolveTR(P,b,x)` use the LU factorization of $\mathbf{P}$ as computed by `KppDecomp` and perform sparse backward and forward substitutions to solve the linear systems $\mathbf{P}\,\boldsymbol{x}$=$\boldsymbol{b}$ and $\mathbf{P}^T\,\boldsymbol{x}$=$\boldsymbol{b}$, respectively. The KPP sparse linear algebra routines are extremely efficient, as shown in Sandu et al. (1996).

3.6 The chemical ODE Hessian

The Hessian is used for sensitivity analysis calculations. It contains second order derivatives of the time derivative func-

tions. More exactly, the Hessian is a 3-tensor such that

$$\mathtt{HESS}_{i,j,k} = \frac{\partial^2(\mathtt{Vdot})_i}{\partial \mathtt{V}_j \, \partial \mathtt{V}_k} , \qquad 1 \le i, j, k \le \mathtt{NVAR} . \qquad (2)$$

With the command `#HESSIAN ON`, KPP produces the routine `Hessian(V,F,RCT,HESS)` which calculates the Hessian `HESS` using the variable (`V`) and fixed (`F`) concentrations, and the rate coefficients (`RCT`).

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`, and `IHESS_K` hold the indices of nonzero entries. Since the time derivative function is smooth, these Hessian matrices are symmetric, `HESS(I,J,K)`=`HESS(I,K,J)`. KPP generates code only for those entries `HESS(I,J,K)` with `J`≤`K`.

The sparsity coordinate vectors `IHESS_I`, `IHESS_J`, and `IHESS_K` are computed by KPP and initialized statically. These vectors are constant as the sparsity pattern of the Hessian does not change during the simulation.

The routines `Hess_Vec(HESS,U1,U2,HU)` and `HessTR_Vec(HESS,U1,U2,HU)` compute the action of the Hessian (or its transpose) on a pair of user-supplied vectors `U1`, `U2`. Sparse operations are employed to produce the result vector `HU`.

3.7 Utility routines

In addition to the chemical system description routines discussed above, KPP generates several utility routines. They are used to set initial values and to update the rate coefficients, e.g. according to current temperature, pressure, and solar zenith angle.

It was shown in Sect. 2.1 that each reaction may contain an equation tag. KPP can generate code that allows to obtain individual reaction rates as well as a string describing the chemical reaction. This information could be used, for example, to analyze the chemical mechanism and identify important reaction cycles as shown by Lehmann (2004).

4 Language-specific code generation

The code generated by KPP for the kinetic model description is organized in a set of separate files. The files associated with the model ROOT are named with the prefix "ROOT_". A list of KPP-generated files is shown in Table 1.

4.1 Fortran90

The generated code is consistent with the Fortran90 standard. It will not exceed the maximum number of 39 continuation lines. If KPP cannot properly split an expression to keep the number of continuation lines below the threshold then it will generate a warning message pointing to the location of this expression.

Table 1. List of KPP-generated files (for Fortran90).

File	Description
ROOT_`Main.f90`	Driver
ROOT_`Precision.f90`	Parameterized types
ROOT_`Parameters.f90`	Model parameters
ROOT_`Global.f90`	Global data headers
ROOT_`Monitor.f90`	Equation info
ROOT_`Initialize.f90`	Initialization
ROOT_`Model.f90`	Summary of modules
ROOT_`Function.f90`	ODE function
ROOT_`Jacobian.f90`	ODE Jacobian
ROOT_`JacobianSP.f90`	Jacobian sparsity
ROOT_`Hessian.f90`	ODE Hessian
ROOT_`HessianSP.f90`	Sparse Hessian data
ROOT_`LinearAlgebra.f90`	Sparse linear algebra
ROOT_`Integrator.f90`	Numerical integration
ROOT_`Stoichiom.f90`	Stoichiometric model
ROOT_`StoichiomSP.f90`	Stoichiometric matrix
ROOT_`Rates.f90`	User-defined rate laws
ROOT_`Util.f90`	Utility input-output
ROOT_`Stochastic.f90`	Stochastic functions
`Makefile`_ROOT	Makefile
ROOT`.map`	Human-readable info

The driver file ROOT_`Main.f90` contains the main Fortran90 program. All other code is enclosed in Fortran modules. There is exactly one module in each file, and the name of the module is identical to the file name but without the suffix `.f90`.

Fortran90 code uses parameterized real types. KPP generates the module ROOT_`Precision` which contains the single and double real kind definitions:

```
INTEGER, PARAMETER ::              &
  SP = SELECTED_REAL_KIND(6,30),   &
  DP = SELECTED_REAL_KIND(12,300)
```

Depending on the user choice of the `#DOUBLE` switch the real variables are of type double or single precision. Changing the parameters of the `SELECTED_REAL_KIND` function in this module will cause a change in the working precision for the whole model.

The global parameters (Sect. 3.1) are defined and initialized in the module ROOT_Parameters. The global variables (Sect. 3.2) are declared in the module ROOT_Global. They can be accessed from within each program unit that uses the global module.

The sparse data structures for the Jacobian (Sect. 3.4) are declared and initialized in the module ROOT_JacobianSP. The sparse data structures for the Hessian (Sect. 3.6) are declared and initialized in the module ROOT_HessianSP.

The code for the ODE function (Sect. 3.3) is in module ROOT_Function. The code for the ODE Jacobian and sparse multiplications (Sect. 3.4) is in module ROOT_Jacobian. The Hessian function and associated sparse multiplications (Sect. 3.6) are in module ROOT_Hessian.

The module ROOT_Stoichiom contains the functions for reactant products and its Jacobian, and derivatives with respect to rate coefficients. The declaration and initialization of the stoichiometric matrix and the associated sparse data structures is done in the module ROOT_StoichiomSP.

Sparse linear algebra routines (Sect. 3.5) are in the module ROOT_LinearAlgebra. The code to update the rate constants and the user defined rate law functions are in module ROOT_Rates. The utility and input/output functions (Sect. 3.7) are in ROOT_Util and ROOT_Monitor.

Matlab-mex gateway routines for the ODE function, Jacobian, and Hessian are discussed in Sect. 4.5.

4.2 Matlab

Matlab code allows for rapid prototyping of chemical kinetic schemes, and for a convenient analysis and visualization of the results. Differences between different kinetic mechanisms can be easily understood. The Matlab code can be used to derive reference numerical solutions, which are then compared against the results obtained with user-supplied numerical techniques. Last but not least Matlab is an excellent environment for educational purposes. KPP/Matlab can be used to teach students fundamentals of chemical kinetics and chemical numerical simulations.

Each Matlab function has to reside in a separate m-file. Function calls use the m-function-file names to reference the function. Consequently, KPP generates one m-function-file for each of the functions discussed in Sects. 3.3, 3.4, 3.5, 3.6, and 3.7. The names of the m-function-files are the same as the names of the functions (prefixed by the model name ROOT).

The global parameters (Sect. 3.1) are defined as Matlab `global` variables and initialized in the file ROOT_parameter_defs.m. The global variables (Sect. 3.2) are declared as Matlab `global` variables in the file ROOT_Global_defs.m. They can be accessed from within each Matlab function by using `global` declarations of the variables of interest.

The sparse data structures for the Jacobian (Sect. 3.4), the Hessian (Sect. 3.6), the stoichiometric matrix and the Jacobian of reaction products are declared as Matlab `global` variables in the file ROOT_Sparse_defs.m. They are initialized in separate m-files, namely ROOT_JacobianSP.m ROOT_HessianSP.m, and ROOT_StoichiomSP.m, respectively.

Two wrappers (ROOT_Fun_Chem.m and ROOT_Jac_SP_Chem.m) are provided for interfacing the ODE function and the sparse ODE Jacobian with Matlab's suite of ODE integrators. Specifically, the syntax of the wrapper calls matches the syntax required by Matlab's integrators like ode15s. Moreover, the Jacobian wrapper converts the sparse KPP format into a Matlab sparse matrix.

4.3 C

The driver file ROOT_Main.c contains the main driver and numerical integrator functions, as well as declarations and initializations of global variables (Sect. 3.2).

The generated C code includes three header files which are `#include`-d in other files as appropriate. The global parameters (Sect. 3.1) are `#define`-d in the header file ROOT_Parameters.h. The global variables (Sect. 3.2) are extern-declared in ROOT_Global.h, and declared in the driver file ROOT.c. The header file ROOT_Sparse.h contains extern declarations of sparse data structures for the Jacobian (Sect. 3.4), Hessian (Sect. 3.6), stoichiometric matrix and the Jacobian of reaction products. The actual declarations of each data structures is done in the corresponding files.

The code for each of the model functions, integration routine, etc. is located in the corresponding file (with extension `.c`) as shown in Table 1.

Finally, Matlab mex gateway routines that allow the C implementation of the ODE function, Jacobian, and Hessian to be called directly from Matlab (Sect. 4.5) are also generated.

4.4 Fortran77

The general layout of the Fortran77 code is similar to the layout of the C code. The driver file ROOT_Main.f contains the main program and the initialization of global variables.

The generated Fortran77 code includes three header files. The global parameters (Sect. 3.1) are defined as parameters and initialized in the header file ROOT_Parameters.h. The global variables (Sect. 3.2) are declared in ROOT_Global.h as common block variables. There are global common blocks for real (GDATA), integer (INTGDATA), and character (CHARGDATA) global data. They can be accessed from within each program unit that includes the global header file.

The header file ROOT_Sparse.h contains common block declarations of sparse data structures for the Jacobian (Sect. 3.4), Hessian (Sect. 3.6), stoichiometric matrix and the

Jacobian of reaction products. These sparse data structures are initialized in four named "block data" statements.

The code for each of the model functions, integration routine, etc. is located in the corresponding file (with extension `.f`) as shown in Table 1.

Matlab-mex gateway routines for the ODE function, Jacobian, and Hessian are discussed in Sect. 4.5.

4.5 Mex interfaces

KPP generates mex gateway routines for the ODE function (ROOT_`mex_Fun`), Jacobian (ROOT_`mex_Jac_SP`), and Hessian (ROOT_`mex_Hessian`), for the target languages C, Fortran77, and Fortran90.

After compilation (using Matlab's mex compiler) the mex functions can be called instead of the corresponding Matlab m-functions. Since the calling syntaxes are identical, the user only has to insert the "mex" string within the corresponding function name. Replacing m-functions by mex-functions gives the same numerical results, but the computational time could be considerably shorter, especially for large kinetic mechanisms.

If possible we recommend to build mex files using the C language, as Matlab offers most mex interface options for the C language. Moreover, Matlab distributions come with a native C compiler (lcc) for building executable functions from mex files. Fortran77 mex files work well on most platforms without additional efforts. The mex files built using Fortran90 may require further platform-specific tuning of the mex compiler options.

5 Applications

In this section we illustrate several applications using KPP.

5.1 Benchmark tests

We performed some model runs to test the stability and efficiency of the KPP integrators. First, we used the very simple Chapman-like stratospheric mechanism. Simulations of one month were made using the 10 different integrators ros2, ros3, ros4, rodas3, rodas4, ros2_manual, kpp_radau5, kpp_sdirk, kpp_seulex, and kpp_lsode. The CPU times used for the runs are shown in Table 2. Radau5, the most accurate integrator, is also the slowest. Since the overhead for the automatic time step control is relatively large in this small mechanism, the integrator ros2_manual with manual time step control is by far the fastest here. However, it is also the least precise integrator, when compared to Radau5 as reference.

As a second example, we have performed runs with the complex chemistry model MECCA (Sander et al. (2005), see also http://www.mpch-mainz.mpg.de/~sander/messy/mecca/) simulating gas and aerosol chemistry in the marine boundary layer. We have selected a subset of the MECCA mechanism with 212 species, 106 gas-phase reactions, 266 aqueous-phase reactions, 154 heterogeneous reactions, 37 photolyses, and 48 aqueous-phase equilibria. The CPU times for 8-day simulations with different integrators are shown in Table 2. Again, Radau5 is the slowest integrator. The Rosenbrock integrators with automatic time step control and LSODE are much faster. The integrator ros2_manual with manual time step control was unable to solve this very stiff system of differential equations.

Table 2. Benchmark tests with the small stratospheric mechanism and with MECCA performed on on a Linux PC with a 2 GHz CPU.

Integrator	stratospheric mechanism CPU time [s]	MECCA CPU time [s]
rodas3	0.42	38.71
kpp_lsode	0.32	39.79
ros3	0.38	41.33
rodas4	0.46	49.92
ros4	0.43	51.09
kpp_seulex	0.50	55.31
kpp_sdirk	0.86	63.24
ros2	0.39	69.43
kpp_radau5	0.49	103.33
ros2_manual	0.08	—

5.2 Direct and adjoined sensitivity studies

KPP has recently been extended with the capability to generate code for direct and adjoint sensitivity analysis. This was described in detail by Sandu et al. (2003) and Daescu et al. (2003). Here, we only briefly summarize these features. The direct decoupled method, build using backward difference formulas (BDF), has been the method of choice for direct sensitivity studies. The direct decoupled approach was extended 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 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.

6 Conclusions

The widely-used software environment KPP for the simulation of chemical kinetics was added the capabilities to generate simulation code in Fortran90 and Matlab. An update of the Fortran77 and C generated code was also performed. The new capabilities will allow researchers to include KPP generated modules in state-of-the-art large scale models, for example in the field of air quality studies. Many of these models are implemented in Fortran90. The Matlab capabilities will allow for a rapid prototyping of chemical kinetic systems, and for the visualization of the results. Matlab also offers an ideal educational environment and KPP can be used in this context to teach introductory chemistry or modeling classes.

The KPP-2.1 source code is distributed under the provisions of the GNU public license (http://www.gnu.org/copyleft/gpl.html) and is available in the electronic supplement to this paper at http://www.atmos-chem-phys.org/acp/6/187/acp-6-187-sp.zip. The source code and the documentation can also be obtained from http://people.cs.vt.edu/~asandu/Software/Kpp.

Acknowledgements. The work of A. Sandu was supported by the awards NSF-CAREER ACI 0413872 and NSF-ITR AP&IM 0205198.

Edited by: M. Ammann

References

Brown, P., Byrne, G., and Hindmarsh, A.: VODE: A Variable Step ODE Solver, SIAM J. Sci. Stat. Comput., 10, 1038–1051, 1989.

Curtis, A. R. and Sweetenham, W. P.: Facsimile/Chekmat User's Manual, Tech. rep., Computer Science and Systems Division, Harwell Lab., Oxfordshire, Great Britain, 1987.

Daescu, D., Sandu, A., and Carmichael, G. R.: Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: II – Validation and numerical experiments, Atmos. Environ., 37, 5097–5114, 2003.

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

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

Djouad, R., Sportisse, B., and Audiffren, N.: Reduction of multiphase atmospheric chemistry, J. Atmos. Chem., 46, 131–157, 2003.

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

Lehmann, R.: An algorithm for the determination of all significant pathways in chemical reaction systems, J. Atmos. Chem., 47, 45–78, 2004.

Leis, J. and Kramer, M.: ODESSA – An Ordinary Differential Equation Solver with Explicit Simultaneous Sensitivity Analysis, ACM Transactions on Mathematical Software, 14, 61–67, 1986.

Radhakrishnan, K. and Hindmarsh, A.: Description and use of LSODE, the Livermore solver for differential equations, NASA reference publication 1327, 1993.

Sander, R., Kerkweg, A., Jöckel, P., and Lelieveld, J.: Technical Note: The new comprehensive atmospheric chemistry module MECCA, Atmos. Chem. Phys., 5, 445–450, 2005, **SRef-ID: 1680-7324/acp/2005-5-445**.

Sandu, A., Potra, F. A., Damian, V., and Carmichael, G. R.: Efficient implementation of fully implicit methods for atmospheric chemistry, J. Comp. Phys., 129, 101–110, 1996.

Sandu, A., Verwer, J. G., Blom, J. G., Spee, E. J., Carmichael, G. R., and Potra, F. A.: Benchmarking stiff ODE solvers for atmospheric chemistry problems II: Rosenbrock solvers, Atmos. Environ., 31, 3459–3472, 1997.

Sandu, A., Daescu, D., and Carmichael, G. R.: Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: I – Theory and software tools, Atmos. Environ., 37, 5083–5096, 2003.

Tang, Y., Carmichael, G. R., Uno, I., Woo, J.-H., Kurata, G., Lefer, B., Shetter, R. E., Huang, H., Anderson, B. E., Avery, M. A., Clarke, A. D., and Blake, D. R.: Impacts of aerosols and clouds on photolysis frequencies and photochemistry during TRACE-P: 2. Three-dimensional study using a regional chemical transport model, J. Geophys. Res., 108D, doi:10.1029/2002JD003100, 2003.

Trentmann, J., Andreae, M. O., and Graf, H.-F.: Chemical processes in a young biomass-burning plume, J. Geophys. Res., 108D, doi:10.1029/2003JD003732, 2003.

Verwer, J., Spee, E., Blom, J. G., and Hunsdorfer, W.: A second order Rosenbrock method applied to photochemical dispersion problems, SIAM Journal on Scientific Computing, 20, 1456–1480, 1999.

von Glasow, R., Sander, R., Bott, A., and Crutzen, P. J.: Modeling halogen chemistry in the marine boundary layer. 1. Cloud-free MBL, J. Geophys. Res., 107D, 4341, doi:10.1029/2001JD000942, 2002.

von Kuhlmann, R., Lawrence, M. G., Crutzen, P. J., and Rasch, P. J.: A model for studies of tropospheric ozone and nonmethane hydrocarbons: Model description and ozone results, J. Geophys. Res., 108D, doi:10.1029/2002JD002893, 2003.