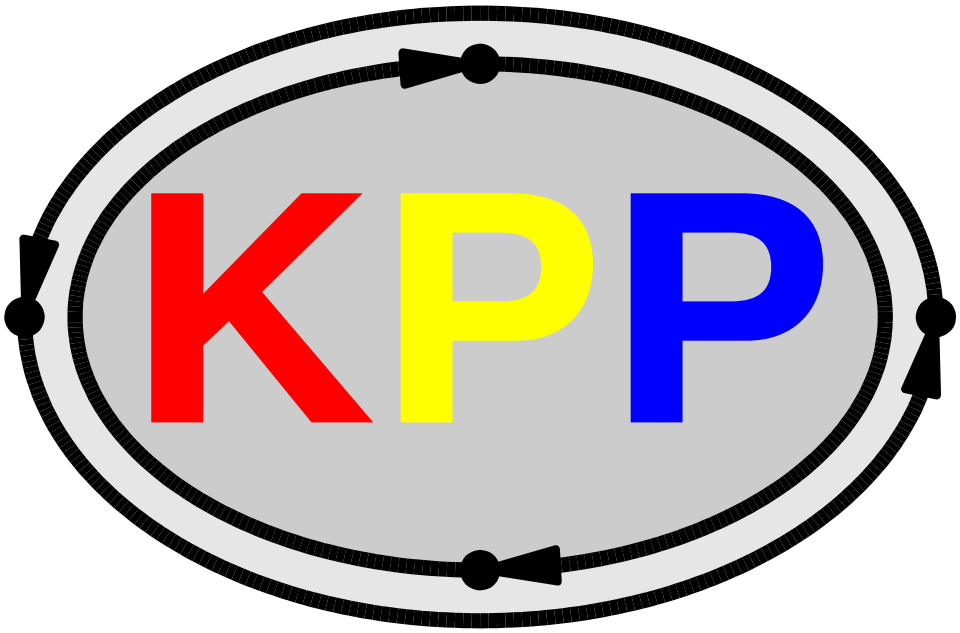

KPP-2.1 User's Manual

The Kinetic PreProcessor KPP
An Environment for the
Simulation of Chemical Kinetic Systems

Adrian Sandu[†] & Rolf Sander[‡]

[†] Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060, USA
`sandu@cs.vt.edu`

[‡] Air Chemistry Department
Max-Planck Institute of Chemistry
PO Box 3060, 55020 Mainz, Germany
`sander@mpch-mainz.mpg.de`

This manual is part of the electronic supplement of our article "Technical note: Simulating chemical systems in Fortran90 and Matlab with the Kinetic PreProcessor KPP-2.1" in Atmos. Chem. Phys. (2005), available at: `http://www.atmos-chem-phys.org`

Date: 2005/07/15

Contents

1 Installation

This section can be skipped if KPP is already installed on your system. If you work under Linux, you can probably use the precompiled executable file that is in the `bin` directory of the distribution. Then you only have to define the `$KPP_HOME` environment variable.

1. Define the `$KPP_HOME` environment variable to point to the complete path where KPP is installed. Also, add the path of the KPP executable to the `$PATH` environment variable. If, for example, KPP is installed in `$HOME/kpp`, under the C shell you have to edit the file `$HOME/.cshrc` and add:

   ```
   setenv KPP_HOME $HOME/kpp
   setenv PATH $PATH:$KPP_HOME/bin
   ```

   If you use the bash shell, edit `$HOME/.bashrc` and add:

   ```
   export KPP_HOME=$HOME/kpp
   export PATH=$PATH:$KPP_HOME/bin
   ```

   After editing `.cshrc` or `.bashrc`, start a new shell to make sure these changes are in effect.

2. Make sure that `sed` is installed on your machine. Type "`which sed`" to test this.

3. Make sure that `yacc` is installed on your machine. Type "`which yacc`" to test this.

4. Make sure that the lexical analizer `flex` is installed on your machine. Type "`flex --version`" to test this. Note down the exact path name where the flex library is installed. The library is called: `libfl.a` or `libfl.sh`.

5. Change to the KPP directory:

   ```
   cd $KPP_HOME
   ```

6. To clean the KPP installation, delete the KPP object files and all the examples with:

   ```
   make clean
   ```

   To delete the KPP executable as well, type:

   ```
   make distclean
   ```

7. Edit `Makefile.defs` and follow the instructions included to specify the compiler, the location of the flex library, etc.

8. Create the kpp executable with:

   ```
   make
   ```

2 Running KPP with an Example Stratospheric Mechanism

Here we consider as an example a very simple Chapman-like mechanism for stratospheric chemistry:

$$\begin{aligned}
O_2 &\xrightarrow{h\nu} 2O && (1)\\
O + O_2 &\rightarrow O_3 && (2)\\
O_3 &\xrightarrow{h\nu} O + O_2 && (3)\\
O + O_3 &\rightarrow 2O_2 && (4)\\
O_3 &\xrightarrow{h\nu} O(^1D) + O_2 && (5)\\
O(^1D) + M &\rightarrow O + M && (6)\\
O(^1D) + O_3 &\rightarrow 2O_2 && (7)\\
NO + O_3 &\rightarrow NO_2 + O_2 && (8)\\
NO_2 + O &\rightarrow NO + O_2 && (9)\\
NO_2 &\xrightarrow{h\nu} NO + O && (10)
\end{aligned}$$

We use the mechanism with the purpose of illustrating the KPP capabilities. However, the software tools are general and can be applied to virtually any kinetic mechanism.

We focus on Fortran90. Particularities of the C, Fortran77, and Matlab languages are discussed in Sections 4.2, 4.3, 4.4, respectively.

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 root name for the simulation. In this paper we will refer to this name as ROOT. Since the root name will be incorporated into Fortran90 module names, it can only contain valid Fortran90 characters, i.e. letters, numbers, and the underscore. 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
```

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

```
#LANGUAGE Fortran90
```

Here, we have chosen Fortran90. See Sect. 3.2.10 for other options.

The data type of the generated model can be switched between single/double precision with the command `#DOUBLE`. The `#INTEGRATOR` command selects a specific numerical

integration routine (from the templates provided by KPP or implemented by the user) and the `#DRIVER` command selects a specific main program. The `#MODEL` command selects a specific kinetic mechanism. In our example the model definition file `small_strato.def` includes the species and the equation files,

```
#INCLUDE small_strato.spc
#INCLUDE small_strato.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 ignore it). The atom file lists the periodic table of elements in an `#ATOM` section. The equation file contains the description of the equations in an `#EQUATIONS` section.

```
#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 file `small_strato.eqn`. Each reaction is described as "the sum of reactants equals the sum of products" and is followed by its rate coefficient. `SUN` is the normalized sunlight intensity, equal to one at noon and zero at night.

```
#EQUATIONS { Stratospheric Mechanism }
<R1>  O2  + hv = 2O       : 2.643E-10*SUN;
<R2>  O   + O2 = O3       : 8.018E-17;
<R3>  O3  + hv = O   + O2 : 6.120E-04*SUN;
<R4>  O   + O3 = 2O2      : 1.576E-15;
<R5>  O3  + hv = O1D + O2 : 1.070E-03*SUN;
<R6>  O1D + M  = O   + M  : 7.110E-11;
<R7>  O1D + O3 = 2O2      : 1.200E-10;
<R8>  NO  + O3 = NO2 + O2 : 6.062E-15;
<R9>  NO2 + O  = NO  + O2 : 1.069E-11;
<R10> NO2 + hv = NO  + O  : 1.289E-02*SUN;
```

To run the model, type:

```
kpp small_strato.kpp
```

Next, compile and run the Fortran90 code:

```
make -fMakefile_small_strato
./small_strato.exe
```

3 Input for KPP

KPP basically handles two types of files: Kinetic description files and auxiliary files. Kinetic description files are in KPP syntax and described in the following sections. Auxiliary files are described in Sect. 3.4. KPP kinetic description files specify the chemical equations, the initial values of each of the species involved, the integration parameters, and many other options. The KPP preprocessor parses the kinetic description files and generates several output files. Files that are written in KPP syntax have one of the suffixes `.kpp`, `.spc`, `.eqn`, or `.def`. An exception is the file `atoms`, which has no suffix.

The following general rules define the structure of a kinetic description file:

- A KPP program is composed of KPP sections, KPP commands and inlined code. Their syntax is presented in the appendix.
- Comments are either enclosed between the curly braces "`{`" and "`}`", or written in a line starting with two slashes "`//`".
- Any name given by the user to denote an atom or a species is restricted to be less than 32 character in length and can only contain letters, numbers, or the underscore character. The first character cannot be a number. All names are case insensitive.

The kinetic description files contain a detailed specification of the chemical model, information about the integration method and the desired type of results. KPP accepts only one of these files as input, but using the `#INCLUDE` command, code from separate files cn be combined. The include files can be nested up to 10 levels. KPP will parse these files as if they were a single big file. By carefully splitting the chemical description, KPP can be configured for a broad range of users. In this way the users can have direct access to that part of the model that they are interested in, and all the other details can be hidden inside several include files. Often, a file with atom definitions is included first, then species definitions, and finally the equations of the chemical mechanism.

3.1 KPP sections

A `#` sign at the beginning of a line followed by a section name starts a new KPP section. Then a list of items separated by semicolons follows. A section ends when another KPP section or command occurs, i.e. when another `#` sign occurs at the beginning of a line. The syntax of an item definition is different for each particular section. Table 1 shows all the sections defined in the KPP language. Each of them will be described separately in the following subsections.

Table 1: KPP sections

name	see Sect.
#ATOMS	3.1.1
#CHECK	3.1.2
#DEFFIX	3.1.3
#DEFVAR	3.1.3
#EQUATIONS	3.1.4
#INITVALUES	3.1.5
#LOOKAT	3.1.6
#LUMP	3.1.7
#MONITOR	3.1.6
#SETFIX	3.1.8
#SETVAR	3.1.8
#TRANSPORT	3.1.9

3.1.1 Atom definitions (#ATOMS)

The atoms that will be further used to specify the components of a species must be declared in an #ATOMS section, e.g.:

```
#ATOMS N; O; Na; Br;
```

Usually, the names of the atoms are the ones specified in the periodic table of elements. For this table there is a predefined file containing all definitions that can be used by the command:

```
#INCLUDE atoms
```

This should be the first line in a KPP input file, because it allows to use any atom in the periodic table of elements throughout the kinetic description file.

3.1.2 Mass balance checking (#CHECK)

KPP is able to do a mass balance checking for all equations. Some chemical equations are not balanced for all atoms, and this might still be correct from a chemical point of view. To accommodate for this, KPP can perform mass balance checking only for the list of atoms specified in the #CHECK section, e.g.:

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

The balance checking for all atoms can be enabled by using the #CHECKALL command. Without #CHECK or #CHECKALL, no checking is performed. The IGNORE atom can also be used to control mass balance checking.

3.1.3 Species definitions (#DEFVAR and #DEFFIX)

There are two ways to declare new species together with their atom composition: #DEFVAR and #DEFFIX. These sections define all the species that will be used in the chemical mechanism. Species can be variable or fixed. The type is implicitly specified by defining the species in the appropriate sections. A species can be considered fixed if its concentration does not vary too much. The variable species are medium or short lived species and their concentrations vary in time. This division of species into different categories is helpful for integrators that benefit from treating them differently.

For each species the user has to declare the atom composition. This information is used for mass balance checking. If the species is a lumped species without an exact composition, it can be ignored. To do this one can declare the predefined atom IGNORE as being part of the species composition. Examples for these sections are:

```
#DEFVAR
  NO2 = N + 2O;
  CH3OOH = C + 4H + 2O;
  HSO4m = IGNORE;
  RCHO = IGNORE;
#DEFFIX
  CO2 = C + 2O;
```

3.1.4 Equations (#EQUATIONS)

The chemical mechanism is specified in the #EQUATIONS section. Each equation is written in the natural way in which a chemist would write it, e.g.:

```
#EQUATIONS
  NO2 + hv = NO + O : 0.533*SUN;
  OH + NO2 = HNO3 : k_3rd(temp,
    cair,2.E-30,3.,2.5E-11,0.,0.6);
```

Only the names of already defined species can be used. The rate coefficient has to be placed at the end of each equation, separated by a colon. The rate coefficient does not necessarily need to be a numerical value. Instead, it can be a valid expression in the target language. If there are several #EQUATIONS sections in the input, their contents will be concatenated.

A minus sign in an equation shows that a secies is consumed in a reaction but it does not affect the reaction rate. For example, the oxidation of methane can be written as:

```
CH4 + OH = CH3OO + H2O - O2 : k_CH4_OH;
```

Often, the stoichiometric factors are integers. However, it is also possible to have non-integer yields, which is very useful to parameterize organic reactions that branch into several side reactions:

```
CH4 + O1D = .75 CH3O2 + .75 OH + .25 HCHO
          + .4 H + .05 H2 : k_CH4_O1D;
```

One restriction is that the list of products must not be empty. If you have such a reaction (e.g. the dry deposition of atmospheric species to the surface), you can define a `DUMMY` species as the product:

```
O3 = DUMMY : v_d_O3;
```

The same equation must not occur twice in the `#EQUATIONS` section. For example, you may have both the gas-phase reaction of N_2O_5 with water in your mechanism and also the heterogeneous reaction on aerosols:

```
N2O5 + H2O = 2 HNO3 : k_gas;
N2O5 + H2O = 2 HNO3 : k_aerosol;
```

These reactions must be merged by adding the rate coefficients:

```
N2O5 + H2O = 2 HNO3 : k_gas+k_aerosol;
```

3.1.5 Initial values (`#INITVALUES`)

The initial concentration values for all species can be defined in the `#INITVALUES` section, e.g.:

```
#INITVALUES
  CFACTOR = 2.5E19;
  NO2 = 1.4E-9;
  CO2 = MyCO2Func();
  ALL_SPEC = 0.0;
```

If no value is specified for a particular species, the default value zero is used. One can set the default values using the generic species names: `VAR_SPEC`, `FIX_SPEC`, and `ALL_SPEC`. In order to use coherent units for concentration and rate coefficients, it is sometimes necessary to multiply each value by a constant factor. This factor can be set by using the generic name `CFACTOR`. Each of the initial values will be multiplied by this factor before being used. If `CFACTOR` is omitted, it defaults to one.

The information gathered in this section is used to generate the `Initialize` subroutine (see Sect. 4.1.3). In more complex 3D models, the initial values are usually taken from some input files or some global data structures. In this case, `#INITVALUES` may not be needed.

3.1.6 Output data selection (`#LOOKAT` and `#MONITOR`)

There are two sections in this category: `#LOOKAT` and `#MONITOR`.

The `#LOOKAT` section instructs the preprocessor what are the species for which the evolution of the concentration, should be saved in a data file. By default, if no `#LOOKAT` section is present, all the species are saved. If an atom is specified in the `#LOOKAT` list then the total mass of the particular atom is reported. This allows to check how the mass of a specific atom was conserved by the integration method. The `#LOOKATALL` command can be used to specify all the species. Output of `#LOOKAT` can be directed to the file ROOT.dat using the utility subroutines described in Sect. 4.1.16.

The `#MONITOR` section defines a different list of species and atoms. This list is used by the driver to display the concentration of the elements in the list during the integration. This may give us a feedback of the evolution in time of the selected species during the integration. The syntax is similar to the `#LOOKAT` section. With the driver `general`, output of `#MONITOR` goes to the screen (STDOUT). The order of the output is: first variable species, then fixes species, finally atoms. It is not the order in the `#MONITOR` command.

Examples for these sections are:

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

3.1.7 Lump species definitions (`#LUMP`)

To reduce the stiffness of some models, various lumping of species may be defined in the `#LUMP` section. The example below shows how the concentration of NO_2 can be replaced by the sum of concentrations for NO_2 and NO which is considered to be a single variable. At the end of the integration, the concentration of NO_2 is computed by substraction from the lumped variable.

```
#LUMP NO2 + NO : NO2
```

3.1.8 Redefining species definitions (`#SETVAR` and `#SETFIX`)

The commands `#SETVAR` and `#SETFIX` change the type of an already defined species. Then, depending on the integration method, one may or may not use the initial classification, or can easily move one species from one category to another. The use of the generic species `VAR_SPEC`, `FIX_SPEC`, and `ALL_SPEC` is also allowed. Examples for these sections are:

```
#SETVAR ALL_SPEC;
#SETFIX H2O; CO2;
```

3.1.9 Transport (`#TRANSPORT`)

The `#TRANSPORT` section is only used for transport chemistry models. It specifies the list of species that needs to be included in the transport model, e.g.:

Table 2: KPP commands

name	see Sect.
#CHECKALL	3.2.16
#DOUBLE	3.2.1
#DRIVER	3.2.2
#DUMMYINDEX	3.2.3
#EQNTAGS	3.2.4
#FUNCTION	3.2.5
#HESSIAN	3.2.6
#INCLUDE	3.2.7
#INTEGRATOR	3.2.8
#INTFILE	3.2.8
#JACOBIAN	3.2.9
#LANGUAGE	3.2.10
#LOOKATALL	3.2.16
#MEX	3.2.11
#MODEL	3.2.12
#REORDER	3.2.13
#STOCHASTIC	3.2.14
#STOICMAT	3.2.15
#TRANSPORTALL	3.2.16

```
#TRANSPORT NO2; CO2; O3; N;
```

One may use a more complex chemical model from which only a couple of species are considered for the transport calculations. The #TRANSPORTALL command is also available as a shorthand for specifying that all the species used in the chemical model have to be included in the transport calculations.

3.2 KPP commands

A command begins on a new line with a # sign, followed by a command name and one or more parameters. A summary of the commands available in KPP is shown in Table 2. Details about each command are given in the following subsections.

3.2.1 Precision control (#DOUBLE)

The #DOUBLE command selects single or double precision arithmetique. ON (the default) means use double precision, OFF means use single precision (see Sect. 4.1.6).

3.2.2 Driver selection (#DRIVER)

The #DRIVER command selects the driver, i.e., the file from which the main function is to be taken. The parameter is a file name, without suffix. The appropriate suffix (.f90, .f, .c, or .m) is automatically appended.

Normally, KPP tries to find the selected driver file in the directory $KPP_HOME/drv/. However, if the supplied file name contains a slash, it is assumed to be absolute. To access a driver in the current directory, the prefix "./" can be used, e.g.:

```
#DRIVER ./mydriver
```

It is possible to choose the empty dummy driver none, if the user wants to include the KPP generated modules into a larger model (e.g. a general circulation or a chemical transport model) instead of creating a stand-alone version of the chemical integrator. The driver none is also selected when the #DRIVER command is missing. If the #DRIVER command occurs twice, the second replaces the first.

3.2.3 Dummy indices (#DUMMYINDEX)

It is possible to declare species in the #SPECIES section that are not used in the #EQUATIONS section. If your model needs to check at run-time if a certain species is included in the current mechanism, you can set #DUMMYINDEX to ON. Then, KPP will set the indices ind_*spc* to zero for all species that do not occur in any reaction. With #DUMMYINDEX OFF (the default), those ind_*spc* are undefined variables. For example, if you frequently switch between mechanisms with and without sulfuric acid, you can use this code:

```
IF (ind_H2SO4=0) THEN
  PRINT *, 'no H2SO4 in current mechanism'
ELSE
  PRINT *, 'c(H2SO4) =', C(ind_H2SO4)
ENDIF
```

3.2.4 Generation of equation tags (#EQNTAGS)

Each reaction in the #EQUATIONS section may start with an equation tag which is enclosed in angle brackets, e.g.:

```
<J1> NO2 + hv = NO + O : 0.533*SUN;
```

With #EQNTAGS set to ON, this equation tag can be used to refer to a specific equation, as described in Sect. 4.1.5. The default for #EQNTAGS is OFF.

3.2.5 The function generation (#FUNCTION)

The #FUNCTION command controls which functions are generated to compute the production/destruction terms for variable species. AGGREGATE generates one function that computes the normal derivatives. SPLIT generates two functions for the derivatives in production and destruction forms.

3.2.6 Generation of Hessian (#HESSIAN)

The option ON (the default) of the #HESSIAN command turns the Hessian generation on (see Sect. 4.1.12). With OFF it is switched off.

3.2.7 File include command (#INCLUDE)

The #INCLUDE command instructs KPP to look for the file specified as a parameter and parse the content of this file before proceeding to the next line. This allows the atoms definition, the species definition and even the equation definition to be shared between several models. Moreover this allows for custom configuration of KPP to accommodate various classes of users. Include files can be either in one of the KPP directories or in the current directory.

3.2.8 Integrator selection (#INTEGRATOR and #INTFILE)

The #INTEGRATOR command selects the integrator definition file. The parameter is the file name of an integrator, without suffix. The effect of:

#INTEGRATOR *integrator*

is similar to:

#INCLUDE $KPP_HOME/int/*integrator*.def

The integrator definition file selects an integrator file with #INTFILE and also defines some suitable options for it. The #INTFILE command 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 suffix. The appropriate suffix (.f90, .f, .c, or .m) is appended and the result selects the file from which the integrator is taken. This file will be copied into the code file in the appropriate place. All integrators have to conform to the same specific calling sequence. Normally, KPP tries to find the selected integrator file in the directory $KPP_HOME/int/. However, if the supplied file name contains a slash, it is assumed to be absolute. To access an integrator in the current directory, the prefix "./" can be used, e.g.:

```
#INTEGRATOR ./mydeffile
#INTFILE ./myintegrator
```

If the #INTEGRATOR command occurs twice, the second replaces the first.

3.2.9 The Jacobian (#JACOBIAN)

The #JACOBIAN command controls which functions are generated to compute the Jacobian. The option OFF inhibits the generation of the Jacobian routine. The option FULL generates the Jacobian as a square (NVAR×NVAR) matrix. It should be used if the integrator needs the whole Jacobians. The options SPARSE_ROW and SPARSE_LU_ROW (the default) both generate the Jacobian in sparse (compressed on rows) format. They should be used if the integrator needs the whole Jacobian, but in a sparse form. The format used is compressed on rows. With SPARSE_LU_ROW, KPP extends the number of nonzeros to account for the fill-in due to the LU decomposition.

3.2.10 Target language selection (#LANGUAGE)

The #LANGUAGE command selects the target language in which the code file is to be generated. Available options are Fortran90, Fortran77, C, or Matlab.

3.2.11 Mex files (#MEX)

Mex is a matlab extension that allows to call functions written in Fortran and C directly from within the Matlab environment. KPP generates the mex interface routines for the ODE function, Jacobian, and Hessian, for the target languages C, Fortran77, and Fortran90. The default is ON. With OFF, no Mex files are generated.

3.2.12 Selcting a chemical model (#MODEL)

The chemical model contains the description of the atoms, species, and chemical equations. It also contains default initial values for the species and default options including the best integrator for the model. In the simplest case, the main kinetic description file, i.e. the one passed as parameter to KPP, can contain just a single line selecting the model. KPP tries to find a file with the name of the model and the suffix .def in the $KPP_HOME/models subdirectory. This file is then parsed. The content of the model definition file is written in the KPP language. The model definition file points to a species file and an equation file. The species file includes further the atom definition file. All default values regarding the model are automatically selected. For convenience, the best integrator and driver for the given model are also automatically selected.

The #MODEL command is optional, and intended for using a predefined model. Users who supply their own reaction mechanism do not need it.

3.2.13 Reordering (#REORDER)

Reordering of the species is performed in order to minimize the fill-in during the LU factorization, and therefore preserve the sparsity structure and increase efficiency. The reordering is done using a diagonal markowitz algorithm. The details are explained in Sandu et al. (1996). The default is ON. OFF means that KPP does not reorder the species. The order of the variables is the order in which the species are declared in the #DEFVAR section.

3.2.14 Stochastic simulation (#STOCHASTIC)

The option ON of the #STOCHASTIC command turns on the generation of code for stochastic kinetic simulations (see Sect. 4.1.15). The default option is OFF.

3.2.15 The Stoichiometric Formulation (#STOICMAT)

Unless this command is set to OFF, KPP generates code for the stoichiometric matrix, the vector of reactant products in each reaction, and the partial derivative of the time derivative function with respect to rate coefficients. These elements are discussed in Sect. 4.1.14.

3.2.16 Shorthand commands (#CHECKALL, #LOOKATALL and #TRANSPORTALL)

KPP defines a couple of shorthand commands. The commands that fall into this category are #CHECKALL, #LOOKATALL and #TRANSPORTALL. All of them have been described in the previous sections.

3.3 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 the target language (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* as explained in Table 3. If two inline commands with the same inline type are declared, then the contents of the second is appended to the first one. In this manual, we show the inline types for Fortran90. The inline types for the other languages are produced by replacing "F90_" by "F77_", "C_", or "MATLAB_", respectively.

3.3.1 Inline type F90_DATA

This inline type was introduced in a previous version of KPP to initialize variables. It is now obsolete but kept for compatibility. For Fortran90, F90_GLOBAL should be used instead.

3.3.2 Inline type F90_GLOBAL

F90_GLOBAL can be used to declare global variables, e.g. for a special rate coefficient:

```
#INLINE F90_GLOBAL
  REAL(dp) :: k_DMS_OH
#ENDINLINE
```

3.3.3 Inline type F90_INIT

F90_INIT can be used to define initial values before the start of the integartion, e.g.:

```
#INLINE F90_INIT
  TSTART = (12.*3600.)
  TEND = TSTART + (3.*24.*3600.)
  DT = 0.25*3600.
  TEMP = 270.
#ENDINLINE
```

3.3.4 Inline type F90_RATES

F90_RATES can be used to add new subroutines to calculate rate coefficients, e.g.:

```
#INLINE F90_RATES
  REAL FUNCTION k_SIV_H2O2(k_298,tdep,cHp,temp)
    ! special rate function for S(IV) + H2O2
    REAL, INTENT(IN) :: k_298, tdep, cHp, temp
    k_SIV_H2O2 = k_298 &
      * EXP(tdep*(1./temp-3.3540E-3)) &
      * cHp / (cHp+0.1)
  END FUNCTION k_SIV_H2O2
#ENDINLINE
```

3.3.5 Inline type F90_RCONST

F90_RCONST can be used to define time-dependent values of rate coefficients that were declared with F90_GLOBAL:

```
#INLINE F90_RCONST
  k_DMS_OH = 1.E-9*EXP(5820./temp)*C(ind_O2)/ &
    (1.E30+5.*EXP(6280./temp)*C(ind_O2))
#ENDINLINE
```

3.3.6 Inline type F90_UTIL

F90_UTIL can be used to define utility subroutines.

3.4 Auxiliary files and the substitution preprocessor

The auxiliary files (listed in Table 4) are templates for integrators, drivers, and utilities. They are inserted into the KPP output after being run through the substitution preprocessor. This preprocessor replaces several placeholders (listed in Table 5) in the template files with their particular values in the model at hand. Usually, only KPP_ROOT and KPP_REAL are needed because the other values can also be obtained via the variables listed in Tab. 8.

KPP_REAL is replaced by the appropriate single or double precision declaration type. Depending on the target language KPP will select the correct declaration type. For

Table 3: Inline types

inline_type	file	placement	usage
F90_DATA	ROOT_Monitor.f90	specification section	(obsolete)
F90_GLOBAL	ROOT_Global.f90	specification section	global variables
F90_INIT	ROOT_Initialize.f90	subroutine Initialize	integration parameters
F90_RATES	ROOT_Rates.f90	executable section	rate law functions
F90_RCONST	ROOT_Rates.f90	subroutine UPDATE_RCONST	USE statements and definitions of rate coefficients
F90_UTIL	ROOT_Util.f90	executable section	utility functions

Table 4: Auxiliary files (for Fortran90)

File	Contents
dFun_dRcoeff.f90	derivatives with respect to reaction rates
dJac_dRcoeff.f90	derivatives with respect to reaction rates
Makefile.f90	unix makefiles
Mex_Fun.f90	mex files
Mex_Jac_SP.f90	mex files
Mex_Hessian.f90	mex files
sutil.f90	Sparse utility functions
tag2num.f90	Function related to equation tags
UpdateSun.f90	Function related to solar zenith angle
UserRateLaws.f90	User-defined rate law functions
util.f90	Input/output utilities

example if one needs to declare an array BIG of size 1000, a declaration like the following must be used:

```
KPP_REAL BIG(1000)
```

When used with the option `#DOUBLE ON`, the above line will be automatically translated into:

```
REAL(dp) BIG(1000)
```

and when used with the option `#DOUBLE OFF`, the same line will become:

```
REAL(sp) BIG(1000)
```

in the resulting Fortran90 output file.

`KPP_ROOT` is replaced by the ROOT file name of the main kinetic description file. In our example where we are processing `small_strato.kpp`, a line in an auxiliary Fortran90 file like

```
USE KPP_ROOT_Monitor
```

will be translated into

```
USE small_strato_Monitor
```

in the generated Fortran90 output file.

4 Output from KPP

4.1 The Fortran90 Code

The code generated by KPP is organized in a set of separate files. Each has a time stamp and a complete description of how it was generated at the begining of the file. The files associated with ROOT are named with a corresponding prefix "ROOT_". The list of files and a short description is shown in Table 6. All subroutines and functions, global parameters, variables, and sparsity data structures are encapsulated in 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`. Fig. 1 shows how these modules are related to each other. A concise list of the main subroutines generated by KPP is shown in Table 7. 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 5: List of symbols replaced by the substitution preprocessor with their particular values for the simulation at hand

Placeholder	Replaced by	Example
KPP_ROOT	the ROOT name	small_strato
KPP_REAL	the real data type	REAL(kind=dp)
KPP_NSPEC	number of species	7
KPP_NVAR	number of variable species	5
KPP_NFIX	number of fixed species	2
KPP_NREACT	number of chemical reactions	10
KPP_NONZERO	number of Jacobian nonzero elements	18
KPP_LU_NONZERO	number of Jacobian nonzero elements, with LU fill-in	19
KPP_NHESS	number of Hessian nonzero elements	10

Table 6: List of model files generated by KPP (for Fortran90). Optional files are only produced under certain circumstances, as specified in the third column.

File	Description	Only if...	see Sect.
ROOT_Main.f90	Driver	#DRIVER $\neq$ none	4.1.1
ROOT_Function.f90	ODE function		4.1.10
ROOT_Global.f90	Global data headers		4.1.9
ROOT_Initialize.f90	Initialization		4.1.3
ROOT_Integrator.f90	Numerical integration		4.1.4
ROOT_LinearAlgebra.f90	Sparse linear algebra		4.1.13
ROOT_Model.f90	Summary of modules		4.1.2
ROOT_Monitor.f90	Equation info		4.1.5
ROOT_Parameters.f90	Model parameters		4.1.8
ROOT_Precision.f90	Parameterized types		4.1.6
ROOT_Rates.f90	User-defined rate laws		4.1.7
ROOT_Util.f90	Utility input-output		4.1.16
ROOT_Jacobian.f90	ODE Jacobian		4.1.11
ROOT_JacobianSP.f90	Jacobian sparsity	#JACOBIAN SPARSE_$*$	4.1.11
ROOT_Hessian.f90	ODE Hessian	#HESSIAN ON	4.1.12
ROOT_HessianSP.f90	Sparse Hessian data	#HESSIAN ON and #JACOBIAN SPARSE_$*$	4.1.12
ROOT_Stochastic.f90	Stochastic functions	#STOCHASTIC ON	4.1.15
ROOT_Stoichiom.f90	Stoichiometric model	#STOICMAT ON	4.1.14
ROOT_StoichiomSP.f90	Stoichiometric matrix	#STOICMAT ON and #JACOBIAN SPARSE_$*$	4.1.14
ROOT_mex_Fun.f90	Matlab interface Fun	#MEX ON	4.1.17
ROOT_mex_Jac_SP.f90	Matlab interface Jac	#MEX ON and #JACOBIAN SPARSE_$*$	4.1.17
ROOT_mex_Hessian.f90	Matlab interface Hess	#MEX ON and #HESSIAN ON	4.1.17
Makefile_ROOT	Makefile		4.1.18
ROOT.map	Human-readable info		4.5

4.1.1 ROOT_Main.f90

ROOT_Main.f90 is the main Fortran90 program. It contains the driver after modifications by the substitution preprocessor. The name of the file is computed by KPP by appending the suffix _Main.f90 to the ROOT name.

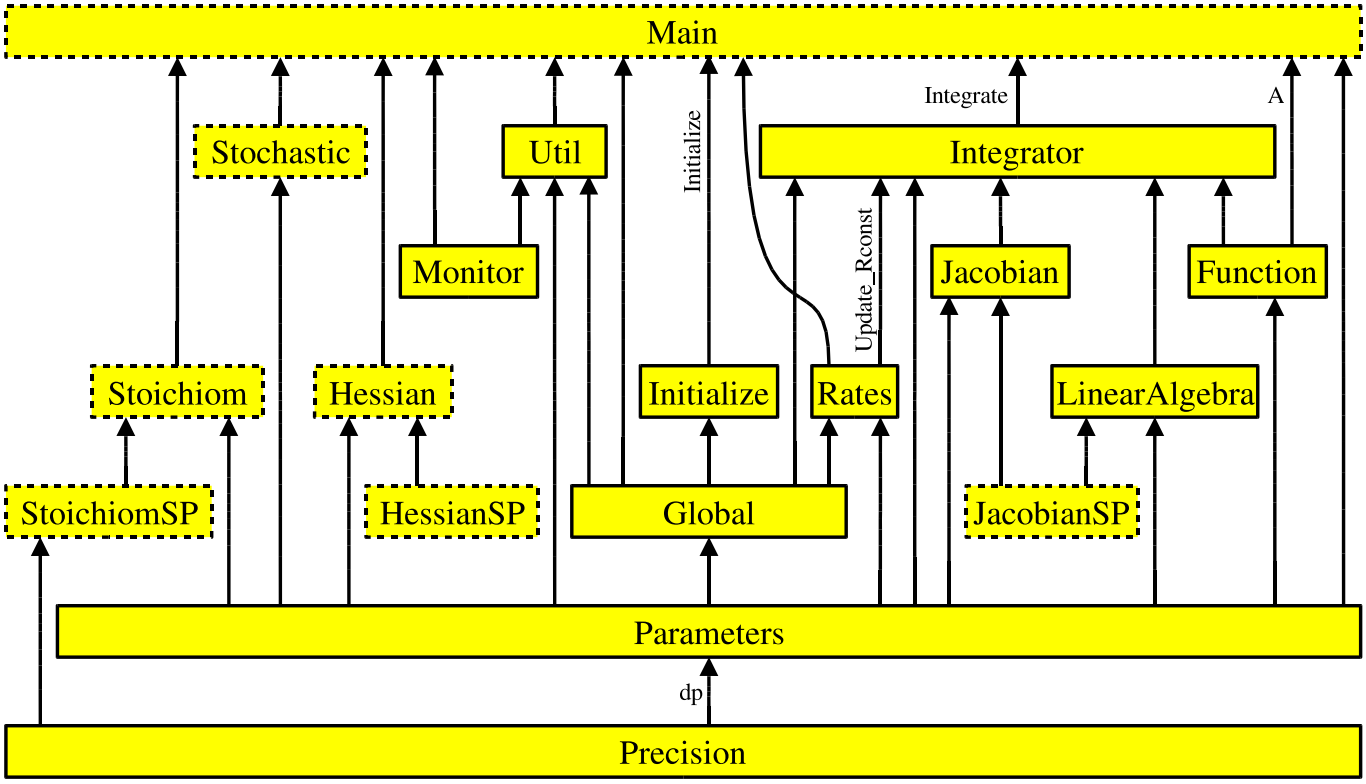

Figure 1: Interdependencies of the KPP-generated files. Each arrow starts at the module that exports a variable or subroutine and points to the module that imports it via the Fortran90 USE instruction. The prefix ROOT_ has been omitted from the module names for better readability. Dotted boxes show optional files that are only produced under certain circumstances, as listed in Tab. 6.

4.1.2 ROOT_Model.f90

The file ROOT_Model.f90 completely defines the model by using all the associated modules.

4.1.3 ROOT_Initialize.f90

The file ROOT_Initialize.f90 contains the subroutine `Initialize` which defines initial values of the chemical species. The driver calls the subroutine `Initialize` once before the time integration loop starts.

4.1.4 ROOT_Integrator.f90

The file ROOT_Integrator.f90 contains the subroutine `INTEGRATE` which is called every time step during the integration. The integrator that was chosen with `#INTEGRATOR` is also included in ROOT_Integrator.f90.

4.1.5 ROOT_Monitor.f90

The file ROOT_Monitor.f90 contains `PARAMETER` arrays with information about the chemical mechanism. The names of all species are included in `SPC_NAMES` and the names of all equations are included in `EQN_NAMES`.

It was shown above (Sect. 3.2.4) that each reaction in the `#EQUATIONS` section may start with an equation tag which is enclosed in angle brackets, e.g.:

```
<J1> NO2 + hv = NO + O : 0.533*SUN;
```

If the equation tags are switched on, KPP also generates the `PARAMETER` array `EQN_TAGS`. In combination with `EQN_NAMES` and the function `tag2num` that converts the equation tag to the KPP-internal equation number, this can be used to describe a reaction:

```
PRINT *,'Reaction J1 is:', &
  EQN_NAMES(tag2num('J1'))
```

4.1.6 ROOT_Precision.f90

Fortran90 code uses parameterized real types. ROOT_Precision.f90 contains the following real kind definitions:

```
! KPP_SP - Single precision kind
  INTEGER, PARAMETER :: &
    SP = SELECTED_REAL_KIND(6,30)
! KPP_DP - Double precision kind
  INTEGER, PARAMETER :: &
    DP = SELECTED_REAL_KIND(12,300)
```

Table 7: List of selected Fortran90 subroutines generated by KPP

Subroutine	Description	File
`Fun`	ODE function	ROOT_Function.f90
`Jac_SP`	ODE Jacobian in sparse format	ROOT_Jacobian.f90
`Jac_SP_Vec`	sparse multiplication	ROOT_Jacobian.f90
`JacTR_SP_Vec`	sparse multiplication	ROOT_Jacobian.f90
`Jac`	ODE Jacobian in full format	ROOT_Jacobian.f90
`Hessian`	ODE Hessian in sparse format	ROOT_Hessian.f90
`Hess_Vec`	Hessian action on vectors	ROOT_Hessian.f90
`HessTR_Vec`	Transposed Hessian action on vectors	ROOT_Hessian.f90
`dFun_dRcoeff`	Derivatives of Fun with respect to rate coefficients	ROOT_Stoichiom.f90
`dJac_dRcoeff`	Derivatives of Jac with respect to rate coefficients	ROOT_Stoichiom.f90
`ReactantProd`	Reactant products	ROOT_Stoichiom.f90
`JacReactantProd`	Jacobian of reactant products	ROOT_Stoichiom.f90
`KppDecomp`	Sparse LU decomposition	ROOT_LinearAlgebra.f90
`KppSolve`	Sparse back substitution	ROOT_LinearAlgebra.f90
`Update_PHOTO`	Update photolysis rate coefficients	ROOT_Rates.f90
`Update_RCONST`	Update all rate coefficients	ROOT_Rates.f90
`Update_SUN`	Update solar intensity	ROOT_Rates.f90
`Initialize`	Set initial values	ROOT_Initialize.f90
`Integrate`	Integrate one time step	ROOT_Integrator.f90
`GetMass`	Check mass balance for selected atoms	ROOT_Util.f90
`Shuffle_kpp2user`	Shuffle concentration vector	ROOT_Util.f90
`Shuffle_user2kpp`	Shuffle concentration vector	ROOT_Util.f90
`InitSaveData`	Utility for `#LOOKAT` command	ROOT_Util.f90
`SaveData`	Utility for `#LOOKAT` command	ROOT_Util.f90
`CloseSaveData`	Utility for `#LOOKAT` command	ROOT_Util.f90
`tag2num`	Calculate reaction number from equation tag	ROOT_Util.f90

Depending on the choice of the `#DOUBLE` command, the real variables are of type double (`REAL(kind=R_8)`) or single precision (`REAL(kind=R_4)`). Changing the parameters of the `SELECTED_REAL_KIND` function in this module will cause a change in the working precision for the whole model.

4.1.7 ROOT_Rates.f90

The code to update the rate constants is in ROOT_Rates.f90. The user defined rate law functions are also placed here.

4.1.8 ROOT_Parameters.f90

The global parameters (Table 8) are defined and initialized in ROOT_Parameters.f90.

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

```
ind_O1D=1, ind_O=2, ind_O3=3,
ind_NO=4, ind_NO2=5
```

and two fixed species (`NFIX`=2)

```
ind_M = 6, ind_O2 = 7.
```

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. Some important simulation parameters generated by KPP are presented in Table 8.

Table 8: List of important simulation parameters and their values for the `small_strato` example

Parameter	Represents	Value
`NSPEC`	No. chemical species	7
`NVAR`	No. variable species	5
`NFIX`	No. fixed species	2
`NREACT`	No. reactions	10
`NONZERO`	No. nonzero entries Jacobian	18
`LU_NONZERO`	As above, after LU factorization	19
`NHESS`	Length, sparse Hessian	10
`NJVRP`	Length, sparse Jacobian `JVRP`	13
`NSTOICM`	Length, stoichiometric matrix	22
`ind_`*spc*	Index of species *spc* in `C()`	
`indf_`*spc*	Index of fixed species *spc* in `FIX()`	

Table 9: List of important global variables

Global variable	Represents
`C(NSPEC)`	Concentrations, all species
`VAR(NVAR)`	Concentrations, variable species
`FIX(NFIX)`	Concentrations, fixed species
`RCONST(NREACT)`	Rate coefficient values
`TIME`	Current integration time
`SUN`	Sun intensity between 0 and 1
`TEMP`	Temperature
`RTOLS`	Relative tolerance (scalar)
`TSTART,TEND`	Simulation start/end time
`DT`	Simulation step
`ATOL(NSPEC)`	Absolute tolerances
`RTOL(NSPEC)`	Relative tolerances
`STEPMIN`	Lower bound for time step
`STEPMAX`	Upper bound for time step
`CFACTOR`	Conversion factor
`SPC_NAMES(NSPEC)`	Names of chemical species
`EQN_NAMES(NREACT)`	Names of chemical equations

4.1.9 ROOT_Global.f90

The global variables (Table 9) are declared in ROOT_Global.f90. Global variables are presented in Table 9.

Both variable and fixed species are stored in the one-dimensional array `C`. The first part (indices from 1 to `NVAR`) contains the variable species, and the second part (indices from `NVAR+1` to `NSPEC`) the fixed species. The total number of species `NSPEC` is the sum of the `NVAR` and `NFIX`. The parts can also be accessed separately through the arrays `VAR` and `FIX`:

```
VAR(1:NVAR) = C(1:NVAR)
FIX(1:NFIX) = C(NVAR+1:NSPEC)
```

4.1.10 ROOT_Function.f90

The code for the ODE function is in ROOT_Function.f90. The chemical reaction mechanism represents a set of ordinary differential equations (ODEs) of dimension `NVAR`. The concentrations of fixed species are parameters in the derivative function. The subroutine `Fun` computes first the vector `A` of reaction rates and then the vector `Vdot` of variable species time derivatives. The input arguments `V`, `F`, and `RCT` are the concentrations of variable species, fixed species, and the rate coefficients, respectively. Below is the Fortran90 code generated by KPP for the ODE function of our `small_strato` example.

```
SUBROUTINE Fun (V, F, RCT, Vdot )
   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
```

4.1.11 ROOT_Jacobian.f90 and ROOT_JacobianSP.f90

The sparse data structures for the Jacobian are declared and initialized in ROOT_JacobianSP.f90. The code for the ODE Jacobian and sparse multiplications is in ROOT_Jacobian.f90. The Jacobian of the ODE function is automatically constructed by KPP. KPP generates the Jacobian subroutine `Jac` or `Jac_SP` where the latter is generated when the sparse format is required. Using the variable species `V`, the fixed species `F`, and the rate coefficients `RCT` as input, the subroutine calculates the Jacobian `JVS`. The default data structures for the sparse compressed on rows Jacobian representation are shown in Table 10 (for the case where the LU fill-in is accounted for). `JVS` stores the `LU_NONZERO` elements of the Jacobian in row order. Each row `I` starts at position `LU_CROW(I)`, and

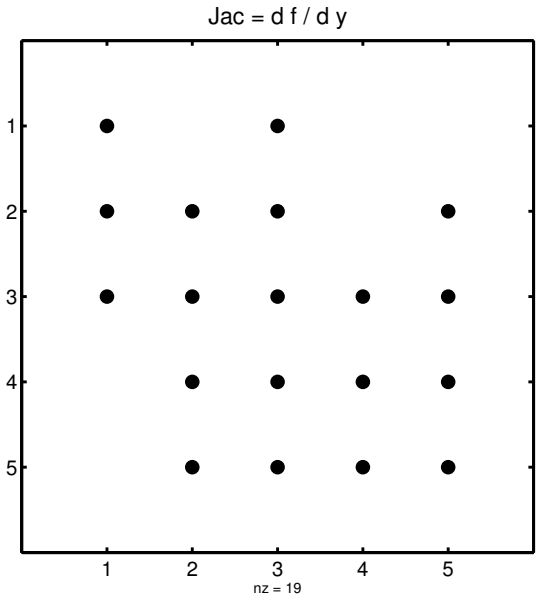

Figure 2: The sparsity pattern of the Jacobian for the `small_strato` example.

Table 10: Sparse Jacobian Data Structures

Global variable	Represents
`JVS(LU_NONZERO)`	Jacobian nonzero elements
`LU_IROW(LU_NONZERO)`	Row indices
`LU_ICOL(LU_NONZERO)`	Column indices
`LU_CROW(NVAR+1)`	Start of rows
`LU_DIAG(NVAR+1)`	Diagonal entries

`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)`. For the `small_strato` example KPP generates the following Jacobian sparse data structure:

```
LU_ICOL = (/ 1,3,1,2,3,5,1,2,3,4, &
             5,2,3,4,5,2,3,4,5 /)
LU_IROW = (/ 1,1,2,2,2,2,3,3,3,3, &
             3,4,4,4,4,5,5,5,5 /)
LU_CROW = (/ 1,3,7,12,16,20 /)
LU_DIAG = (/ 1,4,9,14,19,20 /)
```

This is visualized in Fig. 2. The sparsity coordinate vectors are computed by KPP and initialized statically. These vectors are constant as the sparsity pattern of the Jacobian does not change during the computation.

Two other KPP-generated routines, `Jac_SP_Vec` and `JacTR_SP_Vec` are useful for direct and adjoint sensitivity analysis. They perform sparse multiplication of `JVS` (or its transpose for `JacTR_SP_Vec`) with the user-supplied vector `UV` without any indirect addressing.

Table 11: Sparse Hessian Data

Variable	Represents
`HESS(NHESS)`	Hessian nonzero elements $\mathtt{H}_{i,j,k}$
`IHESS_I(NHESS)`	Index i of element $\mathtt{H}_{i,j,k}$
`IHESS_J(NHESS)`	Index j of element $\mathtt{H}_{i,j,k}$
`IHESS_K(NHESS)`	Index k of element $\mathtt{H}_{i,j,k}$

4.1.12 ROOT_Hessian.f90 and ROOT_HessianSP.f90

The sparse data structures for the Hessian are declared and initialized in ROOT_HessianSP.f90. The Hessian function and associated sparse multiplications are in ROOT_Hessian.f90. The Hessian contains the second order derivatives of the time derivative functions. More exactly, the Hessian is a 3-tensor such that

$$H_{i,j,k} = \frac{\partial^2(\mathrm{d}c/\mathrm{d}t)_i}{\partial c_j\, \partial c_k} \ , \qquad 1 \le i,j,k \le N_{\mathrm{var}} \ . \tag{11}$$

KPP generates the routine `Hessian`. Using the variable species `V`, the fixed species `F`, and the rate coefficients `RCT` as input, the subroutine calculates the Hessian. The Hessian is a very sparse tensor. The sparsity of the Hessian for our `small_strato` example is visualized in Fig. 3. KPP computes the number of nonzero Hessian entries and saves it 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 as illustrated in Table 11. Since the time derivative function is smooth, these Hessian matrices are symmetric, $\mathtt{HESS}_{i,j,k}$=$\mathtt{HESS}_{i,k,j}$. KPP stores only those entries $\mathtt{HESS}_{i,j,k}$ with $j \le k$. The sparsity coordinate vectors `IHESS_I`, `IHESS_J`, and `IHESS_K` are computed by KPP and initialized statically. They are constant as the sparsity pattern of the Hessian does not change during the computation.

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

4.1.13 ROOT_LinearAlgebra.f90

Sparse linear algebra routines are in the file ROOT_LinearAlgebra.f90. 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

$$P\,x = (I - h\gamma J)\,x = b \tag{12}$$

where the matrix $P = I - h\gamma J$ is refered to as the "prediction matrix". I the identity matrix, h the integration

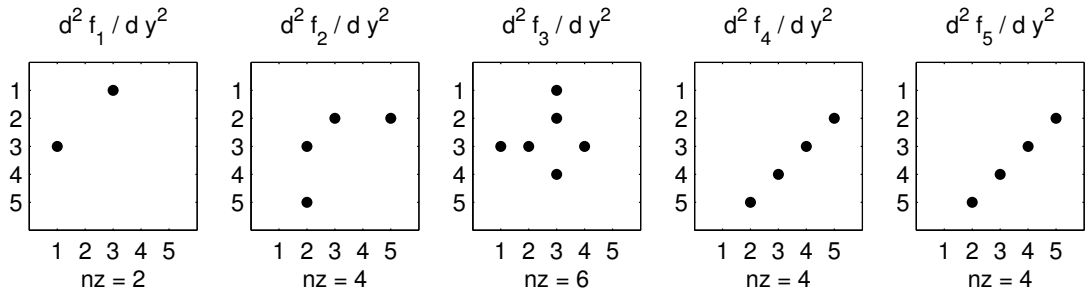

Figure 3: The Hessian of the `small_strato` example.

time step, γ a scalar parameter depending on the method, and J the system Jacobian. The vector b is the system right hand side and the solution 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 P is given by the sparsity structure of the Jacobian, and is produced by KPP (with account for the fill-in) as discussed above.

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

The subroutines `KppSolve` and `KppSolveTR` use the in-place LU factorization P as computed by `KppDecomp` and perform sparse backward and forward substitutions (using P or its transpose). The sparse linear algebra routines `KppDecomp` and `KppSolve` are extremely efficient, as shown by (Sandu et al., 1996).

4.1.14 ROOT_Stoichiom.f90 and ROOT_StoichiomSP.f90

These files contain a description of the chemical mechanism in stoichiometric form. The file ROOT_Stoichiom.f90 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 ROOT_StoichiomSP.f90.

The stoichiometric matrix is constant sparse. For our example the matrix has `NSTOICM=22` nonzero entries out of 50 entries. KPP produces the stoichiometric matrix in sparse, column-compressed format, as shown in Table 12. Elements are stored in columnwise order in the one-dimensional vector of values `STOICM`. Their row and column indices are stored in `IROW_STOICM` and `ICOL_STOICM` respectively. The vector `CCOL_STOICM` contains pointers to the start of each column. For example column `j` starts in the sparse vector at position `CCOL_STOICM(j)` and ends at `CCOL_STOICM(j+1)-1`. The last value `CCOL_STOICM(NVAR+1)=NSTOICM+1` simplifies the handling of sparse data structures.

Table 12: Sparse Stoichiometric Matrix

Global variable	Represents
`STOICM(NSTOICM)`	Stoichiometric matrix
`IROW_STOICM(NSTOICM)`	Row indices
`ICOL_STOICM(NSTOICM)`	Column indices
`CCOL_STOICM(NREACT+1)`	Start of columns

Table 13: Sparse Data for Jacobian of Reactant Products

Global variable	Represents
`JVRP(NJVRP)`	Nonzero elements of JVRP
`ICOL_JVRP(NJVRP)`	Column indices in JVRP
`IROW_JVRP(NJVRP)`	Row indices in JVRP
`CROW_JVRP(NREACT+1)`	Start of rows in JVRP

The subroutine `ReactantProd` computes the reactant products `ARP` for each reaction, and the subroutine `JacReactantProd` computes the Jacobian of reactant products vector, i.e.:

$$\texttt{JVRP} = \partial \texttt{ARP} / \partial \texttt{V} \tag{13}$$

The matrix `JVRP` is sparse and is computed and stored in row compressed sparse format, as shown in Table 13. The parameter `NJVRP` holds the number of nonzero elements. For our example:

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

If `#STOICMAT` is set to `ON`, the stoichiometric formulation allows a direct computation of the derivatives with respect to rate coefficients.

The subroutine dFun_dRcoeff computes the partial derivative DFDR of the ODE function with respect to a subset of NCOEFF reaction coefficients, whose indices are specifies in the array JCOEFF

$$\texttt{DFDR} = \partial \texttt{Vdot} / \partial \texttt{RCT}(\texttt{JCOEFF}) \tag{14}$$

Similarly one can obtain the partial derivative of the Jacobian with respect to a subset of the rate coefficients. More exactly, KPP generates the subroutine dJac_dRcoeff which calculates DJDR, the product of this partial derivative with a user-supplied vector:

$$\texttt{DJDR} = [\partial \texttt{JVS} / \partial \texttt{RCT}(\texttt{JCOEFF})] \times \texttt{U} \tag{15}$$

4.1.15 ROOT_Stochastic.f90

If the generation of stochastic functions is switched on, KPP produces the file ROOT_Stochastic.f90 with the following functions:

Propensity calculates the propensity vector. The propensity function uses the number of molecules of variable (NmlcV) and fixed (NmlcF) species, as well as the stochastic rate coefficients (SCT) to calculate the vector of propensity rates (Propensity). The propensity $\texttt{Prop}_j$ defines the probability that the next reaction in the system is the j^{th} reaction.

StochasticRates converts deterministic rates to stochastic. The stochastic rate coefficients (SCT) are obtained through a scaling of the deterministic rate coefficients (RCT). The scaling depends on the Volume of the reaction container and on the number of molecules which react.

MoleculeChange calculates changes in the number of molecules. When the reaction with index IRCT takes place, the number of molecules of species involved in that reaction changes. The total number of molecules NmlcV is updated by the function.

These functions are used by the Gillespie numerical integrators (direct stochastic simulation algorithm). These integrators are provided in both Fortran90 and C implementations (the template file name is gillespie). Drivers for stochastic simulations are also implemented (the template file name is general_stochastic).

4.1.16 ROOT_Util.f90

The utility and input/output functions are in ROOT_Util.f90. In addition to the chemical system description routines discussed above, KPP generates several utility routines, some of which are summarized in Table 7.

The subroutines InitSaveData, SaveData, and CloseSaveData can be used to print the concentration of the species that were selected with #LOOKAT to the file ROOT.dat.

4.1.17 ROOT_mex_Fun.f90, ROOT_mex_Jac_SP.f90, and ROOT_mex_Hessian.f90

Mex is a Matlab extension. KPP generates the mex routines for the ODE function, Jacobian, and 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 smaller, especially for large kinetic systems.

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. However, the mex files built using Fortran90 may require further platform-specific tuning of the mex compiler options.

4.1.18 The Makefile

KPP produces a Makefile that allows for an easy compilation of all KPP-generated source files. The file name is Makefile_ROOT. The Makefile assumes that the selected driver contains the main program. However, if no driver was selected (i.e. #DRIVER none), it is necessary to add the name of the main program file manually to the Makefile.

4.2 The C Code

The driver file ROOT.c contains the main (driver) and numerical integrator functions, as well as declarations and initializations of global variables. The generated C code includes three header files which are #include-d in other files as appropriate. The global parameters (Table 8) are #define-d in the header file ROOT_Parameters.h. The global variables (Table 9) 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 (Table 10), Hessian (Table 11), stoichiometric matrix (Table 12), and the Jacobian of reaction products (Table 13). The actual declarations of each data structures is done in the corresponding files.

The code for the ODE function (Sect. 4.1.10) is in ROOT_Function.c. The code for the ODE Jacobian and sparse multiplications (Sect. 4.1.11) is in ROOT_Jacobian.c, and the declaration and initialization of the Jacobian sparse data structures (Table 10) is in the file ROOT_JacobianSP.c. Similarly, the Hessian function and associated sparse multiplications (Section 4.1.12)

are in ROOT_Hessian.c, and the declaration and initialization of Hessian sparse data structures (Table 11) in ROOT_HessianSP.c.

The file ROOT_Stoichiom.c contains the functions for reactant products and its Jacobian, and derivatives with respect to rate coefficients (Sect. 4.1.14). The declaration and initialization of the stoichiometric matrix and the associated sparse data structures (Tables 12 and 13) is done in ROOT_StoichiomSP.c.

Sparse linear algebra routines (Sect. 4.1.13) are in the file ROOT_LinearAlgebra.c. The code to update the rate constants and user defined code for rate laws is in ROOT_Rates.c.

Various utility and input/output functions (Sect. 4.1.16) are in ROOT_Util.c and ROOT_Monitor.c.

Finally, mex gateway routines that allow the C implementation of the ODE function, Jacobian, and Hessian to be called directly from Matlab (Sect. 4.1.17) are also generated (in the files ROOT_mex_Fun.c, ROOT_mex_Jac_SP.c, and ROOT_mex_Hessian.c).

4.3 The Fortran77 Code

The general layout of the Fortran77 code is similar to the layout of the C code. The driver file ROOT.f contains the main (driver) and numerical integrator functions.

The generated Fortran77 code includes three header files. The global parameters (Table 8) are defined as parameters and initialized in the header file ROOT_Parameters.h. The global variables (Table 9) 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 (Table 10), Hessian (Table 11), stoichiometric matrix (Table 12), and the Jacobian of reaction products (Table 13). These sparse data structures are initialized in four named block data statements: JACOBIAN_SPARSE_DATA (in ROOT_HessianSP.f), HESSIAN_SPARSE_DATA (in ROOT_HessianSP.f), JVRP_SPARSE_DATA and STOICM_MATRIX (in ROOT_StoichiomSP.f).

The code for the ODE function (Sect. 4.1.10) is in ROOT_Function.f. The code for the ODE Jacobian and sparse multiplications (Sect. 4.1.11) is in ROOT_Jacobian.f. The Hessian function and associated sparse multiplications (Sect. 4.1.12) are in ROOT_Hessian.f.

The file ROOT_Stoichiom.f contains the functions for reactant products and its Jacobian, and derivatives with respect to rate coefficients (Sect. 4.1.14). The declaration and initialization of the stoichiometric matrix and the associated sparse data structures (Tables 12 and 13) is done in the STOICM_MATRIX block data statement.

Sparse linear algebra routines (Sect. 4.1.13) are in the file ROOT_LinearAlgebra.f. The code to update the rate constants is in ROOT_Rates.f, and the utility and input/output functions (Sect. 4.1.16) are in ROOT_Util.f and ROOT_Monitor.f.

Matlab-mex gateway routines for the ODE function, Jacobian, and Hessian are discussed in Sect. 4.1.17.

4.4 The Matlab Code

Matlab (http://www.mathworks.com/products/matlab/) provides a high-level programming environment that allows algorithm development, numerical computations, and data analysis and visualization. The KPP-generated Matlab code allows for a 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 Sections 4.1.10, 4.1.11, 4.1.12, 4.1.13, 4.1.14, and 4.1.16. The names of the m-function-files are the same as the names of the functions (prefixed by the model name ROOT).

The Matlab syntax for calling each function is

```
[Vdot] = Fun    (V, F, RCT);
[JVS ] = Jac_SP (V, F, RCT);
[HESS] = Hessian(V, F, RCT);
```

The global parameters (Table 8) are defined as Matlab global variables and initialized in the file ROOT_parameter_defs.m. The variables of Table 9 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 (Table 10), the Hessian (Table 11), the stoichiometric matrix (Table 12), and the Jacobian of reaction products (Table 13) 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.

4.5 The map file

The map file ROOT.map contains a summary of all the functions, subroutines and data structures defined in the code file, plus a summary of the numbering and category of the species involved.

This file contains supplementary information for the user. Several statistics are listed here, like the total number equations, the total number of species, the number of variable and fixed species. Each species from the chemical mechanism is then listed followed by its type and numbering.

Furthermore it contains the complete list of all the functions generated in the target source file. For each function, a brief description of the computation performed is attached containing also the meaning of the input and output parameters.

5 KPP Internal Structure

This chapter is mainly concerned with describing the internal architecture of the KPP preprocessor. It describes the basic modules and their functionalities, and all the preprocessing analysis performed on the input files. KPP can be very easily configured to suit a broad class of users.

5.1 KPP directory structure

The KPP distribution will unfold a directory $KPP_HOME with the following subdirectories:

- **src/** Contains the KPP source code files, as listed in Table 15.
- **bin/** Contains the KPP executable. The path to this directory needs to be added to the environment PATH variable.
- **util/** Contains different function templates useful for the simulation. Each template file has a suffix that matches the appropriate target language (.f90, .f, .c, or .m). KPP will run the template files through the substitution preprocessor. The user can define their own auxiliary functions by inserting them into the files.

Table 15: Source code files

File	Description
kpp.c	main program
code.c	generic code generation functions
code.h	header file
code_c.c	generation of C code
code_f77.c	generation of Fortran77 code
code_f90.c	generation of Fortran90 code
code_matlab.c	generation of matlab code
debug.c	debugging output
gdata.h	header file
gdef.h	header file
gen.c	generic code generation functions
lex.yy.c	flex/yacc-generated file
scan.h	input for flex/yacc
scan.l	input for flex/yacc
scan.y	input for flex/yacc
scanner.c	evaluate parsed input
scanutil.c	evaluate parsed input
y.tab.c	flex/yacc-generated file
y.tab.h	flex/yacc-generated header file

- **models/** Contains the description of the chemical models. Users can define their own models by placing the model description files in this directory. The KPP distribution contains several models from atmospheric chemistry which can be used as templates for model definitions.
- **drv/** Contains driver templates for chemical simulations. Each driver has a suffix that matches the appropriate target language (.f90, .f, .c, or .m). KPP will run the appropriate driver through the substitution preprocessor. The driver template general provided with the distribution works with any example. Users can define here their own driver templates.
- **int/** Contains numerical time stepping (integrator) routines. The command "#INTEGRATOR *integrator*" will force KPP to look into this directory for a definition file *integrator*.def. This file selects the numerical routine (with the #INTFILE command) and sets the function type, the Jacobian sparsity type, the target language, etc. Each integrator template is found in a file that ends with the appropriate suffix (.f90, .f, .c, or .m). The selected template is processed by the substitution preprocessor. Users can define here their own numerical integration routines.
- **examples/** Contains several model description examples (.kpp files) which can be used as templates for building simulations with KPP.

Table 14: List of Matlab model files

File	Description
ROOT.m	driver
ROOT_parameter_defs.m	Global parameters
ROOT_global_defs.m	Global variables
ROOT_monitor_defs.m	Global monitor variables
ROOT_sparse_defs.m	Global sparsity data
ROOT_Fun_Chem.m	Template for ODE function
ROOT_Fun.m	ODE function
ROOT_Jac_Chem.m	Template for ODE Jacobian
ROOT_Jac_SP.m	ODE Jacobian in sparse format
ROOT_JacobianSP.m	Sparsity data structures
ROOT_Hessian.m	ODE Hessian in sparse format
ROOT_HessianSP.m	Sparsity data structures
ROOT_HessTR_Vec.m	Hessian action on vectors
ROOT_Hess_Vec.m	Transposed Hessian action on vectors
ROOT_stoichiom.m	Derivatives of Fun and Jac with respect to rate coefficients
ROOT_StoichiomSP.m	Sparse data
ROOT_ReactantProd.m	Reactant products
ROOT_JacReactantProd.m	Jacobian of reactant products
ROOT_rates.m	User-defined reaction rate laws
ROOT_Update_PHOTO.m	Update photolysis rate coefficients
ROOT_Update_RCONST.m	Update all rate coefficients
ROOT_Update_SUN.m	Update solar intensity
ROOT_GetMass.m	Check mass balance for selected atoms
ROOT_Initialize.m	Set initial values
ROOT_Shuffle_kpp2user.m	Shuffle concentration vector
ROOT_Shuffle_user2kpp.m	Shuffle concentration vector

- **site-lisp/** Contains the file `kpp.el` which provides a KPP mode for emacs with color highlighting.

5.2 KPP environment variables

In order for KPP to find its components, it has to know the path to the location where the KPP distribution is installed. This is achieved by requiring the `$KPP_HOME` environment variable to be set to the path where KPP is installed.

The PATH variable should be updated to contain the `$KPP_HOME/bin` directory.

There are several optional environment variable that control the places where KPP looks for module files, integrators, and drivers. They are all summarized in Table 16.

5.3 KPP internal modules

5.3.1 Scanner and Parser

This module is responsible for reading the kinetic description files and extracting the information necessary in the code generation phase. We make use of the flex and yacc generic tools in implementing our own scanner and parser. Using these tools this module gathers information from the input files and fills in the following data structures in memory:

- The atom list
- The species list
- The left hand side matrix of coefficients
- The right hand side matrix of coefficients
- The equation rates

Table 16: Environment variables used by KPP

Variable	Description	Default assumed
`$KPP_HOME`	Required, stores the absolute path to the KPP distribution	no default
`$KPP_MODEL`	Optional, specifies additional places were KPP will look for model files before searching the default	`$KPP_HOME/models`
`$KPP_INT`	Optional, specifies additional places were KPP will look for integrator files before searching the default.	`$KPP_HOME/int`
`$KPP_DRV`	Optional, specifies additional places were KPP will look for driver files before searching the default	`$KPP_HOME/drv`

- The option list

Error checking is performed at each step in the scanner and the parser. For each syntax error the exact line and input file, along with an appropriate error message are produced. In most of the cases the exact cause of the error can be identified, therefore the error messages are very precise. Some other errors like mass balance, and equation duplicates, are tested at the end of this phase.

5.3.2 Species reordering

When parsing the input files, the species list is updated as soon as a new species is encountered in a chemical equation. Therefore the ordering of the species is the order in which they appear in the equation description section. This is not a useful order for subsequent operations. The species have to be first sorted such that all variable species and all fixed species are put together. Then if a sparsity structure of the Jacobian is required, it might be better to reorder the species in such a way that the factorization of the Jacobian will preserve the sparsity. This reordering is done using a Markovitz type of algorithm.

5.3.3 Expression trees computation

This is the core of the preprocessor. This module has to generate the production/destruction functions the Jacobian and all the data structure nedeed by these functions. This module has to build a language independent structure of each function and statement in the target source file. Instead of using an intermediate format for this as some other compilers do, KPP generates the intermediate format for just one statement at a time. The vast majority of the statements in the target source file are assignments. The expression tree for each assignment is incrementally build by scanning the coefficient matrices and the rate constant vector. At the end these expression trees are simplified. Similar approaches are applied to function declaration and prototypes, data declaration and initialization.

5.3.4 Code generation

There are basically two modules, each dealing with the syntax particularities of the target language. For example, the C module includes a function that generates a valid C assignment when given an expression tree. Similarly there are functions for data declaration, initializations, comments, function prototypes, etc. Each of these functions produce the code into an output buffer. A language specific routine reads from this buffer and splits the statements into lines to improve readability of the generated code.

6 Numerical methods

The KPP numerical library contains a set of numerical integrators selected to be very efficient in the low to medium accuracy regime (relative errors $\sim 10^{-2} \dots 10^{-5}$). In addition, the KPP numerical integrators preserve the linear invariants (i.e., mass) of the chemical system.

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 each of them KPP implements the tangent linear model (direct decoupled sensitivity) and the adjoint models. The implementations distinguish between sensitivities with respect to initial values and sensitivities with respect to parameters for efficiency.

Note that KPP produces the building blocks for the simulation and also for the sensitivity calculations. It also provides application programming templates. Some minimal programming may be required from the users in order to construct their own application from the KPP building blocks.

In order to offer more control over the integrator, the KPP-generated subroutine `INTEGRATE` provides the optional input parameters `ICNTRL_U` and `RCNTRL_U`. Each of them is an array of 20 elements that allow the fine-tuning of the integrator, e.g. by setting a particular integrator method, tolerances, minimum and maximum step sizes,

Table 17: Symbols used in the decription of the numerical methods implemented in KPP

Symbol	Description
s	Number of stages
t^n	Discrete time moment
h	Time step $h = t^{n+1} - t^n$
y^n	Numerical solution (concentration) at t^n
δy^n	tangent linear solution at t^n
λ^n	Adjoint numerical solution at t^n
$f(\cdot,\cdot)$	The ODE derivative function: $y' = f(t,y)$
$f_t(\cdot,\cdot)$	Partial time derivative $f_t(t,y) = \partial f(t,y)/\partial t$
$J(\cdot,\cdot)$	The Jacobian $J(t,y) = \partial f(t,y)/\partial y$
$J_t(\cdot,\cdot)$	Partial time derivative of Jacobian $J_t(t,y) = \partial J(t,y)/\partial t$
A	The system matrix
$H(\cdot,\cdot)$	The Hessian $H(t,y) = \partial^2 f(t,y)/\partial y^2$
T_i	Internal stage time moment for Runge Kutta and Rosenbrock methods
Y_i	Internal stage solution for Runge Kutta and Rosenbrock methods
k_i, ℓ_i, u_i, v_i	Internal stage vectors for Runge Kutta and Rosenbrock methods, their tangent linear and adjoint models
α_i, α_{ij}, a_{ij}, b_i, c_i, c_{ij}, e_i, m_i	Method coefficients

and more. The exact meaning of the elements depends on the integrator and may change in the future. Please read the comment lines in the individual integrator files `$KPP_HOME/int/*.f90`.

Similarly, to obtain more information about the integration, the subroutine `INTEGRATE` provides the optional output parameters `ISTATUS_U` and `RSTATUS_U`. They are both arrays of 20 elements and contain the length of the last time step, the number of accepted and rejected steps, the number of miscellaneous function calls, and more. Again, for the exact meaning, the reader is refered to the comment lines in the individual integrator files `$KPP_HOME/int/*.f90`.

In the following sections we introduce the numerical methods implemented in KPP. The symbols used in the formulas are explained in Table 17.

6.1 Rosenbrock Methods

An s-stage Rosenbrock method (Hairer and Wanner, 1991, Section IV.7) computes the next-step solution by the formulas

$$
\begin{aligned}
y^{n+1} &= y^n + \sum_{i=1}^{s} m_i k_i \ , \quad \mathrm{Err}^{n+1} = \sum_{i=1}^{s} e_i k_i \qquad (16)\\
T_i &= t^n + \alpha_i h \ , \quad Y_i = y^n + \sum_{j=1}^{i-1} a_{ij} k_j \ ,\\
A &= \left[\frac{1}{h\gamma} - J^T(t^n, y^n) \right]\\
A \cdot k_i &= f\left(T_i, Y_i\right) + \sum_{j=1}^{i-1} \frac{c_{ij}}{h} k_j + h\gamma_i f_t\left(t^n, y^n\right) \ .
\end{aligned}
$$

where s is the number of stages, $\alpha_i = \sum_j \alpha_{ij}$ and $\gamma_i = \sum_j \gamma_{ij}$. The formula coefficients (a_{ij} and γ_{ij}) give the order of consistency and the stability properties. A is the system matrix (in the linear systems to be solved during implicit integration, or in the Newton's method used to solve the nonlinear systems). It is the scaled identity matrix minus the Jacobian.

The coefficients of the methods implemented in KPP are shown in Table 18.

6.1.1 Tangent Linear Model

The method (16) is combined with the sensitivity equations. One step of the method reads

$$
\begin{aligned}
\delta y^{n+1} &= \delta y^n + \sum_{i=1}^{s} m_i \ell_i \qquad (17)\\
T_i &= t^n + \alpha_i h \ , \quad \delta Y_i = \delta y^n + \sum_{j=1}^{i-1} a_{ij} \ell_j\\
A \cdot \ell_i &= J\left(T_i, Y_i\right) \cdot \delta Y_i + \sum_{j=1}^{i-1} \frac{c_{ij}}{h} \ell_j\\
&\quad + \left(H(t^n, y^n) \times k_i\right) \cdot \delta y^n + h\gamma_i J_t\left(t^n, y^n\right) \cdot \delta y^n
\end{aligned}
$$

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

Table 18: Rosenbrock methods implemented in KPP

Method name	Stages (s)	Function calls	Order	Stability properties	Method coefficients
Ros–2	2	2	2(1)	L-stable	$\gamma = 1 + 1/\sqrt{2}$, $a_{2,1} = 1/\gamma$, $c_{2,1} = -2/\gamma$, $m_1 = 3/(2\gamma)$, $m_2 = 1/(2\gamma)$, $e_1 = 1/(2\gamma)$, $e_2 = 1/(2\gamma)$, $\alpha_1 = 0$, $\alpha_2 = 1$, $\gamma_1 = \gamma$, $\gamma_2 = -\gamma$
Ros–3	3	2	3(2)	L-stable	$a_{2,1} = 1$, $a_{3,1} = 1$, $a_{3,2} = 0$, $c_{2,1} = -1.015$, $c_{3,1} = 4.075$, $c_{3,2} = 9.207$, $m_1 = 1$, $m_2 = 6.169$, $m_3 = -0.427$, $e_1 = 0.5$, $e_2 = -2.908$, $e_3 = 0.223$, $\alpha_1 = 0$, $\alpha_2 = 0.436$, $\alpha_3 = 0.436$, $\gamma_1 = 0.436$, $\gamma_2 = 0.243$, $\gamma_3 = 2.185$
Ros–4	4	3	4(3)	L-stable	$a_{2,1} = 2$, $a_{3,1} = 1.868$, $a_{3,2} = 0.234$, $a_{4,1} = a_{3,1}$, $a_{4,2} = a_{3,2}$, $a_{4,3} = 0$, $c_{2,1} = -7.137$, $c_{3,1} = 2.581$, $c_{3,2} = 0.652$, $c_{4,1} = -2.137$, $c_{4,2} = -0.321$, $c_{4,3} = -0.695$, $m_1 = 2.256$, $m_2 = 0.287$, $m_3 = 0.435$, $m_4 = 1.094$, $e_1 = -0.282$, $e_2 = -0.073$, $e_3 = -0.108$, $e_4 = -1.093$, $\alpha_1 = 0$, $\alpha_2 = 1.146$, $\alpha_3 = 0.655$, $\alpha_4 = \alpha_3$, $\gamma_1 = 0.573$, $\gamma_2 = -1.769$, $\gamma_3 = 0.759$, $\gamma_4 = -0.104$
Rodas–3	4	3	3(2)	Stiffly accurate	$a_{2,1} = 0$, $a_{3,1} = 2$, $a_{3,2} = 0$, $a_{4,1} = 2$, $a_{4,2} = 0$, $a_{4,3} = 1$, $c_{2,1} = 4$, $c_{3,1} = 1$, $c_{3,2} = -1$, $c_{4,1} = 1$, $c_{4,2} = -1$, $c_{4,3} = -8/3$, $m_1 = 2$, $m_2 = 0$, $m_3 = 1$, $m_4 = 1$, $e_1 = 0$, $e_2 = 0$, $e_3 = 0$, $e_4 = 1$, $\alpha_1 = 0$, $\alpha_2 = 0$, $\alpha_3 = 1$, $\alpha_4 = 1$, $\gamma_1 = 0.5$, $\gamma_2 = 1.5$, $\gamma_3 = 0$, $\gamma_4 = 0$
Rodas–4	6	5	4(3)	Stiffly accurate	$\alpha_1 = 0$, $\alpha_2 = 0.386$, $\alpha_3 = 0.210$, $\alpha_4 = 0.630$, $\alpha_5 = 1$, $\alpha_6 = 1$, $\gamma_1 = 0.25$, $\gamma_2 = -0.104$, $\gamma_3 = 0.104$, $\gamma_4 = -0.036$, $\gamma_5 = 0$, $\gamma_6 = 0$, $a_{2,1} = 1.544$, $a_{3,1} = 0.946$, $a_{3,2} = 0.255$, $a_{4,1} = 3.314$, $a_{4,2} = 2.896$, $a_{4,3} = 0.998$, $a_{5,1} = 1.221$, $a_{5,2} = 6.019$, $a_{5,3} = 12.537$, $a_{5,4} = -0.687$, $a_{6,1} = a_{5,1}$, $a_{6,2} = a_{5,2}$, $a_{6,3} = a_{5,3}$, $a_{6,4} = a_{5,4}$, $a_{6,5} = 1$, $c_{2,1} = -5.668$, $c_{3,1} = -2.430$, $c_{3,2} = -0.206$, $c_{4,1} = -0.107$, $c_{4,2} = -9.594$, $c_{4,3} = -20.47$, $c_{5,1} = 7.496$, $c_{5,2} = -0.124$, $c_{5,3} = -34$, $c_{5,4} = 11.708$, $c_{6,1} = 8.083$, $c_{6,2} = -7.981$, $c_{6,3} = -31.521$, $c_{6,4} = 16.319$, $c_{6,5} = -6.058$, $m_1 = a_{5,1}$, $m_2 = a_{5,2}$, $m_3 = a_{5,3}$, $m_4 = a_{5,4}$, $m_5 = 1$, $m_6 = 1$, $e_1 = 0$, $e_2 = 0$, $e_3 = 0$, $e_4 = 0$, $e_5 = 0$, $e_6 = 1$

KPP contains tangent linear models (for direct decoupled sensitivity analysis) for each of the Rosenbrock methods (Ros–1, Ros–2, Ros–3, Ros–4, Rodas–3, and Rodas–4). The implementations distinguish between sensitivities with respect to initial values and sensitivities with respect to parameters for efficiency.

6.1.2 The Discrete Adjoint

To obtain the adjoint we first differentiate the method with respect to y_n. Here J denotes the Jacobian and H the Hessian of the derivative function f. The discrete adjoint of the (non-autonomous) Rosenbrock method is

$$\begin{aligned} A \cdot u_i &= m_i \lambda^{n+1} + \sum_{j=i+1}^{s} \left(a_{ji} v_j + \frac{c_{ji}}{h} u_j \right) , \quad (18) \\ v_i &= J^T(T_i, Y_i) \cdot u_i , \quad i = s, s-1, \cdots, 1 , \end{aligned}$$

$$\begin{aligned} \lambda^n &= \lambda^{n+1} + \sum_{i=1}^{s} \left(H(t^n, y^n) \times k_i \right)^T \cdot u_i \\ &\quad + h J_t^T(t^n, y^n) \cdot \sum_{i=1}^{s} \gamma_i u_i + \sum_{i=1}^{s} v_i \end{aligned}$$

KPP contains adjoint models (for direct decoupled sensitivity analysis) for each of the Rosenbrock methods (Ros–1, Ros–2, Ros–3, Ros–4, Rodas–3, and Rodas–4).

6.2 Runge-Kutta methods

A general s-stage Runge-Kutta method is defined as (Hairer et al., 1993, Section II.1)

$$\begin{aligned} y^{n+1} &= y^n + h \sum_{i=1}^{s} b_i k_i , \quad (19) \\ T_i &= t^n + c_i h , \quad Y_i = y^n + h \sum_{j=1}^{s} a_{ij} k_j , \end{aligned}$$

Table 19: Runge-Kutta methods implemented in KPP

Method	File(s)	Description
Radau5	`atm_radau5.f`, `kpp_radau5.f90`	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). While Radau5 is relatively expensive (when compared to the Rosenbrock methods), it is more robust and is useful to obtain accurate reference solutions.
SDIRK4	`kpp_sdirk.f`, `kpp_sdirk.f90`	The implementation is based on the implementation of Hairer and Wanner (1991). SDIRK4 is an L-stable, singly-diagonally-implicit Runge Kutta method of order 4.
SEULEX	`kpp_seulex.f`, `kpp_seulex.f90`	SEULEX is a variable order stiff extrapolation code able to produce highly accurate solutions. The KPP implementation is based on the implementation of Hairer and Wanner (1991).

$$k_i \;=\; f\,(\,T_i,\,Y_i\,)\;,$$

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

KPP numerical library implements a Radau5, a Runge Kutta method of order 5 based on Radau-IIA quadrature (Hairer and Wanner, 1991, Section IV.10). This numerical method is stiffly accurate. The KPP implementation follows the original implementation of Hairer and Wanner (1991). While Radau5 is relatively expensive (when compared to the Rosenbrock methods), it is more robust and is useful to obtain highly accurate reference solutions.

The Runge-Kutta methods implemented in KPP are summarized in Table 19.

6.2.1 Tangent Linear Model

The tangent linear method associated with the Runge Kutta method is

$$\begin{aligned}
\delta y^{n+1} &= \delta y^n + h\sum_{i=1}^{s} b_i \ell_i \;, \qquad (20)\\
\delta Y_i &= \delta y^n + h\sum_{j=1}^{s} a_{ij}\ell_j \;,\\
\ell_i &= J\,(T_i,\,Y_i)\cdot \delta Y_i \;.
\end{aligned}$$

The system (20) is linear and does not require an iterative procedure. However, even for a SDIRK method ($a_{ij} = 0$ for $i > j$ and $a_{ii} = \gamma$) each stage requires the LU factorization of a different matrix.

No Runge Kutta tangent linear model is currently implemented in KPP.

6.2.2 Discrete Adjoint Model

The first order Runge-Kutta adjoint is

$$\begin{aligned}
u_i &= h\,J^T(T_i,Y_i)\cdot\left(b_i\lambda^{n+1} + \sum_{j=1}^{s} a_{ji}u_j\right) \qquad (21)\\
\lambda^n &= \lambda^{n+1} + \sum_{j=1}^{s} u_j \;.
\end{aligned}$$

For $b_i \neq 0$ the Runge-Kutta adjoint can be rewritten as another Runge-Kutta method:

$$\begin{aligned}
u_i &= h\,J^T(T_i,Y_i)\cdot\left(\lambda^{n+1} + \sum_{j=1}^{s} \frac{b_j\,a_{ji}}{b_i}u_j\right) \qquad (22)\\
\lambda^n &= \lambda^{n+1} + \sum_{j=1}^{s} b_j\,u_j \;.
\end{aligned}$$

No Runge Kutta adjoint model is currently implemented in KPP.

6.3 Backward Differentiation Formulas

Backward differentiation formulas (BDF) are linear multistep methods with excellent stability properties for the integration of chemical systems (Hairer and Wanner, 1991, Section V.1). The k-step BDF method reads

$$\sum_{i=0}^{k} \alpha_i y^{n-i} = h_n \beta\, f\,(t^n, y^n) \qquad (23)$$

where the coefficients α_i and β are chosen such that the method has order of consistency k.

The KPP library contains two off-the-shelf, highly popular implementations of BDF methods, described in Table 20.

Table 20: BDF methods implemented in KPP

Method	File(s)	Description
LSODE	`kpp_lsode.f90`	LSODE, the Livermore ODE solver (Radhakrishnan and Hindmarsh, 1993), implements backward differentiation formula (BDF) methods for stiff problems. LSODE has been translated to Fortran90 for the incorporation into the KPP library.
LSODES	`atm_lsodes.f`	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	`kpp_dvode.f`	VODE (Brown et al., 1989) uses another formulation of backward differentiation formulas. The version of VODE present in the KPP library uses directly the KPP sparse linear algebra routines.
ODESSA	`atm_odessa.f`	The BDF-based direct-decoupled sensitivity integrator Odessa (Leis and Kramer, 1986) has been modified to use the KPP sparse linear algebra routines.

7 Differences between KPP-2.1 and Previous Versions

7.1 New features of KPP-2.1

This user manual describes recently added features of KPP as well as those which have been available for a longer period. Here we give an overview about the recent changes:

- Fortran90 output has been available since the preliminary version "1.1-f90-alpha12" provided in Sander et al. (2005).
- Matlab is a new target language (see Sect. 4.4).
- The set of integrators has been extended with a general Rosenbrock integrator, and the corresponding tangent linear and adjoint methods.
- The KPP-generated Fortran90 code has a different file structure than the C or Fortran77 output (see Sects. 4.2 and 4.3).
- An automatically generated Makefile facilitates the compilation of the KPP-generated code (see Sect. 4.1.18).
- Equation tags provide a convenient way to refer to specific chemical reactions (see Sect. 4.1.5).
- The dummy index allows to test if a certain species occurs in the current chemistry mechanism. (see Sect. 3.2.3).
- Lines starting with "`//`" are comment lines.

7.2 Upgrading KPP input files from previous versions to KPP-2.1

KPP users who want to upgrade from previous versions to KPP-2.1 need to make a few modifications to their input files.

- To select the target language, change the previous command name "`#USE`" to "`#LANGUAGE`".
- To access global variables, change "`USE gdata`" to "`USE ROOT_global`".
- Rename all inline types "`*_DECL`" and "`*_DATA`" to "`*_GLOBAL`".

If you have already used the Fortran90 output of the preliminary version "1.1-f90-alpha12" from Sander et al. (2005), these changes are also necessary:

- Change "`#USE Fortran95`" to "`#LANGUAGE Fortran90`".
- Change the names of the indices of the species from "`kpp_*`" to "`ind_*`".
- Rename all inline types from "`F95_*`" to "`F90_*`".
- Since the name of the initialization subroutine has changed, replace

```
USE ROOT_Init, ONLY: initval
CALL initval
```

by

```
USE ROOT_Initialize, ONLY: initialize
CALL initialize
```

8 Acknowledgements

Parts of this user manual are based on the thesis of Damian-Iordache (1996).

References

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

Damian-Iordache, V.: KPP – Chemistry simulation development environment, Master's thesis, University of Iowa, USA, 1996.

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

Hairer, E., Norsett, S., and Wanner, G.: Solving Ordinary Differential Equations I. Nonstiff Problems., Springer-Verlag, Berlin, 1993.

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

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.

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.

A BNF Description of the KPP Language

Following is the BNF-like specification of the language:

program ::= *module* | *module program*

module ::= *section* | *command* | *inline_code*

section ::=
`#ATOMS` *atom_definition_list* |
`#CHECK` *atom_list* |
`#DEFFIX` *species_definition_list* |
`#DEFVAR` *species_definition_list* |
`#EQUATIONS` *equation_list* |
`#INITVALUES` *initvalues_list* |
`#LOOKAT` *species_list atom_list* |
`#LUMP` *lump_list* |
`#MONITOR` *species_list atom_list* |
`#SETFIX` *species_list_plus* |
`#SETVAR` *species_list_plus* |
`#TRANSPORT` *species_list*

command ::=
`#CHECKALL` |
`#DOUBLE [ ON | OFF ]` |
`#DRIVER` *driver_name* |
`#DUMMYINDEX [ ON | OFF ]` |
`#EQNTAGS [ ON | OFF ]` |
`#FUNCTION [ AGGREGATE | SPLIT ]` |
`#HESSIAN [ ON | OFF ]` |
`#INCLUDE` *file_name* |
`#INTEGRATOR` *integrator_name* |
`#INTFILE` *integrator_name* |
`#JACOBIAN [ OFF | FULL | SPARSE_LU_ROW | SPARSE_ROW ]` |
`#LANGUAGE [ Fortran90 | Fortran77 | C | Matlab ]` |
`#LOOKATALL` |
`#MEX [ ON | OFF ]` |
`#MODEL` *model_name* |
`#REORDER [ ON | OFF ]` |
`#STOCHASTIC [ ON | OFF ]` |
`#STOICMAT [ ON | OFF ]` |
`#TRANSPORTALL`

inline_code ::=
`#INLINE` *inline_type*
inline_code
`#ENDINLINE`

atom_count ::=	*integer atom_name* |
	atom_name
atom_definition_list ::=	*atom_definition* |
	atom_definition atom_definition_list
atom_list ::=	*atom_name*; |
	atom_name; *atom_list*
equation ::=	*<equation_tag> expression = expression : rate*; |
	expression = expression : rate;
equation_list ::=	*equation* |
	equation equation_list
expression ::=	*term* |
	term + expression |
	term − expression
initvalues_assignment ::=	*species_name_plus = program_expression*; |
	`CFACTOR` = *program_expression*;
initvalues_list ::=	*initvalues_assignment* |
	initvalues_assignment initvalues_list
inline_type ::=	`F90_RATES` | `F90_RCONST` | `F90_GLOBAL` | `F90_INIT` | `F90_DATA` | `F90_UTIL` |
	`F77_RATES` | `F77_RCONST` | `F77_GLOBAL` | `F77_INIT` | `F77_DATA` | `F77_UTIL` |
	`C_RATES` | `C_RCONST` | `C_GLOBAL` | `C_INIT` | `C_DATA` | `C_UTIL` |
	`MATLAB_RATES` | `MATLAB_RCONST` | `MATLAB_GLOBAL` |
	`MATLAB_INIT` | `MATLAB_DATA` | `MATLAB_UTIL`
lump ::=	*lump_sum : species_name*;
lump_list ::=	*lump* |
	lump lump_list
lump_sum ::=	*species_name* |
	species_name + lump_sum
rate ::=	*number* |
	program_expression
species_composition ::=	*atom_count* |
	atom_count + species_composition |
	`IGNORE`
species_definition ::=	*species_name = species_composition*;
species_definition_list ::=	*species_definition* |
	species_definition species_definition_list
species_list ::=	*species_name*; |
	species_name; *species_list*
species_list_plus ::=	*species_name_plus*; |
	species_name_plus; *species_list_plus*
species_name_plus ::=	*species_name* |
	`VAR_SPEC` |
	`FIX_SPEC` |
	`ALL_SPEC`
term ::=	*number species_name* |
	species_name |
	`hv`