The Kinetic PreProcessor KPP – A Software Environment for Solving Chemical Kinetics

Valeriu Damian

SmithKline Beecham Pharmaceuticals, King of Prussia, PA 19406, U.S.A. E-mail: `damiav01@sbphrd.com`

Adrian Sandu

Department of Computer Science, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931. E-mail: `asandu@mtu.edu`

Mirela Damian

Department of Mathematics and Computer Science, Ursinus College, Collegeville, PA 19426, U.S.A. E-mail: `mdamian@ursinus.edu`

Florian Potra

Department of Mathematics, University of Maryland Baltimore County, Baltimore, MD 21250, U.S.A. E-mail: `potra@math.umbc.edu`

Gregory R. Carmichael*

Department of Chemical and Biochemical Engineering, University of Iowa, Iowa City, IA 52242, U.S.A. E-mail: `gcarmich@cgrer.uiowa.edu`

* Corresponding Author.

Abstract

The KPP kinetic preprocessor is a software tool that assists the computer simulation of chemical kinetic systems. The concentrations of a chemical system evolve in time according to the differential law of mass action kinetics. A computer simulation requires the implementation of the differential system and its numerical integration in time.

KPP translates a specification of the chemical mechanism into FORTRAN or C simulation code that implements the concentration time derivative function and its Jacobian, together with a suitable numerical integration scheme. Sparsity in Jacobian is carefully exploited in order to obtain computational efficiency.

KPP incorporates a library with several widely used atmospheric chemistry mechanisms; users can add their own chemical mechanisms to the library. KPP also includes a comprehensive suite of stiff numerical integrators. The KPP development environment is designed in a modular fashion and allows for rapid prototyping of new chemical kinetic schemes as well as new numerical integration methods.

Keywords: Chemical kinetics, automatic code generation, sparsity, numerical integration.

1 Introduction

The solution of large numbers of coupled partial differential equations is necessary to analyze coupled transport/chemistry problems associated with many chemical systems (e.g., air pollution problems such as acid rain and photochemical smog, catalysis and reactor design, combustion). These are often computationally intensive calculations involving millions of variables. For example, a basic description of the photochemical oxidant cycle in the atmosphere requires the treatment of some 70 species involved in 200 coupled non-linear chemical reactions. The numerical integration of the time evolution of the chemical species involved in reaction networks is a challenging task, and often the computational time is dominated by the solution of the coupled and stiff equations arising from the chemical reactions. Writing the code that simulates the chemical evolution in time of the species involved in the chemical mechanism is tedious and error prone work. In addition the problem solving effort often involves the evaluation of several different integrators and/or exploration of alternative reaction mechanisms (e.g., a reduced form or a more explicit chemical representation).

The kinetic preprocessor (KPP) was designed as a general analysis tool to facilitate the numerical solution of chemical reaction network problems. KPP automatically generates FORTRAN or C code that computes the time-evolution of chemical species, starting with a specification of the chemical mechanism. It also generates the Jacobian and other quantities needed to interface with numerical integration schemes. KPP further allows a rich selection of numerical integration schemes and provides a framework for evaluation new integrators and chemical mechanisms.

2 Simulation of chemical kinetics

To better understand the general kinetic problem, we start with a simple example from stratospheric chemistry. Then we give the general formulation of the law of mass action kinetics. At the end of this section we review several approaches for solving the chemical kinetic problem.

2.1 A chemical kinetic example

KPP was successfully used to treat many chemical mechanisms from tropospheric and stratospheric chemistry, including CBM-IV [6], Lloyd, Atkinson and Lurmann [1, 7], SAPRC [3], NASA HSRP/A-ESA, etc.

For illustration purposes in this paper we consider a very simple example, namely the generalized Chapman-type mechanism (1) for the production of ozone:

$$
\begin{array}{rccccccccl}
r1) & O_2 & + & h\nu & \xrightarrow{k_1} & 2\,O & & & & \left(k_1 = 2.643\times 10^{-10}\cdot\sigma^3\right)\\
r2) & O & + & O_2 & \xrightarrow{k_2} & O_3 & & & & \left(k_2 = 8.018\times 10^{-17}\right)\\
r3) & O_3 & + & h\nu & \xrightarrow{k_3} & O & + & O_2 & & \left(k_3 = 6.120\times 10^{-04}\cdot\sigma\right)\\
r4) & O & + & O_3 & \xrightarrow{k_4} & 2\,O_2 & & & & \left(k_4 = 1.576\times 10^{-15}\right)\\
r5) & O_3 & + & h\nu & \xrightarrow{k_5} & O^{1D} & + & O_2 & & \left(k_5 = 1.070\times 10^{-03}\cdot\sigma^2\right)\\
r6) & O^{1D} & + & M & \xrightarrow{k_6} & O & + & M & & \left(k_6 = 7.110\times 10^{-11}\right)\\
r7) & O^{1D} & + & O_3 & \xrightarrow{k_7} & 2\,O_2 & & & & \left(k_7 = 1.200\times 10^{-10}\right)\\
r8) & NO & + & O_3 & \xrightarrow{k_8} & NO_2 & + & O_2 & & \left(k_8 = 6.062\times 10^{-15}\right)\\
r9) & NO_2 & + & O & \xrightarrow{k_9} & NO & + & O_2 & & \left(k_9 = 1.069\times 10^{-11}\right)\\
r10) & NO_2 & + & h\nu & \xrightarrow{k_{10}} & NO & + & O & & \left(k_{10} = 1.289\times 10^{-02}\cdot\sigma\right)
\end{array}
\qquad (1)
$$

The photolysis rates depend on the normalized sun intensity $\sigma(t)$, which is zero at night and one at noon. This mechanism looks at an important subset of stratospheric O_3 chemistry; i.e. that involves photolysis of molecular oxygen (O_2), creation of ozone (O_3) from atomic (O) and molecular oxygen, and destruction of ozone by photolysis, by interaction with excited atomic oxygen (O^{1D}), and by nitrogen oxydes (NO_2 and NO) catalytic cycle.

For all chemical species in the model we want to trace the evolution of their concentrations in time. The initial concentrations are given, as shown in Table 2.1. The concentration of a species is denoted in this section by square brackets (for instance, $[O]$ is the concentration of atomic oxygen O, etc).

Species	Initial concentration
O^{1D}	9.906E+01 (molecules/cm^3)
O	6.624E+08 (molecules/cm^3)
O_3	5.326E+11 (molecules/cm^3)
O_2	1.697E+16 (molecules/cm^3)
NO	8.725E+08 (molecules/cm^3)
NO_2	2.240E+08 (molecules/cm^3)

Table 1: Initial concentrations for the components of the extended Chapman system (1).

Let us now build the evolution equations of ozone (O_3) according to the law of mass action kinetics. The law states that each chemical reaction progresses at a rate proportional to the concentrations of the reactants; the proportionality constants (named reaction rate coefficients) are shown at the right of each equation in (1). In reaction $r2$ above, the number of molecules of O_3 which are produced each time unit is $k_2[O][O_2]$. Simultaneously, $k_3[O_3]$ ozone molecules are consumed each time unit in reaction $r3$, $k_4[O][O_3]$ molecules in reaction $r4$, $k_5[O_3]$ in reaction $r5$, $k_7[O^{1D}][O_3]$ in reaction $r7$ and $k_8[NO][O_3]$ in reaction $r8$. The rate of variation of ozone concentration is therefore

$$\begin{aligned} \frac{d}{dt}[O_3] &= k_2[O][O_2] - k_3[O_3] - k_4[O][O_3] \\ &\quad -k_5[O_3] - k_7[O^{1D}][O_3] - k_8[NO][O_3] \ . \end{aligned} \tag{2}$$

Similar evolution equations can be written for all the other species. Note that molecular oxygen concentration $[O_2]$ is large and is determined by physical processes rather than chemical processes, therefore it is usually considered constant. The differential equations, together with the set of initial concentrations determine the time evolution of the system completely. If we denote the "ozone production term" by:

$$P_{O_3} = k_2[O][O_2]$$

and the "ozone destruction term" by:

$$D_{O_3} = k_3 + k_4[O] + k_5 + k_7[O^{1D}] + k_8[NO]$$

we obtain:

$$\frac{d}{dt}[O_3] = P_{O3} - D_{O3} \cdot [O_3] \tag{3}$$

This is a general pattern: the evolution of any species can be described in terms of P and D as above. The differential equation of mass action kinetics together with the initial conditions completely determine the concentrations at any future moment. The time evolution of the system is presented in Figure 1. In the following we describe the general chemical kinetic problem.

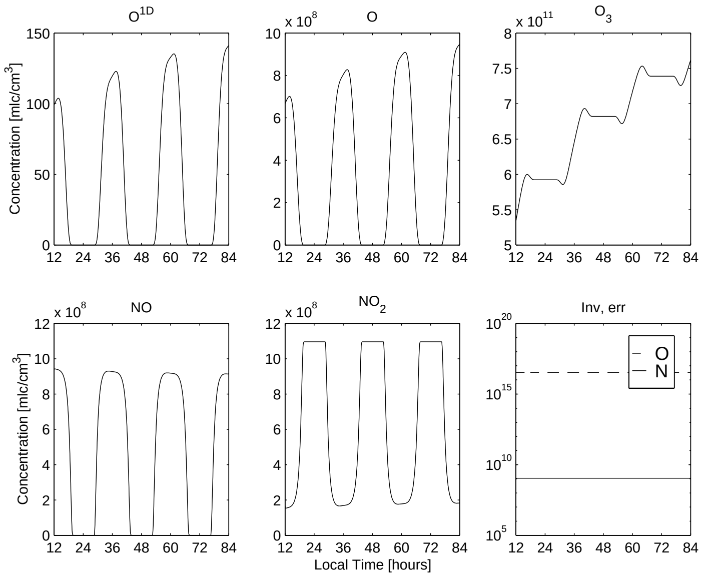

Figure 1: Time evolution of concentrations of the extended Chapman system (1).

2.2 The chemical kinetic problem

Consider a system of n chemical species $y_1, \cdots, y_n$ with R chemical reactions $r_1, \cdots, r_R$. For convenience we denote the concentration of species i also by y_i, and let y be the vector of concentrations of all species involved in the chemical mechanism,

$$y = [\, y_1, \cdots, y_n \,]^T \ .$$

We define k_j to be the rate coefficient of reaction r_j. We also define the *stoichiometric coefficients* $s_{i,j}$ as follows. The stoichiometric coefficient $s^-_{i,j}$ is the number of molecules of species y_i that react (are consumed) in reaction r_j. Similarly, the stoichiometric coefficient $s^+_{i,j}$ is the number of molecules of species y_i that are produced in reaction r_j. Clearly, if y_i is not involved at all in reaction r_j then $s^-_{i,j} = s^+_{i,j} = 0$.

The principle of mass action kinetics states that each chemical reaction – say the $j^{\rm th}$ reaction in our mechanism

$$r_j) \quad \sum s^-_{i,j} y_i \xrightarrow{k_j} \sum s^+_{i,j} y_i \ , \quad 1 \le j \le R \tag{4}$$

progresses at a rate proportional with the concentrations of the reactants; the proportionality constant is the reaction rate coefficient k_j. In general the rate coefficients are time dependent, $k_j = k_j(t)$; for example photolysis reactions $r1$, $r3$, $r5$, and $r10$ are strongest at noon and zero at night.

The reaction velocity (the number of molecules performing the chemical transformation each time unit) is therefore

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

The concentration of species y_i changes at a rate given by the cumulative effect of all chemical reactions,

$$\frac{d}{dt} y_i = \sum_{j=1}^{R} \left(s^+_{i,j} - s^-_{i,j} \right) \omega_j(t,y) \ , \quad i = 1, \cdots, n \tag{5}$$

We organize the stoichiometric coefficients in two matrices,

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

The equation (5) can be rewritten as

$$\frac{d}{dt} y = \left(S^+ - S^- \right) \, \omega\,(t,y) = S\,\omega\,(t,y) = f\,(t,y) \ , \tag{6}$$

where $S = S^+ - S^-$ and $\omega\,(t,y)$ is the vector of all chemical reaction velocities

$$\omega = [\,\omega_1, \cdots, \omega_R\,]^T \ .$$

Clearly some reactions produce y_i, and their rates give the production rates vector

$$S^+\omega\,(t,y) = P(t,y)\ .$$

Other reactions consume y; note that y_i is consumed only if it is a reactant in such a reaction, therefore the destruction rate contains y_i as a factor. We convene to leave out the y_i factors from the definition of the destruction terms. Therefore the destruction term of species i is

$$D_i(t,y) = \sum_{j=1}^{R} s^-_{i,j}\omega_j(t,y)/y_i\ .$$

The diagonal matrix of destruction terms

$$D(t,y) = \mathrm{diag}\left\{D_1(t,y),\cdots,D_n(t,y)\right\}$$

allows the loss rates to be expressed as

$$S^-\omega\,(t,y) = D(t,y)\cdot y\ .$$

If we explicitly separate the positive terms from the negative ones, we derive the *production-destruction* form of the equation

$$\frac{d}{dt}y = P(t,y) - D(t,y)\cdot y\ . \tag{7}$$

To describe the time evolution of the concentrations y_i as given by the mass action kinetics we need the time derivative function either in standard aggregate form $f(t,y)$ as in equation (6), or in split production-destruction form $P(t,y)$, $D(t,y)$ as in equation (7). Given the initial concentrations, the solution of the ordinary differential equations can be traced in time using a *numerical integration method.* Implicit integration methods, suitable for stiff chemical kinetics, also require the evaluation of the Jacobian of the derivative function

$$J = \frac{\partial f(t,y)}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial y_1} & \frac{\partial f_n}{\partial y_2} & \cdots & \frac{\partial f_n}{\partial y_n} \end{bmatrix}\ .$$

2.3 Solving the chemical kinetic problem

There are several possible approaches to translate the chemical reactions into differential equations and numerically solve the latter. The most important ones are summarized below:

(i) **The hard-coded approach**. In this approach the production and the destruction functions related to the chemical equations are derived and coded by hand. This approach has been used in STEM-I and STEM-II developed by Carmichael et. al. [2] using FORTRAN-77. The advantage of this method is that the subroutines that implement the production and the destruction terms can be coded very efficiently. The major disadvantage of this method is that minor changes to the chemical mechanism involve rewriting the whole code from scratch. This makes the hard-coded method very unreliable. Writing such a code is a tedious and error prone work, which poses a big problem to real cases that have hundreds of equations with hundreds of species.

(ii) At the other end is the **totally integrated approach**. In this approach the chemical equations are given in a special file, written in a specific description language. A program parses the equations and stores them in memory as arrays of coefficients, which are used by the production and destruction functions at run time. Thus the code adapts easily to any chemical mechanism. This approach was used by LARKIN in [4]. The drawback of this approach is that all computation is performed at run time, which results in reduced speed.

(iii) A third approach is the **preprocessing approach**. This is the approach that we describe in this paper. Here, as with the totally integrated method, the chemical mechanism is described in a specific language. Then a preprocessing step parses the chemical equations and generates appropriate code that simulates the chemistry kinetics in a high level language (FORTRAN-77 or C). This code is almost as efficient as the code produced using the hard-coded method and is guaranteed to be correct. The code also adapts easily to any changes to the chemical mechanism. This method was partially adopted by CHEMKIN in [5].

The kinetic preprocessor KPP described in this paper fully implements the preprocessing method. It defines a specific language for describing the chemical mechanism, the initial values and the integration options, including the capability to select the numerical integration method and the integration driver. This language is called the **KPP-language**, and is presented in detail in Section 4. KPP generates FORTRAN or C optimized code for the production and destruction functions in split or aggregate form, as well as the Jacobian in either sparse or full format. Special purpose sparse linear algebra routines have been developed [8] and are implemented in KPP. KPP also includes an impressive suite of modern integrators such as VODE, LSODES, RODAS, ROS4, SDIRK, SEULEX, QSSA, EXQSSA, RADAU5, RODAS3 and ROS3. See [9, 10] for a benchmarking of these integrators and references to full descriptions and numerical results.

In what follows we describe KPP from a user perspective and show how to accomplish chemical simulations using the KPP preprocessor.

3 KPP overview

This section contains a high level description of the KPP modules from the user's perspective. The user, presumably a chemist or chemical engineer, describes the application in terms of logical modules such as **chemistry model**, **numerical integrator** and **driver**. For a better understanding of the KPP preprocessor, the following subsections first describe the input files, then the output file and finally the KPP internal modules.

The following subsections present each of these modules starting with the input files, continuing with the output files and concluding with the KPP internal modules.

3.1 KPP input files

Input to KPP is provided in *description files*, which contain commands defined in the KPP language. The KPP language allows the user to specify a set of chemical equations, initial concentration values

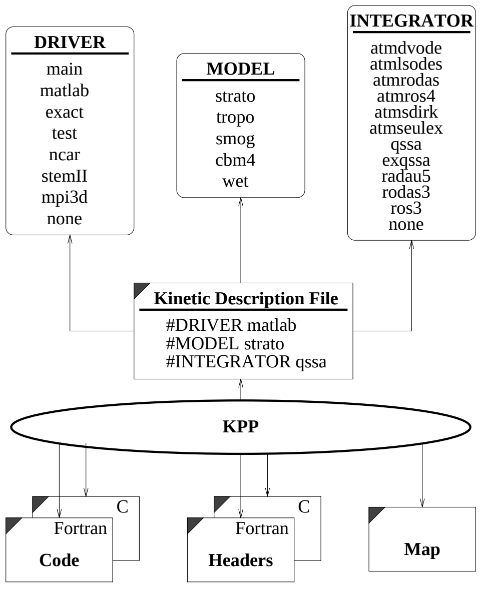

Figure 2: KPP modules

for all species involved, a numerical integration algorithm and a set of integration parameters. The KPP language is described in detail in Section 4.

Kinetic description file. The kinetic description file (***.def**) contains a detailed description of the chemical model including the atoms, chemical species and kinetic equations. It also contains information about the integration method used and the desired type of results. KPP incorporates descriptions of several widely used atmospheric chemistry models: smog model, stratospheric model, tropospheric model, wet deposition model and cbm4 model, as illustrated in Figure 2. These chemical models contain default initial values for chemical species and default options, including the best integrator and driver for the model.

In its simplest form, a kinetic description file may contain a single line selecting one of the predefined models. All default values and options related to the selected model are then used. The default settings can be overridden by selecting a different integrator or driver in the kinetic description file.

KPP accepts only one kinetic description file as input, but we can put together descriptions

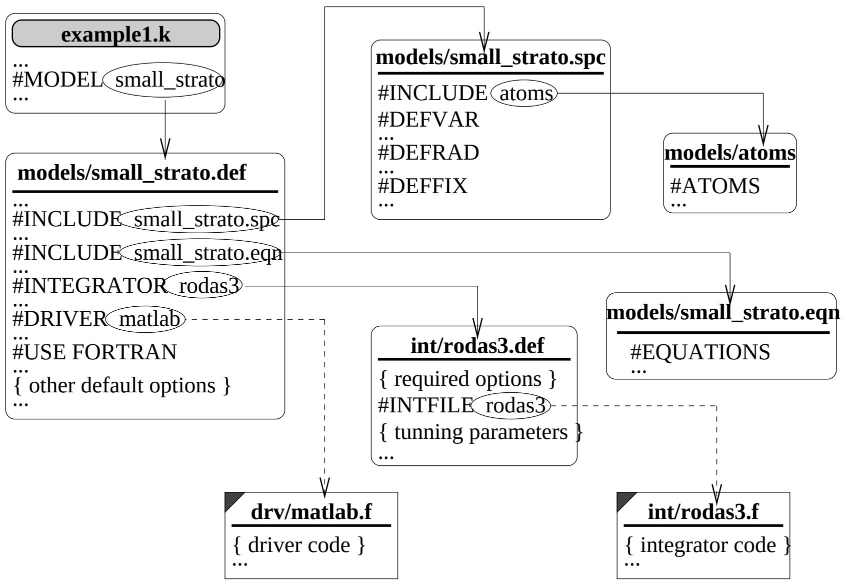

Figure 3: Input files details (default integrator)

from separate files using the #INCLUDE command as described in Section 4.2. These ideas are illustrated in Appendix A.1, which shows the main kinetic description file for a small stratospheric small_strato. Note that the chemical species and chemical equations are defined in two separate kinetic description files (Appendix A.2 and Appendix A.3 respectively), which are included in the main description file using the #INCLUDE command.

By carefully splitting the description of the chemical model, KPP can be configured for a broad range of users. Each user can directly access parts of the chemical model of interest, and keep unwanted details hidden in several include files. A detailed description of the contents of a chemical model definition file is given in Section 4.

Integrator Description File. The kinetic description file points to an integrator description file, also written in the KPP language. The integrator description file in turn contains a reference to a C or FORTRAN source file (depending on the language used) that implements the integrator. In terms of files, the structure of the chemical model is detailed in Figure 3 and Figure 4. In Figure 3, the integrator description file is **int/rodas3.def** and points to the FORTRAN source file **int/rodas3.f**

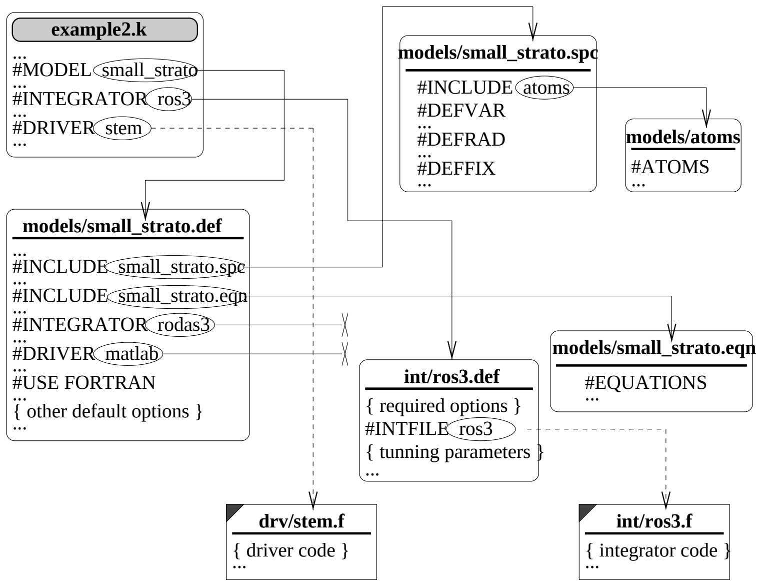

Figure 4: Input files details (selected integrator)

that contains the code for the integrator routine.

Each numerical integrator may require KPP to generate different functions. For instance, some integrators need the derivative function in the aggregate form, while others require the split (production/destruction) form. In addition, implicit integrators require the Jacobian; some integrators use standard linear algebra and need the full Jacobian, while others use sparse linear algebra and require the Jacobian in sparse format. These options are selected through appropriate parameters given in the integrator description file. Some integrators have additional parameters that can be fine tuned for better performance. Values for these parameters are also included in the integrator definition file. For the small stratospheric chemistry mechanism, the integrator description file is given in Appendix B.1.

Driver File. The driver is the main program. The driver is responsible for calling the integrator routine, for reading data from files and writing out the results. Several drivers are built in KPP and they differ from one another by the format of their input and output data files, or by auxiliary files created for interfacing with visualization tools. A program that performs 3D atmospheric chemistry simulation is also called a driver, if it calls the integrator routine generated by KPP for the chemistry integration. Both the driver and integrator code files are regular FORTRAN or C files that implement different drivers and numerical integration algorithms respectively.

3.2 KPP Output files

KPP generates several output files: a *code file*, one or more *header files* and a *map file*. Each file has a time stamp and a complete description of how it was generated. The time stamps are grouped as comments in the target language at the beginning of each output file.

The naming convention used by KPP is that each file generated has the same root name as the kinetic description file and appropriate extension: **.f** or **.c** for the code file (depending on whether FORTRAN or C is used), **.h** and **_s.h** for the header files and **.map** for the map file. For instance, if the name of the kinetic description file is *small_strato.def*, then the map file is called *small_strato.map* and the header files are called *small_strato.h* and *small_strato_s.h*. If FORTRAN is used as target language, then the code file is named *small_strato.f*. If C is used as target language, then the code file is named *small_strato.c*. In the following we give a brief description of the contents of the files generated by KPP.

Code file. This is the main file generated by KPP. It is a complete and correct FORTRAN or C program code that integrates the given chemical mechanism using the selected integration algorithm. Special purpose sparse linear algebra routines are also generated based on the sparsity pattern of the Jacobian, as outlined in [8]. The code can be directly compiled using the FORTRAN or the C compiler. Parts of the code file for the small stratospheric chemistry example are given in Appendix C.2.

Header files. KPP always generates a header file with extension **.h** appended to the root name of the kinetic description file. This header file contains variable and parameter declarations used by all functions in the code file. If FORTRAN is used as a target language, this header file also contains a common block declaration. The header file is normally included in each FORTRAN function. If C is used as a target language, the header file is included only once at the beginning of the C code file. The header file for the small stratospheric chemistry example is given in Appendix C.4.

If the user chooses to have the Jacobian computed by KPP, an additional sparsity header file (***_s.h**) that contains the sparse data structures is generated. The sparsity header file for the stratospheric chemistry example is given in Appendix C.3.

Map file. This file contains supplementary information for the user. Several statistics are listed here. Examples are total number equations, total number of species and the number of radical, variable and fixed species. A list of all species including their name, type and numbering from the chemical mechanism follows. In addition, the map file contains a complete list of all functions generated in the code file. A brief description of the input and output parameters along with the computation performed is generated for each of these functions.

3.3 KPP internal modules

In this section we give a brief description of the internal architecture of the KPP preprocessor. A detailed description of the KPP architecture and its implementation is given in [11]. In the following we briefly describe the basic KPP internal modules and their functionalities. An overall view of the KPP architecture is given in Figure 5.

Scanner and Parser. This module is responsible for reading the kinetic description files and extracting the information necessary in the code generation phase. It stores this information in some internal data structures that define the list of atoms, species, equation rates, options and the coefficient matrices.

We use the Flex and Yacc generic tools to implement the scanner and the parser. The scanner and the parser perform error checking on each input line. For any syntax error encountered in any of the input files, the scanner/parser report a detailed error message along with the exact line number

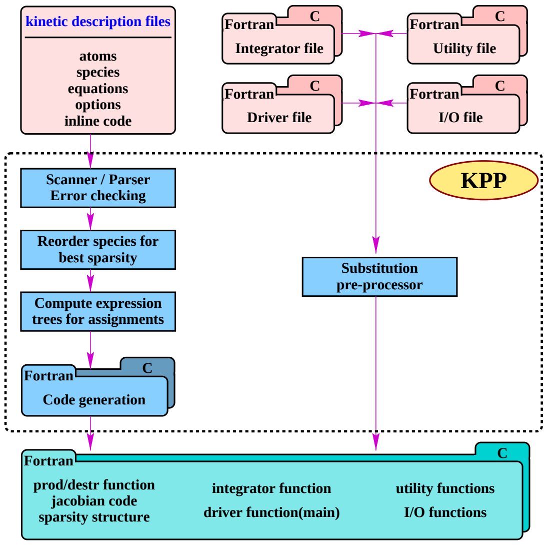

Figure 5: Overall KPP architecture. KPP is representative for the novel generation of software tools we are developing for solving chemistry kinetics

in the input file. In most cases the exact cause of the error can be identified, therefore error messages are very precise. Other errors like mass balance and equation duplicates are checked for at the end of this phase.

Species reordering. Species are reordered using a diagonal Markovitz algorithm to minimize Jacobian fill-in and take maximal advantage of sparsity. This reordering can be turned off is desired.

Expression trees computation. This is the core of the preprocessor. This module generates the production-destruction or aggregate functions along with the Jacobian and all data structures needed by these functions. In doing this efficiently, we build an intermediate structure for each statement in the target code file. Since the vast majority of statements in the target code file are assignments, the intermediate structure is in the form of an expression tree. We build the expression tree incrementally by scanning the coefficient matrices and the rate constant vector, then we simplify this tree to produce a compact output expression. We adopt a similar approach for function declarations and prototypes and data declaration and initialization.

FORTRAN and C code generation. KPP contains two code generation modules, each dealing with the syntax particularities of the FORTRAN 77 and the C languages.

Substitution preprocessor. The substitution processor allows us to include any regular C or FORTRAN code in the generated code, without any changes. Two problems appear here: (i) switching from single to double precision involve replacing all single precision declarations to double precision in the input code (ii) if the input code is in a separate file and uses species defined in the header file, the header file must be included in the input file. However, the name of the header file depends on the model in use. To accommodate for this, KPP defines two tokens: KPP_REAL and KPP_CRTFN.

KPP_CRTFN The preprocessor replaces all occurrences of this token in the input file by the root name of the kinetic description file. For instance, if the line INCLUDE 'KPP_CRTFN.h' appears in the input file and the model **small_strato.def** is used, then this line will be substituted in the generated code by INCLUDE 'small_strato.h'.

KPP_REAL The preprocessor replaces all occurrences of this token by the appropriate single or double precision declaration type. For instance, if the line KPP_REAL BIG(1000) appears in the input file and single precision is used, this line will be substituted in the generated code by REAL BIG(1000). If double precision is used, this line will be substituted by REAL*8 BIG(1000).

4 KPP language

In this section we describe the syntax and semantics of the KPP language and the overall structure of a KPP description file. A KPP description file must conform to the following basic rules:

(i) A KPP description file contains **KPP sections**, **KPP commands** and **program fragments**.

(ii) Anything enclosed between "**{**" and "**}**" is a comment and is ignored by the preprocessor.

(iii) Any name given by the user (to denote an atom or a species, for example) is restricted to be less than 32 characters in length and cannot contain blanks, tabs, new lines, #, +, -, ;, :. All names are **case insensitive**.

4.1 KPP sections

A section begins on a new line with a # sign followed by a **section name**. A section contains a list of items separated by semicolons. The syntax of an item definition is different for each particular section. The sections defined in the KPP language are as follows:

#ATOMS	⟨atom_definition_list⟩
#DEFVAR	⟨species_definition_list⟩
#DEFRAD	⟨species_definition_list⟩
#DEFFIX	⟨species_definition_list⟩
#SETVAR	⟨species_list_plus⟩
#SETRAD	⟨species_list_plus⟩
#SETFIX	⟨species_list_plus⟩
#EQUATIONS	⟨equation_list⟩
#INITVALUES	⟨initvalues_list⟩
#CHECK	⟨atom_list⟩
#LOOKAT	⟨species_list atom_list⟩
#MONITOR	⟨species_list atom_list⟩

In the following we give a detailed description of each of these sections.

Atom definition. The atoms used to specify the components of a species must be declared in an **#ATOMS** section. Usually the names of the atoms are the ones listed in the Periodic Table of Elements. KPP contains a predefined file named **atoms** that defines all atoms in the Periodic Table of Elements, as in:

```
#ATOMS   H; O; N; He; Li; ...
```

To be able to use these definitions in the kinetic description file, we must include the **atoms** file using the command:

```
#INCLUDE atoms
```

This command must appear in the kinetic description file before any statement that uses an atom defined in the atoms file.

Species definition. Species can be **fixed**, **radical** or **variable**. A species is considered **fixed** if its concentration does not change during chemical reactions (e.g. the concentration of O_2 in the atmosphere is driven by physical and not chemical processes). As opposed to fixed species, **radical species** are short life species, i.e., their concentration varies very fast. **Variable** species are medium life species and their concentration varies in time moderately fast.

Fixed, variable and radical species must be declared in the **#DEFFIX**, **#DEFRAD** and **#DEFVAR** sections respectively. These KPP sections define all species involved in the chemical mechanism. The type of a species is implicitly defined by the section that contains the species definition. This division of species into different categories is very useful to integrators that benefit from treating the species differently, based on their type.

For each species the user needs to declare the atom composition. This information is used for mass balance checking. If a species is a generic species and its exact composition is unknown, it can always be ignored. We must inform KPP of any unknown species composition by declaring the predefined atom **IGNORE** as part of the species composition, as in the following example:

```
#DEFVAR
      NO2 = N + 2O;
      RCHO = ignore;
      RO2 = 2O + ignore;
#DEFRAD
```

OH = O + H;

#DEFFIX

CO2 = C + 2O;

Here RCHO and RO2 in photochemical oxydant chemical mechanism in which hydrocarbon reactants are lumped into a relatively small number of species and RCHO represents an aldehyde with a nonspecific organic group R.

We declare species in one category or the other, according to their usual behavior (variable, radical or fixed). Some integrators do not take into account the different types of species involved in the chemical mechanism. In such cases all species should be declared variables. We can redefine the type of an already defined species to be variable by redeclaring it in the **#SETVAR** section:

#SETVAR OH; CO2;

For symmetry, KPP also defines the **#SETRAD** and **#SETFIX** sections, by they are rarely used. KPP also defines a set of generic species named **VAR_SPEC**, **RAD_SPEC**, **FIX_SPEC** and **ALL_SPEC**. The first three generic species in this list refer to all variable, radical and fixed species respectively. The generic species **ALL_SPEC** refers to all species involved in the chemical mechanism. These generic species can be used in any place where a species name is expected:

#DEFVAR ALL_SPEC;

#DEFRAD OH; NO;

These lines state that all species are variable species, except for OH and NO which are radical species. We can change the types of the radical species to variable species subsequently using

#SETVAR RAD_SPEC;

These generic species allow us to easily move all species from one category to another and offer great flexibility in experimenting with different integrators that may or may not take into account the types of the species involved in the chemical mechanism.

Equation specification. The chemical mechanism is specified in the **#EQUATIONS** section. Each equation is written in the natural way, as shown in the example in Appendix A.3. Only names of already defined species can be used. The rate coefficient is always placed at the end of each equation, separated by a colon:

```
O + O2 = O3 : 8.018E-17;
```

The rate coefficient does not necessarily need to be a numerical value. Instead, it can be any valid expression in the target language:

```
OH + NO2 = HNO3 : 2.5*EXP(3.0/TEMP);
```

For the moment, the expression defining the rate coefficient is restricted to be less than 60 characters.

Initial concentration values. The initial concentration values for all species may be defined in the **#INITVALUES** section, as shown in Appendix A.1. For all predefined chemical models, if no value is specified for a particular species, the default value is used. Default values can be set or overridden using the generic species name **ALL_SPEC**:

```
#INITVALUES
    ALL_SPEC = 1.0e-8;
    NO2 = 3.467;
```

These lines state that all species have initial concentration value 1.0e-8, except for NO2 whose concentration is 3.467. Similarly, generic species names **VAR_SPEC**, **RAD_SPEC** and **FIX_SPEC** can be used to set default values for all species in the corresponding category. In order to use coherent units for concentration and rate coefficients, it is sometimes necessary to multiply each concentration value by a constant factor. This factor can be set using the generic name **CFACTOR** (as shown in Appendix A.1). Each of the initial concentration values will be multiplied by this factor before being used.

Mass balance checking. KPP performs mass balance checking for each equation. Some chemical equations are not balanced for all atoms, and this may still be chemically correct. To accommodate for this, KPP performs mass balance checking only for the atoms listed in the **#CHECK** section:

```
#CHECK N; C; O;
```

or, to perform balance checking for all atoms, use

```
#CHECK ALL_SPEC;
```

By default, if this section is missing, no balance checking is performed.

Output data selection. KPP allows us to specify the species whose evolution we are interested in. Such species must be defined in the KPP sections **#LOOKAT** and/or **#MONITOR**:

```
#LOOKAT NO2; CO2; O3; N;
#MONITOR O3; N;
```

The **LOOKAT** section contains a list of species whose evolution is to be saved into a data file by the driver. The drivers built in KPP generate a data file with the same name as the root name as the kinetic description file and extension **.dat**. For instance, if the name of the kinetic description file is *small_strato.def*, then a KPP driver saves the evolution of all species in the LOOKAT section into a file called *small_strato.dat*. To simplify the driver's work, KPP provides three utility functions: *InitSaveData* to open the data file, *SaveData* to actually save the data and *CloseSaveData* to close the data file. The driver may simply call these functions. By default, if no **LOOKAT** section is present, then the evolution of **all** species involved in the chemical mechanism is saved in the data file. If an atom is specified in the **LOOKAT** list, then the total mass of that particular atom is reported. This allows us to check whether the total mass of a specific atom has been conserved by the integration method used.

The **MONITOR** section contains species whose concentrations are displayed at each time step during the integration. This important feature offers instant feedback on the evolution in time of the selected species during integration.

4.2 KPP commands

A command begins on a new line with a # sign, followed by the command name and one or more parameters. Here is the list of all commands available in the KPP language:

```
#INCLUDE file_name
#FUNCTION ⟨ AGGREGATE | SPLIT ⟩
#JACOBIAN ⟨ OFF | ON | SPARSE ⟩
#SPARSEDATA ⟨ OFF | ALL
              LU_ROW | JAC_ROW |
              LU_CROW | JAC_CROW ⟩
#DOUBLE ⟨ OFF | ON ⟩
#CHECKALL
#LOOKATALL
#USE ⟨ FORTRAN | C ⟩
#INTEGRATOR file_name
#DRIVER file_name
#INTFILE file_name
#REORDER ⟨ ON | OFF ⟩
```

In the following we give a detailed description of each of these commands:

#INCLUDE instructs KPP to look for the file with the name specified as a parameter and parse the content of this file before proceeding to the next line. This allows the atom definitions, the species definitions and even the equation definitions to be shared among several models. Moreover this allows for custom configuration of KPP to accommodate various classes of users.

#FUNCTION controls the time derivative functions generated to compute the time evolution of the concentration of the chemical species. Two options are available: AGGREGATE and SPLIT. If the AGGREGATE option is used, KPP generates one function that computes the normal derivatives. If the SPLIT option is used, then KPP generates two functions for the derivatives in production and destruction forms.

#JACOBIAN controls the computation of the Jacobian for the derivative function. Three options are available: OFF, ON and SPARSE. Parameter OFF indicates that the integrator does not need the jacobian and thus the KPP preprocessor does not generate it. Parameter ON indicates that the integrator needs the whole jacobian for variable and radical species. Parameter SPARSE indicates that the integrator needs the whole jacobian, but in a sparse form. The sparse jacobian is stored in *compressed format* in which zeros are not stored explicitly. This is important for applications that require high performance computing, to reduce storage requirements and to also eliminate the need to compute with zeros. The sparse format used is controlled by the SPARSEDATA option; the default is compressed by rows.

#SPARSEDATA controls the sparsity information which is to be generated by KPP. KPP is able to generate sparse data for the jacobian and for the LU decomposition of the jacobian. The sparse matrices are stored in compressed format, in which zeros are not stored explicitly. The simplest format is to store the coordinates (row and column) of the non-zero elements, as shown in the corresponding entry in Table 2. This format can be selected using the options JAC_ROW and LU_ROW for the jacobian and the LU decomposition of the jacobian respectively. Another commonly used format that permits indexed access to rows (but not columns) is *compressed by rows.* This format can be selected using options JAC_CROW and LU_CROW for the jacobian and the LU decomposition of the jacobian respectively. Refer to Table 2 for an example that illustrates the compressed by row format. Three arrays are used in this format: (i) one array **data** to store the nonzero elements of the matrix row by row (ii) a second array **col** of the same size used to store the column positions of the elements in **data** and (iii) a third array **crow** that has one entry for each row of the matrix, and it stores the position in **data** of the first non-zero element of each row of the

Original matrix $\begin{bmatrix} \boxed{a} & \boxed{b} & 0 \\ 0 & 0 & \boxed{c} \\ \boxed{d} & 0 & \boxed{e} \end{bmatrix}$

Nonzero elements	**data:**	a	b	c	d	e
		$\uparrow_1$		$\uparrow_3$	$\uparrow_4$	
Store by coordinates	**row**:	1	1	2	3	3
	col:	1	2	3	1	3
Compressed by row	**crow**:	1	3	4		
	col:	1	2	3	1	3

Table 2: Sparse formats

matrix.

Parameter OFF indicates that no sparse data structure is to be generated by KPP and parameter ALL indicates that *all* sparse data structures discussed above are to be generated by KPP.

#DOUBLE selects single or double precision arithmetic for the integrator. ON means double precision, OFF means single precision.

#CHECKALL is a shorthand command for #CHECK ALL_SPEC described before. It indicates that all species are to be involved in the mass balance checking process.

#LOOKATALL is a shorthand command for #LOOKAT ALL_SPEC described before. It indicates that the concentration of all species must be saved in a data file at each time step during integration.

#USE selects the language used to generate the output code file. Available options are FORTRAN and C. Note that the driver and integrator should also be available in the selected language.

#INTFILE selects the file that contains the integrator routine. This command allows the use of different integration techniques on the same model. The parameter of the command is a file name, without extension. Depending on the language used (FORTRAN or C), the appropriate extension (**.f** or **.c**) is automatically appended. The file given as argument is copied into the code file in the appropriate place. Clearly, this is only possible if all integrators conform to the same specific calling sequence. An integrator routine takes two arguments: TIN (start time) and TOUT (final time). If C is used as target language, the prototype of the integrator routine is:

```
void INTEGRATE( double TIN , double TOUT )
```

If FORTRAN is used as target language, the prototype of the integrator routine is:

```
SUBROUTINE INTEGRATE(TIN, TOUT)
REAL*8 TIN
REAL*8 TOUT
```

Existing general purpose integrators can be embedded in KPP by defining a wrapping function with syntax required by KPP that calls the actual solver with correct arguments.

#INTEGRATOR selects the integrator description file. The parameter is a file name, without extension. The **.def** extension is automatically appended. The effect of:

```
#INTEGRATOR file
```

is identical to:

```
#INCLUDE file.def
```

The integrator description file for the stratospheric example is shown in Appendix (B.1).

#DRIVER selects the driver, which is the file that contains the main function. The parameter is a file name, without extension. Depending on the language used (FORTRAN or C), the appropriate extension (**.f** or **.c**) is automatically appended.

#REORDER specifies whether KPP should reorder the species for best sparsity (option ON) or not (option OFF).

4.3 KPP program fragments

A program fragment begins on a new line with **#INLINE** followed by a fragment type and ends with **#ENDINLINE**. In between there is a piece of code written in FORTRAN on C, depending on the fragment type. Fragment types that start with $F\_$ indicate that FORTRAN code is used and fragment types that start with $C\_$ indicate that C code is used. This piece of code is inserted in the output code file in places also determined by the fragment type. Table 3 lists all fragment types defined in KPP and the position in the generated code where these fragments are placed. If two program fragments with the same fragment type are declared, then one of the following happens:

(a) **override:** The second program fragment overrides the first one. The fragment types that follow this behavior are marked with **override** in Table 3.

(b) **append:** The second program fragment is appended to the first one. The fragment types that follow this behavior are marked with **append** in Table 3.

Examples of program fragments are given in the integrator description file in Appendix B.1.

5 Availability

KPP is publicly available on the web. Both the source code and the documentation can be accessed from

http://www.cs.mtu.edu/~asandu/Kpp or

http://www.cgrer.uiowa.edu/people/vdamian.

Several major environmental research and numerical analysis groups from USA, The Netherlands, France, Germany, Italy, Japan and other countries are currently using KPP.

6 Conclusions

KPP is a symbolic preprocessor for solving chemistry kinetics. It provides a simple natural language to describe the chemical mechanism. In a preprocessing step, KPP parses the chemical equations and generates appropriate output code in a high level language (FORTRAN-77 or C). Because this parsing process is not performed at run time, no overhead is added to the running time to affect the efficiency of the code. Furthermore, the generated code is guaranteed to be correct and adapts easily to any changes to the chemical mechanism. The KPP language used to describe chemical mechanisms is further extended to allow specification of initial values and integration options, including the capability to select the integration method and the integration driver. Special data structures are generated to account for sparsity in the Jacobian.

KPP is written in an expandable way such that new numerical routines can be easily integrated. Moreover the KPP language itself can be easily extended.

KPP has been proved useful to many researchers for chemical model development. Because KPP can be easily configured, it can also be used as a learning tool in general chemistry, as well as a broad range of other courses involving chemical simulations.

Acknowledgements

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

A Small Stratospheric Chemistry Example: KPP input files

A.1 Kinetic Description File

The KPP language allows us to specify the chemical equations, the initial values of each of the species involved and the integration parameters in a file called KPP **kinetic description file**. For the stratospheric model described by the Chapman system (1), the KPP kinetic description file is as follows.

The Kinetic Description File *small_strato.def*:

```
#INCLUDE small_strato.spc
#INCLUDE small_strato.eqn

#USE FORTRAN     {Output Language}
#DOUBLE ON       {Double Precision}
#JACOBIAN SPARSE {Use Sparse DATA STRUCTURES}

#INTEGRATOR rodas3
#DRIVER    general

#LOOKATALL                    {File Output}
#MONITOR O3;N;O2;O;NO;O1D;NO2; {Screen Output}

#INITVALUES   {Initial Values}

CFACTOR = 1.0; {Conversion Factor}
O1D = 9.906E+01 ; O   = 6.624E+08 ;
O3  = 5.326E+11 ; O2  = 1.697E+16 ;
NO  = 8.725E+08 ; NO2 = 2.240E+08 ;
M   = 8.120E+16 ;

#INLINE F_INIT
        TSTART = (12*3600)
        TEND = TSTART + (3*24*3600)
        DT = 0.25*3600
        TEMP = 270
#ENDINLINE
```

A.2 Species Description File

The Species Description File *small_strato.spc*:

```
#INCLUDE atoms {Use the predefined atom list}

#DEFVAR            { Variable Concentration Species}
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            { Fixed Concentration Species}
O2   = O + O;       { Molecular oxygen }
M    = ignore;      { Its composition unimportant }
```

A.3 Equations Description File

The Equations Description File *small_strato.eqn*:

```
#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;
```

B Small Stratospheric Chemistry Example: KPP Library Files

B.1 Integrator Description File

The Integrator Definition File *rodas3.def*:

```
#SETVAR RAD_SPEC     { all radicals considered variables }
#FUNCTION AGGREGATE  { normal function form }
#JACOBIAN SPARSE     { represent Jacobian sparsity ... }
#SPARSEDATA LU_CROW  { ... in compressed row format,
                          with account for fill-in }
#DOUBLE ON           { double precision }
#INTFILE rodas3      { integrator is in file rodas3.f }

#INLINE F_DECL_INT
      INTEGER ISNOTAUTONOM
      COMMON /GDATA/ ISNOTAUTONOM
      REAL*8 STEPSTART
      COMMON /GDATA/ STEPSTART
#ENDINLINE

#INLINE F_INIT_INT
        STEPMIN=0.0001
        STEPMAX=60.
        ISNOTAUTONOM=1
        STEPSTART=STEPMIN
#ENDINLINE
```

C Small Stratospheric Chemistry Example: KPP output files

C.1 The Map File

Extracts from the file *small_strato.map*. The lists of options, parameters and subroutines are not reproduced.

```
### Options ------------------------------------------
FUNCTION - AGGREGATE
JACOBIAN - SPARSE
DOUBLE   - ON

### Parameters ---------------------------------------
(parameter list follows here)

### Species ------------------------------------------
Variable species
  1 = O1D (r)     3 = O3 (r)      5 = NO2 (r)
  2 = O (r)       4 = NO (r)
Radical species
Fixed species
  1 = O2 (r)      2 = M (r)

### Subroutines --------------------------------------
(subroutine list follows here)
```

C.2 Code File

The header file *small_strato.f* contains the following most important elements:

C.2.1 Data structures for the requested sparsity format.

```
C
C Sparse data
C
      BLOCK DATA SPARSE_DATA
      INCLUDE 'small_strato_s.h'
      data LU_ICOL_V / 1, 3,  1,  2,  3,  5,  1,
     *  2,  3,  4,  5,  2,  3,  4,  5,  2,  3,  4,  5 /
      data LU_CROW_V / 1, 3, 7, 12, 16, 20 /
      data LU_DIAG_V / 1, 4,  9, 14, 19, 20 /
      END
```

C.2.2 The subroutine that computes the time derivative of variable concentrations.

```
C ****************************************************************
C
C F_VAR - function for the derivatives of variables - Aggregate form
C   Parameters :
C       V          - Concentration of variable species
C       R          - Concentration of radicals
C       A_VAR      - Aggregate term
```

```
C
C ****************************************************************

      SUBROUTINE F_VAR ( V, R, A_VAR )

      INCLUDE 'small_strato.h'

C V - Concentration of variable species
      REAL*8 V(NVAR)
C R - Concentration of radicals
      REAL*8 R(NRAD)
C A_VAR - Aggregate term
      REAL*8 A_VAR(NVAR)

C Local variables
C A - rate for each equation
      REAL*8 A(NREACT)

C Computation of equation rates
      A(1) = RCONST(1)*F(1)
      A(2) = 8.018e-17*V(2)*F(1)
      A(3) = RCONST(3)*V(3)
      A(4) = 1.576e-15*V(2)*V(3)
      A(5) = RCONST(5)*V(3)
      A(6) = 7.11e-11*V(1)*F(2)
      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) = RCONST(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)
      RETURN
      END

C End of F_VAR function
C ****************************************************************
```

C.2.3 The subroutine that computes the Jacobian of the time derivative function.

```
C ****************************************************************
C
C JACVAR_SP - function for the Jacobian of Variables
C                using sparse matrix representation
C   Parameters :
C       V          - Concentration of variable species
C       R          - Concentration of radicals
```

```
C       JVS       - the Jacobian of variables
C
C ****************************************************************

      SUBROUTINE JACVAR_SP ( V, R, JVS )

      INCLUDE 'small_strato.h'

C V - Concentration of variable species
      REAL*8 V(NVAR)
C R - Concentration of radicals
      REAL*8 R(NRAD)
C JVS - the Jacobian of variables
      REAL*8 JVS(LU_NONZERO_V+1)

C Local variables
C B - temporary matrix
      REAL*8 B(NREACT,NVAR)

C Computation of B(i,j) = dA(i)/dv(j)
      B(2,2) = 8.018e-17*F(1)
      B(3,3) = RCONST(3)
      B(4,2) = 1.576e-15*V(3)
      B(4,3) = 1.576e-15*V(2)
      B(5,3) = RCONST(5)
      B(6,1) = 7.11e-11*F(2)
      B(7,1) = 1.2e-10*V(3)
      B(7,3) = 1.2e-10*V(1)
      B(8,3) = 6.062e-15*V(4)
      B(8,4) = 6.062e-15*V(3)
      B(9,2) = 1.069e-11*V(5)
      B(9,5) = 1.069e-11*V(2)
      B(10,5) = RCONST(10)

C Construct the Jacobian terms from B's
      JVS(1) = -B(6,1)-B(7,1)
      JVS(2) = B(5,3)-B(7,3)
      JVS(3) = B(6,1)
      JVS(4) = -B(2,2)-B(4,2)-B(9,2)
      JVS(5) = B(3,3)-B(4,3)
      JVS(6) = -B(9,5)+B(10,5)
      JVS(7) = -B(7,1)
      JVS(8) = B(2,2)-B(4,2)
      JVS(9) = -B(3,3)-B(4,3)-B(5,3)-B(7,3)-B(8,3)
      JVS(10) = -B(8,4)
      JVS(11) = 0
      JVS(12) = B(9,2)
      JVS(13) = -B(8,3)
      JVS(14) = -B(8,4)
      JVS(15) = B(9,5)+B(10,5)
      JVS(16) = -B(9,2)
      JVS(17) = B(8,3)
      JVS(18) = B(8,4)
      JVS(19) = -B(9,5)-B(10,5)
      RETURN
      END

C End of JACVAR_SP function
C ****************************************************************
```

C.2.4 The numerical integrator subroutine.

```
C ****************************************************************
C
C INTEGRATE - Integrator routine
C   Parameters :
C      TIN       - Start Time for Integration
C      TOUT      - End Time for Integration
C
C ****************************************************************
      SUBROUTINE INTEGRATE( TIN, TOUT )

      INCLUDE 'small_strato.h'

C TIN - Start Time
      REAL*8 TIN
C TOUT - End Time
      REAL*8 TOUT

      INTEGER INFO(5)

      EXTERNAL FUN, JAC_SP

      INFO(1) = ISNOTAUTONOM
        CALL RODAS3(NVAR,TIN,TOUT,STEPMIN,STEPMAX,STEPSTART,
     +                  VAR,ATOL,RTOL,Info,FUN,JAC_SP)

      RETURN
      END


      SUBROUTINE RODAS3(N,T,Tnext,Hmin,Hmax,Hstart,
     +                  y,AbsTol,RelTol,
     +                  Info,FUN,JAC_SP)
      include ' small_strato_s .h'

      (integrator  body here)

      RETURN
      END

      SUBROUTINE FUN(N, T, Y, P)
      INCLUDE 'KPP_CRTFN.h'
      KPP_REAL T, Told
      KPP_REAL Y(NVAR), P(NVAR)
      Told = TIME
      TIME = T
      CALL UPDATE_SUN()
      CALL UPDATE_RCONST()
      CALL F_VAR( Y, RAD, P )
      TIME = Told
      RETURN
      END

      SUBROUTINE JAC_SP(N, T, Y, J)
      INCLUDE 'KPP_CRTFN.h'
      KPP_REAL Told, T
      KPP_REAL Y(NVAR), J(LU_NONZERO_V)
      Told = TIME
      TIME = T
      CALL UPDATE_SUN()
      CALL UPDATE_RCONST()
      CALL JACVAR_SP( Y, RAD, J )
      TIME = Told
      RETURN
      END


C End of INTEGRATE function
C ****************************************************************
```

C.2.5 The driver.

```
C User defined functions


C End user defined functions


C ****************************************************************
C
C MAIN - Main program - driver routine
C   Parameters :
C
C ****************************************************************

      PROGRAM driver


      INCLUDE 'small_strato.h'


      REAL*8 DVAL(NSPEC)
      INTEGER i
      INTEGER NMONITOR_DUMMY, NMASS_DUMMY

      NMONITOR_DUMMY = NMONITOR
      NMASS_DUMMY = NMASS

C ---- TIME VARIABLES ------------------

      TSTART = 0
      TEND = TSTART + 600
      DT = 60.
      TEMP = 298

      STEPMIN = 0.01
      STEPMAX = 900

      RTOLS = 1e-2
      do i=1,NVAR
        RTOL(i) = RTOLS
        ATOL(i) = 1
      end do

      CALL INITVAL()

C ********** TIME LOOP *************************

      CALL InitSaveData()

      write(6,990) (SMONITOR(i), i=1,NMONITOR_DUMMY),
     *             (SMASS(i), i=1,NMASS_DUMMY )
990   FORMAT('done[%] Time[h] ',20(4X,A6))

      TIME = TSTART
      do while (TIME .lt. TEND)

        CALL GETMASS( DVAL )
        write(6,991) (TIME-TSTART)/(TEND-TSTART)*100, TIME/3600.,
     *               (C(MONITOR(i))/CFACTOR, i=1,NMONITOR_DUMMY),
     *               (DVAL(i)/CFACTOR, i=1,NMASS_DUMMY)
991   FORMAT(F6.1,'% ',F7.2,3X,20(E10.4,2X))

        CALL SaveData()

        CALL UPDATE_SUN()
        CALL UPDATE_RCONST()

        CALL INTEGRATE( TIME, TIME+DT )

      end do

      CALL GETMASS( DVAL )
      write(6,991) (TIME-TSTART)/(TEND-TSTART)*100, TIME/3600.,
     *               (C(MONITOR(i))/CFACTOR, i=1,NMONITOR_DUMMY),
     *               (DVAL(i)/CFACTOR, i=1,NMASS_DUMMY)
```

```
      CALL SaveData()

C *********** END TIME LOOP ********

      CALL CloseSaveData()

      STOP
      END

C
C
C End of MAIN function
C ****************************************************************
```

C.3 Sparsity Data Header File

The sparsity header file *small_strato_s.h* contains the specific data that describes the sparse structures. Following is part of this file:

```
      INCLUDE 'small_strato.h'
C
C IROW_V - row indexes for the Jacobian of variables
      INTEGER IROW_V(NONZERO_V)
      COMMON /SDATA/ IROW_V
C ICOL_V - column indexes for the Jacobian of variables
      INTEGER ICOL_V(NONZERO_V)
      COMMON /SDATA/ ICOL_V
C CROW_V - compressed row indexes for the Jacobian of variables
      INTEGER CROW_V(CNVAR)
      COMMON /SDATA/ CROW_V
C DIAG_V - diagonal indexes for the Jacobian of variables
      INTEGER DIAG_V(CNVAR)
      COMMON /SDATA/ DIAG_V
C LU_IROW_V - row indexes for the LU Jacobian of variables
      INTEGER LU_IROW_V(LU_NONZERO_V)
      COMMON /SDATA/ LU_IROW_V
C LU_ICOL_V - column indexes for the LU Jacobian of variables
      INTEGER LU_ICOL_V(19)
      COMMON /SDATA/ LU_ICOL_V
C LU_CROW_V - compressed row indexes for the LU Jacobian of variables
      INTEGER LU_CROW_V(6)
      COMMON /SDATA/ LU_CROW_V
C LU_DIAG_V - diagonal indexes for the LU Jacobian of variables
      INTEGER LU_DIAG_V(6)
      COMMON /SDATA/ LU_DIAG_V
C IROW_R - row indexes for the Jacobian of radicals
      INTEGER IROW_R(NONZERO_R+1)
      COMMON /SDATA/ IROW_R
C ICOL_R - column indexes for the Jacobian of radicals
```

```
      INTEGER ICOL_R(NONZERO_R+1)
      COMMON /SDATA/ ICOL_R
C CROW_R - compressed row indexes for the Jacobian of radicals
      INTEGER CROW_R(CNRAD)
      COMMON /SDATA/ CROW_R
C DIAG_R - diagonal indexes for the Jacobian of radicals
      INTEGER DIAG_R(CNRAD)
      COMMON /SDATA/ DIAG_R
```

C.4 Header File

The header file *small_strato.h*

```
      IMPLICIT REAL*8 ( A-H, O-Z )
C
C NSPEC - The number of species involved
      INTEGER NSPEC
      PARAMETER ( NSPEC = 7 )
C NVAR - The number of Variable species
      INTEGER NVAR
      PARAMETER ( NVAR = 5 )
C NVARACT - The number of Active species
      INTEGER NVARACT
      PARAMETER ( NVARACT = 5 )
C NRAD - The number of Radical species
      INTEGER NRAD
      PARAMETER ( NRAD = 1 )
C NFIX - The number of Fixed species
      INTEGER NFIX
      PARAMETER ( NFIX = 2 )
C NREACT - The number of reactions
      INTEGER NREACT
      PARAMETER ( NREACT = 10 )
C NONZERO_V - Number of nonzero variable elements
      INTEGER NONZERO_V
      PARAMETER ( NONZERO_V = 18 )
C LU_NONZERO_V - Number of nonzero variable LU elements
      INTEGER LU_NONZERO_V
      PARAMETER ( LU_NONZERO_V = 19 )
C NONZERO_R - Number of nonzero radical elements
      INTEGER NONZERO_R
      PARAMETER ( NONZERO_R = 0 )
C NLOOKAT - number of species to look at
      INTEGER NLOOKAT
      PARAMETER ( NLOOKAT = 7 )
C NMONITOR - number of species to monitor
      INTEGER NMONITOR
      PARAMETER ( NMONITOR = 6 )
C NMASS - number of atoms to check mass balance
```

```
      INTEGER NMASS
      PARAMETER ( NMASS = 1 )

C Indexes declaration for variable species
      INTEGER I_O1D
      PARAMETER ( I_O1D = 1 )
      INTEGER I_O
      PARAMETER ( I_O = 2 )
      INTEGER I_O3
      PARAMETER ( I_O3 = 3 )
      INTEGER I_NO
      PARAMETER ( I_NO = 4 )
      INTEGER I_NO2
      PARAMETER ( I_NO2 = 5 )

C Indexes declaration for radical species

C Indexes declaration for fixed species
      INTEGER I_O2
      PARAMETER ( I_O2 = 1 )
      INTEGER I_M
      PARAMETER ( I_M = 2 )

C User defined variables
      REAL*8 STEPSTART
      COMMON /GDATA/ STEPSTART

C End user defined variables
```

References

[1] R.D. Atkinson, D.L. Baulch, R.A. Cox, R.F.JR. Hampson, J.A. Kerr, and J. Troe. Evaluated kinetic and photochemical data for atmospheric chemistry. *Journal of Chemical Kinetics*, 21:115–190, 1989.

[2] G.R. Carmichael, L. Peters, and T. Kitada. A second generation model for regional-scale transport chemistry deposition. *Atmospheric environment*, 20:173–188, 1986.

[3] W. P. L. Carter. Documentation of the SAPRC-99 Chemical Mechanism for VOC Reactivity Assessment. *Draft report to the California Air Resources Board* http://cert.ucr.edu/ carter/-absts.htm, 1999.

[4] U. Nowak. A short user's guide to LARKIN. Technical report, Konrad-Zuse-Zentrum fuer, Informationstechnik Berlin, 1982.

[5] R. J. Kee, F. M. Rupley, and J. A. Miller. CHEMKIN II: A FORTRAN package for the analysis of gas phase chemical kinetics. Technical report, Sandia National Laboratory, Livermore, CA, 1989.

[6] M.W. Gery, G.Z. Whitten, J.P. Killus, and M.C. Dodge. A photochemical kinetics mechanism for urban and regional scale computer modeling. *Journal of Geophysical Research*, 94:12925–12956, 1989.

[7] F.W. Lurmann, A.C. Loyd, and R. Atkinson. A chemical mechanism for use in long-range transport/acid deposition computer modeling. *Journal of Geophysical Research*, 91:10,905–10,936, 1986.

[8] 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.

[9] 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.

[10] 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.

[11] V. Damian. Computational tools for air quality modeling. PhD Thesis, Department of Computer Science, University of Iowa, 1998.

fragment type	language	behavior	code placement
F_DATA	FORTRAN	append	In global data declaration and initialization. Allows for declaration of new variables.
C_DATA	C	append	
F_DATA_INT	FORTRAN	override	
C_DATA_INT	C	override	
F_DECL	FORTRAN	append	In global data declaration in the header file. Used for external variables or common block description.
C_DECL	C	append	
F_DECL_INT	FORTRAN	override	
C_DECL_INT	C	override	
F_INIT	FORTRAN	append	At the end of the initialization routine. Used to define integration parameters.
C_INIT	C	append	
F_INIT_INT	FORTRAN	override	
C_INIT_INT	C	override	
F_UTIL	FORTRAN	append	In the code file outside of any routine. Used to insert any user defined functions and sub-routines.
C_UTIL	C	append	
F_UTIL_INT	FORTRAN	override	
C_UTIL_INT	C	override	

Table 3: Fragment types