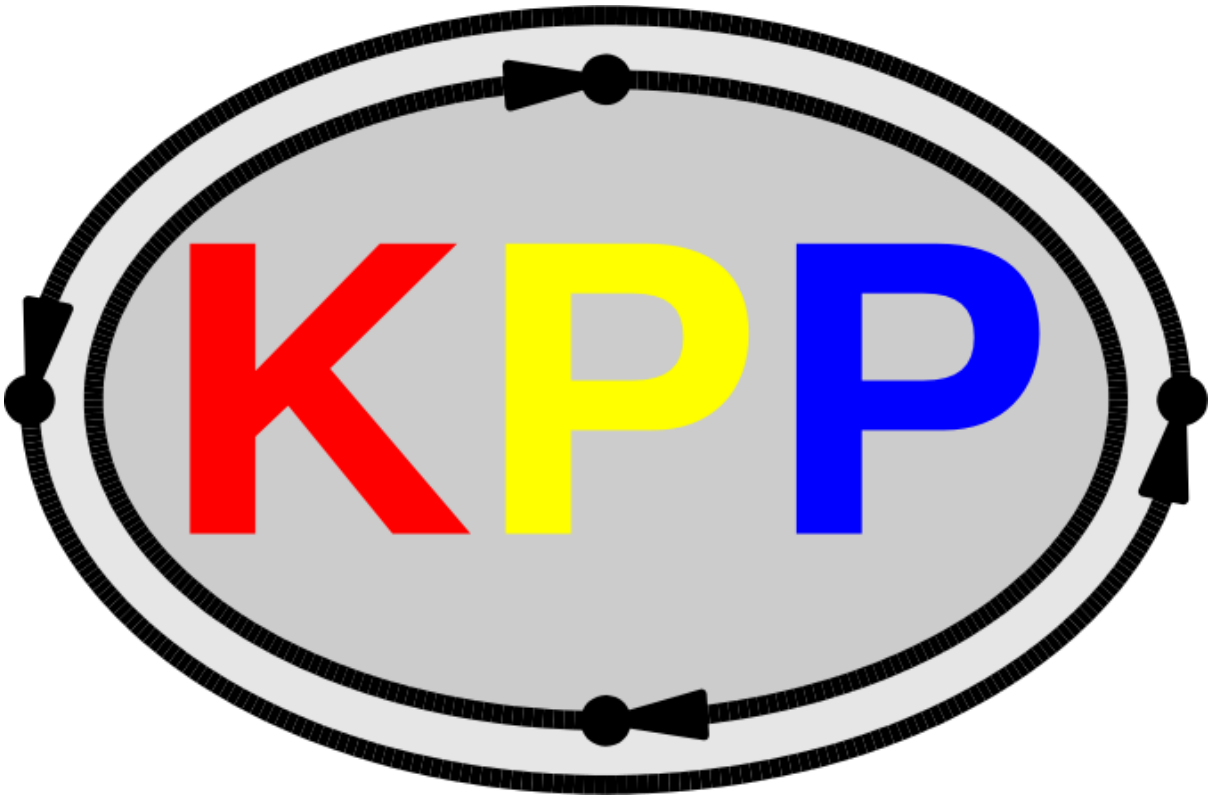


KPP: The Kinetic PreProcessor

Release 3.5.0

A. Sandu, R. Sander, M. Long, H. Lin, R. Yantosca, L. Estrada, L

Aug 17, 2026

Getting Started

1 KPP revision history **6**
1.1 KPP 3.5.0 . . . 7
1.2 KPP 3.4.0 . . . 7
1.3 KPP 3.3.1 . . . 7
1.4 KPP 3.3.0 . . . 7
1.5 KPP 3.2.1 . . . 7
1.6 KPP 3.2.0 . . . 8
1.7 KPP 3.1.1 . . . 8
1.8 KPP 3.1.0 . . . 8
1.9 KPP 3.0.2 . . . 8
1.10 KPP 3.0.1 . . . 8

1.11 KPP 3.0.0 . . . 9
1.12 KPP 2.6.0 . . . 10
1.13 KPP 2.5.0 . . . 10
1.14 KPP 2.4.0 . . . 10
1.15 KPP 2.3.2_gc . . . 10
1.16 KPP 2.3.1_gc . . . 11
1.17 KPP 2.3.0_gc . . . 11
1.18 KPP 2.2.5_gc . . . 11
1.19 KPP 2.2.4_gc . . . 11
1.20 KPP 2.2.3 . . . 12
1.21 KPP 2.1 . . . 13
1.22 KPP 1.1-f90-alpha12 . . . 13

2 Installation **13**
2.1 Download KPP from GitHub . . . 13
2.2 Define the KPP_HOME environment variable . . . 13
2.3 Test if KPP dependencies are installed . . . 15
2.4 Build the KPP executable . . . 17
2.5 Additional steps for macOS . . . 18

3 Running KPP with an example stratospheric mechanism **24**
3.1 1. Create a directory for the example . . . 25
3.2 2. Create a KPP Definition File . . . 25
3.3 3. Build the mechanism with KPP . . . 28
3.4 4. Build and run the small_strato example . . . 30
3.5 5. Cleanup . . . 31

4 Input overview **31**

5 KPP sections **32**
5.1 #ATOMS . . . 32
5.2 #CHECK . . . 32
5.3 #DEFVAR and #DEFFIX . . . 32
5.4 #EQUATIONS . . . 33
5.5 #FAMILIES . . . 34
5.6 #INITVALUES . . . 35
5.7 #LOOKAT and #MONITOR . . . 35
5.8 #SETVAR and #SETFIX . . . 36

6 KPP commands **36**
6.1 #AUTOREDUCE . . . 36
6.2 #DECLARE . . . 36
6.3 #DOUBLE . . . 37
6.4 #DRIVER . . . 37
6.5 #DUMMYINDEX . . . 38
6.6 #EQNTAGS . . . 38
6.7 #FUNCTION . . . 38
6.8 #GRAPH . . . 38
6.9 #HESSIAN . . . 40

6.10 #INCLUDE . . . 40
6.11 #INTEGRATOR . . . 40
6.12 #INTFILE . . . 40
6.13 #JACOBIAN . . . 41
6.14 #LANGUAGE . . . 41
6.15 #MEX . . . 41
6.16 #MINVERSION . . . 41
6.17 #MODEL . . . 42
6.18 #REORDER . . . 42
6.19 #STOCHASTIC . . . 42
6.20 #STOICMAT . . . 42
6.21 #CHECKALL, #LOOKATALL . . . 42
6.22 #UPPERCASEF90 . . . 42

7 Inlined code **43**
7.1 List of inlined types . . . 43
7.2 F90_DATA . . . 43
7.3 F90_GLOBAL . . . 43
7.4 F90_INIT . . . 44
7.5 F90_RATES . . . 44
7.6 F90_RCONST . . . 44
7.7 F90_RCONST_USE . . . 45
7.8 F90_UTIL . . . 45

8 Auxiliary files and the substitution preprocessor **45**
8.1 List of auxiliary files for Fortran90 . . . 46
8.2 List of symbols replaced by the substitution preprocessor . . . 47

9 Filtering out passive species with an absolute tolerance threshold **47**

10 The Fortran90 code **49**
10.1 ROOT_Main . . . 49
10.2 ROOT_Model . . . 50
10.3 ROOT_Initialize . . . 50
10.4 ROOT_Integrator . . . 50
10.5 ROOT_Monitor . . . 50
10.6 ROOT_Precision . . . 50
10.7 ROOT_Rates . . . 51
10.8 ROOT_Parameters . . . 51
10.9 ROOT_Global . . . 52
10.10 ROOT_Function . . . 53
10.11 ROOT_Jacobian and ROOT_JacobianSP . . . 55
10.12 ROOT_Hessian and ROOT_HessianSP . . . 57
10.13 ROOT_LinearAlgebra . . . 58
10.14 ROOT_Stoichiom and ROOT_StoichiomSP . . . 59
10.15 ROOT_Stochastic . . . 60
10.16 ROOT_Util . . . 61
10.17 ROOT_mex_Fun, ROOT_mex_Jac_SP, and ROOT_mex_Hessian . . . 61

11 The C code **62**

12 The Matlab code **63**

13 The Makefile **64**
13.1 all . . . 65
13.2 COMPILER=GFORTRAN . . . 65
13.3 COMPILER=GCC . . . 65
13.4 COMPILER=INTEL . . . 65
13.5 exe . . . 65
13.6 mex . . . 65
13.7 clean . . . 65
13.8 distclean . . . 66

14 The log file **66**

15 Rosenbrock methods **66**
15.1 ROS-2 . . . 67
15.2 ROS-3 . . . 67
15.3 ROS-4 . . . 67
15.4 RODAS-3 . . . 68
15.5 RODAS-3.1 . . . 68
15.6 RODAS-4 . . . 69
15.7 Rosenbrock tangent linear model . . . 70
15.8 Rosenbrock discrete adjoint model . . . 70
15.9 Rosenbrock with mechanism auto-reduction . . . 70
15.10 Rosenbrock with H211b time stepping . . . 71

16 Runge-Kutta (aka RK) methods **71**
16.1 3-stage Runge-Kutta . . . 71
16.2 RADAU5 . . . 72
16.3 SDIRK . . . 72
16.4 SDIRK4 . . . 72
16.5 SEULEX . . . 72
16.6 RK tangent linear model . . . 72
16.7 RK discrete adjoint model . . . 73

17 Backward differentiation methods **73**
17.1 LSODE . . . 73
17.2 VODE . . . 74
17.3 BEULER . . . 74

18 Forward differentiation methods **74**
18.1 FEULER . . . 74

19 Integrator options (`ICNTRL` and `RCNTRL`) **74**
19.1 ICNTRL . . . 75
19.2 RCNTRL . . . 82

20 Integrator status (`ISTATUS` and `RSTATUS`) **84**

20.1 ISTATUS . . . 85
20.2 RSTATUS . . . 87

21 KPP directory structure **88**
21.1 src/ . . . 88
21.2 bin/ . . . 88
21.3 util/ . . . 88
21.4 models/ . . . 89
21.5 drv/ . . . 89
21.6 int/ . . . 89
21.7 examples/ . . . 89
21.8 site-lisp/ . . . 89
21.9 ci-tests/ . . . 89
21.10 ci-pipelines/ . . . 89
21.11 .github/workflows . . . 90

22 KPP environment variables **90**
22.1 KPP_HOME . . . 90
22.2 KPP_FLEX_LIB_DIR . . . 90
22.3 KPP_MODEL . . . 90
22.4 KPP_INT . . . 90
22.5 KPP_DRV . . . 90

23 KPP internal modules **91**
23.1 Scanner and parser . . . 91
23.2 Species reordering . . . 91
23.3 Expression trees computation . . . 91
23.4 Code generation . . . 92

24 Adding new KPP commands **92**

25 Continuous integration tests **92**
25.1 List of continuous integration tests . . . 93
25.2 Running continuous integration tests as a GitHub Action . . . 94
25.3 Running continuous integration tests locally . . . 94

26 BNF description of the KPP language **95**

27 Acknowledgements **99**

28 References **99**

29 Known bugs and issues **99**
29.1 Current bug reports . . . 99
29.2 Bugs that have been resolved . . . 100

30 Support **100**

31 Contributing **100**

32 Editing this User Guide **100**
32.1 Quick start 100
32.2 Learning reStructured Text 102
32.3 Style guidelines 103

References **105**

Index **108**

An Environment for the Simulation of Chemical Kinetic Systems

Adrian Sandu[1], Rolf Sander[2], Michael S. Long[3], Haipeng Lin[4], Robert M. Yantosca[4], Lucas Estrada[4], Lu Shen[5], and Daniel J. Jacob[4]

[1] *Virginia Polytechnic Institute and State University, Blacksburg, VA, USA*
[2] *Max-Planck Institute for Chemistry, Mainz, Germany*
[3] *Renaissance Fiber, LLC, North Carolina, USA*
[4] *Harvard John A. Paulson School of Engineering and Applied Sciences, Cambridge, MA, USA*
[5] *School of Physics, Peking University, Beijing, China*

---

1 2 3 4 5 6

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[7].

[1] https://github.com/KineticPreProcessor/KPP/releases/
[2] https://github.com/KineticPreProcessor/KPP/releases/
[3] https://github.com/KineticPreProcessor/KPP/blob/main/LICENSE.txt
[4] https://doi.org/10.5281/zenodo.6828025
[5] https://kpp.readthedocs.io/en/latest/
[6] https://github.com/KineticPreProcessor/KPP/actions/workflows/run-ci-tests.yml
[7] https://github.com/KineticPreProcessor/KPP/blob/main/CHANGELOG.md

1.1 KPP 3.5.0

- Updated KPP code to exclude passive species from the Rosenbrock error norm computation when the optional `PassiveSpc_ATOL_Threshold` argument is passed to the `Initialize` routine
- Added *F90_ros_passivespc* C-I test and *drv/general_passivespc.f90* driver program
- Added GCC compiler versions 14, 15, and 16 to the "Run C-I tests" GitHub Action
- Fixed several build issues for MacOSX with ARM chipsets
- Fixed emacs font-lock issues in `site-lisp/kpp.el`
- Split ReadTheDocs documentation into smaller files and reorganized documentation folders

1.2 KPP 3.4.0

- Added the *RODAS-3.1* method to all Rosenbrock integrators, activated with `ICNTRL(3)==7`
- Added the *#GRAPH* command, which generates graph-based representations of the chemical mechanism
- Fixed several issues in the `int/runge_kutta.c` integrator that were causing numerical instability

1.3 KPP 3.3.1

- Replaced calls to BLAS functions in F90 integrators with core F90 array operations
- Migrated C-I tests from Microsoft Azure Dev pipelines to GitHub Actions
- Added new C-I tests: `F90_rkadj`, `F90_sd4`, `F90_sdtlm`
- Fixed a compilation issue that prevented `KPP_ROOT_Stochiom.F90` from being compiled when `#STOCHIOM ON` and `#HESSIAN OFF` are used

1.4 KPP 3.3.0

- Added the *rosenbrock_h211b_qssa* integrator as described in Dreger *et al.* [2025], along with the corresponding C-I tests.

1.5 KPP 3.2.1

- Updated the Forward Euler integrator to return the `Texit` value via the `RSTATUS(1)` argument
- Fixed parsing of `INITVALUES` to allow Fortran double-precision exponents
- Added carbon gases mechanism in `models/carbon` folder
- Added the C-I test `F90_feuler` to validate the forward Euler integrator

1.6 KPP 3.2.0

- Added new inline key `F90_RCONST_USE` so that F90 `USE` statements can be inlined into the `Update_RCONST` and `UPDATE_PHOTO` routines
- Updated code in `src/gen.c` to generate the `UPDATE_RCONST` routine with an optional argument `Y`
- Updated C-I tests to print the compiler versions that are used
- Updated `int/sdirk.f90` to a newer version
- Removed `int/beuler.f90`; Users can select Backward Euler with `sdirk` integrator and `ICNTRL(3)=6`
- Added MacOS architecture-specific compilation flags to the build sequence

1.7 KPP 3.1.1

- Use newer Python packages to build ReadTheDocs documentation (see `docs/requirements.txt`)
- Increased `MAX_NO_OF_LINES` and `MAX_EQN` in order to parse the entire MCM mechanism
- Now only add the extra *Aout* argument to *Fun* and *Fun_Split* for target language `Fortran90`. This fixes a Matlab build error.

1.8 KPP 3.1.0

- Added `#AUTOREDUCE` to the list of KPP commands in the ReadTheDocs documentation
- Added `examples/mcm` folder with minimal example for the Master Chemical Mechanism
- Added C-I test for MCM, based on the minimal example
- Removed obsolete input options from the code and documentation

1.9 KPP 3.0.2

- Added a `.readthedocs.yaml` file to the root folder to explicitly state the parameters controlling the documentation build. This will "future-proof" the KPP documentation against issues caused by software updates on the ReadTheDocs platform.

1.10 KPP 3.0.1

- Fixed a segmentation fault that occurred when using the `#STOICMAT` option by using dynamically-sized variables `EqnNr` and `MaxNr` rather than static variables `MAX_EQN` and `MAX_SPECIES`.

1.11 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*.
- 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.
- The dummy integrator `none` does not exist anymore. Thus, commands such as `#INTEGRATOR none` should be removed.

- 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 specified 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`.

1.12 KPP 2.6.0

- Added the **rosenbrock_autoreduce** integrator Lin *et al.* [2023].

1.13 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 documentation).
- 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*.

1.14 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.

1.15 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*).

1.16 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.

1.17 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*.)

1.18 KPP 2.2.5_gc

NOTE: Contains KPP modifications specific to GEOS-Chem.

- Increase `MAX_INLINE` from 20000 to 50000

1.19 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

1.20 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 Rosenbrock 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.

1.21 KPP 2.1

- Described by 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.

1.22 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[8]:

```
$ 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 was cloned. Also add the path of the KPP executable to the `PATH` environment variable. These commands may be placed in your shell startup file, as shown below.

Important

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

The **`git clone`** command above creates a directory named `KPP` (uppercase). Make sure that `KPP_HOME` points to this exact directory name. On case-sensitive Linux filesystems, `KPP` and `kpp` are **different** directories.

If you use bash:

Add these statements to your `$HOME/.bashrc` file:

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

Then apply the settings with:

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

If you use zsh (macOS Catalina and later)

Add these statements to your `$HOME/.zshrc` file:

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

Then apply the settings with:

```
$ source $HOME/.zshrc
```

Important

macOS Catalina (2019) and later versions use **`zsh`** as the default shell instead of **`bash`**. If you are using **`bash`** on macOS, source `~/.bash_profile` rather than `~/.bashrc`, because macOS Terminal opens a login shell by default and login shells do not source `~/.bashrc` automatically. The safest approach is to add the following line to your `~/.bash_profile`:

```
[ -f ~/.bashrc ] && source ~/.bashrc
```

If you use csh or tcsh:

Add these statements to your `$HOME/.cshrc` file:

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

Then apply the settings with:

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

2.3 Test if KPP dependencies are installed

KPP depends on several other Unix/Linux packages. Before using KPP for the first time, test whether these are installed on your system. If any packages are missing, you can install them with your system package manager (e.g. **apt** for Ubuntu, **yum** for Fedora, **homebrew** for macOS) or with Spack[9].

gcc

> **Important**
>
> macOS users: please read *Force macOS to use the GNU gcc compiler* before proceeding further.

KPP uses the GNU Compiler Collection[10] (**gcc**) by default. A version of **gcc** comes pre-installed with most Linux systems. To test whether **gcc** is installed, type:

```
gcc --version
```

This will display version information such as:

```
gcc-15 (GCC 15.3.0) 15.3.0
Copyright (C) 2025 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.
```

sed

The **sed** utility searches for and replaces text in files. To test whether **sed** is 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 whether **bison** is installed, type:

```
which bison
```

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

[9] https://spack.readthedocs.io
[10] https://gcc.gnu.org

flex

> **Important**
>
> macOS users: please read *Install and configure flex on macOS* before proceeding further.

The **flex** (Fast Lexical Analyzer) utility creates a scanner that recognizes the syntax generated by **bison**. To test whether **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. Use the **find** command to locate them:

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

This generates output similar to the following. Look for the entry containing `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 containing the flex library file (in this example `/usr/lib`), define the `KPP_FLEX_LIB_DIR` environment variable in your shell startup file as shown below.

If you use bash:

Add this code to your `~/.bashrc` file:

```
export KPP_FLEX_LIB_DIR=/usr/lib
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${KPP_FLEX_LIB_DIR}
 ↪"
```

Then apply the changes with:

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

If you use zsh:

Add this code to your `~/.zshrc` file:

```
export KPP_FLEX_LIB_DIR=/usr/lib
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${KPP_FLEX_LIB_DIR}
 ↪"
```

Then apply the changes with:

```
$ source ~/.zshrc
```

If you use csh or tcsh:

Add this code to your `~/.cshrc` file:

```
setenv KPP_FLEX_LIB_DIR /usr/lib
setenv LD_LIBRARY_PATH  ${LD_LIBRARY_PATH}:${KPP_FLEX_LIB_DIR}
```

Then apply the changes with:

```
$ source ~/.cshrc
```

KPP will use the path specified by `KPP_FLEX_LIB_DIR` in the following section.

2.4 Build the KPP executable

Change to the `KPP/src` directory:

```
cd $KPP_HOME/src
```

> **Note**
>
> The following **`make clean`** and **`make distclean`** commands are only necessary if you have previously built KPP and wish to start from a clean state. You can skip them on a fresh clone.

To remove previously-built KPP object files and example output:

```
make clean
```

To also remove a previously-built KPP executable:

```
make distclean
```

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

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
```

(continues on next page)

(continued from previous page)

```
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 creates the executable file `$KPP_HOME/bin/kpp`.

To verify that the KPP executable is accessible, type:

```
which kpp
```

The path displayed should match `$KPP_HOME/bin/kpp`.

> **Note**
>
> This check must be performed **after** running **`make`**. The executable does not exist before the build step completes.

2.5 Additional steps for macOS

When installing KPP on a macOS system, some additional configuration steps are necessary. The sections below address each one.

Force macOS to use the GNU gcc compiler

On macOS, the command:

```
$ gcc --version
```

will typically produce output similar to:

```
Apple clang version 17.0.0 (clang-1700.6.4.2)
Target: x86_64-apple-darwin24.6.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

This is because macOS installs **`clang`** as **`gcc`**. Follow the steps below to make your shell use the GNU **`gcc`** compiler instead.

1. Install gcc with Homebrew

```
brew install gcc
```

2. Define compiler settings in your shell startup file

The following code block uses **`brew --prefix`** to locate the Homebrew installation directory automatically. This works correctly on both Intel Macs (`/usr/local`) and Apple Silicon Macs (`/opt/homebrew`).

If you use bash:

Add the following to your `~/.bashrc` (or `~/.bash_profile` if you do not have a `~/.bashrc`):

```
#=============================================================================
# %%%%% Personal settings: Compilers (MacOS) %%%%%
#
# NOTE: MacOSX installs Clang as /usr/bin/gcc, so we have to manually
# force reference to GNU compilers in /usr/local/opt/gcc/bin.

#=============================================================================

# --- Homebrew GCC (override Apple Clang) ----------------------
GCC_BIN="$(brew --prefix gcc)/bin"

if [[ -d "${GCC_BIN}" ]]; then

    # Auto-detect the GCC major version by finding the gcc-NN executable
    # Pipeline: find gcc-NN files --> extract 'gcc-NN' -->
    #           sort numerically --> take highest --> extract number
    GCC_VER=$(find "${GCC_BIN}" -maxdepth 1 -name 'gcc-[0-9]*' 2>/dev/null \
                | grep -oE 'gcc-[0-9]+$'  \
                | sort -t'-' -k2 -n       \
                | tail -1                 \
                | grep -oE '[0-9]+$')

    if [[ "x${GCC_VER}" != "x" ]]; then

        unset CC
        unset CXX
        unset FC
```

(continues on next page)

(continued from previous page)

```
        # --- Interactive shell aliases ----------------------
↪------------------
        alias gcc="${GCC_BIN}/gcc-${GCC_VER}"
        alias cc="${GCC_BIN}/gcc-${GCC_VER}"
        alias g++="${GCC_BIN}/g++-${GCC_VER}"
        alias c++="${GCC_BIN}/g++-${GCC_VER}"
        alias gfortran="${GCC_BIN}/gfortran-${GCC_VER}"

        # --- Build-system variables (make, cmake, autoconf,␣
↪etc.) ------------
        export CC="${GCC_BIN}/gcc-${GCC_VER}"
        export CXX="${GCC_BIN}/g++-${GCC_VER}"
        export FC="${GCC_BIN}/gfortran-${GCC_VER}"
        export F77="${FC}"
    else
        echo "WARNING: Could not detect GCC version in ${GCC_
↪BIN}" >&2
    fi

fi
unset GCC_BIN
unset GCC_VER
```

Then apply the changes with:

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

If you use zsh:

Add the following to your `~/.zshrc`:

```
↪#============================================================================
# %%%%% Personal settings: Compilers (MacOS) %%%%%
#
# NOTE: MacOSX installs Clang as /usr/bin/gcc, so we have to␣
↪manually
# force reference to GNU compilers in /usr/local/opt/gcc/bin.

↪#============================================================================

# --- Homebrew GCC (override Apple Clang) -------------------
↪---------------
GCC_BIN="$(brew --prefix gcc)/bin"

if [[ -d "${GCC_BIN}" ]]; then
```

(continues on next page)

(continued from previous page)

```
    # Auto-detect the GCC major version by finding the gcc-NN␣
↪executable
    # Pipeline: find gcc-NN files --> extract 'gcc-NN' -->
    #           sort numerically --> take highest --> extract␣
↪number
    GCC_VER=$(find "${GCC_BIN}" -maxdepth 1 -name 'gcc-[0-9]*
↪' 2>/dev/null \
                | grep -oE 'gcc-[0-9]+$'  \
                | sort -t'-' -k2 -n       \
                | tail -1                 \
                | grep -oE '[0-9]+$')

    if [[ "x${GCC_VER}" != "x" ]]; then

        unset CC
        unset CXX
        unset FC

        # --- Interactive shell aliases ----------------------
↪------------------
        alias gcc="${GCC_BIN}/gcc-${GCC_VER}"
        alias cc="${GCC_BIN}/gcc-${GCC_VER}"
        alias g++="${GCC_BIN}/g++-${GCC_VER}"
        alias c++="${GCC_BIN}/g++-${GCC_VER}"
        alias gfortran="${GCC_BIN}/gfortran-${GCC_VER}"

        # --- Build-system variables (make, cmake, autoconf,␣
↪etc.) ------------
        export CC="${GCC_BIN}/gcc-${GCC_VER}"
        export CXX="${GCC_BIN}/g++-${GCC_VER}"
        export FC="${GCC_BIN}/gfortran-${GCC_VER}"
        export F77="${FC}"
    else
        echo "WARNING: Could not detect GCC version in ${GCC_
↪BIN}" >&2
    fi

fi
unset GCC_BIN
unset GCC_VER
```

Then apply the changes with:

```
$ source ~/.zshrc
```

3. Verify that your shell now recognizes the GNU gcc compiler

```
gcc --version
```

You should see output similar to:

```
gcc-15 (Homebrew GCC 15.3.0) 15.3.0
Copyright (C) 2025 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 confirms that your shell is using GNU **gcc** rather than **clang**.

Install and configure flex on macOS

If your macOS system does not have the **flex** library installed, install it with Homebrew:

```
brew install flex
```

Unlike Linux package managers, which install the flex library files under `/usr/lib/`, Homebrew installs them under its own prefix directory.

Define the `KPP_FLEX_LIB_DIR` environment variable using **brew --prefix** so that the path is determined automatically regardless of the Homebrew prefix or the installed flex version:

If you use bash:

Add the following to your `~/.bashrc` (or `~/.bash_profile`):

```
export KPP_FLEX_LIB_DIR=$(brew --prefix flex)/lib
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH}:${KPP_FLEX_LIB_
↪DIR}"
```

Then apply the changes with:

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

If you use zsh:

Add the following to your `~/.zshrc`:

```
export KPP_FLEX_LIB_DIR=$(brew --prefix flex)/lib
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH}:${KPP_FLEX_LIB_
↪DIR}"
```

Then apply the changes with:

```
$ source ~/.zshrc
```

Note

macOS uses `DYLD_LIBRARY_PATH` to locate shared libraries at runtime, not `LD_LIBRARY_PATH` (which is the Linux equivalent). If you find that `DYLD_LIBRARY_PATH` is blocked by System Integrity Protection (SIP), you can instead force a hard link:

```
brew link --force flex
```

Request maximum stack memory

macOS imposes a hard limit on stack memory of 65532 bytes, which is considerably less than what is available on typical GNU/Linux systems. To ensure that KPP uses the maximum available stack memory, add the following line to your shell startup file:

If you use bash:

Add this code to your `~/.bashrc` file:

```
ulimit -s 65532
```

Then apply the change with:

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

If you use zsh:

Add this code to your `~/.zshrc` file:

```
ulimit -s 65532
```

Then apply the change with:

```
$ source ~/.zshrc
```

This stack memory restriction means that KPP cannot parse mechanisms with more than approximately 2000 equations and 1000 species. To account for this, the KPP header file `src/gdata.h` defines the `MAX_EQN` and `MAX_SPECIES` parameters conditionally:

```
#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       18000   // Max number of equations
#define MAX_SPECIES    6000   // Max number of species
#endif
```

If KPP cannot parse your mechanism you can adjust `MAX_EQN` and `MAX_SPECIES` in `src/gdata.h` and then rebuild the KPP executable.

macOS is case-insensitive

> **Warning**
>
> macOS uses a case-insensitive filesystem by default. If your project contains two files whose names differ only in case (for example `integrator.F90` and `integrator.f90`), macOS will treat them as the same file. This can cause build failures and unexpected behaviour when working with version control systems such as Git. Ensure that no two files in your project share a case-insensitive name.

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*, and 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 definitions␣
↪    }
#include small_strato.eqn       { Includes file w/ chemical equations ␣
↪    }

#LOOKATALL                      { Output all species to small_strato.
↪dat}
```

(continues on next page)

(continued from previous page)

```
#MONITOR O3;N;O2;O;NO;O1D;NO2;  { Print selected species to screen    ␣
↪   }

#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 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-X.Y.Z.

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:
```

(continues on next page)

(continued from previous page)

```
    - 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
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 successfully 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
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 8192 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 overview

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 and a space "// ".
- Any name given by the user to denote an atom or a species is restricted to be less than 32 characters 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`).

5 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.

5.1 #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.

5.2 #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.

5.3 #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. To ignore mass balance checking for a given species, one can declare the predefined atom **IGNORE** as being part of the species composition. Examples for these sections are:

```
#DEFVAR
  NO2 = N + 2O;
```

(continues on next page)

(continued from previous page)

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

5.4 #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;
```

5.5 #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.

5.6 #INITVALUES

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

```
#INITVALUES
  CFACTOR = 2.5E+19;
  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.

> **Note**
>
> If you are building your mechanism in Fortran 90, note that using floating point constants such as `2.5E+19` will only give you a single precision value (which will incur roundoff error after the 6th or 7th decimal place.
>
> If you are using double precision, you should use Fortran double precision exponents (e.g. `2.5D+19`) instead. This will give you a true double precision value.

5.7 #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 one 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;
```

5.8 #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;
```

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

6.1 #AUTOREDUCE

The **#AUTOREDUCE ON** command can be used with **#INTEGRATOR rosenbrock** to enable *automatic mechanism reduction* as described in Lin *et al.* [2023]. Automatic mechanism reduction is disabled by default.

6.2 #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-optimized code.

6.3 #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.

6.4 #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.

6.5 #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
```

6.6 #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**.

6.7 #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.

6.8 #GRAPH

Off by default, the **#GRAPH** command leverages the pre-existing parsing of stoichiometric structure to create graph-based representations of the chemical mechanism. **#GRAPH** currently supports two options, described below.

#GRAPH stoic

This option generates two CSV files describing the structure of the mechanism based on stoichiometric balances and species composition.

1. `ROOT_BiadjacencyMatrix.csv`: A sparse matrix of the net stoichiometric coefficients for each species in each reaction.

Table 2: Structure of `ROOT_BiadjacencyMatrix.csv`

Column	Header	Description
1	spc_name	Species name.
2	species_index	Number of the species in the chemical mechanism
3	reaction_index	Number of the reaction in the chemical mechanism
4	stoichiometric_coefficient	The net, weighted biadjacency matrix of the bipartite species-reaction graph, useful for automated detection of system-wide stoichiometric invariants and other mass conservation enforcing [Sturm and Wexler, 2022].

2. `ROOT_SpeciesCompositionMatrix.csv`: This is a species composition dense matrix that for each variable species (row), records the atom count of each element present (columns).

Table 3: Structure of `ROOT_SpeciesCompositionMatrix.csv`

Column	Name	Description
1	species_index	Number of the species in the chemical mechanism
2	species_name	Species name.
3 - NSPEC	Species matrix	One column for each atom present in the mechanism (including IGNORE if used for some species). This matrix can be used for automated mass balance checking of mechanism output or individual reactions [Sturm and Silva, 2025].

#GRAPH edgelist

The *#GRAPH stoic* option uses net stoichiometry, which neglects reciprocal reactions (e.g. when the same species is a reactant and a product its net stoichiometry is zero). This **`#GRAPH edgelist`** option generates a `ROOT_Edgelist.csv` file listing the edges of the species-reaction bipartite graph. Each row represents a connection between a species and a reaction, with direction and stoichiometric value.

Column	Name	Description
1	species_index	Species index (starts from 1)
2	reaction_index	Reaction index (starts from 1)
3	From	Starting edge
4	To	Ending edge
5	Stoichiometric value	Directed stoichiometric value

6.9 #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.

6.10 #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.

6.11 #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
```

6.12 #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.

6.13 #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.

6.14 #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*).

6.15 #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.

6.16 #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.

6.17 #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.

6.18 #REORDER

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

6.19 #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**.

6.20 #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*).

6.21 #CHECKALL, #LOOKATALL

The shorthand commands **#CHECKALL** and **#LOOKATALL** apply **#CHECK** and **#LOOKAT**, respectively, to all species in the mechanism.

6.22 #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.

7 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.

7.1 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 4: 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	rate coefficient definitions
F90_RCONST_USE	*ROOT_Rates*	subroutine	rate coefficient definitions
F90_UTIL	*ROOT_Util*	executable section	utility functions

7.2 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.

7.3 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
```

(continues on next page)

(continued from previous page)

```
    ! ... 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.

7.4 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
```

7.5 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
```

7.6 F90_RCONST

This inline type can be used to define time-dependent values of rate coefficients. You may inline 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
```

The inlined code will be placed directly into the subroutines UPDATE_RCONST and UPDATE_PHOTO in the *ROOT_Rates* file.

7.7 F90_RCONST_USE

Similar to *F90_RCONST*, but allows you to inline Fortran-90 USE statements referencing modules where rate coefficients are computed, such as:

```
#INLINE F90_RCONST_USE
  USE MyRateFunctionModule
#ENDINLINE
```

The inlined code will be placed directly into the subroutines UPDATE_RCONST and UPDATE_PHOTO in the *ROOT_Rates* file. USE statements will be placed before Fortran variable definitions and executable statements, as is required by the Fortran-90 language standard.

7.8 F90_UTIL

This inline type can be used to define utility subroutines.

8 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.

8.1 List of auxiliary files for Fortran90

Table 5: 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` and `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.

8.2 List of symbols replaced by the substitution preprocessor

Table 6: 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	19
KPP_FUN_OR_FUN_S	Name of the function to be called	FUN(Y,FIX, RCONST,Ydot)

9 Filtering out passive species with an absolute tolerance threshold

In some external models that use KPP (such as GEOS-Chem) it is common to attach "passive" species to reactions. This is usually done to obtain production or loss rates for diagnostic purposes.

We have created a simple passive species example C-I test (see the chemical mechanism description file `ci-tests/F90_ros_passivespc/F90_ros_passivespc.kpp`). In this example we add 3 passive species (`PASV1`, `PASV2`, `PASV3`) to the `small_strato` mechanism:

```
#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      + PASV1  : (1.200E-10);
<R8>  NO   + O3 = NO2 + O2 + PASV2  : (6.062E-15);
<R9>  NO2  + O  = NO  + O2          : (1.069E-11);
<R10> NO2  + hv = NO  + O  + PASV3  : (1.289E-02) * SUN;
```

Passive species, however, should not be allowed to influence certain computations (such as the Rosenbrock error norm), as this may lead to numerical instability. In the case of the Rosenbrock error norm, we have found that results of key species (such as O3 and NO) can vary depending on the number of passive species that are added to the mechanism.

In KPP 3.5.0 we have introduced the capability to filter out passive species. This feature is currently only available when generating Fortran90 source code with KPP. You may do this as follows:

First, give each passive species an extremely high absolute tolerance (ATOL) value:

```
USE ROOT_Parameters

...

ATOL(ind_PASV1) = 1.0d25
ATOL(ind_PASV2) = 1.0d25
ATOL(ind_PASV3) = 1.0d25
```

Next, pass the optional argument PassiveSpc_ATOL_Threshold to the *Initialize* routine (this needs to be done only once at the model initialization phase):

```
CALL Initialize( PassiveSpc_ATOL_Threshold = 1.0d25 )
```

Any species having a value of ATOL greater than or equal to PassiveSpc_ATOL_Threshold will be denoted as a passive species. Subroutine *Initialize* will then generate the number and list of non-passive species in the following variables, which are stored in the *ROOT_Global* module:

```
! NonPassiveSpc_Count - Number of non-passive species in mechanism
  INTEGER :: NonPassiveSpc_Count
! NonPassiveSpc_Indices - Indices of non-passive species in mechanism
  INTEGER :: NonPassiveSpc_Indices(NVAR)
```

You may then use these variables to exclude passive species from certain computations, such as:

```
USE ROOT_Global

...

INTEGER :: NP, I

! Loop over the number of non-passive species
DO NP = 1, NonPassiveSpc_Count

   ! Only use non-passive species in computation
   I   = NonPassiveSpc_Indices(NP)
   Err = ATOL(I) * RTOL(I) * YMAX

   ... etc ...

ENDDO
```

10 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 beginning 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.

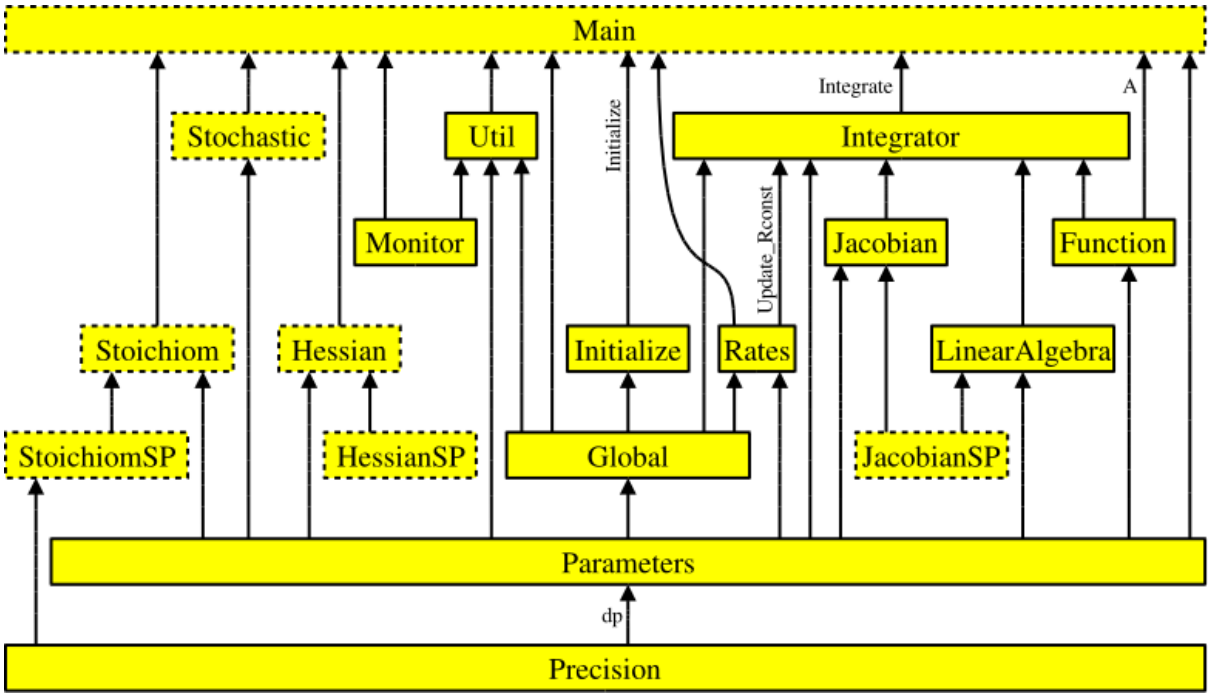


Fig. 1: Figure 1: Interdependencies of the KPP-generated files. Each arrow starts at the module that exports a variable or subroutine and points to the module that imports it via the Fortran90 `USE` instruction. The prefix `ROOT_` has been omitted from module names for better readability. Dotted boxes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

10.1 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.

10.2 ROOT_Model

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

10.3 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.

> **Note**
>
> In KPP 3.5.0 and later versions, you may pass the optional `PassiveSpc_ATOL_Threshold` to subroutine `Initialize` to prevent "passive" species intended for diagnostic purposes from being used in certain computations (such as the Rosenbrock error norm). At present this option is limited to Fortran90. For more information, please see the *Filtering out passive species with an absolute tolerance threshold* section.

10.4 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`.

10.5 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' ) )
```

10.6 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.

10.7 ROOT_Rates

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

Table 7: Fortran90 subroutines in ROOT_Rates

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

10.8 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.

10.9 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
`NonPassiveSpc_Count`	Number of non-passive species (cf. *Filtering out passive species with an absolute tolerance threshold*)
`NonPassiveSpc_Indices`	List of species indices for only the non-passive species

Both variable and fixed species are stored in the one-dimensional array `C`. The first part (indices from `1` to `NVAR`) contains the variable species, and the second part (indices from `NVAR+1` to `NSPEC`) the fixed species. The total number of species 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[11] 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.

10.10 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 `NVAR`. 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 output 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)
```

(continues on next page)

[11] https://openmp.org

(continued from previous page)

```
  REAL(kind=dp) :: F(NVAR)
! RCT - Rate constants (local)
  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
```

10.11 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.

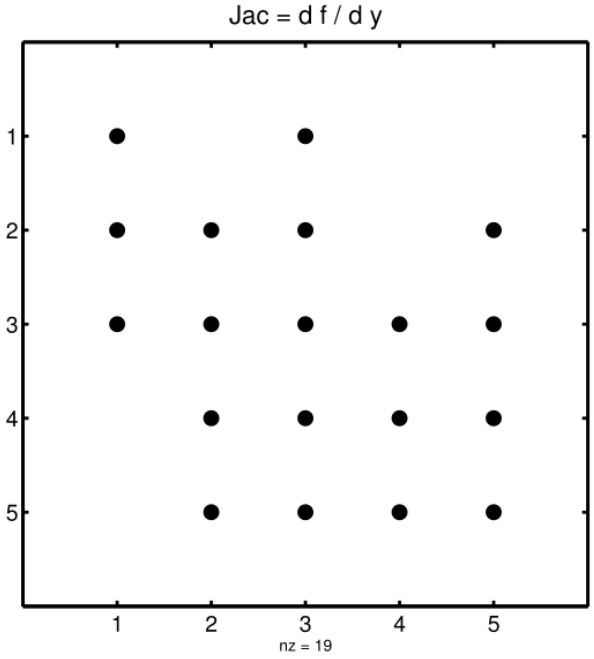


Fig. 2: Figure 2: The sparsity pattern of the Jacobian for the **`small_strato`** example. All nonzero elements are marked with a bullet. Note that even though $\mathbf{J}(3,5)$ is zero, it is also included here because of the fill-in.

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

10.12 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

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

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.*

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*.

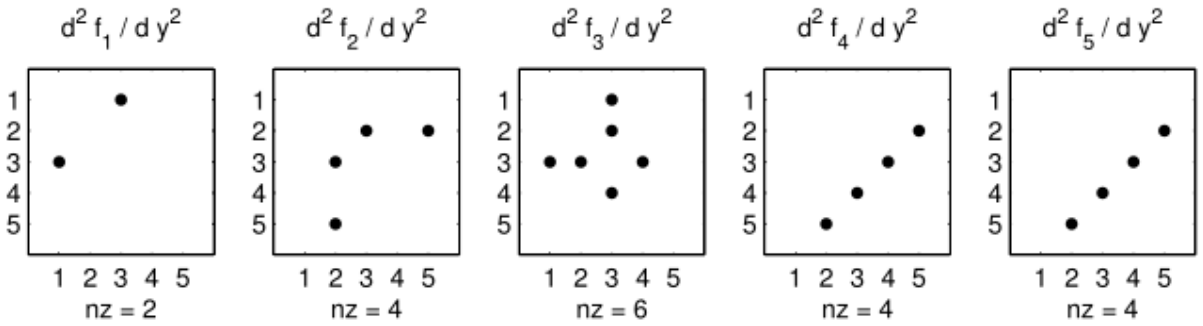


Fig. 3: Figure 3: The Hessian of the small_strato example.

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_K(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_I`, `IHESS_J` and `IHESS_K` are computed by KPP and initialized statically. They are constant as the sparsity pattern of the Hessian does not change during the computation.

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

10.13 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 referred 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 substitution
`KppSolveTR`	Transposed sparse back substitution

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

10.14 ROOT_Stoichiom and ROOT_StoichiomSP

These files 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_StoichiomSP.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 `IROW_STOICM` and `ICOL_STOICM` respectively. The vector `CCOL_STOICM` contains pointers to the start of each column. For example column `j` starts in the sparse vector at position `CCOL_STOICM(j)` and ends at `CCOL_STOICM(j+1)-1`. The last value `CCOL_STOICM(NREACT+1) = NSTOICM+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

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

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:

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

10.15 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`).

10.16 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_Opt`	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*).

10.17 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.

11 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. *Global parameters* are `#include`-d in the header file `ROOT_Parameters.h`.
2. *Global variables* are extern-declared in `ROOT_Global.h` and declared in the driver file `ROOT.c`.
3. Extern declarations of sparse data structures for the *Jacobian*, *Hessian* and *stoichiometric matrix*, and the *Jacobian of reaction products* are in the header file `ROOT_Sparse.h`. The actual declarations of each data structure is done in the corresponding files.

The code for the *ODE function* is in `ROOT_Function.c`.

The *ODE Jacobian and sparse multiplications* 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* are in `ROOT_Hessian.c`, and the declaration and initialization of Hessian sparse data structures are in `ROOT_HessianSP.c`.

Functions for *reactant products and its Jacobian, and derivatives with respect to rate coefficients* are in the file `ROOT_Stoichiom.c`.

The declaration and initialization of the *stoichiometric matrix and the associated sparse data structures* is done in `ROOT_StoichiomSP.c`.

Sparse linear algebra routines 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 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 are also generated in the files `ROOT_mex_Fun.c`, `ROOT_mex_Jac_SP.c`, and `ROOT_mex_Hessian.c`.

12 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[12] 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.

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.

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

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_stoichiomSP.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 solar intensity
ROOT_GetMass.m	Check mass balance for selected atoms
ROOT_Initialize.m	Set initial values
ROOT_Shuffle_kpp2user.m	Shuffle concentration vector
ROOT_Shuffle_user2kpp.m	Shuffle concentration vector

13 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.

Use this command to compile KPP-generated source code into an executable:

```
$ make -f Makefile_ROOT [target]
```

Where [target] is one of the following optional values:

13.1 all

Synonym for *exe*.

13.2 COMPILER=GFORTRAN

Builds KPP-generated Fortran-90 code into an executable with the GNU **gfortran** compiler.

This is the default setting for when **#LANGUAGE Fortran90** is specified.

13.3 COMPILER=GCC

Builds KPP-generated C code into an executable with the GNU **gcc** compiler.

This is the default setting for when **#LANGUAGE C** is specified.

13.4 COMPILER=INTEL

Compiles Fortran-90 code generated by KPP into an executable with the GNU **gfortran** compiler.

Compiles C code generated by KPP into an executable with the Intel **icc** compiler.

13.5 exe

Compiles KPP-generated source code into an executable. This is the default option.

13.6 mex

Compiles Matlab extensions into the executable when *#MEX ON* is specified.

13.7 clean

Removes the following files:

- Object files (`*.o`)
- Fortran90 module files (`*mod`)
- Executable files (`*.exe`)
- *The log file* (`ROOT.log`)
- Species concentrations specified with *#LOOKAT* (`*.dat`)
- Comma-separated variable files generated by *#GRAPH* (`*.csv`)
- Matlab extensions generated by *#MEX* (`ROOT*.mexglx`)
- Output from the Matlab executable (`results.m`)

13.8 distclean

Removes all of the files removed by *clean*, as well as:

- KPP-generated C source code (`ROOT*.c`)
- KPP-generated Fortran90 source code (`ROOT*.f90`, `ROOT*.F90`)
- KPP-generated Matlab source code (`ROOT*.m`)

14 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.

15 Rosenbrock methods

Integrator file: `int/rosenbrock.f90`

An s-stage Rosenbrock method (cf. Section IV.7 in 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:

15.1 ROS-2

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

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

15.2 ROS-3

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

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

15.3 ROS-4

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

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

15.4 RODAS-3

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

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

15.5 RODAS-3.1

- Reference: Long *et al.* [2026, submitted]
- Stages (s): 4
- Function calls: 3
- Order: 3(2)
- Stability properties: Stiffly-accurate
- Method Coefficients:

$$
\begin{aligned}
a_{2,1} &= 0 & a_{3,1} &= 0.646601929740551 \\
a_{3,2} &= 0.409567801987914 & a_{4,1} &= 0.646601929740551 \\
a_{4,2} &= 0.409567801987914 & a_{4,3} &= 1 \\
c_{2,1} &= 4.198495621784201 & c_{3,1} &= 3.711590161613010 \\
c_{3,2} &= -1.787771994729384 & c_{4,1} &= 4.458898153216104 \\
c_{4,2} &= -2.024095448516552 & c_{4,3} &= -2.626700600119396 \\
m_1 &= 0.646601929740551 & m_2 &= 0.409567801987914 \\
m_3 &= 1 & m_4 &= 1 \\
e_1 &= 0 & e_2 &= 0 \\
e_3 &= 0 & e_4 &= 1 \\
\alpha_1 &= 0 & \alpha_2 &= 0 \\
\alpha_3 &= 1 & \alpha_4 &= 1 \\
\gamma_1 &= 0.515 & \gamma_2 &= 1.628546001287715 \\
\gamma_3 &= 0 & \gamma_4 &= 0
\end{aligned}
$$

15.6 RODAS-4

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

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

15.7 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.

15.8 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*).

15.9 Rosenbrock with mechanism auto-reduction

Integrator file: `int/rosenbrock_autoreduce.f90`

Mechanism auto-reduction (described in Lin *et al.* [2023]) expands previous work by Santillana *et al.* [2010] and 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.* [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.

15.10 Rosenbrock with H211b time stepping

Integrator file: `int/rosenbrock_h211b_qssa.f90`

H211b time stepping according to Söderlind [2003], as implemented by Dreger *et al.* [2025].

16 Runge-Kutta (aka RK) methods

A general s-stage Runge-Kutta method is defined as (see Section II.1 of Hairer *et al.* [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:

16.1 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

16.2 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 [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.

16.3 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 [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

16.4 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 [1991].

16.5 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 [1991].

16.6 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.

16.7 RK discrete adjoint model

The first order Runge-Kutta adjoint is

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

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

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

17 Backward differentiation methods

Backward differentiation formulas (BDF) are linear multistep methods with excellent stability properties for the integration of chemical systems (cf. 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:

17.1 LSODE

Integrator file: `int/lsode.f90`

LSODE, the Livermore ODE solver (Radhakrishnan and Hindmarsh [1993]), implements backward differentiation formula (BDF) methods for stiff problems. LSODE has been translated to Fortran90 for the incorporation into the KPP library.

> **Attention**
>
> We have discovered that the current implementation of the LSODE integrator is not thread-safe for OpenMP parallelization[13]. When LSODE is called from within an OpenMP parallel loop, the integration will fail because key internal variables in LSODE will be overwritten by concurrent threads.

[13] https://www.openmp.org/

17.2 VODE

Integrator file: `int/dvode.f90`

VODE (Brown *et al.* [1989]) uses another formulation of backward differentiation formulas. The version of VODE present in the KPP library uses directly the KPP sparse linear algebra routines.

17.3 BEULER

Integrator file: `int/sdirk.f90`

Backward Euler integration method. To request this method, make sure you select

```
#INTEGRATOR sdirk
```

in your definition file, and then set `ICNTRL(3) = 6`.

18 Forward differentiation methods

18.1 FEULER

Integrator file: `int/feuler.f90`

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

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

19 Integrator options (`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/`.

19.1 ICNTRL

Table 20: 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	18
beuler		Y	Y	Y	Y	Y	s								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			
rosen-brock	Y	Y	Y	Y											Y	Y		
rosen-brock_tlm	Y	Y	Y	Y								s			Y			
rosen-brock_autc	Y	Y	Y	Y								s	s	s	Y	Y		
rosen-brock_h21	Y	Y	Y	Y											Y	Y		s
runge_kutt		Y	Y	Y	Y	s	s	s	s	s	Y				Y			
runge_kutt		Y	Y	Y	Y	Y				s	Y				Y			
runge_kutt		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)

Specifies the time-dependence of the function F to be integrated:

ICNTRL(1)	Time dependence of F
0	$F = F(t, y)$ aka non-autonomous
1	$F = F(y)$ aka autonomous

ICNTRL(2)

Specifies the dimensionality of the absolute (`ATOL`) and relative (`RTOL`) tolerances. These can be expressed by either a scalar or individually for each species in a vector.

ICNTRL(2)	Dimensionality of `ATOL` and `RTOL`
0	`NVAR` -dimensional vector
1	scalar

ICNTRL(3)

Selects a specific integration method.

Integrator	ICNTRL(3)	Method selected
rosenbrock	0 or 4	Rodas3 (default)
rosenbrock_adj	1	Ros2
rosenbrock_autoreduce	2	Ros3
rosenbrock_tlm	3	Ros4
	5	Rodas4
	6	Rang3
	7	Rodas3.1
runge_kutta	0 or 1	Radau-2A (default)
runge_kutta_adj	2	Lobatto-3C
runge_kutta_tlm	3	Gauss
	4	Radau-1A
	5	Lobatto-3A
sdirk	0 or 1	Sdirk-2A (default)
sdirk_adj	2	Sdirk-2B
sdirk_tlm	3	Sdirk-3A
	4	Sdirk-4A
	5	Sdirk-4B
	6	Backward Euler

ICNTRL(4)

Specifies the maximum number of integration steps.

ICNTRL(5)

Specifies the maximum number of Newton iterations.

ICNTRL(6)

Selects integrator-specific settings.

Integrator	ICNTRL(6)	Option selected
rosenbrock_adj	0 thru 6	Selection of a particular Rosenbrock method for the continuous adjoint integration (see `ICNTRL(3)`)
radau5 runge_kutta_ runge_kutta_	0	Starting values for Newton iterations are interpolated (default)
sdirk sdirk_adj sdirk_tlm	1	Starting values for Newton iterations are zero

ICNTRL(7)

Selects options for adjoint integrators.

Table 21: Selection of adjoint algorithm

Integrator	ICNTRL(7)	Adjoint algorithm selected
rosenbrock_adj	0 or 2	Discrete adjoint with method `ICNTRL(3)` (default)
	1	No adjoint
	3	Fully adaptive continuous adjoint with method `ICNTRL(6)`
	4	Simplified continuous adjoint with method `ICNTRL(6)`

Table 22: Method to solve the linear Adj equations

Integrator	ICNTRL(7)	Method selected
runge_kutta_	0 or 1	Modified Newton re-using LU (default)
	2	Direct solution (additional one LU factorization of 3Nx3N matrix per step; good for debugging
	3	Adaptive solution (if Newton does not converge, switch to direct)
sdirk_adj	0	Modified Newton re-using LU (default)
sdirk_tlm	1	Direct solution (additional one LU factorization per stage)

ICNTRL(8)

Determines if LU factorization will be checkpointed at each step (for adjoint integrators only).

Integrator	ICNTRL(8)	Option selected
rosenbrock_adj runge_kutta_adj	0	Do not save LU factorization at each step (default)
sdirk_adj	1	Save LU factorization at each step

ICNTRL(9)

Selects options for adjoint and tangent linear method (TLM) integrators.

Table 23: Selection of adjoint algorithm

Integrator	ICNTRL(9)	Adjoint algorithm selected
runge_kutta_adj	0 or 2	Discrete adjoint with method `ICNTRL(3)` (default)
	1	No adjoint
	3	Fully adaptive continuous adjoint with method `ICNTRL(6)`
	4	Simplified continuous adjoint with method `ICNTRL(6)`

Table 24: Selection of tangent linear method (TLM) error estimation strategy

Integrator	ICNTRL(9)	Strategy selected
runge_kutta_t	0	Base number of iterations as forward solution
sdirk_tlm	1	USE `ATOL_tlm` and `RTOL_tlm` to calculate error estimation for TLM at Newton stages

ICNTRL(10)

Selects integrator-specific options.

Integrator	ICNTRL(10)	Option selected
lsode	user supplied	Maximum order of the integration formula allowed
runge_kutta	0	Error estimation strategy: one additional stage at `c=0` (default)
runge_kutta_ runge_kutta_	1	Error estimation strategy: Two additional stages at `c=0` and SDIRK at `c=1`, stiffly accurate
seulex	0	No verbose output
	1	Dense verbose output

ICNTRL(11)

Selects integrator-specific settings.

Integrator	ICNTRL(11)	Option selected
seulex	user supplied	Maximum number of columns in the extrapolation. Default is 12.
radau5 runge_kutta	0	Gustaffson step size control
runge_kutta_adj runge_kutta_tlm	1	Classical step size control

ICNTRL(12)

Selects integrator-specific settings.

Integrator	ICNTRL(12)	Option selected
rosenbrock_autoreduc	0	Disable mechanism auto-reduction (i.e. acts in the same way as the `rosenbrock` integrator)
	1	Enables mechanism auto-reduction, set threshold in `RCNTRL(12)`
rosenbrock_tlm	0	TLM truncation error is not used
runge_kutta_tlm	1	TLM truncation error is computed and used
sdirk_tlm		
seulex	0	Nsequence = 2 (default)
	1	Nsequence = 1,2,3,4,6,8,12,16,24,32,48,...
	2	Nsequence = 2,3,4,6,8,12,16,24,32,48,64,...
	3	Nsequence = 1,2,3,4,5,6,7,8,9,10,...
	4	Nsequence = 2,3,4,5,6,7,8,9,10,11,...

ICNTRL(13)

Selects integrator-specific settings.

Integrator	ICNTRL(13	Option selected
rosenbrock_autorec	0	In auto-reduction, disables scanning species P and L rates throughout the internal timesteps of the integrator.
	1	In auto-reduction, enables scanning species P and L rates throughout the internal timesteps of the integrator, for repartitioning.
seulex	0, 1	Sets the `lambda` parameter for verbose output.

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)

Determines which `Update_*` subroutines are called within the integrator.

ICNTRL(15)	Option selected
-1	Do not call any `Update_*` subroutines
0	Use the integrator-specific default values
1	Call `Update_RCONST` from within the integrator.
2	Call `Update_PHOTO` from within the integrator
3	Call `Update_RCONST` and `Update_PHOTO` from within the integrator.
4	Call `Update_SUN` from within the integrator
5	Call `Update_SUN` and `Update_RCONST` from within the integrator
6	Call `Update_SUN` and `Update_PHOTO` from within the integrator.
7	Call `Update_SUN`, `Update_PHOTO`, and `Update_RCONST` from within the integrator.

Calling `Update_RCONST` may be necessary when reaction rate coefficients depend on the concentration of a specific species, e. g.:

```
HSO3m + HSO5m + Hp = 2 HSO4m + Hp : k_aqueous( C(ind_Hp) );
```

This ensures that the concentration `C(ind_Hp)` at the specific integration time is used when the reaction rate coefficient is updated within the integrator.

ICNTRL(16)

Specifies how negative values should be handled.

ICNTRL(16)	Option selected
0	Leave negative values unchanged
1	Set negative values to zero
2	Print warning and continue
3	Print error message and stop

ICNTRL(17)

Selects the amount of verbose output that will be generated.

ICNTRL(17)	Option selected
0	Only return error number
1	Full verbose error output

ICNTRL(18)

Currently not used.

ICNTRL(19)

Currently not used.

ICNTRL(20)

Currently not used.

19.2 RCNTRL

Table 25: 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_a	Y	Y	Y	Y	Y	Y	Y												
rosenbrock	Y	Y	Y	Y	Y	Y	Y												
rosenbrock_t	Y	Y	Y	Y	Y	Y	Y												
rosenbrock_a	Y	Y	Y	Y	Y	Y	Y					s		s					
rosenbrock_l	Y	Y	Y	Y	Y	Y	Y								s	s	s	s	
runge_l	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
runge_l	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
runge_l	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk4	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk_a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y								
sdirk_t	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_lea																			

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`

Different, solver-specific settings for `seulex`.

RCNTRL(11)

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

Different, solver-specific settings for `seulex`.

RCNTRL(12)

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

Different, solver-specific settings for `seulex`.

RCNTRL(13)

Solver-specific settings for `seulex`.

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)`.

Different, solver-specific settings for `seulex`.

RCNTRL(15) - RCNTRL(18)

Solver-specific settings for `rosenbrock_h211b_qssa`.

Different, solver-specific settings for `seulex`.

RCNTRL(19)

Solver-specific settings for `seulex`.

RCNTRL(20)

Currently not used.

20 Integrator status (`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/`.

20.1 ISTATUS

Table 26: 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	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_adj	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_autoreduce	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_h211b_qssa	Y	Y	Y	Y	Y	Y	Y	Y	
rosenbrock_tlm	Y	Y	Y	Y	Y	Y	Y	Y	Y
runge_kutta	Y	Y	Y	Y	Y	Y	Y	Y	
runge_kutta_adj	Y	Y	Y	Y	Y	Y	Y	Y	
runge_kutta_tlm	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk_adj	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk_tlm	Y	Y	Y	Y	Y	Y	Y	Y	
sdirk4	Y	Y	Y	Y	Y	Y	Y	Y	
seulex	Y	Y	Y	Y	Y	Y	Y		
tau_leap									

ISTATUS(1)

Number of times that the *function containing the chemical ODE system* was called.

ISTATUS(2)

Number times *the Jacobian matrix* was constructed.

ISTATUS(3)

Total number of integration timesteps.

ISTATUS(4)

Number of accepted timesteps.

ISTATUS(5)

Number of rejected timesteps (except at very beginning).

ISTATUS(6)

Number of LU decompositions that were performed.

ISTATUS(7)

Number of forward/backward substitutions that were performed.

ISTATUS(8)

Number of singular matrix decompositions that were performed.

ISTATUS(9)

Number of times *the Hessian matrix* was evaluated.

ISTATUS(10) .. ISTATUS(20)

Currently not used.

20.2 RSTATUS

Table 27: 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	Y			
gillespie				
lsode	Y	Y		
radau5				
rosenbrock	Y	Y	Y	
rosenbrock_adj	Y	Y	Y	
rosenbrock_autoreduce	Y	Y	Y	s
rosenbrock_h211b_qssa	Y	Y	Y	
rosenbrock_tlm	Y	Y	Y	
runge_kutta	Y	Y	Y	
runge_kutta_adj	Y	Y	Y	
runge_kutta_tlm	Y	Y	Y	
sdirk	Y	Y	Y	
sdirk_adj	Y	Y	Y	
sdirk_tlm	Y	Y	Y	
sdirk4	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.

21 KPP directory structure

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

21.1 src/

Contains the KPP source code files:

Table 28: 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

21.2 bin/

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

21.3 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.

21.4 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.

21.5 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.

21.6 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.

21.7 examples/

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

21.8 site-lisp/

Contains the file `kpp.el` which provides a KPP mode for emacs with color highlighting.

21.9 ci-tests/

Contains directories defining several *Continuous integration tests*.

21.10 .ci-pipelines/

Hidden directory containing a YAML file with settings for automatically running the continuous integration tests on Azure DevOps Pipelines[14]

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*.

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

21.11 .github/workflows

Contains configuration files for GitHub Actions[15] that will run automatically when commits are pushed or when pull requests are opened.

22 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 variables that control the places where KPP looks for module files, integrators, and drivers:

22.1 KPP_HOME

Required, stores the absolute path to the KPP distribution.

Default setting: none.

22.2 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.

22.3 KPP_MODEL

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

Default setting: `$KPP_HOME/models`.

22.4 KPP_INT

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

Default setting: `$KPP_HOME/int`.

22.5 KPP_DRV

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

Default setting: `$KPP_HOME/drv`.

[15] https://github.com/KineticPreProcessor/KPP/actions

23 KPP internal modules

23.1 Scanner and parser

This module is responsible for reading the kinetic description files and extracting the information necessary in the code generation phase. We make use of the flex and 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.

23.2 Species reordering

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

23.3 Expression trees computation

This is the core of the preprocessor. This module generates the production/destruction functions, the Jacobian and all the data structure needed 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.

23.4 Code generation

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

24 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:

1. 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 ); }`
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.
3. Add documentation to 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*
4. Add command to `site-lisp/kpp.el` (Emacs configuration file)

25 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.

C-I tests will run automatically as a GitHub Action[16] when commits are pushed to the KPP Github repository[17], or when new pull requests are opened. You may also run the integration tests *locally on your own computer*.

25.1 List of continuous integration tests

Table 29: Continuous integration tests

C-I test	Language	Model	Integrator	What it tests
`C_rk`	C	small_strato	runge_kutta	*Runge-Kutta integrator*
`C_rosadj`	C	small_strato	rosenbrock_adj	*Rosenbrock discrete adjoin*
`C_sd`	C	small_strato	sdirk	*SDIRK integrator*
`C_sdadj`	C	small_strato	sdirk_adj	*SDIRK discrete adjoint*
`C_small_strato`	C	small_strato	rosenbrock	*Rosenbrock integrator*
`F90_feuler`	Fortran90	carbon	feuler	*Forward Euler integrator*
`F90_graph`	Fortran90	small_strato	rosenbrock	*#GRAPH command*
`F90_lsode`	Fortran90	small_strato	lsode	*LSODE integrator*
`F90_mcm`	Fortran90	mcm	rosenbrock	Master Chemical Mechanis
`F90_mcm_h211b`	Fortran90	mcm	rosenbrock_h211b_qssa	*Rosenbrock int. w/ H211b*
`F90_radau`	Fortran90	saprc99	radau5	*RADAU5 integrator*
`F90_rk`	Fortran90	small_strato	runge_kutta	*Runge-Kutta integrator*
`F90_rkadj`	Fortran90	small_strato	runge_kutta_adj	*Runge-Kutta discrete adjoi*
`F90_rktlm`	Fortran90	small_strato	runge_kutta_tlm	*Runge-Kutta tangent linear*
`F90_ros`	Fortran90	small_strato	rosenbrock	*Rosenbrock integrator*
`F90_rosadj`	Fortran90	small_strato	rosenbrock_adj	*Rosenbrock discrete adjoin*
`F90_ros_autoreduce`	Fortran90	saprc99	rosenbrock_autoreduce	*Rosenbrock integrator w/ a*
`F90_rosenbrock`	Fortran90	saprc99	rosenbrock	*Rosenbrock integrator*
`F90_ros_h211b`	Fortran90	saprc99	rosenbrock_h211b_qssa	*Rosenbrock int. w/ H211b*
`F90_ros_passivespc`	Fortran90	small_strato	rosenbrock	Excluding passive species
`F90_ros_split`	Fortran90	small_strato	rosenbrock	*#FUNCTION SPLIT*
`F90_rostlm`	Fortran90	small_strato	rosenbrock_tlm	*Rosenbrock tangent linear*
`F90_ros_upcase`	Fortran90	saprc99	rosenbrock	*#UPPERCASEF90 ON*
`F90_saprc_2006`	Fortran90	saprcnov	rosenbrock	*Rosenbrock integrator*
`F90_sd4`	Fortran90	small_strato	sdirk4	*Rosenbrock integrator*
`F90_sd`	Fortran90	small_strato	sdirk	*SDIRK integrator*
`F90_sdadj`	Fortran90	small_strato	sdirk_adj	*SDIRK discrete adjoint*
`F90_sdtlm`	Fortran90	small_strato	sdirk_tlm	*SDIRK tangent linear mode*
`F90_seulex`	Fortran90	saprcnov	seulex	*SEULEX integrator*
`F90_small_strato`	Fortran90	small_strato	rosenbrock	*Rosenbrock integrator*
`X_minver`	Fortran90	small_strato	runge_kutta	*#MINVERSION command*

Notes about C-I tests:

1. `F90_ros_split` uses **`#FUNCTION SPLIT`**.
2. `F90_ros_upcase` uses **`#UPPERCASEF90 ON`**.

[16] https://github.com/KineticPreProcessor/KPP/actions
[17] https://github.com/KineticPreProcessor/KPP

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`).

25.2 Running continuous integration tests as a GitHub Action

The files needed to run the C-I tests are described below.

run-ci-tests.yml

Path: `$KPP_HOME/.github/workflows/run-ci-tests.yml`

Description: Configuration file with commands to download KPP, load libraries, and run the C-I tests as a GitHub Action.

C-I tests will run automatically when a commit is pushed to any branch at https://github.com/KineticPreProcessor/KPP, or when a new pull request is opened there. 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.

ci-testing-script.sh

Path: `$KPP_HOME/.ci-pipelines/ci-testing-script.sh`

Description: Runs the KPP C-I tests as a GitHub Action, or on a local computer system.

ci-cleanup-script.sh

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

Description: Removes compiler-generated files (e.g. `*.o`, `.mod` , and `.exe`) from C-I test folders.

ci-common-defs.sh

Path: `$KPP_HOME/.ci-pipelines/ci-common-defs.sh`

Description Contains common variable and function definitions needed by *ci-testing-script.sh* and *ci-cleanup-script.sh*.

25.3 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`.

26 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                 ␣
 ↪          |
                            #LOOKAT species_list atom_list              ␣
 ↪          |
                            #MONITOR species_list atom_list             ␣
 ↪          |
                            #SETFIX species_list_plus                   ␣
 ↪          |
                            #SETVAR species_list_plus

command ::=                 #CHECKALL                                   ␣
 ↪          |
                            #DECLARE [ SYMBOL | VALUE ]                 ␣
 ↪          |
                            #DOUBLE [ ON | OFF ]                        ␣
 ↪          |
                            #DRIVER driver_name                         ␣
 ↪          |
                            #DUMMYINDEX [ ON | OFF ]                    ␣
 ↪          |
```

(continues on next page)

(continued from previous page)

```
                              #EQNTAGS [ ON | OFF ]                          ␣
↪           |
                              #FUNCTION [ AGGREGATE | SPLIT ]                ␣
↪           |
                              #GRAPH [ OFF | STOIC | EDGELIST ]              ␣
↪           |
                              #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 ]                       ␣
↪           |
                              #STOICMAT [ 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
```

(continues on next page)

(continued from previous page)

```
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_0-9]+".

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

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

initvalues_list ::=        initvalues_assignment
 ↪          |
                           initvalues_assignment initvalues_list

inline_type ::=            F90_RATES    | F90_RCONST    | F90_GLOBAL
 ↪          |
                           F90_INIT     | F90_DATA      | F90_UTIL
 ↪          |
                           F77_RATES    | F77_RCONST    | F77_GLOBAL
 ↪          |
                           F77_INIT     | F77_DATA      | F77_UTIL
 ↪          |
                           C_RATES      | C_RCONST      | C_GLOBAL
 ↪          |
                           C_INIT       | C_DATA        | C_UTIL
 ↪          |
                           MATLAB_RATES | MATLAB_RCONST | MATLAB_
↪GLOBAL          |
                           MATLAB_INIT  | MATLAB_DATA   | MATLAB_UTIL
```

(continues on next page)

(continued from previous page)

```
rate ::=                    number                                          ␣
 ↪          |
                            program_expression

species_composition ::=     atom_count                                      ␣
 ↪          |
                            atom_count + species_composition                ␣
 ↪          |
                            IGNORE

species_definition ::=      species_name = species_composition;

species_definition_list := species_definition                               ␣
 ↪          |
                            species_definition species_definition_list

species_list ::=            species_name;                                   ␣
 ↪          |
                            species_name; species_list

species_list_plus ::=       species_name_plus;                              ␣
 ↪          |
                            species_name_plus; species_list_plus

species_name ::=            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
```

27 Acknowledgements

This work has been supported by:

- The US EPA Science to Achieve Results (EPA-STAR)[18] program (grant # R840014[19]);
- The NASA Modeling, Analysis, and Prediction (MAP)[20] program;
- The NASA Atmospheric Composition Modeling and Analysis (ACMAP)[21] program;
- The NASA Advanced Systems Information Technology (AIST)[22] program (grant # AIST-18-0011[23])

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[24] and for assistance with debugging.

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.

Stuart Lacy wrote an export function for the Master Chemical Mechanism[25]. It creates KPP files that can be used out-of-the-box for the small model in the `examples/mcm` directory.

Simon Rosanka improved the `Update_RCONST` subroutine by providing time-dependent concentration values via an optional parameter.

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

28 References

29 Known bugs and issues

Please see our Issue tracker on GitHub[36] for a list of recent bugs and fixes.

29.1 Current bug reports

These bug reports (on GitHub)[37] are currently unresolved. We hope to fix these in future releases.

[18] https://www.epa.gov/research-grants/air-research-grants
[19] https://cfpub.epa.gov/ncer_abstracts/index.cfm/fuseaction/display.abstractDetail/abstract_id/11083/report/0
[20] https://astg.pages.smce.nasa.gov/website/esm/
[21] https://airbornescience.nasa.gov/category/Discipline/Atmospheric_Composition_Modeling_and_Analysis_Program
[22] https://esto.nasa.gov/aist
[23] https://esto.nasa.gov/project-selections-for-aist-18/#martin
[24] https://azure.microsoft.com/en-us/services/devops/pipelines/
[25] https://mcm.york.ac.uk/MCM/export
[36] https://github.com/KineticPreProcessor/KPP/issues
[37] https://github.com/KineticPreProcessor/KPP/issues?q=is%3Aissue%20state%3Aopen%20label%3Abug

LSODE integrator is not thread-safe

We have discovered that the current implementation of the LSODE integrator is not thread-safe for OpenMP parallelization[38]. When LSODE is called from within an OpenMP parallel loop, the integration will fail because key internal variables in LSODE will be overwritten by concurrent threads.

29.2 Bugs that have been resolved

These bugs (reported on GitHub)[39] have been resolved.

30 Support

The support guidelines can be found here[40].

31 Contributing

The Contributing guidelines can be found here[41].

32 Editing this User Guide

This user guide is generated with Sphinx[42]. Sphinx is an open-source Python project designed to make writing software documentation easier. The documentation is written in *reStructuredText (reST)*, a plaintext markup language that Sphinx extends for software documentation. The source for the documentation is the `docs/source` directory in top-level of the source code (and its subdirectories).

32.1 Quick start

First-time setup: Install Sphinx

To build this user guide on your local machine, you need to install Sphinx and its dependencies, which are listed in the table below.

[38] https://www.openmp.org/
[39] https://github.com/KineticPreProcessor/KPP/issues?q=is%3Aissue%20state%3Aclosed%20label%3Abug
[40] https://github.com/KineticPreProcessor/KPP/blob/main/SUPPORT.md
[41] https://github.com/KineticPreProcessor/KPP/blob/main/CONTRIBUTING.md
[42] https://www.sphinx-doc.org/

Package	Description	Version
sphinx	Creates online user manual documentation from markup text files	8.2.3
sphinx-autobuild[43]	Dynamically builds Sphinx documentation and displays it in a browser	2024.2.4
sphinx_rtd_theme[44]	Sphinx theme for ReadTheDocs	3.1.0
sphinxcontrib-bibtex[45]	Inserts LaTeX-style bibliography citations into ReadTheDocs documentation	2.6.5
docutils[46]	Processes plaintext documentation into HTML and other formats	0.21.2
myst-parser[47]	Parses text for docutils	5.0.0
jinja2[48]	Replaces tokenized strings with text	3.1.6

We recommend that you create a standalone **Conda** environment to install Sphinx and its dependencies. The YAML file `docs/read_the_docs_environment.yml` contains the proper package specifications. Use these commands:

```
$ cd docs
$ conda env create -n rtd_env --file=read_the_docs_environment.yml
```

This step only needs to be done once.

Build the documentation

1. Activate the **Conda** environment containing **Sphinx** and its dependencies:

   ```
   $ conda activate rtd_env
   ```

2. Navigate to the `docs/` folder:

   ```
   (rtd_env) $ cd docs      # Skip if you are already in the docs␣
   ↪folder
   ```

3. Check out the `docs/dev` branch of this repository, as this is the branch from which the **latest** ReadTheDocs version will be built:

   ```
   (rtd_env) $ git checkout docs/dev    # Skip if you are already on␣
   ↪the docs/dev branch
   ```

4. Start the **sphinx-autobuild** server:

[43] https://github.com/sphinx-doc/sphinx-autobuild
[44] https://github.com/readthedocs/sphinx_rtd_theme
[45] https://pypi.org/project/sphinxcontrib-bibtex/
[46] https://docutils.sourceforge.io/
[47] https://myst-parser.readthedocs.io/en/latest/
[48] https://jinja.palletsprojects.com/en/stable/

```
(rtd_env) $ sphinx-autobuild source build/html
```

This will parse the reST-format files in the `docs/source/` directory tree and generate new HTML files in `docs/build/html`.

5. Remove any HTML files (in `docs/build/html`) that might be left behind from a previous build:

```
(rtd_env) $ make clean
```

6. Open a web browser and navigate to `localhost:8000`.
7. Open your favorite text editor and start making changes to the reST-format documentation files in the `docs/source` directory tree. While **`sphinx-autobuild`** is running, you will see your updates rendered in the web browser as soon as you save your changes to disk.
8. Once you are satisfied with your edits, commit your changes to Git and push the documentation to the `docs/dev` remote branch of this repository,
9. Remove the generated HTML documentation files:

```
(rtd_env) $ make clean
```

10. Halt the **`sphinx-autobuild`** server by typing **`CTRL-C`**.
11. Deactivate the **`Conda`** environment:

```
(rtd_env) $ conda deactivate
```

32.2 Learning reStructured Text

ReadTheDocs documentation is generated from text files in **reStructured Text (reST)**, which is an easy-to-read, what-you-see-is-what-you-get plaintext markup language. It is the default markup language used by Sphinx.

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[49]: (single best resource; however, it's better read than skimmed)
- Official reStructuredText reference[50] (there is *a lot* of information here)

[49] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html
[50] https://docutils.sourceforge.io/docs/user/rst/quickref.html

- Presentation by Eric Holscher[51] (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[52]

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

32.3 Style guidelines

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.

Titles and headers

Element	reST Markup
Section header (aka "Heading 1")	Overline by # and underline by #
Sub-section header (aka "Heading 2")	Overline by = and underline by =
Sub-sub-section header (aka "Heading 3")	Underline by -
Sub-sub-sub-section header (aka "Heading 4")	Underline by ~
Sub-sub-sub-sub-section header (aka "Heading 5")	Underline by ^

References and links

Element	reST Markup Example	Rendered text
Reference to a named anchor	`:ref:`editing_this_user_`	*Quick start*
Renamed reference to a named anchor	`:ref:`Getting Started <editing_this_user_guide`	*Getting Started*
HTML link	`` `KPP Manual <https://kpp.readthedocs.io>`__ ``	KPP Manual[53]

[51] https://www.youtube.com/watch?v=eWNiwMwMcr4
[52] https://www.youtube.com/watch?v=DSIuLnoKLd8
[53] https://kpp.readthedocs.io

Other common style elements

Element	reST Markup Example	Rendered text
File paths	`:file:`myfile.nc``	myfile.nc
Directories	`:file:`/usr/bin``	/usr/bin
Program names	`:program:`cmake``	**cmake**
OS-level commands	`:program:`rm -rf``	**rm -rf**
Environment variables	`:envvar:`$HOME``	$HOME
Inline code or code variables	`:code:`PRINT*, "HELLO!"``	PRINT*, "HELLO!"
Inline literal text	`:literal:`$``	$

Indented code and text blocks

Code snippets should use `.. code-block:: <language>` directives:

Python

```
.. code-block:: python

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

Renders as:

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

Fortran

```
.. code-block:: Fortran

   DO I = 1, 10
      PRINT*, I
   ENDDO
```

Renders as:

```fortran
DO I = 1, 10
   PRINT*, I
ENDDO
```

Bash

```
.. code-block:: bash

   #!/bin/bash

   for f in *.nc; do
       echo $f
   done
```

Renders as:

```bash
#!/bin/bash

for f in *.nc; do
    echo $f
done
```

Command line (aka “console”)

```
.. code-block:: console

   $ ls -l $HOME
```

Renders as:

```
$ ls -l $HOME
```

No formatting

```
.. code-block:: none

   This text renders without any special formatting.
```

Renders as:

```
This text renders without any special formatting.
```

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[26].

[26] https://doi.org/10.5194/ACP-4-1461-2004

[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.

[Dreger 2025] Dreger, R., Kirfel, T., Pozzer, A., Rosanka, S., Sander, R., and Taraborrelli, D. Optimized step size control within the rosenbrock solvers for stiff chemical ordinary differential equation systems in kpp version 2.2.3_rs4. *Geosci. Model Dev.*, 18:4273–4291, 2025. doi:10.5194/GMD-18-4273-2025[27].

[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.*, pages 2022MS003293, 2023. doi:10.1029/2022MS003293[28].

[Long et al., 2026] Long, M. S., Sturm, O. P., Shah, V., and M, Y. R. Step size-optimized integrator coefficients for a 3rd order rosenbrock solver for stiff atmospheric chemical mechanisms - RODAS3.1. *Earth and Space Science*, 2026, submitted.

[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.

[Soederlind 2003] Söderlind, G. Digital filters in adaptive time-stepping. *ACM TOMS*, 29:1–26, 2003. doi:10.1145/641876.641877[29].

[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[30].

[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[31].

[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

[27] https://doi.org/10.5194/GMD-18-4273-2025
[28] https://doi.org/10.1029/2022MS003293
[29] https://doi.org/10.1145/641876.641877
[30] https://doi.org/10.5194/ACP-5-445-2005
[31] https://doi.org/10.5194/ACP-6-187-2006

atmospheric chemistry models. *Atmos. Environ.*, 44(35):4426–4431, 2010. doi:10.1016/j.atmosenv.2010.07.044[32].

[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[33].

[Sturm and Wexler 2022] Sturm, P. O. and Wexler, A. S. Conservation laws in a neural network architecture: enforcing the atom balance of a julia-based photochemical model (v0.2.0). *Geosci. Model Dev.*, 15(8):3417–3431, 2022. doi:10.5194/gmd-15-3417-2022[34].

[Sturm and Silva 2025] Sturm, P. O. and Silva, S. J. A nudge to the truth: atom conservation as a hard constraint in models of atmospheric composition using a species-weighted correction. *Journal of Geophysical Research: Machine Learning and Computation*, 2(1):99–108, 2025. doi:10.1029/2025JH000895[35].

[32] https://doi.org/10.1016/j.atmosenv.2010.07.044
[33] https://doi.org/10.5194/gmd-13-2475-2020
[34] https://doi.org/10.5194/gmd-15-3417-2022
[35] https://doi.org/10.1029/2025JH000895

Index

Symbols

$HOME, 104
$KPP_HOME, 88, 90

D

DYLD_LIBRARY_PATH, 23

E

environment variable
    $HOME, 104
    $KPP_HOME, 88, 90
    DYLD_LIBRARY_PATH, 23
    FLEX_HOME, 12
    FLEX_LIB_DIR, 12
    KPP_FLEX_LIB_DIR, 9, 16, 17, 22, 90
    KPP_HOME, 13, 14, 25
    LD_LIBRARY_PATH, 23
    PATH, 13, 88, 90

F

FLEX_HOME, 12
FLEX_LIB_DIR, 12

K

KPP_FLEX_LIB_DIR, 9, 16, 17, 22, 90
KPP_HOME, 13, 14, 25

L

LD_LIBRARY_PATH, 23

P

PATH, 13, 88, 90