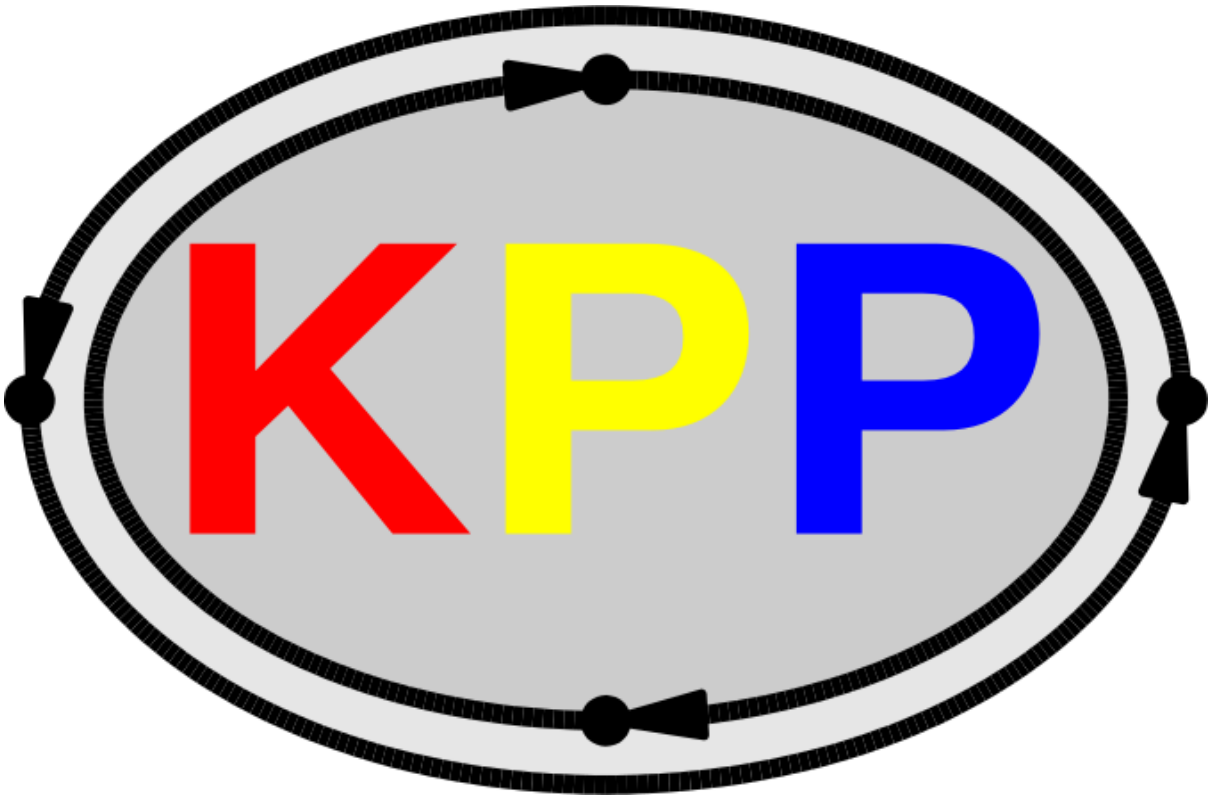

KPP: The Kinetic PreProcessor

Release 3.0.0

A. Sandu, R. Sander, M. Long, H. Lin, and R. Yantosca

Nov 09, 2022

Getting Started

This site provides instructions for **KPP**, the Kinetic PreProcessor.

Contributions (e.g., suggestions, edits, revisions) would be greatly appreciated. See *Editing this User Guide* and our contributing guidelines. If you find something hard to understand—let us know!

1 KPP revision history

Only the major new features are listed here. For a detailed description of the changes, read CHANGELOG.md[1].

1.1 KPP 3.0.0

Attention: When you are upgrading from an older KPP version to KPP 3.0.0 or later versions, a few minor changes in your code may be necessary:

- The `atoms` file is now called `atoms.kpp`. Thus, you have to change `#INCLUDE atoms` to `#INCLUDE atoms.kpp` in your KPP input file.
- The utility functions `ARR`, `ARR2`, `k_3rd` and `k_arr` have been replaced by the new set of the consistent functions `ARR_abc`, `ARR_ab`, `ARR_ac`, `k3rd_jpl`, `k3rd_jpl_activation`, and `k3rd_iupac`. We recommend to upgrade to the new functions, which all use the temperature from the `temp` variable in `ROOT_Global.f90`. Alternatively, it is possible to copy the old functions into a separate file and make them available via *F90_RCONST*.

[1] https://github.com/KineticPreProcessor/KPP/blob/main/CHANGELOG.md

- If you have been using `ICNTRL(5)` for maximal order in the `lsode` integrator, you now have to use `ICNTRL(10)` instead. The index 5 in the `ICNTRL` array is now used consistently for the maximum number of Newton iterations in all integrators.

- Updated the search for the **`flex`** library in `src/Makefile.defs`. The build process will look for the **`flex:`** library file (either `libfl.so` or `libfl.a` file in several standard locations first. If not found, the build process will look in the path specfiied by environment variable `KPP_FLEX_LIB_DIR`.
- Added content to ReadTheDocs pages and fixed several formatting issues.
- Fixed various minor issues in generating C-language code.
- Fixed various minor issues in generating Matlab-language code.
- C-I tests folders have been renamed for clarity. Also refactored the scripts used to submit C-I tests. Updated the Dockerfile to always request Ubuntu 20.04 and an AMD64 platform, so that the same libraries will always be used when running C-I tests on Azure DevOps.
- Fortran type `DOUBLE_COMPLEX` is now replaced by `COMPLEX(kind=dp)`.
- Fixed incorrect license metadata in `.zenodo.json`, which is used to auto-generate a DOI with each KPP release on Github.
- Added extra `free()` statements in `src/gen.c` to avoid memory leaks.
- `Fun()` no longer uses Vdotout since it can be retrieved from `Vdot`.
- Fixed a bug in `int/feuler.f90`, where the wrong argument was being passed to routine `Fun`.

1.2 KPP 2.6.0

- Added the **rosenbrock_autoreduce** integrator Lin *et al.* [[Lin et al., 2022]].

1.3 KPP 2.5.0

- Merged updates from the GEOS-Chem development stream (versions *KPP 2.2.4_gc*, *KPP 2.2.5_gc*, *KPP 2.3.0_gc*, *KPP 2.3.1_gc*, *KPP 2.3.2_gc*) into the mainline KPP development stream. Previously hardwired code has been removed and replaced with code selectable via KPP commands.
- Added a new forward-Euler method integrator (`feuler.f90`).
- Added KPP commands **`#MINVERSION`** and **`#UPPERCASEF90`** (along with corresponding continuous integration tests).
- Added optional variables `Aout` and `Vdotout` to subroutine Fun().
- Replaced Fortran `EQUIVALENCE` statements with thread-safe pointer assignments (Fortran90 only).

- Converted the KPP user manual to Sphinx/ReadTheDocs format (this now replaces the prior ReadTheDocs documentaton).
- Added updates to allow KPP to be built on MacOS X systems.
- Added **`small_strato`** C-I test that uses the exact same options as is described in *Running KPP with an example stratospheric mechanism*.

1.4 KPP 2.4.0

- Added new integrators: `beuler.f90`, `rosenbrock_mz.f90`, `rosenbrock_posdef.f90`, `rosenbrock_posdef_h211b_qssa.f90`.
- Several memory sizes (`MAX_EQN`, ...) have been increased to allow large chemical mechanisms.
- Added new Makefile target: `list`.
- Added LaTeX User Manual.
- Now use `ICNTRL(15)` to decide whether or not to toggle calling the `Update_SUN`, `Update_RCONST`, and `Update_PHOTO` routines from within the integrator.

1.5 KPP 2.3.2_gc

NOTE: Contains KPP Modifications specific to GEOS-Chem.

- Added workaround for F90 derived-type objects in inlined code (i.e. properly parse `State_Het%xArea`, etc).
- Updated Github issue templates.
- `MAX_INLINE` (max # of inlined code lines to read) has been increased to 200000.
- Commented out the `Update_Sun()` functions in `update_sun.F90`, `update_sun.F`. (NOTE: These have been restored in *KPP 2.5.0*).
- Default rate law functions are no longer written to `gckpp_Rates.F90`. (NOTE: These have been restored in *KPP 2.5.0*).

1.6 KPP 2.3.1_gc

NOTE: KPP modifications specific to GEOS-Chem.

ALSO NOTE: ReadTheDocs documentation has been updated in *KPP 2.5.0* to remove GEOS-Chem specific information.

- Added documentation for ReadTheDocs.
- Added Github issue templates.
- README.md now contains the ReadTheDocs badge.

- README.md now points to kpp.readthedocs.io for documentation.

1.7 KPP 2.3.0_gc

NOTE: Contains KPP modifications specific to GEOS-Chem.

- Added `README.md` for the GC_updates branch.
- Added MIT license for the GC_updates branch.
- Add `Aout` argument to return reaction rates from `SUBROUTINE Fun`.
- Rename `KPP/kpp_2.2.3_01` directory to `KPP/kpp-code`.
- Now write `gckpp_Model.F90` and `gckpp_Precision.F90` from `gen.c`.
- Do not write file creation & time to KPP-generated files (as this will cause Git to interpret each file as a new file to be added).
- Now create Fortran-90 source code files with `*.F90` instead of `*.f90`. (NOTE: In *KPP 2.5.0*, this can specified with the *#UPPERCASEF90* command.)
- Remove calls to UPDATE_SUN and UPDATE_RCONST from all `*.f90` integrators. (NOTE: This has been restored in *KPP 2.5.0*.)

1.8 KPP 2.2.5_gc

NOTE: Contains KPP modifications specific to GEOS-Chem.

- Increase `MAX_INLINE` from 20000 to 50000

1.9 KPP 2.2.4_gc

NOTE: Contains KPP modifications specific to GEOS-Chem.

- Add MIT license files for GC_updates branch and update `README.md` accordingly
- Create `README.md` for main branch
- Set `FLEX_LIB_DIR` using `FLEX_HOME` env variable if it is defined.
- Added an exponential integrator.
- Added array to `*_Monitor` for family names (`FAM_NAMES`) string vector.
- Added functionality for Prod/Loss families using *#FAMILIES* token.
- Add scripts necessary to build a new mechanism for GEOS-Chem
- Completed the prod/loss option (token: `#FLUX [on/off]`)
- Added `OMP THREADPRIVATE` to LinearAlgebra.F90
- Added `rosenbrock_split.def` integrator definition

- Added `OMPThreadPrivate` function for F77.
- Added declaration of `A` in *ROOT_Function*
- Added `OMP THREADPRIVATE` Functionality to F90 output.
- Completed the split-form Function for F90.
- Increase maximum number of equations.
- Increase `MAX_FAMILIES` parameter from 50 to 300
- Extend equation length limit to 200 characters.
- Also changed the species name for a family to the family name itself.
- Modified Families to minimize the number of additional species created
- Renamed and change indexing convention
- Removed unnecessary files

1.10 KPP 2.2.3

- A new function called `k_3rd_iupac` is available, calculating third-order rate coefficients using the formula used by IUPAC [[Atkinson et al., 2004]].
- While previous versions of KPP were using **`yacc`** (yet another compiler compiler), the current version has been modified to be compatible with the parser generator **`bison`**, which is the successor of **`yacc`**.
- New Runge-Kutta integrators were added: `kpp_dvode.f90`, `runge_kutta.f90`, `runge_kutta_tlm.f90`, `sdirk_adj.f90`, `sdirk_tlm.f90`.
- New Rosebrock method `Rang3` was added.
- The new KPP command **`#DECLARE`** was added (see: *#DECLARE*).
- Several vector and array functions from **`BLAS`** (`WCOPY`, `WAXPY`, etc.) were replaced by Fortran90 expressions.

1.11 KPP 2.1

- Described by Sandu and Sander [[Sandu and Sander 2006]].
- Matlab is a new target language (see: *The Matlab code*).
- 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: *The Fortran90 code*).
- An automatically generated Makefile facilitates the compilation of the KPP-generated code (see: *The Makefile*).

- Equation tags provide a convenient way to refer to specific chemical reactions (see: *#LOOKAT and #MONITOR*.
- The dummy index allows to test if a certain species occurs in the current chemistry mechanism. (see: *#DUMMYINDEX*)
- Lines starting with `//` are comment lines.

1.12 KPP 1.1-f90-alpha12

- First KPP version with Fortran90 [[Sander et al., 2005]].

2 Installation

This section can be skipped if KPP is already installed on your system.

2.1 Download KPP from Github

Clone the KPP source code from the KPP Github repository[2]:

```
$ cd $HOME
$ git clone https://github.com/KineticPreProcessor/KPP.git
```

This will create a directory named KPP in your home directory.

2.2 Define the KPP_HOME environment variable

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 you are using the Unix C-shell (aka **csh**), add add these statements to your `$HOME/.cshrc` file:

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

and then apply the settings with:

```
$ source $HOME/.cshrc
```

If, on the other hand, you are using the Unix **bash** shell, add these statements to your `$HOME/.bashrc` file:

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

[2] https://github.com/KineticPreProcessor/KPP

and then apply the settings with:

```
$ source $HOME/.bashrc
```

Now if you type:

```
$ echo $PATH
```

You should see the `$KPP_HOME/bin` directory placed at the end of the `PATH` variable.

2.3 Test if KPP dependencies are installed on your system

KPP depends on several other Unix packages. Before using KPP for the first time, test if these are installed on your system. If any of these packages are missing, you can install them with your system's package manager (e.g. **`apt`** for Ubuntu, **`yum`** for Fedora, **`homebrew`** for MacOS, etc.), or with Spack[3].

gcc

Important: You might have to follow some *additional configuration and installation steps* regarding **`gcc`** on MacOS X systems.

KPP uses the GNU Compiler Collection[4] (aka **`gcc`**) by default. A version of **`gcc`** comes pre-installed with most Linux or MacOS systems. To test if **`gcc`** is installed on your system, type:

```
$ gcc --version
```

This will display the version information, such as:

```
gcc (GCC) 11.2.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. ␣
↪There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR␣
↪PURPOSE.
```

[3] https://spack.readthedocs.io
[4] https://gcc.gnu.org/

sed

The **sed** utility is used to search for and replace text in files. To test if **sed** has been installed, type:

```
$ which sed
```

This will print the path to **sed** on your system.

bison

The **bison** utility parses the chemical mechanism file into a computer-readable syntax. To test **bison** is installed, type:

```
$ which bison
```

This will print the path to **bison** on your system.

flex

Important: You might have to follow some *additional configuration and installation steps* regarding **flex** on MacOS X systems.

The **flex** (the Fast Lexical Analyzer) creates a scanner that can recognize the syntax generated by **bison**. To test if **flex** is installed, type:

```
$ which flex
```

This will print the path to **flex** on your system.

You will also need to specify the path to the **flex** library files (`libfl.so` or `libfl.a`) in order to *build the KPP executable*. This can be done with the **find** command:

```
$ find /usr/ -name "*libfl*" -print
```

This will generate a list of file paths such as shown below. Look for the text `libfl.`:

```
/usr/include/libflashrom.h
/usr/lib/gthumb/extensions/libflicker.so
/usr/lib/gthumb/extensions/libflicker_utils.so
/usr/lib/libflashrom.so.1.0.0
/usr/lib/libfl.so               # <---- This is the flex library␣
↪file
# ... etc ...
```

Once you have located the directory where flex library file resides (which in this example is `/usr/lib`), use it to define the `KPP_FLEX_LIB_DIR` environment variable in your `.bashrc` (or `.bash_aliases` file if you have one):

```
export KPP_FLEX_LIB_DIR=/usr/lib
```

And then apply the changes with:

```
. ~/.bashrc
```

KPP will use the path specified by `KPP_FLEX_LIB_DIR` during the compilation sequence (described in the next section).

2.4 Build the KPP executable

Change to the KPP/src directory:

```
$ cd $KPP_HOME/src
```

To clean a previously-built KPP installation, delete the KPP object files and all the examples with:

```
$ make clean
```

To delete a previoulsy-built KPP executable as well, type:

```
$ make distclean
```

KPP will use **`gcc`** as the default compiler. If you would like to use a different compiler (e.g. **`icc`**), then edit `src/Makefile.defs` to add your compiler name.

Create the KPP executable with:

```
$ make
```

You should see output similar to:

```
gcc -g -Wall -Wno-unused-function -I/usr/include -c code.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c code_c.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c code_f77.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c code_f90.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c code_matlab.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c debug.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c gen.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c kpp.c
flex -olex.yy.c scan.l
bison -d -o y.tab.c scan.y
gcc -g -Wall -Wno-unused-function -I/usr/include -c lex.yy.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c scanner.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c scanutil.c
gcc -g -Wall -Wno-unused-function -I/usr/include -c y.tab.c
gcc -g -Wall -Wno-unused-function code.o code_c.o
    code_f77.o code_f90.o code_matlab.o debug.o gen.o kpp.o
    lex.yy.o scanner.o scanutil.o y.tab.o -L/usr/lib -lfl -o kpp
```

This will create the executable file `$KPP_HOME/bin/kpp`.

2.5 Instructions for MacOS X users

When installing KPP on a MacOS X system, some additional configuration and installation steps may be necessary.

Force MacOS to recognize the gcc compiler

On MacOS X, if you type:

```
$ gcc --version
```

you will probably see output similar to:

```
Apple clang version 13.1.6 (clang-1316.0.21.2.5)
Target: x86_64-apple-darwin21.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

This is because MacOS X installs **`clang`** as **`gcc`**. To force MacOS X to recognize the **`gcc`** compiler, follow these steps:

1. Use the **`homebrew`** package manager to install **`gcc`**:

   ```
   $ brew install gcc
   ```

2. Type this command:

   ```
   $ ls /usr/local/Cellar/gcc/*/bin/ | grep gcc
   ```

   You should see output such as:

   ```
   gcc-11*
   gcc-ar-11*
   gcc-nm-11*
   gcc-ranlib-11*
   # ... etc ...
   ```

   This output indicates **`gcc`** major version 11 has been installed, and that the gcc executable is called `gcc-11`. (Your version may differ.)

3. Add the following code block to your `.bashrc` file (or to your `.bash_aliases` file if you have one). This will define aliases that will override **`clang`** with **`gcc`**.

   ```
   ↪#==============================================================================
   # Compiler settings (MacOS)
   #
   ```

(continues on next page)

(continued from previous page)

```
# NOTE: MacOSX installs Clang as /usr/bin/gcc, so we have to␣
↪manually
# force reference to gcc-11, g++-11, and gfortran-11, which␣
↪HomeBrew
# installs to /usr/local/bin.  (bmy, 10/28/21)

↪#=============================================================================
alias gcc=gcc-11
alias g++=g++-11
alias gfortran=gfortran-11
export CC=gcc
export CXX=g++-11
export FC=gfortran-11
export F77=gfortran-11
```

Then apply the changes with:

```
$ . ~/.bashrc
```

4. To check if your shell now recognizes the **gcc** compiler, type:

```
$ gcc --version
```

You should see output similar to:

```
gcc-11 (Homebrew GCC 11.3.0_1) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. ␣
↪There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A␣
↪PARTICULAR PURPOSE.
```

This now indicates that your compiler is **gcc** and not **clang**.

Install flex with homebrew

If your MacOS X computer does not have the **flex** library installed, then you can install it with **homebrew**:

```
$ brew install flex
```

Unlike Linux pacakge managers, which would install the **flex** library files in the path `/usr/lib/`, **homebrew** will install it to a path such as `/usr/local/Cellar/flex/X.Y.Z/lib/`.

To find the version of **flex** that has been installed by **homebrew**, type:

```
$ ls /usr/local/Cellar/flex
```

and you will get a listing such as:

```
2.6.4_2
```

This indicates that the version of **flex** on your system is `2.6.4_2` (the `_2` denotes the number of bug-fix updates since version `2.6.4` was released).

The **flex** library files (`libfl.so` or `libfl.a`) will be found in `lib/` subfolder. In this example, the path will be:

```
/usr/local/Cellar/flex/2.6.4_2/lib
```

Knowing this, you can now define the `KPP_FLEX_LIB_DIR` environment variable *as described above*:

```
export FLEX_LIB_DIR=/usr/local/Cellar/flex/2.6.4_2/lib
```

Request maximum stack memory

MacOS X has a hard limit of 65332 bytes for stack memory. This is much less memory than what is available on GNU/Linux operating systems such as Ubuntu, Fedora, etc.

To make sure you are using the maximum amount of stack memory on MacOS X add this command to your `.bashrc` file:

```
ulimit -s 65532
```

and then apply the change with:

```
$ . ~/.bashrc
```

This stack memory limit means that KPP will not be able to parse mechanisms with more than about 2000 equations and 1000 species. Because of this, we have added an `#ifdef` block to KPP header file `src/gdata.h` to define the `MAX_EQN` and `MAX_SPECIES` parameters accordingly:

```
#ifdef MACOS
#define MAX_EQN         2000     // Max number of equations (MacOS␣
↪only)
#define MAX_SPECIES     1000     // Max number of species   (MacOS␣
↪only)
#else
#define MAX_EQN        11000     // Max number of equations
#define MAX_SPECIES     6000     // Max number of species
#endif
```

If you find that KPP will not parse your mechanism, you can increase `MAX_EQN` and decrease `MAX_SPECIES` (or vice-versa) as needed, and then *rebuild the KPP executable*.

Know that MacOS X is case-insenstive

If you have two files with identical names except for case (e.g. `integrator.F90` and `integrator.f90`) then MacOS X will not be able to tell them apart. Because of this, you may encounter an error if you try to commit such files into Git, etc.

3 Running KPP with an example stratospheric mechanism

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

$$
\begin{aligned}
O_2 + h\nu &\longrightarrow 2O && (R1)\\
O + O_2 &\longrightarrow O_3 && (R2)\\
O_3 + h\nu &\longrightarrow O + O_2 && (R3)\\
O + O_3 &\longrightarrow 2O_2 && (R4)\\
O_3 + h\nu &\longrightarrow O(^1D) + O_2 && (R5)\\
O(^1D) + M &\longrightarrow O + M && (R6)\\
O(^1D) + O_3 &\longrightarrow 2O_2 && (R7)\\
NO + O_3 &\longrightarrow NO_2 + O_2 && (R8)\\
NO_2 + O &\longrightarrow NO + O_2 && (R9)\\
NO_2 + h\nu &\longrightarrow NO + O && (R10)
\end{aligned}
$$

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 and Matlab languages are discussed in the *#LANGUAGE* section.

Important: Most of the recent KPP developments described in this manual have been added into the Fortran90 language. We look to members of the KPP user community to spearhead development in C, Matlab, and other languages.

The KPP input files (with suffix `.kpp`) specify the *target model*, the *target language*, the *integrator* the *driver program*. etc. The file name (without the `.kpp`) serves as the root name for the simulation. Here 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.

The sections below outline the steps necessary to build and run a "box-model" simulation with an example mechanism.

3.1 1. Create a directory for the example

Create a directory in which to build and run the example mechanism:

```
$ cd $HOME
$ mkdir small_strato_example
$ cd small_strato_example
```

In the following sections we will refer to `$HOME/small_strato_example` as "the example directory".

3.2 2. Create a KPP Definition File

Create a KPP definition file in the example directory. The name of this file will always be `ROOT.kpp`, where `ROOT` is the name of the chemical mechanism.

For this example, write the following lines into a file named `small_strato.kpp` in the example directory:

```
#MODEL      small_strato
#LANGUAGE   Fortran90
#INTEGRATOR rosenbrock
#DRIVER     general
```

Important: KPP will look for the relevant files (e.g. mechanism definition, driver, etc.) in the proper subdirectories of `KPP_HOME`. Therefore you won't need to copy these manually to the example directory.

We will now look at the *KPP commands* in `small_strato.kpp`.

#MODEL small_strato

The *#MODEL* command selects a specific kinetic mechanism (in this example, **`small_strato`**). KPP will look in the path `$KPP_HOME/models/` for the *model definition file* `small_strato.def` which contains the following code in the *KPP language*:

```
#include small_strato.spc       { Includes file w/ species␣
↪definitons     }
#include small_strato.eqn       { Includes file w/ chemical␣
↪equations     }

#LOOKATALL                      { Output all species to small_
↪strato.dat}
#MONITOR O3;N;O2;O;NO;O1D;NO2;   { Print selected species to screen ␣
↪      }
```

(continues on next page)

(continued from previous page)

```
#CHECK O; N;                        { Check Mass Balance of oxygen &␣
↪nitrogen }

#INITVALUES                         { Set initial values of species    ␣
↪      }
  CFACTOR = 1.     ;               { and set units conversion factor␣
↪to 1    }
  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 ;

{ Fortran code to be inlined into ROOT_Global }
#INLINE F90_INIT
  TSTART = (12*3600)
  TEND = TSTART + (3*24*3600)
  DT = 0.25*3600
  TEMP = 270
#ENDINLINE

{ Matlab code to be inlined into ROOT_Global }
#INLINE MATLAB_INIT
  global TSTART TEND DT TEMP
  TSTART = (12*3600);
  TEND = TSTART + (3*24*3600);
  DT = 0.25*3600;
  TEMP = 270;
#ENDINLINE

{ C code to be inlined into ROOT_GLOBAL }
#INLINE C_INIT
  TSTART = (12*3600);
  TEND = TSTART + (3*24*3600);
  DT = 0.25*3600;
  TEMP = 270;
#ENDINLINE
```

The *definition file* `small_strato.def` uses the *#INCLUDE* command to include the *species file* and the *equation file*. It also specifies parameters for running a "box-model" simulation, such as *species initial values*, start time, stop, time, and timestep (cf. *Inlined Code*).

The *species file* `small_strato.spc` lists all the species in the model. Some of them are variable, meaning that their concentrations change according to the law of mass action kinetics. Others are fixed, with the concentrations determined by physical and not chemical factors (cf. *#DEFVAR and #DEFFIX*). For each species its atomic composition is given (unless the user chooses to ignore it).

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

The species file also includes the *atoms file* (`atoms.kpp`), which defines the chemical elements in the *#ATOMS* section.

The *equation file* `small_strato.eqn` contains the description of the equations in an *#EQUATIONS* section. The chemical kinetic mechanism is specified in the *KPP language*. Each reaction is described as "the sum of reactants equals the sum of products" and, after a colon, is followed by its rate coefficient. `SUN` is the normalized sunlight intensity, equal to one at noon and zero at night. Equation tags, e.g. `<R1>`, are optional.

```
#EQUATIONS { Small Stratospheric Mechanism }


<R1>  O2   + hv = 2O           : (2.643E-10) * SUN*SUN*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*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;
```

#LANGUAGE Fortran90

The *#LANGUAGE* command selects the language for the KPP-generated solver code. In this example we are using Fortran90.

#INTEGRATOR rosenbrock

The *#INTEGRATOR* command selects a numerical integration routine from the templates provided in the `$KPP_HOME/int` directory, or implemented by the user.

In this example, the *Rosenbrock integrator* and the Fortran90 language have been been specified. Therefore, the file `$KPP_HOME/int/rosenbrock.f90` will be used.

#DRIVER general

The *#DRIVER* command selects a specific main program (located in the `$KPP_HOME/drv` directory):

1. `general_adj.f90` : Used with integrators that use the discrete adjoint method
2. `general_tlm.f90` : Used with integrators that use the tangent-linear method
3. `general.f90` : Used with all other integrators.

In this example, the `rosenbrock.f90` integrator does not use either adjoint or tangent-linear methods, so the `$KPP_HOME/drv/general.f90` will be used.

3.3 3. Build the mechanism with KPP

Now that all the necessary files have been copied to the example directory, the **`small_strato`** mechanism can be built. Type:

```
$ kpp small_strato.kpp
```

You should see output similar to:

```
This is KPP-3.0.0.

KPP is parsing the equation file.
KPP is computing Jacobian sparsity structure.
KPP is starting the code generation.
KPP is initializing the code generation.
KPP is generating the monitor data:
    - small_strato_Monitor
KPP is generating the utility data:
    - small_strato_Util
KPP is generating the global declarations:
    - small_strato_Main
KPP is generating the ODE function:
    - small_strato_Function
KPP is generating the ODE Jacobian:
    - small_strato_Jacobian
    - small_strato_JacobianSP
KPP is generating the linear algebra routines:
    - small_strato_LinearAlgebra
KPP is generating the Hessian:
    - small_strato_Hessian
    - small_strato_HessianSP
KPP is generating the utility functions:
    - small_strato_Util
KPP is generating the rate laws:
    - small_strato_Rates
KPP is generating the parameters:
    - small_strato_Parameters
```

(continues on next page)

(continued from previous page)

```
KPP is generating the global data:
    - small_strato_Global
KPP is generating the stoichiometric description files:
    - small_strato_Stoichiom
    - small_strato_StoichiomSP
KPP is generating the driver from general.f90:
    - small_strato_Main
KPP is starting the code post-processing.

KPP has succesfully created the model "small_strato".
```

This will generate the Fortran90 code needed to solve the **`small_strato`** mechanism. The file listing should be similar to:

```
atoms.kpp                       small_strato.kpp
general.f90                     small_strato_LinearAlgebra.f90
Makefile_small_strato           small_strato_Main.f90
rosenbrock.def                  small_strato_mex_Fun.f90
rosenbrock.f90                  small_strato_mex_Hessian.f90
small_strato.def                small_strato_mex_Jac_SP.f90
small_strato.eqn                small_strato_Model.f90
small_strato_Function.f90       small_strato_Monitor.f90
small_strato_Global.f90         small_strato_Parameters.f90
small_strato_Hessian.f90        small_strato_Precision.f90
small_strato_HessianSP.f90      small_strato_Rates.f90
small_strato_Initialize.f90     small_strato.spc@
small_strato_Integrator.f90     small_strato_Stoichiom.f90
small_strato_Jacobian.f90       small_strato_StoichiomSP.f90
small_strato_JacobianSP.f90     small_strato_Util.f90
```

KPP creates Fortran90 beginning with the mechanism name (which is `ROOT_ = small_strato_` in this example). KPP also generates a human-readable summary of the mechanism (`small_strato.log`) as well as the Makefile `Makefile_small_strato` that can be used to build the executable.

3.4 4. Build and run the small_strato example

To compile the Fortran90 code generated by KPP into an executable, type:

```
$ make -f Makefile_small_strato
```

You will see output similar to this:

```
gfortran -cpp -O -g  -c small_strato_Precision
gfortran -cpp -O -g  -c small_strato_Precision.f90
gfortran -cpp -O -g  -c small_strato_Parameters.f90
gfortran -cpp -O -g  -c small_strato_Global.f90
gfortran -cpp -O -g  -c small_strato_Function.f90
```

(continues on next page)

(continued from previous page)

```
gfortran -cpp -O -g  -c small_strato_JacobianSP.f90
gfortran -cpp -O -g  -c small_strato_Jacobian.f90
gfortran -cpp -O -g  -c small_strato_HessianSP.f90
gfortran -cpp -O -g  -c small_strato_Hessian.f90
gfortran -cpp -O -g  -c small_strato_StoichiomSP.f90
gfortran -cpp -O -g  -c small_strato_Stoichiom.f90
gfortran -cpp -O -g  -c small_strato_Rates.f90
gfortran -cpp -O -g  -c small_strato_Monitor.f90
gfortran -cpp -O -g  -c small_strato_Util.f90
gfortran -cpp -O -g  -c small_strato_LinearAlgebra.f90
gfortran -cpp -O -g  -c small_strato_Initialize.f90
gfortran -cpp -O -g  -c small_strato_Integrator.f90
gfortran -cpp -O -g  -c small_strato_Model.f90
gfortran -cpp -O -g  -c small_strato_Main.f90
gfortran -cpp -O -g  small_strato_Precision.o    small_strato_
↪Parameters.o    small_strato_Global.o small_strato_Function.o␣
↪small_strato_JacobianSP.o small_strato_Jacobian.o small_strato_
↪HessianSP.o small_strato_Hessian.o small_strato_Stoichiom.o small_
↪strato_StoichiomSP.o small_strato_Rates.o   small_strato_Util.o  ␣
↪small_strato_Monitor.o small_strato_LinearAlgebra.o small_strato_
↪Main.o          small_strato_Initialize.o small_strato_Integrator.
↪o    small_strato_Model.o  -o small_strato.exe
```

Once compilation has finished, you can run the **`small_strato`** example by typing:

```
$ ./small_strato.exe | tee small_strato.log
```

This will run a "box-model" simulation forward several steps in time. You will see the concentrations of selected species at several timesteps displayed to the screen (aka the Unix stdout stream) as well as to a log file (`small_strato.log`).

If your simulation results exits abruptly with the `Killed` error, you probably need to increase your stack memory limit. On most Linux systems the default stacksize limit is 8 kIb = or 8192 kB. You can max this out with the following commands:

If you are using bash, type:

```
$ ulimit -s unlimited
```

If you are using csh, type:

```
$ limit stacksize unlimited
```

3.5 5. Cleanup

If you wish to remove the executable (`small_strato.exe`), as well as the object (`*.o`) and module (`*.mod`) files generated by the Fortran compiler, type:

```
$ make -f Makefile_small_strato clean
```

If you also wish to remove all the files that were generated by KPP (i.e. `small_strato.log` and `small_strato_*.f90`), type:

```
$ make -f Makefile_small_strato distclean
```

4 Input for KPP

KPP basically handles two types of input 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 the section entitled *Auxiliary files and the substitution preprocessor*.

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`.

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 *BNF description of the KPP language*.
- 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 can 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, the atom definitions (`atoms.kpp`) are included first, then species definitions (`*.spc`), and finally the equations of the chemical mechanism (`*.eqn`).

4.1 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.

#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.kpp
```

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.

#CHECK

KPP is able to do mass balance checks 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.

#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 fixed species does not vary through chemical reactions.

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, its composition 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;
```

#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:

```
#EQUATIONS

<R1> NO2 + hv = NO + O3P :  6.69e-1*(SUN/60.0);
<R2> O3P + O2 + AIR = O3 :  ARR_ac(5.68e-34,  -2.80);
<R3> O3P + O3 = 2O2 :       ARR_ab(8.00e-12, 2060.0);
<R4> O3P + NO + AIR = NO2 : ARR_ac(1.00e-31,  -1.60);
//... etc ...
```

Note: The above example is taken from the **saprc99** mechanism (see `models/saprc99.eqn`), with some whitespace deleted for clarity. Optional *equation tags* are specified by text within < > angle brackets. Functions that compute **saprc99** equation rates (e.g. `ARR_ac`, `ARR_ab`) are defined in `util/UserRateLaws.f90` and `util/UserRateLawsInterfaces.f90`.

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 (or a call to an *inlined rate law function*) 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 species 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;
```

However, it should be noted that using negative products may lead to numerical instabilities.

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 + 0.4 H + .05 H2 : k_CH4_
↪O1D;
```

KPP provides two pre-defined dummy species: `hv` and `PROD`. Using dummy species does not affect the numerics of the integrators. It only serves to improve the readability of the equations. For photolysis reactions, `hv` can be specified as one of the reagents to indicate that light ($h\nu$) is needed for this reaction, e.g.:

```
NO2 + hv = NO + O : J_NO2;
```

When the products of a reaction are not known or not important, the dummy species `PROD` should be used as a product. This is necessary because the KPP syntax does not allow an empty list of products. For example, the dry deposition of atmospheric ozone to the surface can be written as:

```
O3 = PROD : 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 `N2O5` 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;
```

#FAMILIES

Chemical families (for diagnostic purposes) may be specified in the **#FAMILIES** section as shown below. Family names beginning with a `P` denote production, and those beginning with an `L` denote loss.

```
#FAMILIES
  POx : O3 + NO2 + 2NO3 + HNO3 + ... etc. add more species as␣
↪needed ...
  LOx : O3 + NO2 + 2NO3 + HNO3 + ... etc. add more species as␣
↪needed ...
  PCO : CO;
  LCO : CO;
  PSO4 : SO4;
  LCH4 : CH4;
  PH2O2 : H2O2;
```

KPP will examine the chemical mechanism and create a dummy species for each defined family. Each dummy species will archive the production and loss for the family. For example, each molecule of CO that is produced will be added to the `PCO` dummy species. Likewise, each molecule of CO that is consumed will be added to the `LCO` dummy species. This will allow the `PCO` and `LCO` species to be later archived for diagnostic purposes. Dummy species for chemical families will not be included as active species in the mechanism.

#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 (cf *ROOT_Initialize*). 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.

#LOOKAT and #MONITOR

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

The 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 the section entitled *ROOT_Util*.

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 fixed 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;
```

#LUMP

To reduce the stiffness of some models, various lumping of species may be defined in the **#LUMP** section. In the example below, species NO and NO2 are summed and treated as a single lumped variable, NO2. Following integration, the individual species concentrations are recomputed from the lumped variable.

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

#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;
```

#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.:

```
#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.

4.2 KPP commands

A KPP command begins on a new line with a # sign, followed by a command name and one or more parameters. Details about each command are given in the following subsections.

Table 1: Default values for KPP commands

KPP command	default value
#CHECKALL	
#DECLARE	SYMBOL
#DOUBLE	ON
#DRIVER	none
#DUMMYINDEX	OFF
#EQNTAGS	OFF
#FUNCTION	AGGREGATE
#HESSIAN	ON
#INCLUDE	
#INTEGRATOR	
#INTFILE	
#JACOBIAN	SPARSE_LU_ROW
#LANGUAGE	
#LOOKATALL	
#MEX	ON
#MINVERSION	
#MODEL	
#REORDER	ON
#STOCHASTIC	OFF
#STOICMAT	ON
#TRANSPORTALL	
#UPPERCASEF90	OFF

#DECLARE

The **#DECLARE** command determines how constants like dp, NSPEC, NVAR, NFIX, and NREACT are inserted into the KPP-generated code. **#DECLARE SYMBOL** (the default) will declare array variables using parameters from the *ROOT_Parameters* file. **#DECLARE VALUE** will replace each parameter with its value.

For example, the global array variable C is declared in the *ROOT_Global* file generated by KPP. In the **small_strato** example (described in *Running KPP with an example stratospheric mechanism*), C has dimension NSPEC=7. Using **#DECLARE SYMBOL** will generate the following code in *ROOT_Global*:

```
! C - Concentration of all species
  REAL(kind=dp), TARGET :: C(NSPEC)
```

Whereas **#DECLARE VALUE** will generate this code instead:

```
! C - Concentration of all species
  REAL(kind=dp), TARGET :: C(7)
```

We recommend using **#DECLARE SYMBOL**, as most modern compilers will automatically replace each parameter (e.g. NSPEC) with its value (e.g 7). However, if you are using a very

old compiler that is not as sophisticated, **#DECLARE VALUE** might result in better-optmized code.

#DOUBLE

The **#DOUBLE** command selects single or double precision arithmetic. **ON** (the default) means use double precision, **OFF** means use single precision (see the section entitled *ROOT_Precision*).

Important: We recommend using double precision whenever possible. Using single precision may lead to integration non-convergence errors caused by roundoff and/or underflow.

#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`, `.F90`, `.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 command occurs twice, the second replaces the first.

#DUMMYINDEX

It is possible to declare species in the *#DEFVAR and #DEFFIX* sections 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 to **#DUMMYINDEX ON**. Then, KPP will set the indices to zero for all species that do not occur in any reaction. With **#DUMMYINDEX OFF** (the default), those 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
```

#EQNTAGS

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

```
<R1> NO2 + hv = NO + O3P :  6.69e-1*(SUN/60.0);
```

With **#EQNTAGS** set to **ON**, this equation tag can be used to refer to a specific equation (cf. *ROOT_Monitor*). The default for **#EQNTAGS** is **OFF**.

#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.

#HESSIAN

The option **ON** (the default) of the **#HESSIAN** command turns the Hessian generation on (see section *ROOT_Hessian and ROOT_HessianSP*). With **OFF** it is switched off.

#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 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.

#INTEGRATOR

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

```
#INTEGRATOR integrator_name
```

is similar to:

```
#INCLUDE $KPP_HOME/int/integrator_name.def
```

The **#INTEGRATOR** command allows the use of different integration techniques on the same model. If it occurs twice, the second replaces the first. Normally, KPP tries to find the selected integrator files 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

Attention: **#INTFILE** is used internally by KPP but should not be used by the KPP user. Using *#INTEGRATOR* alone suffices to specify an integrator.

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. The parameter of the command is a file name, without suffix. The appropriate suffix (`.f90`, `.F90`, `.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.

#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 x NVAR` matrix. It should only 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.

#LANGUAGE

Attention: The **Fortran77** language option is deprecated in *KPP 2.5.0* and later versions. All further KPP development will only support Fortran90.

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

You can select the suffix (`.F90` or `.f90`) to use for Fortran90 source code generated by KPP (cf. *#UPPERCASEF90*).

#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 **#MEX ON**. With **#MEX OFF**, no Mex files are generated.

#MINVERSION

You may restrict a chemical mechanism to use a given version of KPP or later. To do this, add

```
#MINVERSION X.Y.Z
```

to the definition file.

The version number (`X.Y.Z`) adheres to the Semantic Versioning style (https://semver.org), where `X` is the major version number, `Y` is the minor version number, and `Z` is the bugfix (aka "patch") version number.

For example, if **#MINVERSION 2.4.0** is specified, then KPP will quit with an error message unless you are using KPP 2.4.0 or later.

#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 a suitable 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.

#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.* [[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.

#STOCHASTIC

The option **ON** of the **#STOCHASTIC** command turns on the generation of code for stochastic kinetic simulations (see the section entitled *ROOT_Stochastic*. The default option is **OFF**.

#STOICMAT

Unless the **#STOICMAT** 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 (cf. *ROOT_Stoichiom and ROOT_StoichiomSP*).

#CHECKALL, #LOOKATALL, #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.

#UPPERCASEF90

If you have selected **#LANGUAGE Fortran90** option, KPP will generate source code ending in `.f90` by default. Setting **#UPPERCASEF90 ON** will tell KPP to generate Fortran90 code ending in `.F90` instead.

4.3 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, 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 shown in *KPP inlined types*. If two inline commands with the same inline type are declared, then the contents of the second is appended to the first one.

List of inlined types

In this manual, we show the inline types for Fortran90. The inline types for the other languages are produced by replacing `F90` by `C`, or `matlab`, respectively.

Table 2: KPP inlined types

Inline_type	File	Placement	Usage
F90_DATA	*ROOT_Monitor*	specification section	(obsolete)
F90_GLOBAL	*ROOT_Global*	specification section	global variables
F90_INIT	*ROOT_Initialize*	subroutine	integration parameters
F90_RATES	*ROOT_Rates*	executable section	rate law functions
F90_RCONST	*ROOT_Rates*	subroutine	statements and definitions of rate coefficients
F90_UTIL	*ROOT_Util*	executable section	utility functions

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.

F90_GLOBAL

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

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

Inlining code can be useful to introduce additional state variables (such as temperature, humidity, etc.) for use by KPP routines, such as for calculating rate coefficients.

If a large number of state variables needs to be held in inline code, or require intermediate computation that may be repeated for many rate coefficients, a derived type object should be used for efficiency, e.g.:

```
#INLINE F90_GLOBAL
  TYPE, PUBLIC :: ObjGlobal_t
     ! ... add variable fields to this type ...
  END TYPE ObjGlobal_t
  TYPE(ObjGlobal_t), TARGET, PUBLIC :: ObjGlobal
#ENDINLINE
```

This global variable `ObjGlobal` can then be used globally in KPP.

Another way to avoid cluttering up the KPP input file is to `#include` a header file with global variables:

```
#INLINE F90_GLOBAL
! Inline common variables into KPP_ROOT_Global.f90
#include "commonIncludeVars.f90"
#ENDINLINE
```

In future versions of KPP, the global state will be reorganized into derived type objects as well.

F90_INIT

This inline type can be used to define initial values before the start of the integration, e.g.:

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

F90_RATES

This inline type 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
```

F90_RCONST

This inline type can be used to define time-dependent values of rate coefficients. You may inline `USE` statements that reference modules where rate coefficients are computed, e.g.:

```
#INLINE F90_RCONST
  USE MyRateFunctionModule
#ENDINLINE
```

or define variables directly, e.g.:

```
#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
```

Note that the `USE` statements must precede any variable definitions.

The inlined code will be placed directly into the `UPDATE_RCONST` routine in the *ROOT_Rates* function.

F90_UTIL

This inline type can be used to define utility subroutines.

4.4 Auxiliary files and the substitution preprocessor

The *auxiliary files* in the `$KPP_HOME/util` subdirectory 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 placeholder symbols* 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 *KPP inlined types*.

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 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 command **#DOUBLE ON**, the above line will be automatically translated into:

```
REAL(kind=dp) :: BIG(1000)
```

and when used with the command **#DOUBLE OFF**, the same line will become:

```
REAL(kind=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.

List of auxiliary files for Fortran90

Table 3: 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` and `Makefile_upper_F90`	Makefiles to build Fortran-90 code.
`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` and `UserRateLawsInterfaces.f90`	User-defined rate-law functions.
`util.f90`	Input/output utilities.

List of symbols replaced by the substitution preprocessor

Table 4: Symbols and their replacements

Symbol	Replacement	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_LU_NHESS	Number of Hessian nonzero elements	10
KPP_FUN_OR_FUN_SPLIT	Name of the function to be called	`FUN(Y,FIX, RCONST,Ydot)`

4.5 Controlling the Integrator with ICNTRL and RCNTRL

In order to offer more control over the integrator, KPP provides the arrays ICNTRL (integer) and RCNTRL (real). Each of them is an array of 20 elements that allow the fine-tuning of the integrator. All integrators (except for tau_leap and gillespie) use ICNTRL and RCNTRL. Details can be found in the comment lines of the individual integrator files in $KPP_HOME/int/.

ICNTRL

Table 5: Summary of ICNTRL usage in the f90 integrators.
Here, Y = used, and s = solver-specific usage.

ICNTRL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
beuler		Y	Y	Y	Y	Y									Y		
dvode															Y		
exponen-tial																	
feuler															Y	Y	Y
gillespie																	
lsode		Y		Y						s					Y		
radau5		Y		Y	Y	Y					Y				Y		
rosen-brock_adj	Y	Y	Y	Y		s	s	s							Y		
rosenbrock	Y	Y	Y	Y											Y	Y	
rosen-brock_tlm	Y	Y	Y	Y								s			Y		
rosen-brock_autoreduce	Y	Y	Y	Y								s	s	s	Y	Y	
runge_kutta_adj		Y	Y	Y	Y	s	s	s	s	s	Y				Y		
runge_kutta		Y	Y	Y	Y	Y				s	Y				Y		
runge_kutta_tlm	Y	Y		Y	Y	s		s	s	Y	s				Y		
sdirk4		Y		Y											Y		
sdirk_adj		Y	Y	Y	Y	Y	s	s							Y		
sdirk		Y	Y	Y	Y	Y									Y		
sdirk_tlm		Y	Y	Y	Y	Y	s		s			s			Y		
seulex	Y	Y		Y						s	s	s	s	s	Y		
tau_leap																	

ICNTRL(1)

= 1: $F = F(y)$, i.e. independent of t (autonomous)

= 0: $F = F(t, y)$, i.e. depends on t (non-autonomous)

ICNTRL(2)

The absolute (ATOL) and relative (RTOL) tolerances can be expressed by either a scalar or individually for each species in a vector:

= 0 : NVAR -dimensional vector

= 1 : scalar

ICNTRL(3)
Selection of a specific method.

ICNTRL(4)
Maximum number of integration steps.

ICNTRL(5)
Maximum number of Newton iterations.

ICNTRL(6)
Starting values of Newton iterations (only avaialble for some of the integrators).

= 0 : Interpolated

= 1 : Zero

ICNTRL(11)
Gustafsson step size controller

ICNTRL(12)
(Solver-specific for rosenbrock_autoreduce) Controls whether auto-reduction of the mechanism is performed. If set to = 0, then the integrator behaves the same as rosenbrock.

ICNTRL(13)
(Solver-specific for rosenbrock_autoreduce) Controls whether in auto-reduction species production and loss rates are scanned throughout the internal time steps of the integrator for repartitioning.

ICNTRL(14)
(Solver-specific for rosenbrock_autoreduce) If set to > 0, then the threshold is calculated based on the max of production and loss rate of the species ID specified in ICNTRL(14) multiplied by RCNTRL(14).

ICNTRL(15)
This determines which Update_* subroutines are called within the integrator.

= -1 : Do not call any Update_* subroutines

= 0 : Use the integrator-specific default values

> 1 : A number between 1 and 7, derived by adding up bits with values 4, 2, and 1. The first digit (4) activates Update_SUN. The second digit (2) activates Update_PHOTO. The third digit (1) activates Update_RCONST. |

For example ICNTRL(15)=6) (4+2) will activate the calls to Update_SUN and Update_PHOTO, but not to Update_RCONST.

ICNTRL(16)
Treatment of negative concentrations:

= 0 : Leave negative values unchanged

= 1 : Set negative values to zero

= 2 : Print warning and continue

= 3 : Print error message and stop

ICNTRL(17)

Verbosity:

= 0 : Only return error number

= 1 : Verbose error output

ICNTRL(18) ... ICNTRL(20)

currently not used

Table 6: Summary of RCNTRL usage in the f90 integrators.
Here, Y = used, and s = solver-specific usage.

RCNTRL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
beuler	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
dvode																			
exponential																			
feuler																			
gillespie																			
lsode	Y	Y	Y																
radau5		Y		Y	Y	Y	Y	Y	Y	Y	Y								
rosenbrock_adj	Y	Y	Y	Y	Y	Y	Y												
rosenbrock	Y	Y	Y	Y	Y	Y	Y												
rosenbrock_tlm	Y	Y	Y	Y	Y	Y	Y												
rosenbrock_autoreduce	Y	Y	Y	Y	Y	Y	Y					s		s					
runge_kutta_adj	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
runge_kutta	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
runge_kutta_tlm	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk4	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk_adj	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk_tlm	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
seulex	Y	Y	Y	Y	Y	Y	Y	Y		s	s	s	s	s	s	s	s	s	s
tau_leap																			

RCNTRL(1)
`Hmin`, the lower bound of the integration step size. It is not recommended to change the default value of zero.

RCNTRL(2)
`Hmax`, the upper bound of the integration step size.

RCNTRL(3)
`Hstart`, the starting value of the integration step size.

RCNTRL(4)
`FacMin`, lower bound on step decrease factor.

RCNTRL(5)
FacMax, upper bound on step increase factor.

RCNTRL(6)
FacRej, step decrease factor after multiple rejections.

RCNTRL(7)
FacSafe, the factor by which the new step is slightly smaller than the predicted value.

RCNTRL(8)
ThetaMin. If the Newton convergence rate is smaller than ThetaMin, the Jacobian is not recomputed.

RCNTRL(9)
NewtonTol, the stopping criterion for Newton's method.

RCNTRL(10)
Qmin

RCNTRL(11)
Qmax. If Qmin < Hnew/Hold < Qmax, then the step size is kept constant and the LU factorization is reused.

RCNTRL(12)
(Solver-specific for rosenbrock_autoreduce) Used to specify the threshold for auto-reduction partitioning, if ICNTRL(12) = 1, and ICNTRL(14) = 0. Will be ignored if ICNTRL(14) > 0.

RCNTRL(14)
(Solver-specific for rosenbrock_autoreduce) Used to specify the multiplier for threshold for auto-reduction partitioning, if ICNTRL(12) = 1, and ICNTRL(14) > 0, RCNTRL(14) is multiplied against max of production and loss rates of species ICNTRL(14) to produce the partitioning threshold, ignoring RCNTRL(12).

RCNTRL(10) ... RCNTRL(19)
(Solver-specific for seulex)

RCNTRL(20)
currently not used

5 Output from KPP

This chapter describes the source code files that are generated by KPP.

5.1 The Fortran90 code

The code generated by KPP is organized in a set of separate files. Each has 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_` A short description of each file is contained in the following sections.

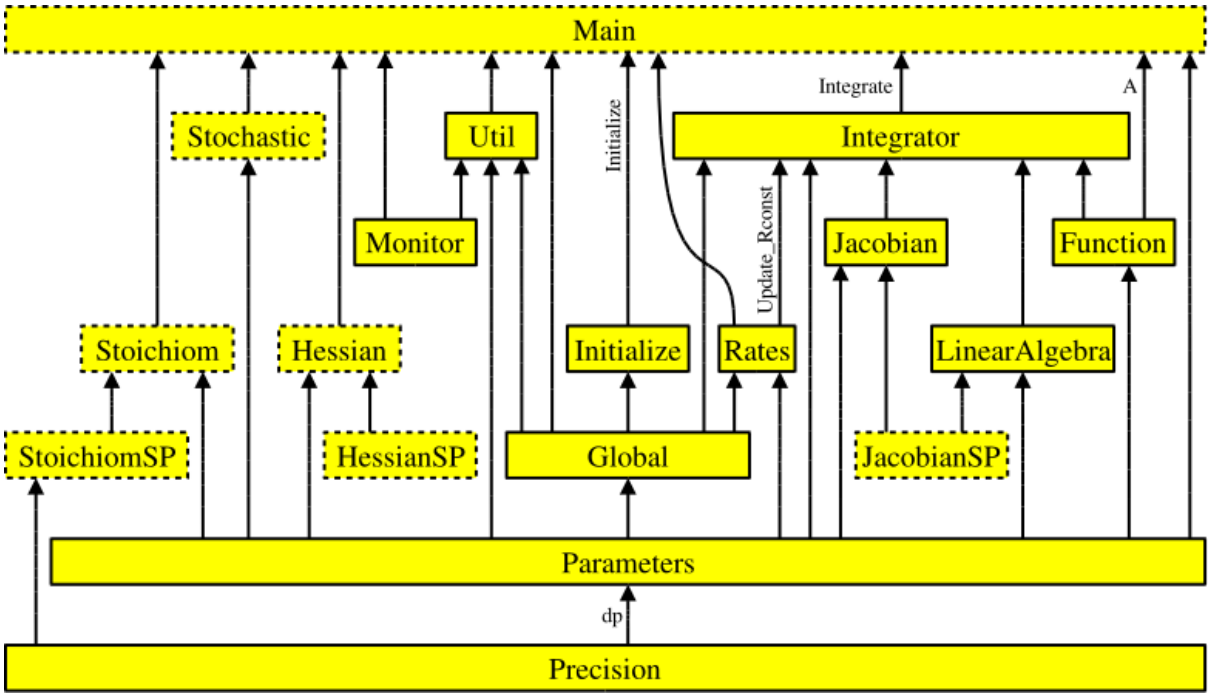

Fig. 1: Figure 1: Interdependencies of the KPP-generated files. Each arrow starts at the module that exports a variable or subroutine and points to the module that imports it via the Fortran90 `USE` instruction. The prefix `ROOT_` has been omitted from module names for better readability. Dotted boxes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

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` or `.F90`. *Figure 1 (above)* shows how these modules are related to each other. The generated code is consistent with the Fortran90 standard. It may, however, exceed the official maximum number of 39 continuation lines.

Tip: The default Fortran90 file suffix is `.f90`. To have KPP generate Fortran90 code ending in `.F90` instead, add the command **`#UPPERCASEF90 ON`** to the KPP definition file.

ROOT_Main

`ROOT_Main.f90` (or `.F90`) root 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 to the root name.

Using **`#DRIVER none`** will skip generating this file.

ROOT_Model

The file `ROOT_Model.f90` (or `.F90`) unifies all model definitions in a single module. This simplifies inclusion into external Fortran programs.

ROOT_Initialize

The file `ROOT_Initialize.f90` (or `.F90`) contains the subroutine `Initialize`, which defines initial values of the chemical species. The driver calls the subroutine once before the time integration loop starts.

ROOT_Integrator

The file `ROOT_Integrator.f90` (or `.F90`) contains the subroutine `Integrate`, which is called every time step during the integration. The integrator that was chosen with the *#INTEGRATOR* command is also included in this file. In case of an unsuccessful integration, the module root provides a short error message in the public variable `IERR_NAME`.

ROOT_Monitor

The file `ROOT_Monitor.f90` (`.F90`) contains 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 (cf. *#EQNTAGS*) that each reaction in the section may start with an equation tag which is enclosed in angle brackets, e.g.:

```
<R1> NO2 + hv = NO + O3P :  6.69e-1*(SUN/60.0e0);
```

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 tag number, this can be used to describe a reaction:

```
PRINT*, 'Reaction 1 is:', EQN_NAMES( tag2num( 'R1' ) )
```

ROOT_Precision

Fortran90 code uses parameterized real types. `ROOT_Precision.f90` (or `.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)
```

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

ROOT_Rates

The code to update the rate constants is in `ROOT_Rates.f90` (or `.F90`). The user defined rate law functions (cf. *Fortran90 subrotutines in ROOT_Rates*) are also placed here.

Table 7: Fortran90 subrotutines in ROOT_Rates

Function	Description
`Update_PHOTO`	Update photolysis rate coefficients
`Update_RCONST`	Update all rate coefficients
`Update_SUN`	Update sun intensity

ROOT_Parameters

Global parameters are defined and initialized in `ROOT_Parameters.f90` (or `.F90`):

Table 8: Parameters Declared in ROOT_Parameters

Parameter	Represents	Example
`NSPEC`	No. chemical species (`NVAR` + `NFIX`)	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`	

Example values listed in the 3rd column are taken from the **small_strato** mechanism (cf. *Running KPP with an example stratospheric mechanism*).

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

```
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.

ROOT_Global

Several global variables are declared in `ROOT_Global.f90` (or `.F90`):

Table 9: Global Variables Declared in ROOT_Global

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

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 to `NVAR+1` to `NSPEC`) the fixed species. The total number of species is the sum of the `NVAR` and `NFIX`. The parts can also be accessed separately through pointer variables `VAR` and `FIX`, which point to the proper elements in `C`.

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

Important: In previous versions of KPP, Fortran90 code was generated with `VAR` and `FIX`

being linked to the C array with an EQUIVALENCE statement. This construction, however, is not thread-safe, and it prevents KPP-generated Fortran90 code from being used within parallel environments (e.g. such as an OpenMP[5] parallel loop).

We have modified *KPP 2.5.0* and later versions to make KPP-generated Fortran90 code thread-safe. VAR and FIX are now POINTER variables that point to the proper slices of the C array. They are also nullified when no longer needed. VAR and FIX are now also kept internal to the various integrator files located in the $KPP_HOME/int directory.

ROOT_Function

The chemical ODE system for our **small_strato** example (described in *Running KPP with an example stratospheric mechanism*) is:

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

where square brackets denote concentrations of the species. The code for the ODE function is in ROOT_Function.f90 (or .F90) The chemical reaction mechanism represents a set of ordinary differential equations (ODEs) of dimension . The concentrations of fixed species are parameters in the derivative function. The subroutine computes first the vector A of reaction rates and then the vector Vdot of variable species time derivatives. The input arguments V, F, RCT are the concentrations of variable species, fixed species, and the rate coefficients, respectively. A and Vdot may be returned to the calling program (for diagnostic purposes) with optional ouptut argument Aout. Below is the Fortran90 code generated by KPP for the ODE function of our **small_strato** example.

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

! V - Concentrations of variable species (local)
  REAL(kind=dp) :: V(NVAR)
! F - Concentrations of fixed species (local)
  REAL(kind=dp) :: F(NVAR)
! RCT - Rate constants (local)
```

(continues on next page)

[5] https://openmp.org

(continued from previous page)

```
  REAL(kind=dp) :: RCT(NREACT)
! Vdot - Time derivative of variable species concentrations
  REAL(kind=dp) :: Vdot(NVAR)
! Aout - Optional argument to return equation rate constants
  REAL(kind=dp), OPTIONAL :: Aout(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)

  !### Use Aout to return equation rates
  IF ( PRESENT( Aout ) ) Aout = A

! 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
```

ROOT_Jacobian and ROOT_JacobianSP

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

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

It defines how the temporal change of each chemical species depends on all other species. For example, $\mathbf{J}(5,2)$ shows that NO_2 (species number 5) is affected by O (species number 2) via reaction R9. The sparse data structures for the Jacobian are declared and initialized in `ROOT_JacobianSP.f90` (or `.F90`). The code for the ODE Jacobian and sparse multiplications is in `ROOT_Jacobian.f90` (or `.F90`).

Tip: Adding either **`#JACOBIAN SPARSE_ROW`** or **`#JACOBIAN SPARSE_LU_ROW`** to the KPP definition file will create the file `ROOT_JacobianSP.f90` (or `.F90`).

The Jacobian of the ODE function is automatically constructed by KPP. KPP generates the Jacobian subroutine `Jac` or `JacSP` 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 (for the case where the LU fill-in is accounted for) are:

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

JVS stores the LU_NONZERO elements of the Jacobian in row order. Each row I starts at position LU_CROW(I), and LU_CROW(NVAR+1) = LU_NONZERO+1. The location of the I-th diagonal element is LU_DIAG(I). The sparse element JVS(K) is the Jacobian entry in row LU_IROW(K) and column LU_ICOL(K). 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 Figure 2 below.. 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.

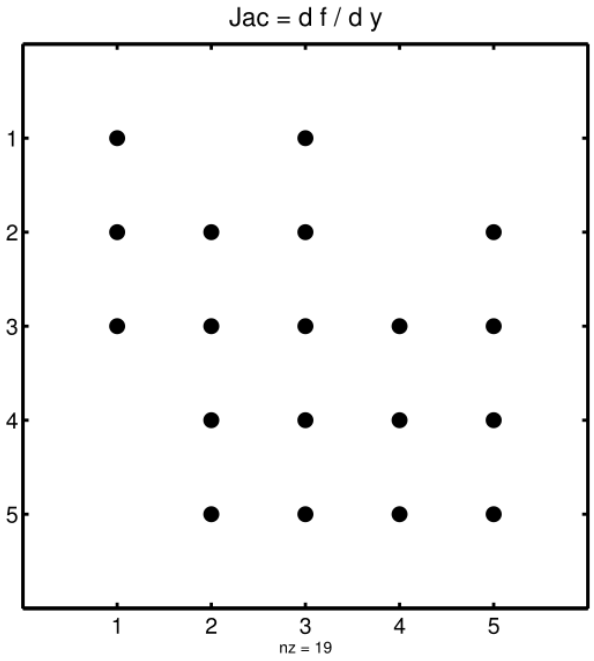

Fig. 2: Figure 2: The sparsity pattern of the Jacobian for the **small_strato** example. All non-zero elements are marked with a bullet. Note that even though $\mathbf{J}(3,5)$ is zero, it is also included here because of the fill-in.

Two other KPP-generated routines, Jac_SP_Vec and JacTR_SP_Vec (see *Fortran90 sub-*

routines in ROOT_Jacobian) 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: Fortran90 subroutines in ROOT_Jacobian

Function	Description
`Jac_SP`	ODE Jacobian in sparse format
`Jac_SP_Vec`	Sparse multiplication
`JacTR_SP_Vec`	Sparse multiplication
`Jac`	ODE Jacobian in full format

ROOT_Hessian and ROOT_HessianSP

The sparse data structures for the Hessian are declared and initialized in `ROOT_Hessian.f90` (or `.F90`). The Hessian function and associated sparse multiplications are in `ROOT_HessianSP.f90` (or `.F90`).

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

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

KPP generates the routine `Hessian`:

Table 12: Fortran90 functions in ROOT_Hessian

Function	Description
`Hessian`	ODE Hessian in sparse format
`Hess_Vec`	Hessian action on vectors
`HessTR_Vec`	Transposed Hessian action on vectors

Using the variable species `V`, the fixed species `F`, and the rate coefficients `RCT` as input, the subroutine `Hessian` calculates the Hessian. The Hessian is a very sparse tensor. The sparsity of the Hessian for our example is visualized in *Figure 3: The Hessian of the small_strato example.*

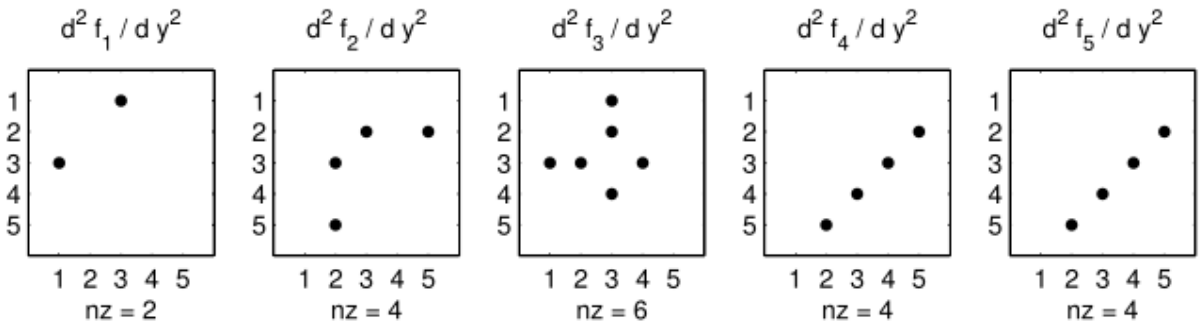

Fig. 3: Figure 3: The Hessian of the small_strato example.

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 *Sparse Hessian Data*.

Table 13: Sparse Hessian Data

Variable	Represents
`HESS(NHESS)`	Hessian nonzero elements $H_{i,j,k}$
`IHESS_I(NHESS)`	Index i of element $H_{i,j,k}$
`IHESS_J(NHESS)`	Index j of element $H_{i,j,k}$
`IHESS_J(NHESS)`	Index k of element $H_{i,j,k}$

Since the time derivative function is smooth, these Hessian matrices are symmetric, $\texttt{HESS}_{\texttt{i,j,k}}$=$\texttt{HESS}_{\texttt{i,k,j}}$. KPP stores only those entries $\texttt{HESS}_{\texttt{i,j,k}}$ with $j \leq k$. The sparsity coordinate vectors `IHESS_1`, `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.

ROOT_LinearAlgebra

Sparse linear algebra routines are in the file `ROOT_LinearAlgebra.f90` (or `.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

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

where the matrix $P = I - h\gamma J$ is refered to as the "prediction matrix". I the identity matrix, h the integration 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` (see *Fortran90 functions in ROOT_LinearAlgebra*) 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.

Table 14: Fortran90 functions in ROOT_LinearAlgebra

Function	Description
KppDecomp	Sparse LU decomposition
KppSolve	Sparse back subsitution
KppSolveTR	Transposed sparse back substitution

The subroutines KppSolve and KppSolveTr and use the in-place LU factorization P as computed by 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.* [[Sandu et al., 1996]].

ROOT_Stoichiom and ROOT_StoichiomSP

These files contain contain a description of the chemical mechanism in stoichiometric form. The file ROOT_Stoichiom.f90 (or .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_StochiomSP.f90 (or .F90).

Tip: Adding **#STOICMAT ON** to the KPP definition file will create the file ROOT_Stoichiom.f90 (or .F90) Also, if either **#JACOBIAN SPARSE ROW** or **#JACOBIAN SPARSE_LU_ROW** are also added to the KPP definition file, the file ROOT_StoichiomSP.f90 (or .F90) will also be created.

The stoichiometric matrix is constant sparse. For our example the matrix NSTOICM=22 has 22 nonzero entries out of 50 entries. KPP produces the stoichiometric matrix in sparse, column-compressed format, as shown in *Sparse Stoichiometric Matrix*. Elements are stored in columnwise order in the one-dimensional vector of values STOICM. Their row and column indices are stored in ICOL_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) = NSTOICHM+1 simplifies the handling of sparse data structures.

Table 15: Sparse Stoichiometric Matrix

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

Table 16: Fortran90 functions in ROOT_Stoichiom

Variable	Represents
dFun_dRcoeff	Derivatives of Fun w/r/t rate coefficients
dJac_dRcoeff	Derivatives of Jac w/r/t rate coefficients
ReactantProd	Reactant products
JacReactantProd	Jacobian of reactant products

The subroutine ReactantProd (see *Fortran90 functions in ROOT_Stoichiom*) computes the reactant products ARP for each reaction, and the subroutine JacReactantProd computes the Jacobian of reactant products vector, i.e.:

$$\mathtt{JVRP} = \partial \mathtt{ARP} / \partial \mathtt{V}$$

The matrix JVRP is sparse and is computed and stored in row compressed sparse format, as shown in *Fortran90 functions in ROOT_Hessian*. The parameter NJVRP holds the number of nonzero elements. For our **small_strato** 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 /)
```

Table 17: Sparse Data for Jacobian of Reactant Products

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

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 specified in the array

$$\mathtt{DFDR} = \partial \mathtt{Vdot} / \partial \mathtt{RCT}(\mathtt{JCOEFF})$$

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 dJacR_dCoeff, which calculates DJDR, the product of this partial derivative with a user-supplied vector U:

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

ROOT_Stochastic

If the generation of stochastic functions is switched on (i.e. when the command **#STOCHASTIC ON** is added to the KPP definition file), KPP produces the file `ROOT_Stochastic.f90` (or `.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 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`.).

ROOT_Util

In addition to the chemical system description routines discussed above, KPP generates several utility subroutines and functions in the file `ROOT_Util.f90` (or `.F90`).

Table 18: Fortran90 subroutines and functions in ROOT_Util

Function	Description
`GetMass`	Check mass balance for selected atoms
`Shuffle_kpp2user`	Shuffle concentration vector
`Shuffle_user2kpp`	Shuffle concentration vector
`InitSaveData`	Utility for **#LOOKAT** command
`SaveData`	Utility for **#LOOKAT** command
`CloseSaveData`	Utility for **#LOOKAT** command
`tag2num`	Calculate reaction number from equation tag
`Integrator_Update_Options`	Choose `Update_RCONST/PHOTO/SUN`

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` (cf. *#LOOKAT and #MONITOR*).

ROOT_mex_Fun, ROOT_mex_Jac_SP, and ROOT_mex_Hessian

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.

Tip: To generate Mex files, add the command **#MEX ON** to the KPP definition file.

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

5.2 The C code

Important: Some run-time options for C-language integrators (specified in the *ICNTRL and RCNTRL arrays*) do not exactly correspond to the Fortran90 run-time options. We will standardize run-time integrator options across all target languages in a future KPP release.

The driver file `ROOT.c` contains the main (driver) program 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.

1. The global parameters (cf. *Parameters Declared in ROOT_Parameters*) are `#include`-d in the header file `ROOT_Parameters.h`
2. The global variables (cf. *Global Variables Declared in ROOT_Global*) are extern-declared in `ROOT_Global.h` and declared in the driver file `ROOT.c`.
3. The header file `ROOT_Sparse.h` contains extern declarations of sparse data structures for the Jacobian (cf. *Sparse Jacobian Data Structures*),Hessian (cf. *Sparse Hessian Data*) and stoichiometric matrix (cf. *Sparse Stoichiometric Matrix*), and the Jacobian of reaction products (cf. *Sparse Data for Jacobian of Reactant Products*). The actual declarations of each datastructures is done in the corresponding files.

The code for the ODE function (see section *ROOT_Function*) is in `ROOT_Function.c`. The code for the ODE Jacobian and sparse multiplications (cf. *ROOT_Jacobian and ROOT_JacobianSP*) is in `ROOT_Jacobian.c`, and the declaration and initialization of the Jacobian sparse data structures is in the file `ROOT_JacobianSP.c`. Similarly, the Hessian function and associated sparse multiplications (cf. *ROOT_Hessian and ROOT_HessianSP*) are

in ROOT_Hessian.c, and the declaration and initialization of Hessian sparse data structures are 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 (cf. *ROOT_Stoichiom and ROOT_StoichiomSP*) . The declaration and initialization of the stoichiometric matrix and the associated sparse data structures (cf. *Sparse Stoichiometric Matrix*) is done in ROOT_StoichiomSP.c.

Sparse linear algebra routines (cf. *ROOT_LinearAlgebra*) 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 (cf. *ROOT_Util*) 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 (cf. *ROOT_mex_Fun, ROOT_mex_Jac_SP, and ROOT_mex_Hessian*) are also generated (in the files ROOT_mex_Fun.c, ROOT_mex_Jac_SP.c, and ROOT_mex_Hessian.c).

5.3 The Matlab code

Important: Some run-time options for Matlab-language integrators (specified in the *ICNTRL and RCNTRL arrays*) do not exactly correspond to the Fortran90 run-time options. We will standardize run-time integrator options across all target languages in a future KPP release.

Matlab[6] 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. KPP/Matlab can also 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 the sections entitled *ROOT_Function* , *ROOT_Jacobian and ROOT_JacobianSP*, *ROOT_Hessian and ROOT_HessianSP*, *ROOT_Stoichiom and ROOT_StoichiomSP*, *ROOT_Util*. The names of the m-function-files are the same as the names of the functions (prefixed by the model name ROOT.

The variables of *Parameters Declared in ROOT_Parameters* are defined as Matlab global variables and initialized in the file ROOT_parameter_defs.m. The variables of *Global Variables Declared in ROOT_Global* are declared as Matlab global variables in the file ROOT_global_defs.m. They can be accessed from within each Matlab function by using declarations of the variables of interest.

[6] http://www.mathworks.com/products/matlab/

The sparse data structures for the Jacobian (cf. *Sparse Jacobian Data Structures*), the Hessian (cf. *Sparse Hessian Data*), the stoichiometric matrix (cf. *Sparse Stoichiometric Matrix*), and the Jacobian of reaction (see *Sparse Data for Jacobian of Reactant Products*) are declared as Matlab `global` variables in the file `ROOT_Sparse_defs.m`. They are initialized in separate m-files, namely `ROOT_JacobianSP.m`, `ROOT_HessianSP.m`, and `ROOT_StoichiomSP.m` respectively.

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

Table 19: List of Matlab model files

Function	Description
`ROOT.m`	Driver
`ROOT_parameter_defs.m`	Global parameters
`ROOT_global_defs.m`	Global 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`	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_Hess_Vec.m`	Hessian action on vectors
`ROOT_HessTR_Vec.m`	Transposed Hessian action on vectors
`ROOT_stoichiom.m`	Derivatives of Fun and Jac w/r/t rate coefficients
`ROOT_stochiomSP.m`	Sparse data
`ROOT_ReactantProd.m`	Reactant products
`ROOT_JacReactantProd.m`	Jacobian of reactant products
`ROOT_Rates.m`	User-defined rate reaction laws
`ROOT_Update_PHOTO.m`	Update photolysis rate coefficients
`ROOT_Update_RCONST.m`	Update all rate coefficients
`ROOT_Update_SUN.m`	Update sola 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

5.4 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.

5.5 The log file

The log file `ROOT.log` 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.

5.6 Output from the Integrators (`ISTATUS` and `RSTATUS`)

In order to obtain more information about the integration, KPP provides the arrays `ISTATUS` (integer) and `RSTATUS` (real). Each of them is an array of 20 elements. Array elements not listed here are currently not used. Details can be found in the comment lines of the individual integrator files in `$KPP_HOME/int/`.

Table 20: Summary of ISTATUS usage in the f90 integrators. Here, Y = used.

ISTATUS	1	2	3	4	5	6	7	8	9
beuler	Y	Y	Y	Y	Y	Y	Y	Y	
dvode									
exponential									
feuler									
gillespie									
lsode	Y	Y	Y						
radau5	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_adj	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_tlm	Y	Y	Y	Y	Y	Y	Y	Y	Y
rosenbrock_autoreduce	Y	Y	Y	Y	Y	Y	Y	Y	
runge_kutta_adj	Y	Y	Y	Y	Y	Y	Y	Y	
runge_kutta	Y	Y	Y	Y	Y	Y	Y	Y	
runge_kutta_tlm	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk4	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk_adj	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk_tlm	Y	Y	Y	Y	Y	Y	Y	Y	
seulex	Y	Y	Y	Y	Y	Y	Y		
tau_leap									

`ISTATUS(1)`
Number of function calls.

`ISTATUS(2)`
Number of Jacobian calls.

`ISTATUS(3)`
Number of steps.

`ISTATUS(4)`
Number of accepted steps.

`ISTATUS(5)`
Number of rejected steps (except at very beginning).

`ISTATUS(6)`
Number of LU decompositions.

`ISTATUS(7)`
Number of forward/backward substitutions.

`ISTATUS(8)`
Number of singular matrix decompositions.

ISTATUS(9)
Number of Hessian calls.

ISTATUS(10) ... ISTATUS(20)
Currently not used.

RSTATUS

Table 21: Summary of RSTATUS usage in the f90 integrators. Here, Y = used, s = solver specific usage.

RSTATUS	1	2	3	4
beuler	Y	Y	Y	
dvode				
exponential				
feuler				
gillespie				
lsode	Y	Y		
radau5				
rosenbrock_adj	Y	Y	Y	
rosenbrock	Y	Y	Y	
rosenbrock_tlm	Y	Y	Y	
rosenbrock_autoreduce	Y	Y	Y	s
runge_kutta_adj	Y	Y	Y	
runge_kutta	Y	Y	Y	
runge_kutta_tlm	Y	Y	Y	
sdirk4	Y	Y		
sdirk_adj	Y	Y	Y	
sdirk	Y	Y	Y	
sdirk_tlm	Y	Y	Y	
seulex				
tau_leap				

RSTATUS(1)
`Texit`, the time corresponding to the computed Y upon return.

RSTATUS(2)
`Hexit`: the last accepted step before exit.

RSTATUS(3)
`Hnew`: The last predicted step (not yet taken. For multiple restarts, use `Hnew` as `Hstart` in the subsequent run.

RSTATUS(4)
(Solver-specific for `rosenbrock_autoreduce`) `AR_thr`: used to output the calculated (used) auto-reduction threshold for the integration. Useful when `ICNTRL(10) > 0` where the threshold is dynamically determined based on a given species.

RSTATUS(5) ... RSTATUS(20)
Currently not used.

6 Information for KPP developers

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

6.1 KPP directory structure

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

src/
Contains the KPP source code files:

Table 22: KPP 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_f90.c`	Generation of F90 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 generated file
`scan.h`	Input for Flex and Bison
`scan.l`	Input for Flex
`scan.y`	Input for Bison
`scanner.c`	Evaluate parsed input
`scanutil.c`	Evaluate parsed input
`y.tab.c`	Bison generated file
`y.tab.h`	Bison generated header file

bin/
Contains the KPP executable. This directory should be added to the `PATH` environment variable.

util/
Contains different function templates useful for the simulation. Each template file has a suffix that matches the appropriate target language (Fortran90, C, or Matlab). KPP will run the template files through the substitution preprocessor (cf. *List of symbols replaced*

by the substitution preprocessor). The user can define their own auxiliary functions by inserting them into the files.

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 (Fortran90, C, or Matlab). KPP will run the appropriate driver through the substitution preprocessor (cf. *List of symbols replaced by the substitution preprocessor*). Users can also define their own driver templates here.

int/
: Contains numerical solvers (integrators). The **#INTEGRATOR** command will force KPP to look into this directory for a definition file with suffix `.def`. This file selects the numerical solver etc. Each integrator template is found in a file that ends with the appropriate suffix (`.f90`, `.c`, or `.m`). The selected template is processed by the substitution preprocessor (cf. *List of symbols replaced by the substitution preprocessor*). Users can define their own numerical integration routines in the `user_contributed` subdirectory.

examples/
: Contains several model description examples (`.kpp` files) which can be used as templates for building simulations with KPP.

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

ci-tests/
: Contains directories defining several *Continuous integration tests*.

.ci-pipelines/
: Hidden directory containing a YAML file with settings for automatically running the continuous integration tests on Azure DevOps Pipelines[7]

  Also contains bash scripts (ending in `.sh`) for running the continuous integration tests either automatically in Azure Dev Pipelines, or manually from the command line. For more information, please see *Continuous integration tests*.

[7] https://azure.microsoft.com/en-us/services/devops/pipelines/

6.2 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 setting the `$KPP_HOME` environment variable to the path where KPP is installed.

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

There are also several optional environment variable that control the places where KPP looks for module files, integrators, and drivers:

`KPP_HOME`
: Required, stores the absolute path to the KPP distribution.

  Default setting: none.

`KPP_FLEX_LIB_DIR`
: Optional. Use this to specify the path to the *flex library file* (`libfl.so` or `libfl.a`) that are needed to *build the KPP executable*. The KPP build sequence will use the path contained in `KPP_FLEX_LIB_DIR` if the flex library file cannot be found in `/usr/lib`, `/usr/lib64`, and similar standard library paths.

`KPP_MODEL`
: Optional, specifies additional places where KPP will look for model files before searching the default location.

  Default setting: `$KPP_HOME/models`.

`KPP_INT`
: Optional, specifies additional places where KPP will look for integrator files before searching the default.

  Default setting: `$KPP_HOME/int`.

`KPP_DRV`
: Optional specifies additional places where KPP will look for driver files before searching the default directory.

  Default setting: `$KPP_HOME/drv`.

6.3 KPP internal modules

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 bison 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
- 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. Some other errors like mass balance, and equation duplicates, are tested at the end of this phase.

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 algorithm.

Expression trees computation

This is the core of the preprocessor. This module generates the production/destruction functions, the Jacobian and all the data structure nedeed by these functions. It builds 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 built 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.

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.

6.4 Adding new KPP commands

To add a new KPP command, the source code has to be edited at several locations. A short summary is presented here, using `NEWCMD` as an example:

- Add the new command to several files in the `src/` directory:
  - `scan.h`: add `void CmdNEWCMD( char *cmd );`
  - `scan.l`: add `{ "NEWCMD", PRM_STATE, NEWCMD },`
  - `scanner.c`: add `void CmdNEWCMD( char *cmd )`
  - `scan.y`:
    - Add `%token NEWCMD`
    - Add `| NEWCMD PARAMETER`
    - Add `{ CmdNEWCMD( $2 ); }`
- Add *Continuous integration tests*:
  - Create a new directory `ci-tests/ros_newcmd/ros_newcmd.kpp`
  - Add new *Continuous integration tests* to the `ci-tests` directory and update the scripts in the `.ci-pipelines` directory.
- Other:
  - Explain in user manual `docs/source/*/*.rst`:
    - Add to Table *Default values for KPP commands*
    - Add a new subsection to *KPP commands*
    - Add to the Table *BNF description of the KPP language*
  - Add to `site-lisp/kpp.el`

6.5 Continuous integration tests

KPP contains several continuous integration (aka C-I) tests. Each C-I test calls KPP to generate source code for a given *chemical mechanism*, *integrator*, and *target language*, and then runs a short "box model" simulation with the generated code. C-I tests help to ensure that new features and updates added to KPP will not break any existing functionality.

The continuous integration tests will run automatically on Azure DevOps Pipelines[8] each time a commit is pushed to the KPP Github repository[9]. You can also run the integration tests *locally on your own computer*.

[8] https://azure.microsoft.com/en-us/services/devops/pipelines/
[9] https://github.com/KineticPreProcessor/KPP

List of continuous integration tests

Table 23: Continuous integration tests

C-I test	Language	Model	Integrator
C_rk	C	small_strato	runge_kutta
C_rosadj	C	small_strato	rosenbrock_adj
C_sd	C	small_strato	sdirk
C_sdadj	C	small_strato	sdirk_adj
C_small_strato	C	small_strato	rosenbrock
F90_lsode	Fortran90	small_strato	lsode
F90_radau	Fortran90	saprc99	radau5
F90_rk	Fortran90	small_strato	runge_kutta
F90_rktlm	Fortran90	small_strato	runge_kutta_tlm
F90_ros	Fortran90	small_strato	rosenbrock
F90_ros_autoreduce	Fortran90	saprc99	rosenbrock_autoreduce
F90_ros_split	Fortran90	small_strato	rosenbrock
F90_ros_upcase	Fortran90	saprc99	rosenbrock
F90_rosadj	Fortran90	small_strato	rosenbrock_adj
F90_rosenbrock	Fortran90	saprc99	rosenbrock
F90_rostlm	Fortran90	small_strato	rosenbrock_tlm
F90_saprc_2006	Fortran90	saprcnov	rosenbrock
F90_sd	Fortran90	small_strato	sdirk
F90_sdadj	Fortran90	small_strato	sdirk_adj
F90_seulex	Fortran90	saprcnov	seulex
F90_small_strato	Fortran90	small_strato	rosenbrock
X_minver	Fortran90	small_strato	runge_kutta

Notes about C-I tests:

1. `F90_ros_split` also uses **`#FUNCTION SPLIT`**.
2. `F90_ros_upcase` also uses **`#UPPERCASEF90 ON`**.
3. `F90_small_strato` is the example from *Running KPP with an example stratospheric mechanism*.
4. `X_minver` tests if the *#MINVERSION* command works properly.

Each continuous integration test is contained in a subdirectory of `$KPP_HOME/ci-tests`. In each subdirectory is a KPP definition file (ending in `.kpp`).

Running continuous integration tests on Azure DevOps Pipelines

The files needed to run the C-I tests are located in the `$KPP_HOME/.ci-pipelines` directory:

Table 24: Files needed to execute C-I tests

File	Description
`Dockerfile`	File containing specifications for the Docker container that will be used to run C-I tests on Azure DevOps Pipelines. Also contains commands needed to run the C-I scripts in the Docker container.
`build_testing.yml`	Contains options for triggering C-I tests on Azure DevOps Pipelines.
`ci-testing-script.sh`	Driver script for running C-I tests. Can be used on Azure DevOps Pipelines or on a local computer.
`ci-cleanup-script.sh`	Script to remove compiler-generated files (e.g. `*.o`, `.mod`, and `.exe`) from C-I test folders.
`ci-common-script.sh`	Script with common variable and function definitions needed by `ci-testing-script.sh` and `ci-cleanup-script.sh`.

The `Dockerfile` contains the software environment for Azure DevOps Pipelines[10]. You should not have to update this file.

File `build_testing.yml` defines the runtime options for Azure DevOps Pipelines. The following settings determine which branches will trigger C-I tests:

```
# Run a C-I test when a push to any branch is made.
trigger:
  branches:
    include:
      - '*'
pr:
  branches:
    include:
      - '*'
```

Currently this is set to trigger the C-I tests when a commit or pull request is made to any branch of https://github.com/KineticPreProcessor/KPP. This is the recommended setting, but you can restrict this so that only pushes or pull requests to certain branches will trigger the C-I tests.

The script `ci-testing-script.sh` executes all of the C-I tests whenever a push or a pull request is made to the selected branches in the KPP Github repository.

[10] https://azure.microsoft.com/en-us/services/devops/pipelines/

Running continuous integration tests locally

To run the C-I tests on a local computer system, use this command:

```
$ $KPP_HOME/.ci-pipelines/ci-testing-script.sh | tee ci-tests.log
```

This will run all C-I tests on your own computer system and pipe the results to a log file. This will easily allow you to check if the results of the C-I tests are identical to C-I tests that were run on a prior commit or pull request.

To remove the files generated by the continuous integration tests, use this command:

```
$ $KPP_HOME/.ci-pipelines/ci-cleanup-script.sh
```

If you add new C-I tests, be sure to add the name of the new tests to the variable `GENERAL_TESTS` in `ci-common-defs.sh`.

7 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 $\sim 10^{-2} \ldots 10^{-5}$). In addition, the KPP numerical integrators preserve the linear invariants (i.e., mass) of the chemical system.

KPP implements several Rosenbrock methods: ROS–2 (Verwer *et al.* [[Verwer et al., 1999]]), ROS–3 (Sandu *et al.* [[Sandu et al., 1997b]]), RODAS–3 (Sandu *et al.* [[Sandu et al., 1997b]]), ROS–4 (Hairer and Wanner [[Hairer and Wanner 1991]]), and RODAS–4 (Hairer and Wanner [[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.

The symbols used in the formulas of the following sections are:

Table 25: Symbols used in numerical methods

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

7.1 Rosenbrock methods

Integrator file: `int/rosenbrock.f90`

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

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

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 below:

ROS-2

- Stages (s): 2
- Funcion calls: 2
- Order: 2(1)
- Stability properties: L-stable
- Method Coefficients:

$$\begin{array}{lll} \gamma = 1 + 1/sqrt2 & a_{2,1} = 1/\gamma & c_{2,1} = -2/\gamma \\ m_1 = 3/(2\gamma) & m_2 = 1/(2\gamma) & e_1 = 1/(2\gamma) \\ e_2 = 1/(2\gamma) & \alpha_1 = 0 & \alpha_2 = 1 \\ \gamma_1 = \gamma & \gamma_2 = -\gamma & \end{array}$$

ROS-3

- Stages (s): 3
- Funcion calls: 2
- Order: 3(2)
- Stability properties: L-stable
- Method Coefficients:

$$\begin{array}{lll} a_{2,1} = 1 & a_{3,1} = 1 & a_{3,2} = 0 \\ c_{2,1} = -1.015 & c_{3,1} = 4.075 & c_{3,2} = 9.207 \\ m_1 = 1 & m_2 = 6.169 & m_3 = -0.427 \\ e_1 = 0.5 & e_2 = -2.908 & e_3 = 0.223 \\ alpha_1 = 0 & \alpha_2 = 0.436 & \alpha_3 = 0.436 \\ \gamma_1 = 0.436 & \gamma_2 = 0.243 & \gamma_3 = 2.185 \end{array}$$

ROS-4

- Stages (s): 4
- Funcion calls: 3
- Order: 4(3)
- Stability properties: L-stable
- Method Coefficients:

$$
\begin{aligned}
a_{2,1} &= 2 & a_{3,1} &= 1.868 & a_{3,2} &= 0.234 \\
a_{4,1} &= a_{3,1} & a_{4,2} &= a_{3,2} & a_{4,3} &= 0 \\
c_{2,1} &= -7.137 & c_{3,1} &= 2.581 & c_{3,2} &= 0.652 \\
c_{4,1} &= -2.137 & c_{4,2} &= -0.321 & c_{4,3} &= -0.695 \\
m_1 &= 2.256 & m_2 &= 0.287 & m_3 &= 0.435 \\
m_4 &= 1.094 & e_1 &= -0.282 & e_2 &= -0.073 \\
e_3 &= -0.108 & e_4 &= -1.093 & \alpha_1 &= 0 \\
\alpha_2 &= 1.146 & \alpha_3 &= 0.655 & \alpha_4 &= \alpha_3 \\
\gamma_1 &= 0.573 & \gamma_2 &= -1.769 & \gamma_3 &= 0.759 \\
\gamma_4 &= -0.104 & & & &
\end{aligned}
$$

RODAS-3

- Stages (s): 4
- Funcion calls: 3
- Order: 3(2)
- Stability properties: Stiffly-accurate
- Method Coefficients:

$$
\begin{aligned}
a_{2,1} &= 0 & a_{3,1} &= 2 & a_{3,2} &= 0 \\
a_{4,1} &= 2 & a_{4,2} &= 0 & a_{4,3} &= 1 \\
c_{2,1} &= 4 & c_{3,1} &= 1 & c_{3,2} &= -1 \\
c_{4,1} &= 1 & c_{4,2} &= -1 & c_{4,3} &= -8/3 \\
m_1 &= 2 & m_2 &= 0 & m_3 &= 1 \\
m_4 &= 1 & e_1 &= 0 & e_2 &= 0 \\
e_3 &= 0 & e_4 &= 1 & \alpha_1 &= 0 \\
\alpha_2 &= 0 & \alpha_3 &= 1 & \alpha_4 &= 1 \\
\gamma_1 &= 0.5 & \gamma_2 &= 1.5 & \gamma_3 &= 0 \\
\gamma_4 &= 0 & & & &
\end{aligned}
$$

RODAS-4

- Stages (s): 6
- Funcion calls: 5
- Order: 4(3)
- Stability properties: Stiffly-accurate
- Method Coefficients:

$$
\begin{aligned}
\alpha_1 &= 0 & \alpha_2 &= 0.386 & \alpha_3 &= 0.210 \\
\alpha_4 &= 0.630 & \alpha_5 &= 1 & \alpha_6 &= 1 \\
\gamma_1 &= 0.25 & \gamma_2 &= -0.104 & \gamma_3 &= 0.104 \\
\gamma_4 &= -0.036 & \gamma_5 &= 0 & \gamma_6 &= 0 \\
a_{2,1} &= 1.544 & a_{3,1} &= 0.946 & a_{3,2} &= 0.255 \\
a_{4,1} &= 3.314 & a_{4,2} &= 2.896 & a_{4,3} &= 0.998 \\
a_{5,1} &= 1.221 & a_{5,2} &= 6.019 & a_{5,3} &= 12.537 \\
a_{5,4} &= -0.687 & a_{6,1} &= a_{5,1} & a_{6,2} &= a_{5,2} \\
a_{6,3} &= a_{5,3} & a_{6,4} &= a_{5,4} & a_{6,5} &= 1 \\
c_{2,1} &= -5.668 & c_{3,1} &= -2.430 & c_{3,2} &= -0.206 \\
c_{4,1} &= -0.107 & c_{4,2} &= -9.594 & c_{4,3} &= -20.47 \\
c_{5,1} &= 7.496 & c_{5,2} &= -0.124 & c_{5,3} &= -34 \\
c_{5,4} &= 11.708 & c_{6,1} &= 8.083 & c_{6,2} &= -7.981 \\
c_{6,3} &= -31.521 & c_{6,4} &= 16.319 & c_{6,5} &= -6.058 \\
m_1 &= a_{5,1} & m_2 &= a_{5,2} & m_3 &= a_{5,3} \\
m_4 &= a_{5,4} & m_5 &= 1 & m_6 &= 1 \\
e_1 &= 0 & e_2 &= 0 & e_3 &= 0 \\
e_4 &= 0 & e_5 &= 0 & e_6 &= 1
\end{aligned}
$$

Rosenbrock tangent linear model

Integrator file: `int/rosenbrock_tlm.f90`

The Tangent Linear method is combined with the sensitivity equations. One step of the method reads:

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

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

KPP contains tangent linear models (for direct decoupled sensitivity analysis) for each of the Rosenbrock methods (*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.

Rosenbrock discrete adjoint model

Integrator file: `int/rosenbrock_adj.f90`

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

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

KPP contains adjoint models (for direct decoupled sensitivity analysis) for each of the Rosenbrock methods (*ROS-2*, *ROS-3*, *ROS-4*, *RODAS-3*, *RODAS-4*).

Rosenbrock with mechanism auto-reduction

Integrator file: `int/rosenbrock_autoreduce.f90`

Mechanism auto-reduction (described in Lin *et al.* [[Lin et al., 2022]]) expands previous work by Santillana *et al.* [[Santillana et al., 2010]] and Shen *et al.* [[Shen et al., 2020]] to a computationally efficient implementation in KPP, avoiding memory re-allocation, re-compile of the code, and on-the-fly mechanism reduction based on dynamically determined production and loss rate thresholds.

We define a threshold δ which can be fixed (as in Santillana *et al.* [[Santillana et al., 2010]]) or determined by the production and loss rates of a "target species" scaled by a factor

$$\delta = max(P_{target}, L_{target}) * \alpha_{target}.$$

For each species i, the species is partitioned as "slow" iff.

$$max(P_i, L_i) < \delta$$

if the species is partitioned as "slow", it is solved explicitly (decoupled from the rest of the mechanism) using a first-order approximation. Otherwise, "fast" species are retained in the implicit Rosenbrock solver.

7.2 Runge-Kutta (aka RK) methods

A general s-stage Runge-Kutta method is defined as (see Section II.1 of Hairer *et al.* [[Hairer Norsett and Wanner 1987]])

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

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) $n \times s$.

The Runge-Kutta methods implemented in KPP are summarized below:

3-stage Runge-Kutta

Integrator file: `int/runge_kutta.f90`

Fully implicit 3-stage Runge-Kutta methods. Several variants are available:

- RADAU-2A: order 5
- RADAU-1A: order 5
- Lobatto-3C: order 4
- Gauss: order 6

RADAU5

Integrator file: `int/radau5.f90`

This Runge-Kutta method of order 5 based on RADAU-IIA quadrature is stiffly accurate. The KPP implementation follows the original implementation of Hairer and Wanner [[Hairer and Wanner 1991]], Section IV.10. While RADAU5 is relatively expensive (when compared to the Rosenbrock methods), it is more robust and is useful to obtain accurate reference solutions.

SDIRK

Integrator file: `int/sdirk.f90`,

SDIRK is an L-stable, singly-diagonally-implicit Runge-Kutta method. The implementation is based on Hairer and Wanner [[Hairer and Wanner 1991]]. Several variants are available:

- Sdirk 2a, 2b: 2 stages, order 2
- Sdirk 3a: 3 stages, order 2
- Sdirk 4a, 4b: 5 stages, order 4

SDIRK4

Integrator file: `int/sdirk4.f90`

SDIRK4 is an L-stable, singly-diagonally-implicit Runge-Kutta method of order 4. The implementation is based on Hairer and Wanner [[Hairer and Wanner 1991]].

SEULEX

Integrator file: `int/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 [[Hairer and Wanner 1991]].

RK tangent linear model

The tangent linear method associated with the Runge-Kutta method is

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

The system 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.

RK discrete adjoint model

The first order Runge-Kutta adjoint is

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

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

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

7.3 Backward differentiation formulas

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

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

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 the following sections:

LSODE

Integrator file: `int/lsode.f90`

LSODE, the Livermore ODE solver (Radhakrishnan and Hindmarsh [[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.

VODE

Integrator file: `int/dvode.f90`

VODE (Brown *et al.* [[Brown Byrne and Hindmarsh 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.

BEULER

Integrator file: `int/beuler.f90`

Backward Euler integration method.

7.4 Other integration methods

FEULER

Integrator file: `int/feuler.f90`

Forward Euler is an explicit integration method for non-stiff problems. FEULER computes y^{n+1} as

$$y^{n+1} = y^n + hf\left(t^n, y^n\right)$$

8 BNF description of the KPP language

Following is the BNF-like specification of the KPP 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
 ↪              |
                            #FAMILIES family_list
 ↪              |
                            #INITVALUES initvalues_list
 ↪              |
```

(continues on next page)

(continued from previous page)

```
                                #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
↪                |
                                #DECLARE [ SYMBOL | VALUE ]
↪                |
                                #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 ]
↪                |
                                #MINVERSION minimum_version_number
↪                |
                                #MODEL model_name
↪                |
                                #REORDER [ ON | OFF ]
↪                |
                                #STOCHASTIC [ ON | OFF ]
↪                |
```

(continues on next page)

(continued from previous page)

```
                              #STOICHMAT [ ON | OFF ]
 ↪              |
                              #TRANSPORTALL [ ON | OFF ]
 ↪              |
                              #UPPERCASEF90 [ ON | OFF ]

inline_code ::=               #INLINE inline_type
                              inline_code
                              #ENDINLINE

atom_count ::=                integer atom_name
 ↪              |
                              atom_name

atom_definition_list :=       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

equation_tag ::=              Alphanumeric expression, also including
 ↪the
                              underscore. In scan.l it is defined as
                              "[a-zA-Z_0-0]+".

expression ::=                term
 ↪              |
                              term + expression
 ↪              |
                              term - expression

initvalues_assignment :=      species_name_plus = program_expression;
 ↪              |
                              CFACTOR = program_expression

initvalues_list ::=           initvalues_assignment
 ↪              |
                              initvalues_assignment initvalues_list
```

(continues on next page)

(continued from previous page)

```
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;                        ␣
↪             |
```

(continues on next page)

(continued from previous page)

```
                                species_name_plus; species_list_plus

species_name ::=                Alphanumeric expression, also including␣
 ↪the
                                underscore, starting with a letter.  In
                                scan.l it is defined as "[a-zA-Z_][a-ZA-
 ↪Z_0-9]*".
                                Its maximum length is 32.

species_name_plus ::=           species_name                            ␣
 ↪              |
                                VAR_SPEC                                ␣
 ↪              |
                                FIX_SPEC                                ␣
 ↪              |
                                ALL_SPEC

term ::=                        number species_name                     ␣
 ↪              |
                                species_name                            ␣
 ↪              |
                                PROD                                    ␣
 ↪              |
                                hv
```

9 Acknowledgements

This work has been supported by:

- The US EPA Science to Achieve Results (EPA-STAR)[11] program (grant # R840014[12]);
- The NASA Modeling, Analysis, and Prediction (MAP)[13] program;
- The NASA Atmospheric Composition Modeling and Analysis (ACMAP)[14] program;
- The NASA Advanced Systems Information Technology (AIST)[15] program (grant # AIST-18-0011[16])

We thank Jason Lander for his suggestions how to migrate from **`yacc`** to **`bison`**.

We would also like to thank Lucas Estrada for his assistance in setting up the *Continuous integration tests* on Azure DevOps Pipelines[17]. and for assistance with debugging.

[11] https://www.epa.gov/research-grants/air-research-grants
[12] https://cfpub.epa.gov/ncer_abstracts/index.cfm/fuseaction/display.abstractDetail/abstract_id/11083/report/0
[13] https://map.nasa.gov
[14] https://airbornescience.nasa.gov/category/Discipline/Atmospheric_Composition_Modeling_and_Analysis_Program
[15] https://esto.nasa.gov/aist
[16] https://esto.nasa.gov/project-selections-for-aist-18/#martin
[17] https://azure.microsoft.com/en-us/services/devops/pipelines/

Shaddy Ahmed and Jennie Thomas helped us with the Matlab output of KPP.

We thank Domenico Taraborrelli for providing the rosenbrock_posdef_h211b_qssa solver.

Parts of this user manual are based on Damian-Iordache [[Damian-Iordache 1996]].

10 References

11 Known Bugs

Bugs are discussed at the KPP repository Github issues page[24].

12 Support

The support guidelines can be found here[25].

13 Contributing

The Contributing guidelines can be found here[26].

14 Editing this User Guide

This user guide is generated with Sphinx[27]. Sphinx is an open-source Python project designed to make writing software documentation easier. The documentation is written in a reStructuredText (it's similar to markdown), which Sphinx extends for software documentation. The source for the documentation is the `docs/source` directory in top-level of the source code.

[24] https://github.com/KineticPreProcessor/KPP/issues/
[25] https://github.com/KineticPreProcessor/KPP/blob/main/SUPPORT.md
[26] https://github.com/KineticPreProcessor/KPP/blob/main/CONTRIBUTING.md
[27] https://www.sphinx-doc.org/

14.1 Quick start

To build this user guide on your local machine, you need to install Sphinx. Sphinx is a Python 3 package and it is available via **`pip`**. This user guide uses the Read The Docs theme, so you will also need to install `sphinx-rtd-theme`. It also uses the sphinxcontrib-bibtex[28] and recommonmark[29] extensions, which you'll need to install.

```
$ cd $KPP_HOME/docs
$ pip install -r requirements.txt
```

Installing with `requirements.txt` will make sure that the proper versions of Sphinx and its dependencies will be installed.

To build this user guide locally, navigate to the `docs/` directory and make the `html` target.

```
$ cd $KPP_HOME/docs
$ make html
```

This will build the user guide in `docs/build/html`, and you can open `index.html` in your web-browser. The source files for the user guide are found in `docs/source`.

Note: You can clean the documentation with `make clean`.

14.2 Learning reST

Writing reST can be tricky at first. Whitespace matters, and some directives can be easily miswritten. Two important things you should know right away are:

- Indents are 3-spaces
- "Things" are separated by 1 blank line. For example, a list or code-block following a paragraph should be separated from the paragraph by 1 blank line.

You should keep these in mind when you're first getting started. Dedicating an hour to learning reST will save you time in the long-run. Below are some good resources for learning reST.

- reStructuredText primer[30]: (single best resource; however, it's better read than skimmed)
- Official reStructuredText reference[31] (there is *a lot* of information here)
- Presentation by Eric Holscher[32] (co-founder of Read The Docs) at DjangoCon US 2015 (the entire presentation is good, but reST is described from 9:03 to 21:04)
- YouTube tutorial by Audrey Tavares's[33]

[28] https://pypi.org/project/sphinxcontrib-bibtex/
[29] https://recommonmark.readthedocs.io/
[30] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html
[31] https://docutils.sourceforge.io/docs/user/rst/quickref.html
[32] https://www.youtube.com/watch?v=eWNiwMwMcr4
[33] https://www.youtube.com/watch?v=DSIuLnoKLd8

A good starting point would be Eric Holscher's presentations followed by the reStructuredText primer.

14.3 Style guidelines

Important: This user guide is written in semantic markup. This is important so that the user guide remains maintainable. Before contributing to this documentation, please review our style guidelines (below). When editing the source, please refrain from using elements with the wrong semantic meaning for aesthetic reasons. Aesthetic issues can be addressed by changes to the theme.

For **titles and headers**:

- Section headers should be underlined by `#` characters
- Subsection headers should be underlined by `-` characters
- Subsubsection headers should be underlined by `^` characters
- Subsubsubsection headers should be avoided, but if necessary, they should be underlined by `"` characters

File paths (including directories) occuring in the text should use the `:file:` role.

Program names (e.g. **`cmake`**) occuring in the text should use the `:program:` role.

OS-level commands (e.g. **`rm`**) occuring in the text should use the `:command:` role.

Environment variables occuring in the text should use the `:envvar:` role.

Inline code or code variables occuring in the text should use the `:code:` role.

Code snippets should use `.. code-block:: <language>` directive like so

```
.. code-block:: python

   import gcpy
   print("hello world")
```

The language can be "none" to omit syntax highlighting.

For command line instructions, the "console" language should be used. The `$` should be used to denote the console's prompt. If the current working directory is relevant to the instructions, a prompt like `gcuser:~/path1/path2$` should be used.

Inline literals (e.g. the `$` above) should use the `:literal:` role.

References

[Atkinson et al., 2004] Atkinson, R., Baulch, D. L., Cox, R. A., Crowley, J. N., Hampson, R. F., Hynes, R. G., Jenkin, M. E., Rossi, M. J., and Troe, J. Evaluated kinetic and photochemical data for atmospheric chemistry: volume I – gas phase reactions of Ox, HOx, NOx, and SOx species. *Atmos. Chem. Phys.*, 4:1461–1738, 2004. doi:10.5194/ACP-4-1461-2004[18].

[Brown Byrne and Hindmarsh 1989] Brown, P. N., Byrne, G. D., and Hindmarsh, A. C. VODE: a variable step ode solver. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051, 1989.

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

[Hairer Norsett and Wanner 1987] Hairer, E., Norsett, S. P., and Wanner, G. *Solving Ordinary Differential Equations I. Nonstiff Problems.* Springer-Verlag, Berlin, 1987.

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

[Lin et al., 2022] Lin, H., Long, M. S., Sander, R., Sandu, A., Yantosca, R. M., Estrada, L. A., Shen, L., and Jacob, D. J. An adaptive auto-reduction solver for speeding up integration of chemical kinetics in atmospheric chemistry models: implementation and evaluation within the kinetic pre-processor (KPP) version 3.0.0. *J. Adv. Model. Earth Syst.*, submitted, 2022.

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

[Sander et al., 2005] 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. doi:10.5194/ACP-5-445-2005[19].

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

[Sandu and Sander 2006] Sandu, A. and Sander, R. Technical note: simulating chemical systems in fortran90 and matlab with the kinetic preprocessor kpp-2.1. *Atmos. Chem. Phys.*, 6:187–195, 2006. doi:10.5194/ACP-6-187-2006[20].

[Sandu et al., 1997b] 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. doi:10.1016/S1352-2310(97)83212-8[21].

[Santillana et al., 2010] Santillana, M., Le Sager, P., Jacob, D. J., and Brenner, M. P. An adaptive reduction algorithm for efficient chemical calculations in global

[18] https://doi.org/10.5194/ACP-4-1461-2004
[19] https://doi.org/10.5194/ACP-5-445-2005
[20] https://doi.org/10.5194/ACP-6-187-2006
[21] https://doi.org/10.1016/S1352-2310(97)83212-8

atmospheric chemistry models. *Atmos. Environ.*, 44(35):4426–4431, 2010. doi:10.1016/j.atmosenv.2010.07.044[22].

[Shen et al., 2020] Shen, L., Jacob, D. J., Santillana, M., Wang, X., and Chen, W. An adaptive method for speeding up the numerical integration of chemical mechanisms in atmospheric chemistry models: application to GEOS-Chem version 12.0.0. *Geosci. Model Dev.*, 13:2475–2486, 2020. doi:10.5194/gmd-13-2475-2020[23].

[Verwer et al., 1999] Verwer, J., Spee, E. J., 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.

[22] https://doi.org/10.1016/j.atmosenv.2010.07.044
[23] https://doi.org/10.5194/gmd-13-2475-2020

Index

Symbols

B

C

D

E

F

I

K

M

P

R

S

U