RMG-Py and Arkane Documentation

Release 2.3.0

William H. Green, Richard H. West, and the RMG Team

Dec 20, 2018

CONTENTS

RMG is an automatic chemical reaction mechanism generator that constructs kinetic models composed of elementary chemical reaction steps using a general understanding of how molecules react. This documentation is for the newer Python version of RMG that we call RMG-Py.

I want to. . .	Resource
analyze models & search databases	RMG website resources (no download needed)
create mechanisms automatically	Download RMG with the RMG User's Guide
make transition state theory calculations	Run Arkane after downloading RMG. See the Arkane User's Guide
post an issue with RMG	GitHub issues page
contribute to the RMG project	RMG developer's wiki

Arkane is developed and distributed as part of RMG-Py, but can be used as a stand-alone application for Thermochemistry, Transition State Theory, and Master Equation chemical kinetics calculations. Its user guide is also included.

The last section of this documentation covers some of the more in depth theory behind RMG and Arkane.

Please visit http://reactionmechanismgenerator.github.io/RMG-Py/ for the most up to date documentation and source code. You may refer to the separate *RMG-Py API Reference* document to view the details of RMG-Py's modules and subpackages.

RMG USER'S GUIDE

For any questions related to RMG and its usage and installation, please post an issue at https://github.com/ReactionMechanismGenerator/RMG-Py/issues and the RMG developers will get back to you as soon as we can. You can also search for your problem on the issues page to see if there are already solutions in development. Alternatively, you can email us at rmg_dev@mit.edu.

1.1 Introduction

Reaction Mechanism Generator (RMG) is an automatic chemical reaction mechanism generator that constructs kinetic models composed of elementary chemical reaction steps using a general understanding of how molecules react. This version is written in Python, and called RMG-Py.

1.1.1 License

RMG is an open source program, available to the general public free of charge. The primary RMG code is distributed under the terms of the MIT/X11 License. However, RMG has a number of dependencies of various licenses, some of which may be more restrictive. **It is the user's responsibility to ensure these licenses have been obtained.**

```
Copyright (c) 2002-2018 Prof. William H. Green (whgreen@mit.edu),
Prof. Richard H. West (r.west@neu.edu) and the RMG Team (rmg_dev@mit.edu)

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the 'Software'),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

1.2 Overview of Features

Thermodynamics estimation using group additivity. Group additivity based on Benson's groups provide fast and reliable thermochemistry estimates. A standalone utility for estimating heat of formation, entropy, and heat capacity is also included.

Rate-based model enlargement Reactions are added to the model based on their rate, fastest first.

Rate-based termination. The model enlargement stops when all excluded reactions are slower than a given threshold. This provides a controllable error bound on the kinetic model that is generated.

Extensible libraries Ability to include reaction models on top of the provided reaction families.

Pressure-dependent reaction networks. Dissociation, combination, and isomerization reactions have the potential to have rate coefficients that are dependent on both temperature and pressure, and RMG is able to estimate both for networks of arbitrary complexity with a bounded error.

Simultaneous mechanism generation for several conditions. Concurrent generation of a reaction mechanism over multiple temperature and pressure conditions. Mechanisms generated this way are valid over a range of reaction conditions.

Dynamic simulation to a target conversion or time. Often the desired simulation time is not known *a priori*, so a target conversion is preferred.

Transport properties estimation using group additivity The Lennard-Jones sigma and epsilon parameters are estimated using empirical correlations (based on a species' critical properties and acentric factor). The critical properties are estimated using a group-additivity approach; the acentric factor is also estimated using empirical correlations. A standalone application for estimating these parameters is provided, and the output is stored in CHEMKIN-readable format.

1.3 Installation

Note: It is recommended that RMG be installed with Python 2.7, although it has been previously tested that Python 2.5 and 2.6 may also work. Dependency issues render it incompatible with Python 3.x releases.

For any questions related to RMG and its usage and installation, please post an issue at https://github.com/ReactionMechanismGenerator/RMG-Py/issues and the RMG developers will get back to you as soon as we can. You can also search for your problem on the issues page to see if there are already solutions in development. Alternatively, you can email us at rmg_dev@mit.edu

1.3.1 For Basic Users: Binary Installation Using Anaconda

It is highly recommended to use the Python platform Anaconda to perform the installation of RMG-Py. A binary installation is recommended for users who want to use RMG out of the box, and are not interested in changing or recompiling the RMG code or making many additions to RMG's thermodynamic and kinetics databases.

Binary Installation Using Anaconda for Unix-Based Systems: Linux and Mac OSX

- Download and install the Anaconda Python Platform for Python 2.7 (make sure not to install Python 3.0+, which is incompatible with RMG). When prompted to append Anaconda to your PATH, select or type Yes.

- Install both RMG and the RMG-database binaries through the Terminal. Dependencies will be installed automatically. It is safest to make a new Anaconda environment for RMG and its dependencies. Type the following command into the Terminal to create the new environment named 'rmg_env' containing the latest stable version of the RMG program and its database.

```
conda create -c rmg --name rmg_env rmg rmgdatabase
```

Whenever you wish to use it you must first activate the environment:

```
source activate rmg_env
```

- Optional: If you wish to use the *QMTP interface* with MOPAC to run quantum mechanical calculations for improved thermochemistry estimates of cyclic species, please obtain a legal license through the MOPAC License Request Form. Once you have it, type the following into your Terminal

```
mopac password_string_here
```

- You may now run an RMG test job. Save the Minimal Example Input File to a local directory. Use the Terminal to run your RMG job inside that folder using the following command

```
rmg.py input.py
```

You may now use RMG-Py, Arkane, as well as any of the *Standalone Modules* included in the RMG-Py package.

Updating your binary installation of RMG in Linux or Mac OSX

If you had previously installed a binary version of the RMG package, you may check and update your installation to the latest stable version available on Anaconda by typing the following command on the Terminal

```
source activate rmg_env
conda update rmg rmgdatabase -c rmg
```

Binary Installation Using Anaconda for Windows

- Download and install the Anaconda Python Platform for Python 2.7 (make sure not to install Python 3.0+, which is incompatible with RMG). We recommend changing the default install path to `C:\Anaconda\` in order to avoid spaces in the install path and be easily accessible. It is recommended to append Anaconda to your PATH as well as setting it as your default Python executable. All other settings can remain as their defaults.
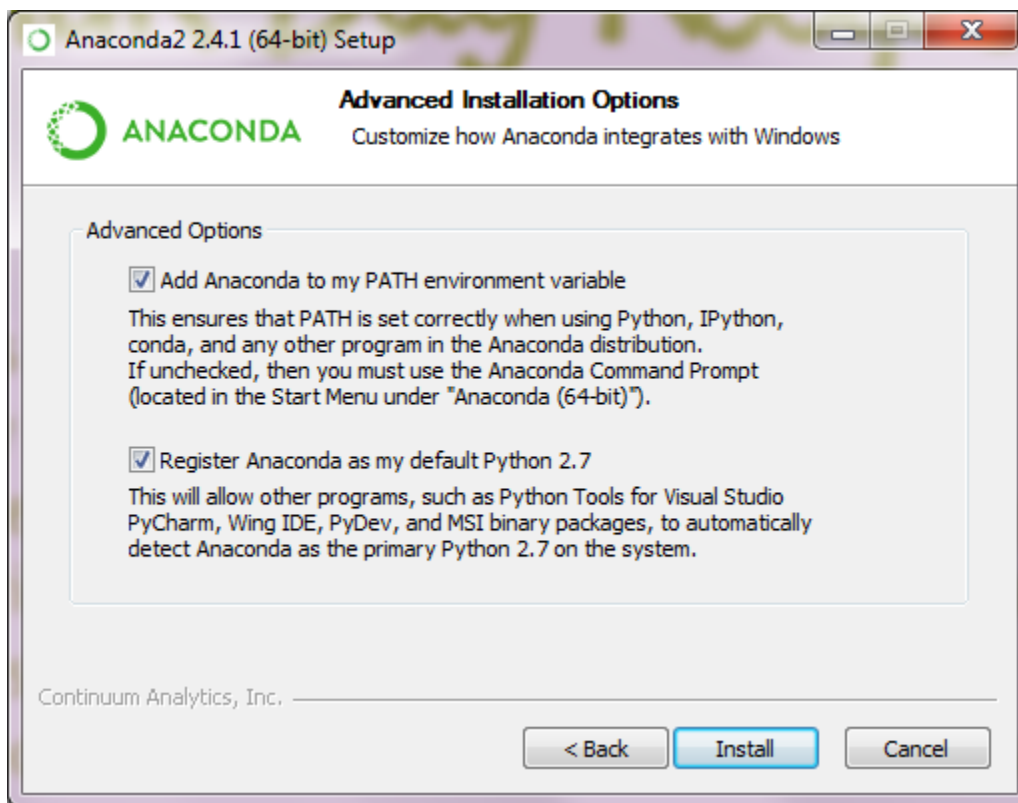
- Now we want to install both RMG and the RMG-database binaries via the command prompt. Dependencies will be installed automatically. It is safest to make a new Anaconda environment for RMG and all its dependencies. Open a command prompt (either by finding it in your Program Files or by searching for `cmd.exe`. You may need to run the command prompt as an administrator: to do this open up a file explorer and navigate to `C:\Windows\System32` and find the file `cmd.exe`; right click on this file and select "run as administrator") and type the following to create the new environment named '`rmg_env`' containing the latest stable version of the RMG program and its database.

```
conda create -c rmg --name rmg_env rmg rmgdatabase
```

- Whenever you wish to use it you must first activate the environment in the command prompt by typing:

```
activate rmg_env
```

- Optional: If you wish to use the *QMTP interface* with MOPAC to run quantum mechanical calculations for improved thermochemistry estimates of cyclic species, please obtain a legal license through the MOPAC License Request Form.Once you have it, type the following into your command prompt (while the environment is activated)

```
mopac password_string_here
```

- Now you must *set the RMG environment variable in Windows* to allow your system to find the RMG python files more easily.

- If you set any new environment variables, you must now close and reopen the command prompt so that those environment variables can be refreshed and used.

- You may now run an RMG test job. Save the Minimal Example Input File to a local directory. Use the command prompt to run your RMG job inside that folder by using the following command

```
activate rmg_env
python %RMGPy%\rmg.py input.py
```

You may now use RMG-Py, Arkane, as well as any of the *Standalone Modules* included in the RMG-Py package.

Updating your binary installation of RMG for Windows

If you had previously installed a binary version of the RMG package, you may check and update your installation to the latest stable version available on Anaconda by typing the following command in a Command Prompt

```
source activate rmg_env
conda update rmg rmgdatabase -c rmg
```

Setting up Windows Environment Variables for RMG

Setting environment variables in Windows allows for easier shortcutting and usage of RMG scripts and packages.

Setting the `RMGPy` variable

- If you have a search bar available in your start menu, search for "environment variables" and select "Edit environment variables for your account". Alternatively, navigate to this settings window by first going to "Control Panel > System", then clicking "Advanced system settings".
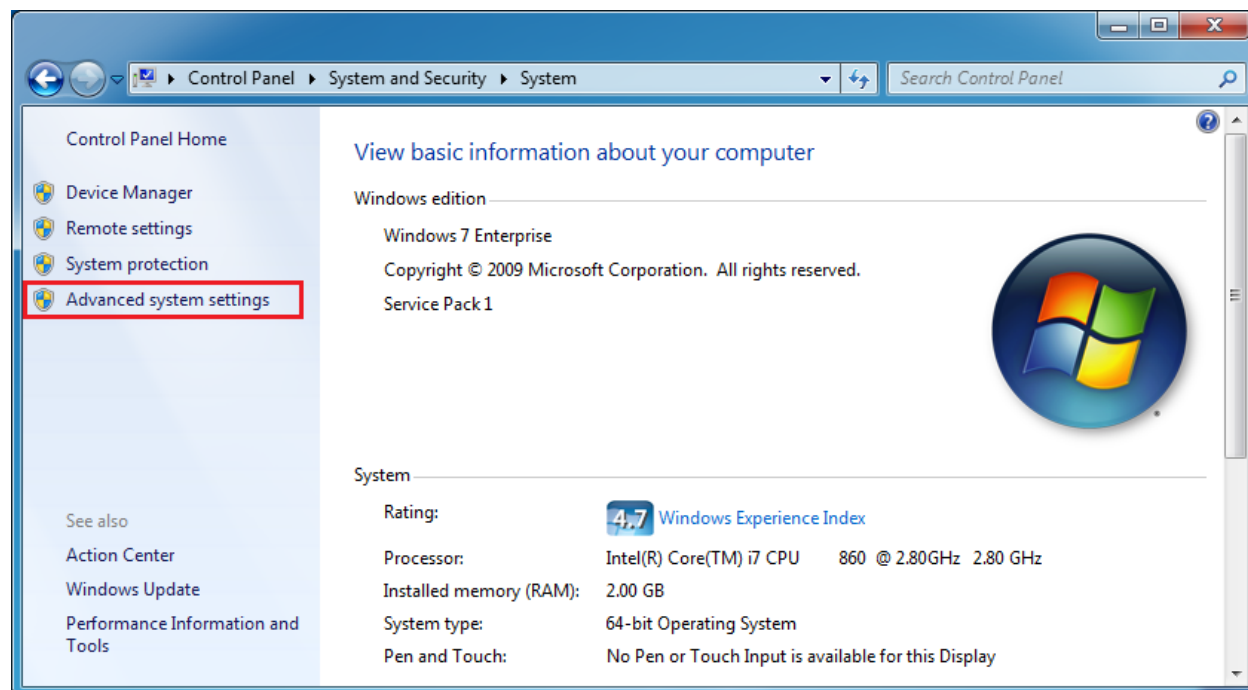
- Once the "System Properties" window opens, click on "Environment Variables..." in the "Advanced" tab.
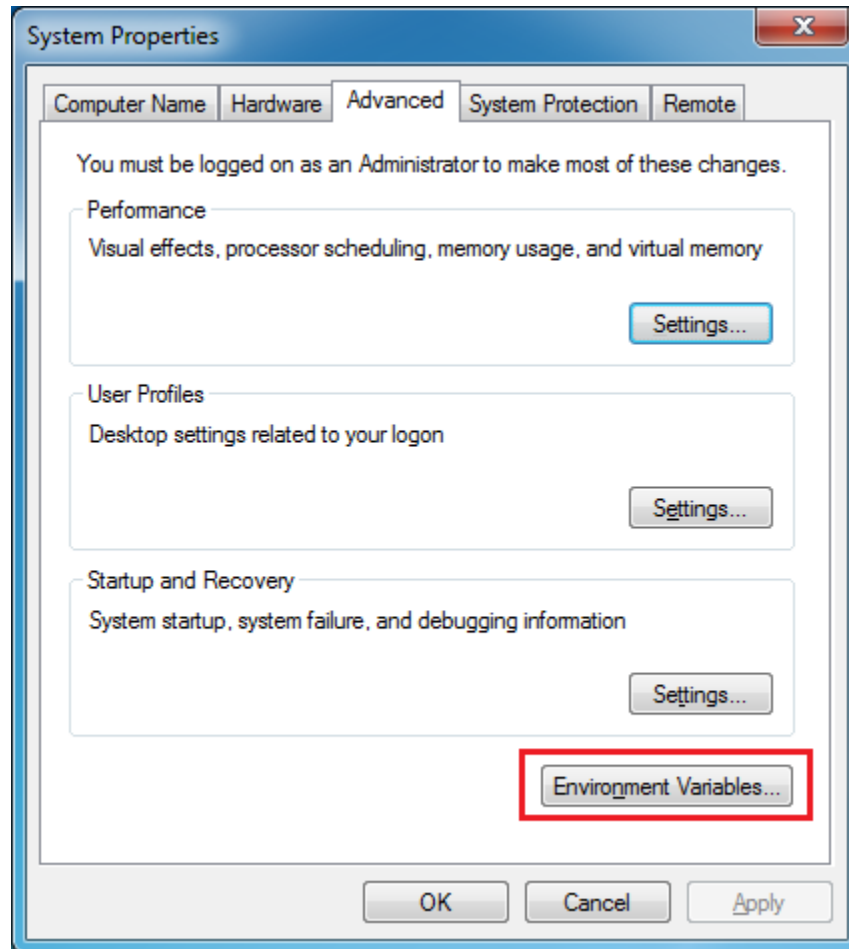
- Once the "Environment Variables" window opens, click on "New" under the "User variables".

Environment Variables

User variables for Connie

Variable	Value
PATH	C:\Anaconda;C:\Anaconda\Scripts;C:\A...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New...   Edit...   Delete

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	1
OS	Windows_NT

New...   Edit...   Delete

OK   Cancel
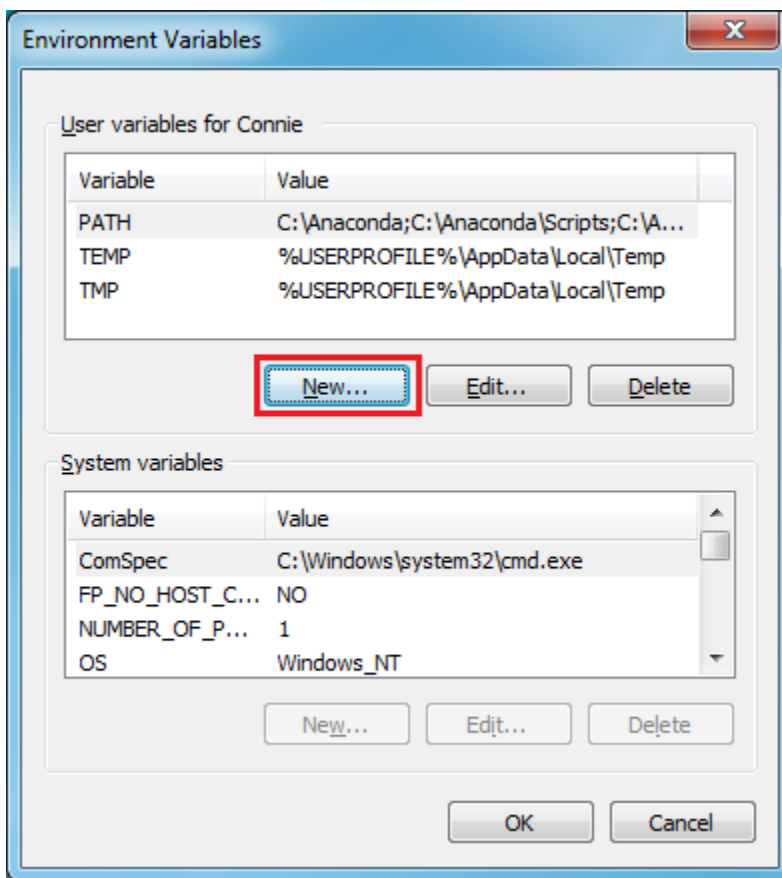
- Set a new variable with the name `RMGPy` with the appropriate value directed at your RMG path.

  If you installed the binary version of RMG, the environment value should be set to:

  ```
  C:\Anaconda\envs\rmg_env\Scripts\
  ```

  where `C:\Anaconda` can be replaced by wherever your Anaconda was installed.

  Your screen might look like this:

Edit User Variable

Variable name: RMGPy

Variable value: C:\Anaconda\envs\rmg_env\Scripts

OK   Cancel
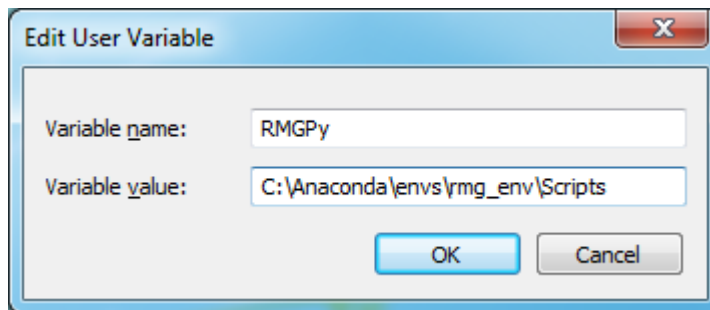
  If you are installing RMG by source, you can similarly set your `RMGPy` variable to the source directory, such as

  ```
  C:\Code\RMG-Py
  ```

- Click "Ok" on all screens to confirm the changes.

Note: If you set any new environment variables, you must close and reopen any command prompts previously open

before the changes can take effect.

Optional: Setting a Permanent Anaconda Environment for RMG

If you use Anaconda solely for RMG, it may be more convenient to set your `PATH` variable to be permanently directed to the RMG environment. This will allow you to run RMG easily without having to type `activate rmg_env` in the command prompt every time.

Similarly to setting the environment variable for `RMGPy`, go to "Edit environment variables for your account" and click edit on the `PATH` variable. Replace the paths containing the Anaconda main directory with the RMG environment in Anaconda.

For example a path such as

```
C:\Anaconda\Scripts\
```

should be changed to:

```
C:\Anaconda\envs\rmg_env\Scripts\
```

Note that `C:\Anaconda` should be wherever your Anaconda was installed.

1.3.2 For Developers: Installation by Source Using Anaconda Environment

RMG-Py can now be built by source using the Anaconda Python Platform to assist in installing all necessary dependencies. This is recommended for a developer who may be altering the RMG source code or someone who expects to manipulate the databases extensively. You will also be able to access the latest source code updates and patches through Github.

Installation by Source Using Anaconda Environment for Unix-based Systems: Linux and Mac OSX

- Download and install the Anaconda Python Platform for Python 2.7 (make sure not to install Python 3.0+, which is incompatible with RMG). When prompted to append Anaconda to your PATH, select or type Yes. Install the Anaconda folder inside your home directory (typically `/home/YourUsername/` in Linux and `/Users/YourUsername` in Mac).

- Install Git, the open source version control package through the Terminal. **For Mac OS X**: Git is already packages with OS X 10.9 or later, but requires installation of Xcode's Command Line Tools. Skip the git installation and run it through the terminal, where you will be prompted to install the Command Line Tools if they are not already installed.

  ```
  sudo apt-get install git
  ```

- Install the latest versions of RMG and RMG-database through cloning the source code via Git. Make sure to start in an appropriate local directory where you want both RMG-Py and RMG-database folders to exist.

  ```
  git clone https://github.com/ReactionMechanismGenerator/RMG-Py.git
  git clone https://github.com/ReactionMechanismGenerator/RMG-database.git
  ```

- Now create the anaconda environment for RMG-Py

  For Linux users:

```
cd RMG-Py
conda env create -f environment_linux.yml
```

For Mac users:

```
cd RMG-Py
conda env create -f environment_mac.yml
```

- Compile RMG-Py after activating the anaconda environment

```
source activate rmg_env
make
```

- Modify environment variables. Add RMG-Py to the PYTHONPATH to ensure that you can access RMG modules from any folder. Modify your `~/.bashrc` file by adding the following line

```
export PYTHONPATH=$PYTHONPATH:YourFolder/RMG-Py/
```

NOTE: Make sure to change `YourFolder` to the path leading to the `RMG-Py` code. Not doing so will lead to an error stating that python cannot find the module `rmgpy`.

- If you wish to always be able to run RMG-Py, you can modify the anaconda path to point to the RMG environment. Modify the following line in your `~/.bashrc` file

```
export PATH=~/anaconda/bin:$PATH
```

by changing it to the following line

```
export PATH=~/anaconda/envs/rmg_env/bin:$PATH
```

be sure to either close and reopen your terminal to refresh your environment variables, or type the following command

```
source ~/.bashrc
```

- Optional: If you wish to use the *QMTP interface* with MOPAC to run quantum mechanical calculations for improved thermochemistry estimates of cyclic species, please obtain a legal license through the MOPAC License Request Form. Once you have it, type the following into your Terminal

```
mopac password_string_here
```

You may now use RMG-Py, Arkane, as well as any of the *Standalone Modules* included in the RMG-Py package.

Test Suite

There are a number of basic tests you can run on the newly installed RMG. It is recommended to run them regularly to ensure the code and databases are behaving normally.

- **Unit test suite**: this will run all the unit tests in the `rmgpy` package

```
cd RMG-Py
make test
```

- **Database test suite**: this will run the database unit tests to ensure that groups, rate rules, and libraries are well formed

```
cd RMG-Py
make test-database
```

Running Examples

A number of basic examples can be run immediately. Additional example input files can be found in the `RMG-Py/examples` folder. Please read more on *Example Input Files* in the documentation.

- **Minimal Example**: this will run an Ethane pyrolysis model. It should take less than a minute to complete. The results will be in the `RMG-Py/testing/minimal` folder:

```
cd RMG-Py
make eg1
```

- **Hexadiene Example**: this will run a Hexadiene model with pressure dependence and QMTP. Note that you must have MOPAC installed for this to run. The results will be in the `RMG-Py/testing/hexadiene` folder:

```
cd RMG-Py
make eg2
```

- **Liquid Phase Example**: this will run a liquid phase RMG model. The results will be in the `RMG-Py/testing/liquid_phase` folder

```
cd RMG-Py
make eg3
```

- **ThermoEstimator Example**: this will run the *Thermo Estimation Module* on a few molecules. Note that you must have MOPAC installed for this to run completely. The results will be in the `RMG-Py/testing/thermoEstimator` folder

```
cd RMG-Py
make eg4
```

Installation by Source Using Anaconda Environment for Windows

- Download and install the Anaconda Python Platform for Python 2.7 (make sure not to install Python 3.0+, which is incompatible with RMG). We recommend changing the default install path to `C:\Anaconda\` in order to avoid spaces in the install path and be easily accessible. It is recommended to append Anaconda to your PATH as well as setting it as your default Python executable. All other settings can remain as their defaults.
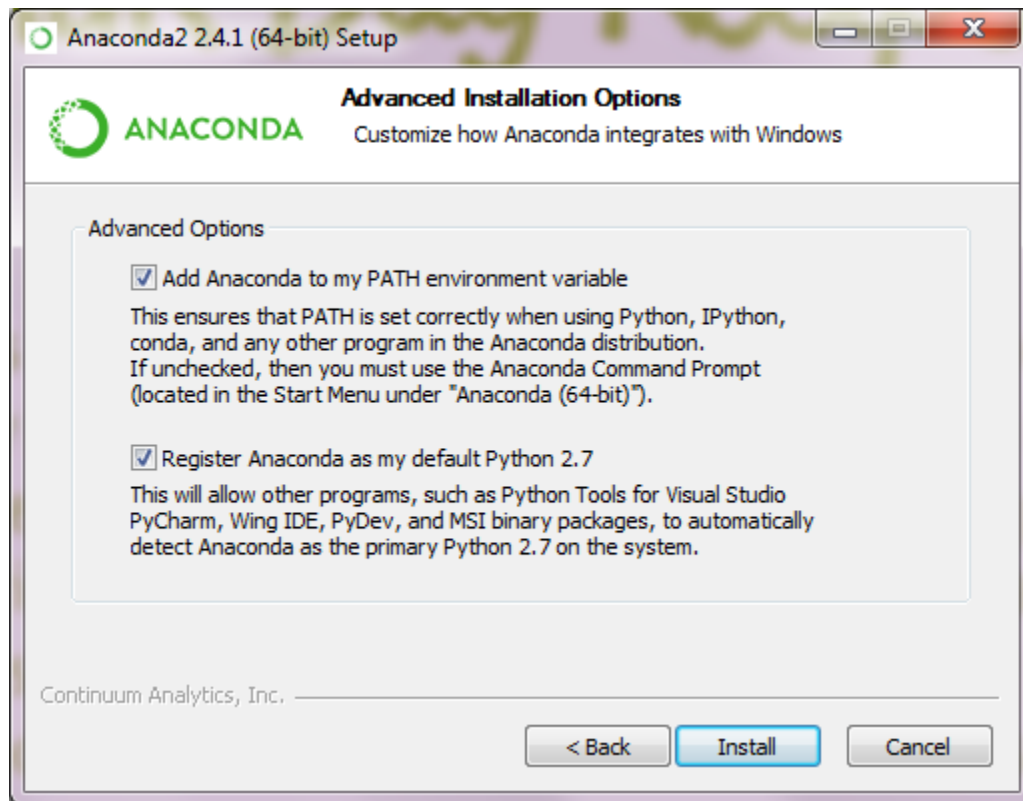
- Install Git, the open source version control package. When asked, append Git tools to your Command Prompt. It is also recommended to commit Unix-style line endings:
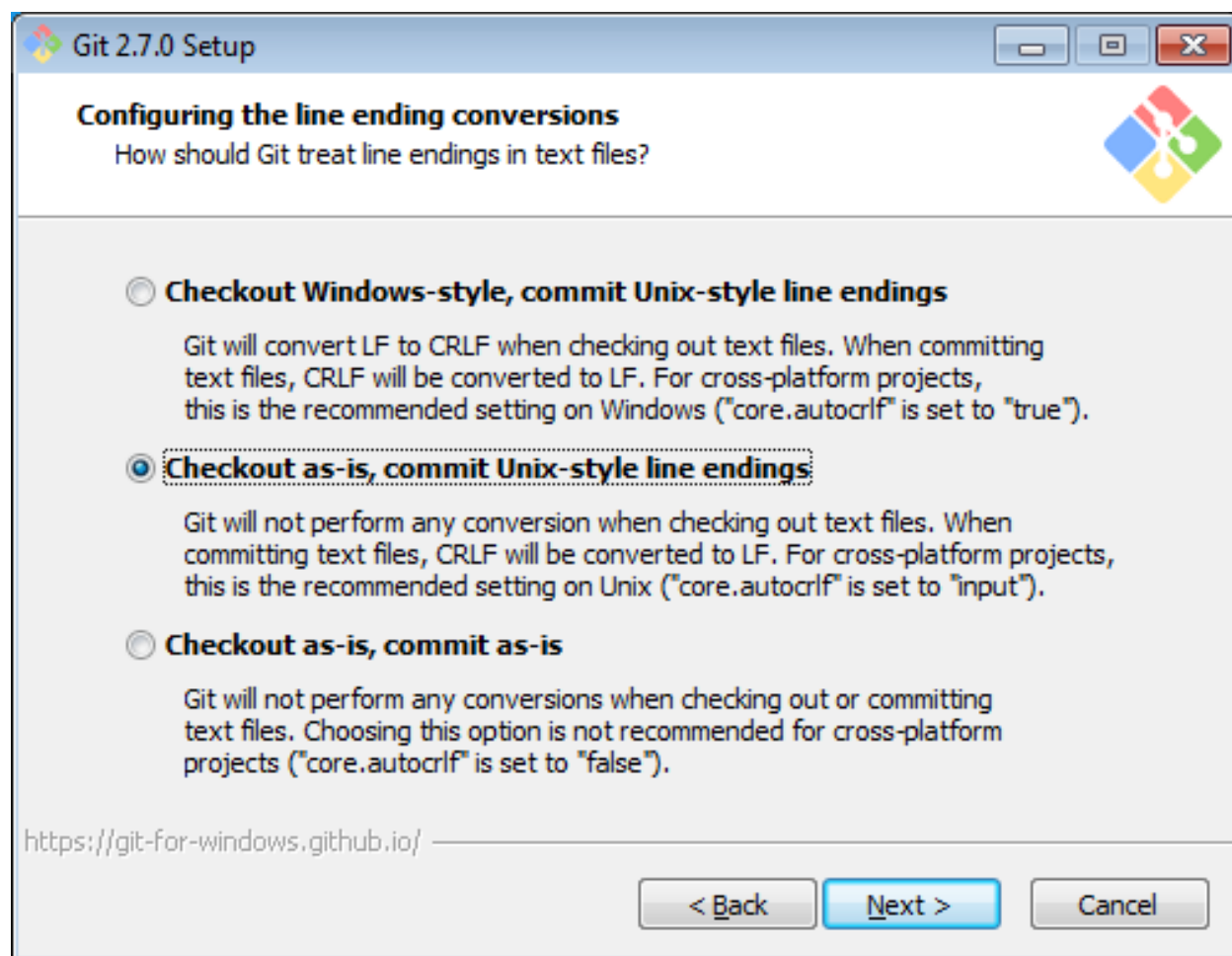
- Open Git CMD or a command prompt (either by finding it in your Program Files or by searching for `cmd.exe`. You may have to run the command prompt as an administrator. To do so right click on `cmd.exe`. and select Run as Administrator). Install the latest versions of RMG and RMG-database through cloning the source code via Git. Make sure to start in an appropriate local directory where you want both RMG-Py and RMG-database folders to exist. We recommend creating a folder such as `C:\Code\`

```
git clone https://github.com/ReactionMechanismGenerator/RMG-Py.git
git clone https://github.com/ReactionMechanismGenerator/RMG-database.git
```

- Create and activate the RMG Anaconda environment

```
cd RMG-Py
conda env create -f environment_windows.yml
activate rmg_env
```

Every time you open a new command prompt and want to complie or use RMG, you must reactivate this environment by typing

```
activate rmg_env
```

- Now you can compile RMG-Py

```
cd RMG-Py
mingw32-make
```

- Now it is recommended to modify your system's environment variables. Please see *Setting the RMG environment variable in Windows* for more information.

  Additionally, set the `PYTHONPATH` environment variable to the path of your RMG-Py source folder to ensure that you can access RMG modules from any python prompt. The prompt might look like this:
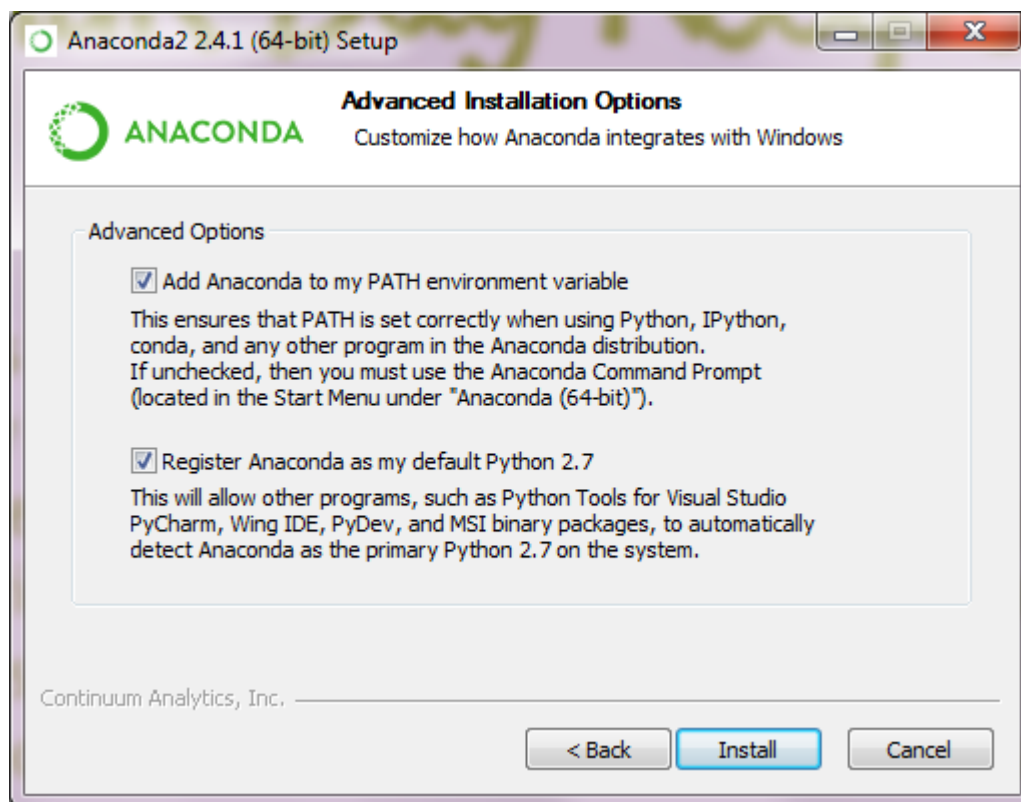
- If you set any new environment variables, you must now close and reopen the command prompt so that those environment variables can be refreshed and used.

- Optional: If you wish to use the *QMTP interface* with MOPAC to run quantum mechanical calculations for improved thermochemistry estimates of cyclic species, please obtain a legal license through the MOPAC License Request Form. Once you have it, type the following into your command prompt

```
mopac password_string_here
```

You may now use RMG-Py, Arkane, as well as any of the *Standalone Modules* included in the RMG-Py package.

Test Suite

There are a number of basic tests you can run on the newly installed RMG. It is recommended to run them regularly to ensure the code and databases are behaving normally.

- **Unit test suite**: this will run all the unit tests in the `rmgpy` package

```
cd RMG-Py
mingw32-make test
```

- **Database test suite**: this will run the database unit tests to ensure that groups, rate rules, and libraries are well formed

```
cd RMG-Py
mingw32-make test-database
```

Running Examples

A number of basic examples can be run immediately. Additional example input files can be found in the `RMG-Py\examples` folder. Please read more on *Example Input Files* in the documentation.

- **Minimal Example**: this will run an Ethane pyrolysis model. It should take less than a minute to complete. The results will be in the `RMG-Py\testing\minimal` folder:

```
cd RMG-Py
mingw32-make eg1
```

- **Hexadiene Example**: this will run a Hexadiene model with pressure dependence and QMTP. Note that you must have MOPAC installed for this to run. The results will be in the `RMG-Py\testing\hexadiene` folder:

```
cd RMG-Py
mingw32-make eg2
```

- **Liquid Phase Example**: this will run a liquid phase RMG model. The results will be in the `RMG-Py\testing\liquid_phase` folder

```
cd RMG-Py
mingw32-make eg3
```

- **ThermoEstimator Example**: this will run the *Thermo Estimation Module* on a few molecules. Note that you must have MOPAC installed for this to run completely. The results will be in the `RMG-Py\testing\thermoEstimator` folder

```
cd RMG-Py
mingw32-make eg4
```

Updating the RMG-Py Source Code

It is recommended to keep yourself up to date with the latest patches and bug fixes by RMG developers, which is maintained on the official repository at https://github.com/ReactionMechanismGenerator/RMG-Py/ You can view the latest changes by viewing the commits tab on the repository. To update your source code, you can "pull" the latest changes from the official repo by typing the following command in the Command Prompt

```
cd RMG-Py
git pull https://github.com/ReactionMechanismGenerator/RMG-Py.git master
```

We also recommend updating the RMG-database regularly. The repo itself can be found at https://github.com/ReactionMechanismGenerator/RMG-database/

```
cd RMG-database
git pull https://github.com/ReactionMechanismGenerator/RMG-database.git master
```

For more information about how to use the Git workflow to make changes to the source code, please refer to the handy Git Tutorial

1.3.3 For Developers: Direct Installation by Source without Anaconda

The installation approach in this section is not recommended and also not maintained by RMG developer team. This is only a record for people who cannot use Anaconda.

Linux Installation

RMG-Py and all of its dependencies may be easily installed through a short series of Terminal commands. The instructions listed below were written for Ubuntu 12.04 and should generally apply to other distributions.

Warning: This installation method is no longer actively maintained, and is not guaranteed to work as written.

- Install compilers and libraries:

```
sudo apt-get install git g++ gfortran python-dev liblapack-dev
sudo apt-get install python-openbabel python-setuptools python-pip
```

- After creating a Github account, generate your public key:

```
cd ~; ssh-keygen                        # press enter to save to the default directory
                                        # create a password if desired
cat .ssh/id_rsa.pub
```

Copy this public key to your Github profile.

- Install dependencies:

```
sudo apt-get install libpng-dev libfreetype6-dev graphviz

sudo pip install numpy          # install NumPy before other packages

sudo pip install scipy cython nose matplotlib quantities sphinx psutil xlwt

cd ~
git clone https://github.com/ReactionMechanismGenerator/PyDAS.git
git clone https://github.com/ReactionMechanismGenerator/PyDQED.git
cd PyDAS; make F77=gfortran; sudo make install; cd ..
cd PyDQED; make F77=gfortran; sudo make install; cd ..
```

- Install RDKit

Full installation instructions: http://code.google.com/p/rdkit/wiki/GettingStarted Be sure to **build it with InChI support.** Here's a synopsis:

```
cd ~
sudo apt-get install flex bison build-essential python-numpy cmake python-dev sqlite3
sudo apt-get install libsqlite3-dev libboost-dev libboost-python-dev libboost-regex-dev
git clone https://github.com/rdkit/rdkit.git
cd rdkit
export RDBASE=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDBASE/lib
export PYTHONPATH=$PYTHONPATH:$RDBASE
cd External/INCHI-API
./download-inchi.sh
cd ../../
mkdir build
cd build
cmake .. -DRDK_BUILD_INCHI_SUPPORT=ON
```

```
make
make install
```

You'll need various environment variables set (you may want to add these to your *.bash_profile* file), eg.:

```
export RDBASE=$HOME/rdkit # CHECK THIS (maybe you put RDKit somewhere else)
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RDBASE/lib
export PYTHONPATH=$PYTHONPATH:$RDBASE  # (or some other way to make sure it's on your␣
→Python path)
```

- The following dependencies are also required for core RMG functions and must be installed from source before building RMG:

  pyrdl: RingDecomposerLib, used for ring perception. Download from https://github.com/rareylab/RingDecomposerLib. Requires CMAKE to compile.

  lpsolve: Mixed integer linear programming solver. Download from https://sourceforge.net/projects/lpsolve/. Python extension also required.

- Install RMG-Py:

```
cd ~
git clone https://github.com/ReactionMechanismGenerator/RMG-database.git
git clone https://github.com/ReactionMechanismGenerator/RMG-Py.git
sudo pip install -r RMG-Py/requirements.txt
cd RMG-Py
make
```

- Run an example:

```
python rmg.py examples/rmg/minimal/input.py
```

Verify your installation by opening the resulting output.html file under the "examples/rmg/minimal" directory.

You can also use the Makefile targets to test and run examples:

```
make test
make eg1
make eg2
```

- Compiling RMG-Py with Sensitivity Analysis:

  Running sensitivity analysis in RMG-Py requires the prerequisite DASPK solver and DASPK compiled wrapper in PyDAS. To do so first compile daspk in PyDAS and agree to download the daspk31.tgz file when prompted.

```
cd PyDAS/
make
make install
```

Then compile RMG-Py normally. It will automatically be compiled with sensitivity analysis if DASPK is found.

```
cd RMG-Py
make clean-solver
make
```

Note that using this option will allow RMG to both run with and without sensitivity.

MacOS X Installation

There are a number of dependencies for RMG-Py. This page will guide you through installing them. You will need the Command Line Tools for XCode. If you are not using Anaconda to install RMG-Py, we highly recommend the Homebrew package manager. The following instructions assume that you have installed Homebrew and its requirements. We recommend using a Virtual Environment for your Python packages, but this is optional (without it you may need to add *sudo* before some commands to solve permission errors).

You will also need gfortran, Python, Numpy and Scipy. We typically install them using homebrew-python but other methods may work as well.

Warning: This installation method is no longer actively maintained, and is not guaranteed to work as written.

- For example:

```
brew tap homebrew/python
brew install numpy
brew install scipy
brew install matplotlib --with-cairo --with-ghostscript --with-ticl-tk --with-pyqt --with-
↪pygtk --withgtk3
```

- Install git if you don't already have it (you may also like some graphical interfaces like mxcl's GitX or GitHub for Mac):

```
brew update
brew install git
```

- Optional (but recommended for Nitrogen-chemistry nomenclature): install OpenBabel:

```
brew install open-babel --with-python --HEAD
```

- Install RDKit:

```
brew tap rdkit/rdkit
brew install rdkit --with-inchi
brew link --overwrite rdkit
```

  You'll need to set an environment variable to use it, eg. put this in your *~/.bash_profile* file:

```
export RDBASE=/usr/local/share/RDKit
```

- The following dependencies are also required for core RMG functions and must be installed from source before building RMG:

  pyrdl: RingDecomposerLib, used for ring perception. Download from https://github.com/rareylab/RingDecomposerLib. Requires CMAKE to compile.

  lpsolve: Mixed integer linear programming solver. Download from https://sourceforge.net/projects/lpsolve/. Python extension also required.

- Make a directory to put everything in:

```
mkdir ~/Code
```

- Get the RMG-Py source code and the RMG-database from GitHub:

```
cd ~/Code
git clone https://github.com/ReactionMechanismGenerator/RMG-database.git
git clone https://github.com/ReactionMechanismGenerator/RMG-Py.git
```

- Install the Python dependencies listed in the `RMG-Py/requirements.txt` file using *pip* (do `easy_install pip` if you don't already have it):

```
pip install -r RMG-Py/requirements.txt
```

- Get and build PyDQED:

```
cd ~/Code
git clone https://github.com/ReactionMechanismGenerator/PyDQED.git
cd PyDQED
export LIBRARY_PATH=$(dirname $(gfortran -print-file-name=libgfortran.a))
make
make install
```

- Get and build PyDAS:

```
cd ~/Code
git clone https://github.com/ReactionMechanismGenerator/PyDAS.git
cd PyDAS
export LIBRARY_PATH=$(dirname $(gfortran -print-file-name=libgfortran.a))
make
make install
```

- Build RMG-Py:

```
cd ~/Code/RMG-Py
make -j4
```

- Run an example:

```
cd ~/Code/RMG-Py/
python rmg.py examples/rmg/minimal/input.py
```

   Verify your installation by opening the resulting output.html file under the "examples/rmg/minimal" directory.

   You can also use the Makefile targets to test and run examples:

```
cd ~/Code/RMG-Py/
make test
make eg1
make eg2
```

To run with on-the-fly Quantum Mechanics calculations, you will also need to install MOPAC or Gaussian, then run *make QM*.

1.3.4 Dependencies

Please visit the page below for detailed information on all of RMG's dependencies and their license restrictions

Dependencies

List of Dependencies

Briefly, RMG depends on the following packages, almost all of which can be found in the RMG anaconda channel as binary packages.

- **boost:** portable C++ source libraries
- **cairo:** a 2D vector graphics library with support for multiple backends including image buffers, PNG, PostScript, PDF, and SVG file output. Used for molecular diagram generation
- **cairocffi:** a set of Python bindings and object-oriented API for cairo
- **coverage:** code coverage measurement for Python
- **cython:** compiling Python modules to C for speed up
- **ffmpeg:** (optional) used to encode videos, necessary for generating video flux diagrams
- **gaussian:** (optional) commerical software program for quantum mechanical calculations. Must be installed separately.
- **gcc:** GNU compiler collection for C,C++, and Fortran. (MinGW is used in windows)
- **gprof2dot:** converts Python profiling output to a dot graph
- **graphviz:** generating flux diagrams
- **jinja2:** Python templating language for html rendering
- **jupyter:** (optional) for using IPython notebooks
- **lpsolve:** mixed integer linear programming solver, used for resonance structure generation. Must also install Python extension.
- **markupsafe:** implements XML/HTML/XHTML markup safe strings for Python
- **matplotlib:** library for making plots
- **mock:** for unit-testing
- **mopac:** semi-empirical software package for QM calculations
- **muq:** (optional) MIT Uncertainty Quantification library, used for global uncertainty analysis
- **networkx:** (optional) network analysis for reaction-path analysis IPython notebook
- **nose:** advanced unit test controls
- **numpy:** fast matrix operations
- **openbabel:** chemical toolbox for speaking the many languages of chemical data
- **psutil:** system utilization diagnostic tool
- **pydas:** differential algebraic system solver
- **pydot:** interface to Dot graph language
- **pydqed:** constrained nonlinear optimization
- **pyparsing:** a general parsing module for python
- **pyrdl:** RingDecomposerLib for graph ring perception
- **pyzmq:** Python bindings for zeroMQ
- **quantities:** unit conversion
- **rdkit:** open-source cheminformatics toolkit
- **scipy:** fast mathematical toolkit
- **scoop:** parallelization of Python code
- **setuptools:** for packaging Python projects

- **sphinx:** documentation generation

- **symmetry:** calculating symmetry numbers of chemical point groups

- **xlwt:** generating Excel output files

License Restrictions on Dependencies

All of RMG's dependencies except the ones listed below are freely available and compatible with RMG's open source MIT license (though the specific nature of their licenses vary).

- **pydas**: The DAE solvers used in the simulations come from Linda Petzold's research group at UCSB. For running sensitivity analysis in RMG, the DASPK 3.1 solver is required, which "is subject to copyright restrictions" for non-academic use. Please visit their website for more details. To run RMG without this restriction, one may switch to compiling with the DASSL solver instead in RMG, which is "available in the public domain."

If you wish to do on-the-fly quantum chemistry calculations of thermochemistry (advisable for fused cyclic species in particular, where the ring corrections to group additive estimates are lacking), the then you will need the third-party software for the QM calculations:

- **gaussian**: Gaussian03 and Gaussian09 are currently supported and commercially available. See http://www.gaussian.com for more details.

- **mopac** MOPAC can be found at http://openmopac.net/. Though it is not free for industrial use, it is free for non-profit and academic research use.

1.3.5 Installation FAQ

This section collects frequently asked questions on installation of RMG.

FAQ collection

- Got an error of `Segmentation fault:11` after installing RMG on my machine?

  Segmentation fault is a typical error in C code, caused by a program trying to read or write an illegal memory location, i.e. one it is not allowed to access. The most common cause in RMG is a conflict between two different versions of a shared library. RMG has some dependencies which are written in C++, e.g. rdkit, openbabel. If you compile one of these with a different version of some compiler library, or you compile RMG using one version and run it with another, you will often get a Segmentation fault. Chances are those packages are not up to date, or maybe your environmental variable `PATH` is messed up so that the wrong version of something is being found. Please see one example from a user having same Segmentation fault issue.

- How can I install RMG-Py without Anaconda?

  Usually we don't recommend installing RMG-Py without Anaconda because it takes longer and is easier to get trouble with package management. But one still can try direct installation on Linux or MacOS by following *Linux instruction* or *MacOS instruction*. The RMG team does not use this install approach internally any more, so these instructions are not actively maintained.

- Windows binary installation gives `WindowsError: [Error 5]`?

  Error 5 is access is denied, so this is either a permissions error, or an issue with the Windows file lock. These posts suggest rebooting the computer (in case it's a file lock), and running the anaconda prompt, from which you run `conda create -c rmg --name rmg_env rmg rmgdatabase`, as an administrator (in case it's a permissions error). Please checkout one example from a user having Windows binary installation issue.

- I get something like `IOError: [Errno 13] Permission denied: 'C:\\RMG.log'`

  You do not have permission to write to the log file. Try running the RMG from a different folder that you do have write permission to, such as within your user's documents directory, or else try running the command prompt as an Administrator (so that you have write permission everywhere). See for example issue #817.

If you have any other errors please report them by opening an issue, and for general questions ask in the RMG-Py chat room.

1.4 Creating Input Files

The syntax and parameters within an RMG input file are explained below. We recommend trying to build your first input file while referencing one of the *Example Input Files*.

1.4.1 Syntax

The format of RMG-Py `input.py` is based on Python syntax.

Each section is made up of one or more function calls, where parameters are specified as text strings, numbers, or objects. Text strings must be wrapped in either single or double quotes.

1.4.2 Datasources

This section explains how to specify various reaction and thermo data sources in the input file.

Thermo Libraries

By default, RMG will calculate the thermodynamic properties of the species from Benson additivity formulas. In general, the group-additivity results are suitably accurate. However, if you would like to override the default settings, you may specify the thermodynamic properties of species in the ThermoLibrary. When a species is specified in the ThermoLibrary, RMG will automatically use those thermodynamic properties instead of generating them from Benson's formulas. Multiple libraries may be created, if so desired. The order in which the thermo libraries are specified is important: If a species appears in multiple thermo libraries, the first instance will be used.

Now in RMG, you have two types of thermo libraries: gas and liquid thermo libraries. As species thermo in liquid phase depends on the solvent, those libraries can only be used in liquid phase simulation with the corresponding solvent. Gas phase thermo library can be used either in gas phase simulation or in liquid phase simulation. (see more details on the two *thermo library types* and *how to use thermo librairies in liquid phase simulation*)

Please see Section *editing thermo database* for more details. In general, it is best to leave the ThermoLibrary set to its default value. In particular, the thermodynamic properties for H and H2 must be specified in one of the primary thermo libraries as they cannot be estimated by Benson's method.

For example, if you wish to use the GRI-Mech 3.0 mechanism [GRIMech3.0] as a ThermoLibrary in your model, the syntax will be:

```
thermoLibraries = ['primaryThermoLibrary','GRI-Mech3.0']
```

This library is located in the `$RMG/RMG-database/input/thermo/libraries` directory. All "Locations" for the ThermoLibrary field must be with respect to the `$RMG/RMG-database/input/thermo/libraries` directory.

Note: Checks during the initialization are maid to avoid users to use "liquid thermo librairies" in gas phase simulations or to use "liquid phase libraries" obtained in another solvent that the one defined in the input file in liquid phase simulations.

Reaction Libraries

The next section of the `input.py` file specifies which, if any, Reaction Libraries should be used. When a reaction library is specified, RMG will first use the reaction library to generate all the relevant reactions for the species in the core before going through the reaction templates. Unlike the Seed Mechanism, reactions present in a Reaction Library will not be included in the core automatically from the start.

You can specify your own reaction library in the location section. In the following example, the user has created a reaction library with a few additional reactions specific to n-butane, and these reactions are to be used in addition to the Glarborg C3 library:

```
reactionLibraries = [('Glarborg/C3',False)],
```

The keyword False/True permits user to append all unused reactions (= kept in the edge) from this library to the chemkin file. True means those reactions will be appended. Using just the string inputs would lead to a default value of *False*. In the previous example, this would look like:

```
reactionLibraries = ['Glarborg/C3'],
```

The reaction libraries are stored in `$RMG-database/input/kinetics/libraries/` and the *Location:* should be specified relative to this path.

Because the units for the Arrhenius parameters are given in each mechanism, the different mechanisms can have different units.

Note: While using a Reaction Library the user must be careful enough to provide all instances of a particular reaction in the library file, as RMG will ignore all reactions generated by its templates. For example, suppose you supply the Reaction Library with butyl_1 –> butyl_2. Although RMG would find two unique instances of this reaction (via a three- and four-member cyclic Transition State), RMG would only use the rate coefficient supplied by you in generating the mechanism.

RMG will not handle irreversible reactions correctly, if supplied in a Reaction Library.

Seed Mechanisms

The next section of the `input.py` file specifies which, if any, Seed Mechanisms should be used. If a seed mechanism is passed to RMG, every species and reaction present in the seed mechanism will be placed into the core, in addition to the species that are listed in the *List of species* section.

For details of the kinetics libraries included with RMG that can be used as a seed mechanism, see *Reaction Libraries*.

You can specify your own seed mechanism in the location section. Please note that the oxidation library should not be used for pyrolysis models. The syntax for the seed mechanisms is similar to that of the primary reaction libraries.

```
seedMechanisms = ['GRI-Mech3.0']
```

The seed mechanisms are stored in `RMG-database/input/kinetics/libraries/`

As the units for the Arrhenius parameters are given in each mechanism, different mechanisms can have different units. Additionally, if the same reaction occurs more than once in the combined mechanism, the instance of it from the first mechanism in which it appears is the one that gets used.

Kinetics Depositories

Kinetics depositories store reactions which can be used for rate estimation. Depositories are divided by the sources of the data. Currently, RMG database has two depositories. The main depository is *training* which contains reactions from various sources. This depository is loaded by default and can be disabled by adding *'!training'* to the list of depositories. The *NIST* depository contains reactions taken from NIST's gas kinetics database. The *kineticsDepositories* argument in the input file accepts a list of strings describing which depositories to include.:

```
kineticsDepositories = ['training']
```

Kinetics Families

In this section users can specify the particular reaction families that they wish to use to generate their model. This can be specified with any combination of specific families and predefined sets from RMG-database/input/kinetics/families/recommended.py.

For example, you can use only the H_Abstraction family to build the model:

```
kineticsFamilies = 'H_Abstraction'
```

You can also specify multiple families in a list:

```
kineticsFamilies = ['H_Abstraction', 'Disproportionation', 'R_Recombination']
```

To use a predefined set, simply specify its name:

```
kineticsFamilies = 'default'
```

You can use a mix of predefined sets and kinetics families:

```
kineticsFamilies = ['default', 'SubstitutionS']
```

It is also possible to request the inverse of a particular list:

```
kineticsFamilies = ['!default', '!SubstitutionS']
```

This will load all kinetics families except the ones in `'default'` and `'SubstitutionS'`.

Finally, you can also specify `'all'` or `'none'`, which may be useful in certain cases.

Kinetics Estimator

The last section is specifying that RMG is estimating kinetics of reactions from rate rules. For more details on how kinetic estimations is working check *Kinetics Estimation*:

```
kineticsEstimator = 'rate rules'
```

The following is an example of a database block, based on above chosen libraries and options:

```
database(
        thermoLibraries = ['primaryThermoLibrary', 'GRI-Mech3.0'],
        reactionLibraries = [('Glarborg/C3',False)],
        seedMechanisms = ['GRI-Mech3.0'],
        kineticsDepositories = ['training'],
        kineticsFamilies = 'defult',
        kineticsEstimator = 'rate rules',
)
```

1.4.3 List of species

Species to be included in the core at the start of your RMG job are defined in the species block. The label, reactive or inert, and structure of each reactant must be specified.

The label field will be used throughout your mechanism to identify the species. Inert species in the model can be defined by setting reactive to be False. Reaction families will no longer be applied to these species, but reactions of the inert from libraries and seed mechanisms will still be considered. For all other species the reactive status must be set as True. The structure of the species can be defined using either by using SMILES or adjacencyList.

The following is an example of a typical species item, based on methane using SMILE or adjacency list to define the structure:

```
species(
        label='CH4',
        reactive=True,
        structure=SMILES("C"),
)

species(
        label='CH4',
        reactive=True,
        structure=adjacencyList(
                """
                1 C 0
                """
)
```

1.4.4 Reaction System

Every reaction system we want the model to be generated at must be defined individually. Currently, RMG can only model constant temperature and pressure systems. Future versions will allow for variable temperature and pressure. To define a reaction system we need to define the temperature, pressure and initial mole fractions of the reactant species. The initial mole fractions are defined using the label for the species in the species block. Reaction system simulations terminate when one of the specified termination criteria are satisfied. Termination can be specied to occur at a specific time, at a specific conversion of a given initial species or to occur at a given terminationRateRatio, which is the characteristic flux in the system at that time divided by the maximum characteristic flux observed so far in the system (measure of how much chemistry is happening at a moment relative to the main chemical process).

The following is an example of a simple reactor system:

```
simpleReactor(
        temperature=(1350,'K'),
        pressure=(1.0,'bar'),
        initialMoleFractions={
```

```
                "CH4": 0.104,
                "H2": 0.0156,
                "N2": 0.8797,
        },
        terminationConversion={
                'CH4': 0.9,
        },
        terminationTime=(1e0,'s'),
        terminationRateRatio=0.01,
        sensitivity=['CH4','H2'],
        sensitivityThreshold=0.001,

)
```

Troubleshooting tip: if you are using a goal conversion rather than time, the reaction systems may reach equilibrium below the goal conversion, leading to a job that cannot converge physically. Therefore it is may be necessary to reduce the goal conversion or set a goal reaction time.

For sensitivity analysis, RMG-Py must be compiled with the DASPK solver, which is done by default but has some dependency restrictions. (See *License Restrictions on Dependencies* for more details.) The sensitivity and sensitivityThrehold are optional arguments for when the user would like to conduct sensitivity analysis with respect to the reaction rate coefficients for the list of species given for `sensitivity`.

Sensitivity analysis is conducted for the list of species given for `sensitivity` argument in the input file. The normalized concentration sensitivities with respect to the reaction rate coefficients dln(C_i)/dln(k_j) are saved to a csv file with the file name `sensitivity_1_SPC_1.csv` with the first index value indicating the reactor system and the second naming the index of the species the sensitivity analysis is conducted for. Sensitivities to thermo of individual species is also saved as semi normalized sensitivities dln(C_i)/d(G_j) where the units are given in 1/(kcal mol-1). The sensitivityThreshold is set to some value so that only sensitivities for dln(C_i)/dln(k_j) > sensitivityThreshold or dlnC_i/d(G_j) > sensitivityThreshold are saved to this file.

Note that in the RMG job, after the model has been generated to completion, sensitivity analysis will be conducted in one final simulation (sensitivity is not performed in intermediate iterations of the job).

Advanced Setting: Range Based Reactors

Under this setting rather than using reactors at fixed points, reaction conditions are sampled from a range of conditions. Conditions are chosen using a weighted stochastic grid sampling algorithm. An implemented objective function measures how desirable it is to sample from a point condition (T, P, concentrations) based on prior run conditions (weighted by how recent they were and how many objects they returned). Each iteration this objective function is evaluated at a grid of points spaning the reactor range (the grid has 20^N points where N is the number of dimensions). The grid values are then normalized to one and a grid point is chosen with probability equal to its normalized objective function value. Then a random step of maximum length sqrt(2)/2 times the distance between grid points is taken from that grid point to give the chosen condition point. The random numbers are seeded so that this does not make the algorithm non-deterministic.

These variable condition reactors run a defined number of times (`nSims`) each reactor cycle. Use of these reactors tends to improve treatment of reaction conditions that otherwise would be between reactors and reduce the number of simulations needed by focusing on reaction conditions at which the model terminates earlier. An example with sensitivity analysis at a specified reaction condition is available below:

```
simpleReactor(
        temperature=[(1000,'K'),(1500,'K')],
        pressure=[(1.0,'bar'),(10.0,'bar')],
        nSims=12,
```
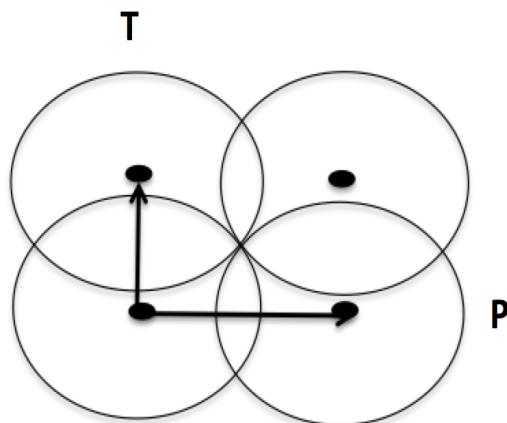
The T and P range, the grid points and the circle of conditions chosen from at each grid point.

```
        initialMoleFractions={
        "ethane": [0.05,0.15],
        "O2": 0.1,
        "N2": 0.9,
        },
        terminationConversion={
        'ethane': 0.1,
        },
        terminationTime=(1e1,'s'),
        sensitivityTemperature = (1000,'K'),
        sensitivityPressure = (10.0,'bar'),
        sensitivityMoleFractions = {"ethane":0.1,"O2":0.9},
        sensitivity=["ethane","O2"],
        sensitivityThreshold=0.001,
        balanceSpecies = "N2",
        )
```

Note that increasing `nSims` improves convergence over the entire range, but convergence is only guaranteed at the last set of `nSims` reaction conditions. Theoretically if `nSims` is set high enough the RMG model converges over the entire interval. Except at very small values for `nSims` the convergence achieved is usually as good or superior to that achieved using the same number of evenly spaced fixed reactors.

If there is a particular reaction condition you expect to converge more slowly than the rest of the range there is virtually no cost to using a single condition reactor (or a ranged reactor at a smaller range) at that condition and a ranged reactor with a smaller value for nSims. This is because the fixed reactor simulations will almost always be useful and keep the overall RMG job from terminating while the ranged reactor samples the faster converging conditions.

What you should actually set `nSims` to is very system dependent. The value you choose should be at least 2 + N where N is the number of dimensions the reactor spans (T=>N=1, T and P=>N=2, etc. . .). There may be benefits to setting it as high as 2 + 5N. The first should give you convergence over most of the interval that is almost always better than the same number of fixed reactors. The second should get you reasonably close to convergence over the entire range for N <= 2.

For gas phase reactors if normalization of the ranged mole fractions is undesirable (eg. perhaps a specific species mole fractions needs to be kept constant) one can use a `balanceSpecies`. When a `balanceSpecies` is used instead of

normalizing the mole fractions the concentration of the defined `balanceSpecies` is adjusted to maintain an overall mole fraction of one. This ensures that all species except the `balanceSpecies` have mole fractions within the range specified.

1.4.5 Simulator Tolerances

The next two lines specify the absolute and relative tolerance for the ODE solver, respectively. Common values for the absolute tolerance are 1e-15 to 1e-25. Relative tolerance is usually 1e-4 to 1e-8:

```
simulator(
    atol=1e-16,
    rtol=1e-8,
    sens_atol=1e-6,
    sens_rtol=1e-4,
)
```

The `sens_atol` and `sens_rtol` are optional arguments for the sensitivity absolute tolerance and sensitivity relative tolerances, respectively. They are set to a default value of 1e-6 and 1e-4 respectively unless the user specifies otherwise. They do not apply when sensitivity analysis is not conducted.

1.4.6 Model Tolerances

Model tolerances dictate how species get included in the model. For more information, see the theory behind how RMG builds models using the *Flux-based Algorithm*. For running an initial job, it is recommended to only change the `toleranceMoveToCore` and `toleranceInterruptSimulation` values to an equivalent desired value. We find that typically a value between `0.01` and `0.05` is best. If your model cannot converge within a few hours, more advanced settings such as *reaction filtering* or *pruning* can be turned on to speed up your simulation at a slight risk of omitting chemistry.

```
model(
    toleranceMoveToCore=0.1,
    toleranceInterruptSimulation=0.1,
)
```

- `toleranceMoveToCore` indicates how high the edge flux ratio for a species must get to enter the core model. This tolerance is designed for controlling the accuracy of final model.

- `toleranceInterruptSimulation` indicates how high the edge flux ratio must get to interrupt the simulation (before reaching the `terminationConversion` or `terminationTime`). This value should be set to be equal to `toleranceMoveToCore` unless the advanced *pruning* feature is desired.

Advanced Setting: Speed Up by Filtering Reactions

For generating models for larger molecules, RMG-Py may have trouble converging because it must find reactions on the order of $(n_{reaction\ sites})^{n_{species}}$. Thus it can be further sped up by pre-filtering reactions that are added to the model. This modification to the algorithm does not react core species together until their concentrations are deemed high enough. It is recommended to turn on this flag when the model does not converge with normal parameter settings. See *Filtering Reactions within the Flux-based Algorithm*. for more details.

```
model(
    toleranceMoveToCore=0.1,
    toleranceInterruptSimulation=0.1,
    filterReactions=True,
```

```
    filterThreshold=5e8,
)
```

Additional parameters:

- `filterReactions`: set to `True` if reaction filtering is turned on. By default it is set to False.

- `filterThreshold`: click *here* for more description about its effect. Default: `5e8`

Advanced Setting: Speed Up by Pruning

For further speed-up, it is also possible to perform mechanism generation with pruning of "unimportant" edge species to reduce memory usage.

A typical set of parameters for pruning is:

```
model(
    toleranceMoveToCore=0.5,
    toleranceInterruptSimulation=1e8,
    toleranceKeepInEdge=0.05,
    maximumEdgeSpecies=200000,
    minCoreSizeForPrune=50,
    minSpeciesExistIterationsForPrune=2,
    )
```

Additional parameters:

- `toleranceKeepInEdge` indicates how low the edge flux ratio for a species must be to keep on the edge. This should be set to zero, which is its default.

- `maximumEdgeSpecies` indicates the upper limit for the size of the edge. The default value is set to `1000000` species.

- `minCoreSizeForPrune` ensures that a minimum number of species are in the core before pruning occurs, in order to avoid pruning the model when it is far away from completeness. The default value is set to 50 species.

- `minSpeciesExistIterationsForPrune` is set so that the edge species stays in the job for at least that many iterations before it can be pruned. The default value is 2 iterations.

Recommendations:

We recommend setting `toleranceKeepInEdge` to not be larger than 10% of `toleranceMoveToCore`, based on a pruning case study. In order to always enable pruning, `toleranceInterruptSimulation` should be set as a high value, e.g. 1e8. `maximumEdgeSpecies` can be adjusted based on user's RAM size. Usually 200000 edge species would cause memory shortage of 8GB computer, setting `maximumEdgeSpecies = 200000` (or lower values) could effectively prevent memory crash.

Additional Notes:

Note that when using pruning, RMG will not prune unless all reaction systems reach the goal reaction time or conversion without exceeding the `toleranceInterruptSimulation`. Therefore, you may find that RMG is not pruning even though the model edge size exceeds `maximumEdgeSpecies`, or an edge species has flux below the `toleranceKeepInEdge`. This is a safety check within RMG to ensure that species are not pruned too early, resulting in inaccurate chemistry. In order to increase the likelihood of pruning you can try increasing `toleranceInterruptSimulation` to an arbitrarily high value.

As a contrast, a typical set of parameters for non-pruning is:

```
model(
    toleranceKeepInEdge=0,
    toleranceMoveToCore=0.5,
    toleranceInterruptSimulation=0.5,
)
```

where `toleranceKeepInEdge` is always 0, meaning all the edge species will be kept in edge since all the edge species have positive flux. `toleranceInterruptSimulation` equals to `toleranceMoveToCore` so that ODE simulation get interrupted once discovering a new core species. Because the ODE simulation is always interrupted, no pruning is performed.

Please find more details about the theory behind pruning at *Pruning Theory*.

Advanced Setting: Thermodynamic Pruning

Thermodynamic pruning is an alternative to flux pruning that does not require a given simulation to complete to remove excess species. The thermodynamic criteria is calculated by determining the minimum and maximum Gibbs energies of formation (Gmin and Gmax) among species in the core. If the Gibbs energy of formation of a given species is G the value of the criteria is (G-Gmax)/(Gmax-Gmin). All of the Gibbs energies are evaluated at the highest temperature used in all of the reactor systems. This means that a value of 0.2 for the criterion implies that it will not add species that have Gibbs energies of formation greater than 20% of the core Gibbs energy range greater than the maximum Gibbs energy of formation within the core.

For example

```
model(
    toleranceMoveToCore=0.5,
    toleranceInterruptSimulation=0.5,
    toleranceThermoKeepSpeciesInEdge=0.5,
    maximumEdgeSpecies=200000,
    minCoreSizeForPrune=50,
    )
```

Advantages over flux pruning:

Species are removed immediately if they violate tolerance Completing a simulation is unnecessary for this pruning so there is no need to waste time setting the interrupt tolerance higher than the movement tolerance. Will always maintain the correct `maximumEdgeSpecies`.

Primary disadvantage:

Since we determine whether to add species primarily based on flux, at tight tolerances this is more likely to kick out species RMG might otherwise have added to core.

Advanced Setting: Taking Multiple Species At A Time

Taking multiple objects (species, reactions or pdepNetworks) during a given simulation can often decrease your overall model generation time over only taking one. For this purpose there is a `maxNumObjsPerIter` parameter that allows RMG to take that many species, reactions or pdepNetworks from a given simulation. This is done in the order they trigger their respective criteria.

You can also set `terminateAtMaxObjects=True` to cause it to terminate when it has the maximum number of objects allowed rather than waiting around until it hits an interrupt tolerance. This avoids additional simulation time, but will also make it less likely to finish simulations, which can affect flux pruning.

For example

```
model(
        toleranceKeepInEdge=0.0,
        toleranceMoveToCore=0.1,
        toleranceInterruptSimulation=0.3,
        maxNumObjsPerIter=2,
        terminateAtMaxObjects=True,
)
```

Note that this can also result in larger models, however, sometimes these larger models (from taking more than one object at a time) pick up chemistry that would otherwise have been missed.

Advanced Setting: Dynamics Criterion

While the flux criterion works very well for identifying new species that have high flux and therefore will likely be high throughput or high concentration species, it provides few automatic guarantees about how well a given model will accurately represent the concentrations of the involved species. The dynamics criterion is more complex than the flux criterion, but in general it is a measure of how much impact a given reaction will have on the concentrations of core species. A more detailed explanation is available in the theory section: *Dynamics Criterion*.

Reasonable values for the dynamics criterion range typically between 2-30.

For example

```
model(
        toleranceMoveToCore=0.1,
        toleranceInterruptSimulation=0.1,
        toleranceMoveEdgeReactionToCore=10.0,
        toleranceMoveEdgeReactionToCoreInterrupt=5.0,
)
```

Note that it is highly recommended to use the dynamics criterion only alongside the flux criterion and not by itself.

Advanced Setting: Surface Algorithm

One common issue with the dynamics criterion is that it treats all core species equally as discussed in our theory section: *Dynamics Criterion*. Because of this, if the dynamics criterion is set too low it enters a feedback loop where it adds species and then since it can't get those species' concentrations right it adds more species and so on. In order to avoid this feedback loop the surface algorithm was developed. It creates a new partition called the *surface* that is considered part of the core. We will refer to the part of the core that is not part of the surface as the *bulk core*. When operating without the dynamics criterion everything moves from edge to the bulk core as usual; however the dynamics criterion is managed differently. When using the surface algorithm most reactions pulled in by the dynamics criterion enter the surface instead of the bulk core. However, unlike movement to bulk core a constraint is placed on movement to the surface. Any reaction moved to the surface must have either both reactants or both products in the bulk core. This prevents the dynamics criterion from pulling in reactions to get the concentrations of species in the surface right avoiding the feedback loop. To avoid important species being trapped in the surface we also add criteria for movement from surface to bulk core based on flux or dynamics criterion.

For example

```
model(
        toleranceMoveToCore=0.1,
        toleranceInterruptSimulation=0.1,
        toleranceMoveEdgeReactionToCore=30.0,
        toleranceMoveEdgeReactionToCoreInterrupt=5.0,
        toleranceMoveEdgeReactionToSurface=10.0,
```

```
        toleranceMoveSurfaceSpeciesToCore=.01,
        toleranceMoveSurfaceReactionToCore=5.0,
)
```

1.4.7 On the fly Quantum Calculations

This block is used when quantum mechanical calculations are desired to determine thermodynamic parameters. These calculations are only run if the molecule is not included in a specified thermo library. The `onlyCyclics` option, if `True`, only runs these calculations for cyclic species. In this case, group additive estimates are used for all other species.

Molecular geometries are estimated via RDKit [RDKit]. Either MOPAC (2009 and 2012) or GAUSSIAN (2003 and 2009) can be used with the semi-empirical pm3, pm6, and pm7 (pm7 only available in MOPAC2012), specified in the software and method blocks. A folder can be specified to store the files used in these calculations, however if not specified this defaults to a *QMfiles* folder in the output folder.

The calculations are also only run on species with a maximum radical number set by the user. If a molecule has a higher radical number, the molecule is saturated with hydrogen atoms, then quantum mechanical calculations with subsequent hydrogen bond incrementation is used to determine the thermodynamic parameters.

The following is an example of the quantum mechanics options

```
quantumMechanics(
        software='mopac',
        method='pm3',
        fileStore='QMfiles',
        scratchDirectory = None,
        onlyCyclics = True,
        maxRadicalNumber = 0,
        )
```

1.4.8 Machine Learning-based Thermo Estimation

This block is used to turn on the machine learning module to estimate thermodynamic parameters. These calculations are only run if the molecule is not included in a specified thermo library. There are a number of different settings that can be specified to tune the estimator so that it only tries to estimate some species. This is useful because the machine learning model may perform poorly for some molecules and group additivity may be more suitable. Using the machine learning estimator for fused cyclic species of moderate size or any species with significant proportions of oxygen and/or nitrogen atoms will most likely yield better estimates than group additivity.

The available options with their default values are

```
mlEstimator(
    thermo=True,
    name='main',
    minHeavyAtoms=1,
    maxHeavyAtoms=None,
    minCarbonAtoms=0,
    maxCarbonAtoms=None,
    minOxygenAtoms=0,
    maxOxygenAtoms=None,
    minNitrogenAtoms=0,
    maxNitrogenAtoms=None,
```

```
    onlyCyclics=False,
    minCycleOverlap=0,
    H298UncertaintyCutoff=(3.0, 'kcal/mol'),
    S298UncertaintyCutoff=(2.0, 'cal/(mol*K)'),
    CpUncertaintyCutoff=(2.0, 'cal/(mol*K)')
)
```

`name` is the name of the folder containing the machine learning model architecture and parameters in the RMG database. The next several options allow setting limits on the numbers of atoms. `onlyCyclics` means that only cyclic species will be estimated. `minCycleOverlap` specified the minimum number of atoms that must be shared between any two cycles. For example, if there are only disparate monocycles or no cycles in a species, the overlap is zero; "spiro" cycles have an overlap of one; "fused" cycles have an overlap of two; and "bridged" cycles have an overlap of at least three. Note that specifying any value greater than zero will automatically restrict the machine learning estimator to only consider cyclic species regardless of the `onlyCyclics` setting.

If the estimated uncertainty of the thermo prediction is greater than any of the `UncertaintyCutoff` values, then machine learning estimation is not used for that species.

1.4.9 Pressure Dependence

This block is used when the model should account for pressure dependent rate coefficients. RMG can estimate pressure dependence kinetics based on `Modified Strong Collision` and `Reservoir State` methods. The former utilizes the modified strong collision approach of Chang, Bozzelli, and Dean [Chang2000], and works reasonably well while running more rapidly. The latter utilizes the steady-state/reservoir-state approach of Green and Bhatti [Green2007], and is more theoretically sound but more expensive.

The following is an example of pressure dependence options

```
pressureDependence(
        method='modified strong collision',
        maximumGrainSize=(0.5,'kcal/mol'),
        minimumNumberOfGrains=250,
        temperatures=(300,2000,'K',8),
        pressures=(0.01,100,'bar',5),
        interpolation=('Chebyshev', 6, 4),
        maximumAtoms=16,
)
```

The various options are as follows:

Method used for estimating pressure dependent kinetics

To specify the modified strong collision approach, this item should read

```
method='Modified Strong Collision'
```

To specify the reservoir state approach, this item should read

```
method='Reservoir State'
```

For more information on the two methods, consult the following resources :

Grain size and minimum number of grains

Since the $k(E)$ requires discretization in the energy space, we need to specify the number of energy grains to use when solving the Master Equation. The default value for the minimum number of grains is 250; this was selected to balance the speed and accuracy of the Master Equation solver method. However, for some pressure-dependent networks, this number of energy grains will result in the pressure-dependent $k(T, P)$ being greater than the high-P limit

```
maximumGrainSize=(0.5,'kcal/mol')
minimumNumberOfGrains=250
```

Temperature and pressure for the interpolation scheme

To generate the $k(T, P)$ interpolation model, a set of temperatures and pressures must be used. RMG can do this automatically, but it must be told a few parameters. We need to specify the limits of the temperature and pressure for the fitting of the interpolation scheme and the number of points to be considered in between this limit. For typical combustion model temperatures of the experiments range from 300 - 2000 K and pressure 1E-2 to 100 bar

```
temperatures=(300,2000,'K',8)
pressures=(0.01,100,'bar',5)
```

Interpolation scheme

To use logarithmic interpolation of pressure and Arrhenius interpolation for temperature, use the line

```
interpolation=('PDepArrhenius',)
```

The auxillary information printed to the Chemkin chem.inp file will have the "PLOG" format. Refer to Section 3.5.3 of the `CHEMKIN_Input.pdf` document and/or Section 3.6.3 of the `CHEMKIN_Theory.pdf` document. These files are part of the CHEMKIN manual.

To fit a set of Chebyshev polynomials on inverse temperature and logarithmic pressure axes mapped to [-1,1], specify *''Chebyshev'* interpolation. You should also specify the number of temperature and pressure basis functions by adding the appropriate integers. For example, the following specifies that six basis functions in temperature and four in pressure should be used

```
interpolation=('Chebyshev', 6, 4)
```

The auxillary information printed to the Chemkin chem.inp file will have the "CHEB" format. Refer to Section 3.5.3 of the `CHEMKIN_Input.pdf` document and/or Section 3.6.4 of the `CHEMKIN_Theory.pdf` document.

Regarding the number of polynomial coeffients for Chebyshev interpolated rates, plese refer to the `rmgpy.kinetics. Chebyshev` documentation. The number of pressures and temperature coeffcents should always be smaller than the respective number of user-specified temperatures and pressures.

Maximum size of adduct for which pressure dependence kinetics be generated

By default pressure dependence is run for every system that might show pressure dependence, i.e. every isomerization, dissociation, and association reaction. In reality, larger molecules are less likely to exhibit pressure-dependent behavior than smaller molecules due to the presence of more modes for randomization of the internal energy. In certain cases involving very large molecules, it makes sense to only consider pressure dependence for molecules smaller than some user-defined number of atoms. This is specified e.g. using the line

```
maximumAtoms=16
```

to turn off pressure dependence for all molecules larger than the given number of atoms (16 in the above example).

1.4.10 Miscellaneous Options

Miscellaneous options:

```
options(
    name='Seed',
    generateSeedEachIteration=True,
    saveSeedToDatabase=True,
    units='si',
    saveRestartPeriod=(1,'hour'),
    generateOutputHTML=True,
    generatePlots=False,
    saveSimulationProfiles=True,
    verboseComments=False,
    saveEdgeSpecies=True,
    keepIrreversible=True,
    trimolecularProductReversible=False,
)
```

The `name` field is the name of any generated seed mechanisms

Setting `generateSeedEachIteration` to `True` tells RMG to save and update a seed mechanism and thermo library during the current run

Setting `saveSeedToDatabase` to `True` tells RMG (if generating a seed) to also save that seed mechanism and thermo library directly into the database

The `units` field is set to `si`. Currently there are no other unit options.

The `saveRestartPeriod` indictes how frequently you wish to save restart files. For very large/long RMG jobs, this process can take a significant amount of time. In such cases, the user may wish to increase the time period for saving these restart files.

Setting `generateOutputHTML` to `True` will let RMG know that you want to save 2-D images (png files in the local `species` folder) of all species in the generated core model. It will save a visualized HTML file for your model containing all the species and reactions. Turning this feature off by setting it to `False` may save memory if running large jobs.

Setting `generatePlots` to `True` will generate a number of plots describing the statistics of the RMG job, including the reaction model core and edge size and memory use versus execution time. These will be placed in the output directory in the plot/ folder.

Setting `saveSimulationProfiles` to `True` will make RMG save csv files of the simulation in .csv files in the `solver/` folder. The filename will be `simulation_1_26.csv` where the first number corresponds to the reaciton system, and the second number corresponds to the total number of species at the point of the simulation. Therefore, the highest second number will indicate the latest simulation that RMG has complete while enlarging the core model. The information inside the csv file will provide the time, reactor volume in m^3, as well as mole fractions of the individual species.

Setting `verboseComments` to `True` will make RMG generate chemkin files with complete verbose commentary for the kinetic and thermo parameters. This will be helpful in debugging what values are being averaged for the kinetics. Note that this may produce very large files.

Setting `saveEdgeSpecies` to `True` will make RMG generate chemkin files of the edge reactions in addition to the core model in files such as `chem_edge.inp` and `chem_edge_annotated.inp` files located inside the `chemkin`

folder. These files will be helpful in viewing RMG's estimate for edge reactions and seeing if certain reactions one expects are actually in the edge or not.

Setting `keepIrreversible` to `True` will make RMG import library reactions as is, whether they are reversible or irreversible in the library. Otherwise, if `False` (default value), RMG will force all library reactions to be reversible, and will assign the forward rate from the relevant library.

Setting `trimolecularProductReversible` to `False` will not allow families with three products to react in the reverse direction. Default is `True`.

1.4.11 Species Constraints

RMG can generate mechanisms with a number of optional species constraints, such as total number of carbon atoms or electrons per species. These are applied to all of RMG's reaction families.

```
generatedSpeciesConstraints(
    allowed=['input species','seed mechanisms','reaction libraries'],
    maximumCarbonAtoms=10,
    maximumOxygenAtoms=2,
    maximumNitrogenAtoms=2,
    maximumSiliconAtoms=2,
    maximumSulfurAtoms=2,
    maximumHeavyAtoms=10,
    maximumRadicalElectrons=2,
    maximumSingletCarbenes=1,
    maximumCarbeneRadicals=0,
    maximumIsotopicAtoms=2,
    allowSingletO2 = False,
)
```

An additional flag `allowed` can be set to allow species from either the input file, seed mechanisms, or reaction libraries to bypass these constraints. Note that this should be done with caution, since the constraints will still apply to subsequent products that form.

Note that under all circumstances all forbidden species will still be banned unless they are manually removed from the database. See *Kinetics Database* for more information on forbidden groups.

By default, the `allowSingletO2` flag is set to `False`. See *Representing Oxygen* for more information.

1.4.12 Staging

It is now possible to concatenate different model and simulator blocks into the same run in stages. Any given stage will terminate when the RMG run terminates and then the current group of model and simulator parameters will be switched out with the next group and the run will continue until that stage terminates. Once the last stage terminates the run ends normally. This is currently enabled only for the model and simulator blocks.

There must be the same number of each of these blocks (although only having one simulator block and many model blocks is enabled as well) and RMG will enter each stage these define in the order they were put in the input file.

To enable easier manipulation of staging a new parameter in the model block was developed maxNumSpecies that is the number of core species at which that stage (or if it is the last stage the entire model generation process) will terminate.

For example

```
model(
        toleranceKeepInEdge=0.0,
        toleranceMoveToCore=0.1,
        toleranceInterruptSimulation=0.1,
        maximumEdgeSpecies=100000,
        maxNumSpecies=100
)
```

1.5 Example Input Files

Perhaps the best way to learn the input file syntax is by example. To that end, a number of example input files and their corresponding output have been given in the `examples` directory. Two of the RMG jobs are shown below.

1.5.1 Ethane pyrolysis (Minimal)

This is the minimal example file characterizing a very basic system for ethane pyrolysis and should run quickly if RMG is set up properly. It does not include any calculation of pressure-dependent reaction rates.

```
# Data sources
database(
    thermoLibraries = ['primaryThermoLibrary'],
    reactionLibraries = [],
    seedMechanisms = [],
    kineticsDepositories = ['training'],
    kineticsFamilies = 'default',
    kineticsEstimator = 'rate rules',
)

# List of species
species(
    label='ethane',
    reactive=True,
    structure=SMILES("CC"),
)

# Reaction systems
simpleReactor(
    temperature=(1350,'K'),
    pressure=(1.0,'bar'),
    initialMoleFractions={
        "ethane": 1.0,
    },
    terminationConversion={
        'ethane': 0.9,
    },
    terminationTime=(1e6,'s'),
)

simulator(
    atol=1e-16,
    rtol=1e-8,
)
```

(continues on next page)

```
model(
    toleranceKeepInEdge=0.0,
    toleranceMoveToCore=0.1,
    toleranceInterruptSimulation=0.1,
    maximumEdgeSpecies=100000,
)

options(
    units='si',
    saveRestartPeriod=None,
    generateOutputHTML=True,
    generatePlots=False,
    saveEdgeSpecies=True,
    saveSimulationProfiles=True,
)
```

1.5.2 1,3-hexadiene pyrolysis

This example models the pyrolysis of 1,3-hexadiene and demonstrates the effect of turning on the pressure-dependence module within RMG.

```
# Data sources
database(
    thermoLibraries = ['primaryThermoLibrary', 'GRI-Mech3.0'],
    reactionLibraries = [],
    seedMechanisms = [],
    kineticsDepositories = ['training'],
    kineticsFamilies = 'default',
    kineticsEstimator = 'rate rules',
)

# Constraints on generated species
generatedSpeciesConstraints(
    maximumRadicalElectrons = 2,
)

# List of species
species(
    label='HXD13',
    reactive=True,
    structure=SMILES("C=CC=CCC"),
)
species(
    label='CH4',
    reactive=True,
    structure=SMILES("C"),
)
species(
    label='H2',
    reactive=True,
    structure=adjacencyList(
        """
        1 H u0 p0 {2,S}
        2 H u0 p0 {1,S}
        """),
```

```
)
species(
    label='N2',
    reactive=False,
    structure=InChI("InChI=1/N2/c1-2"),
)

# Reaction systems
simpleReactor(
    temperature=(1350,'K'),
    pressure=(1.0,'bar'),
    initialMoleFractions={
        "HXD13": 6.829e-4,
        "CH4": 0.104,
        "H2": 0.0156,
        "N2": 0.8797,
    },
    terminationConversion={
        'HXD13': 0.9,
    },
    terminationTime=(1e0,'s'),
)

simulator(
    atol=1e-16,
    rtol=1e-8,
)

model(
    toleranceKeepInEdge=0.0,
    toleranceMoveToCore=0.5,
    toleranceInterruptSimulation=0.5,
    maximumEdgeSpecies=100000
)

quantumMechanics(
    software='mopac',
    method='pm3',
    # fileStore='QMfiles', # relative to where it is run. Defaults within the output folder.
    scratchDirectory = None, # not currently used
    onlyCyclics = True,
    maxRadicalNumber = 0,
    )

pressureDependence(
    method='modified strong collision',
    maximumGrainSize=(0.5,'kcal/mol'),
    minimumNumberOfGrains=250,
    temperatures=(300,2000,'K',8),
    pressures=(0.01,100,'bar',5),
    interpolation=('Chebyshev', 6, 4),
)

options(
    units='si',
    saveRestartPeriod=(1,'hour'),
    generateOutputHTML=False,
```

```
    generatePlots=False,
)
```

1.5.3 Commented input file

This is a fully commented input file with all optional blocks for new users to better understand the options of rmg input files

```python
# Data sources
database(
    # overrides RMG thermo calculation of RMG with these values.
    # libraries found at http://rmg.mit.edu/database/thermo/libraries/
    # if species exist in multiple libraries, the earlier libraries overwrite the
    # previous values
    thermoLibraries=['BurkeH2O2', 'primaryThermoLibrary', 'DFT_QCI_thermo', 'CBS_QB3_1dHR'],
    # overrides RMG kinetics estimation if needed in the core of RMG.
    # list of libraries found at http://rmg.mit.edu/database/kinetics/libraries/
    # libraries can be input as either a string or tuple of form ('library_name',True/False)
    # where a `True` indicates that all unused reactions will be automatically added
    # to the chemkin file at the end of the simulation. Placing just string values
    # defaults the tuple to `False`. The string input is sufficient in almost
    # all situations
    reactionLibraries=[('C3', False)],
    # seed mechanisms are reactionLibraries that are forced into the initial mechanism
    # in addition to species listed in this input file.
    # This is helpful for reducing run time for species you know will appear in
    # the mechanism.
    seedMechanisms=['BurkeH2O2inN2', 'ERC-FoundationFuelv0.9'],
    # this is normally not changed in general RMG runs.  Usually used for testing with
    # outside kinetics databases
    kineticsDepositories='default',
    # lists specific families used to generate the model. 'default' uses a list of
    # families from RMG-Database/input/kinetics/families/recommended.py
    # a visual list of families is available in PDF form at RMG-database/families
    kineticsFamilies='default',
    # specifies how RMG calculates rates.  currently, the only option is 'rate rules'
    kineticsEstimator='rate rules',
)

# List of species
# list initial and expected species below to automatically put them into the core mechanism.
# 'structure' can utilize method of SMILES("put_SMILES_here"),
# adjacencyList("""put_adj_list_here"""), or InChI("put_InChI_here")
# for molecular oxygen, use the smiles string [O][O] so the triplet form is used
species(
    label='butane',
    reactive=True,  # this parameter is optional if true
    structure=SMILES("CCCC"),
)
species(
    label='O2',
    structure=SMILES("[O][O]"),
)
species(
    label='N2',
```

```
    reactive=False,
    structure=adjacencyList("""
    1 N u0 p1 c0 {2,T}
    2 N u0 p1 c0 {1,T}
    """),
)
# You can list species not initially in reactor to make sure RMG includes them in the mechanism
species(
    label='QOOH',
    reactive=True,
    structure=SMILES("OOCC[CH]C")
)
species(
    label='CO2',
    reactive=True,
    structure=SMILES("O=C=O")
)

# Reaction systems
# currently RMG models only constant temperature and pressure as homogeneous batch reactors.
# two options are: simpleReactor for gas phase or liquidReactor for liquid phase
# use can use multiple reactors in an input file for each condition you want to test.
simpleReactor(
    # specifies reaction temperature with units
    temperature=(700, 'K'),
    # specifies reaction pressure with units
    pressure=(10.0, 'bar'),
    # list initial mole fractions of compounds using the label from the 'species' label.
    # RMG will normalize if sum/=1
    initialMoleFractions={
        "N2": 4,
        "O2": 1,
        "butane": 1. / 6.5,
    },
    # the following two values specify when to determine the final output model
    # only one must be specified
    # the first condition to be satisfied will terminate the process
    terminationConversion={
        'butane': .99,
    },
    terminationTime=(40, 's'),
    # the next two optional values specify how RMG computes sensitivities of
    # rate coefficients with respect to species concentrations.
    # sensitivity contains a list of species' labels to conduct sensitivity analysis on.
    # sensitvityThreshold is the required sensitiviy to be recorded in the csv output file
    # sensitivity=['CH4'],
    # sensitivityThreshold=0.0001,
)

# liquidReactor(
#     temperature=(500, 'K'),
#     initialConcentrations={
#         "N2": 4,
#         "O2": 1,
#         "CO": 1,
#     },
#     terminationConversion=None,
```

```
#     terminationTime=(3600, 's'),
#     sensitivity=None,
#     sensitivityThreshold=1e-3
# )
# liquid reactors also have solvents, you can specify one solvent
# list of solvents available at : http://rmg.mit.edu/database/solvation/libraries/solvent/
# solvation('water')

# determines absolute and relative tolerances for ODE solver and sensitivities.
# normally this doesn't cause many issues and is modified after other issues are
# ruled out
simulator(
    atol=1e-16,
    rtol=1e-8,
    # sens_atol=1e-6,
    # sens_rtol=1e-4,
)

# used to add species to the model and to reduce memory usage by removing unimportant␣
↪additional species.
# all relative values are normalized by a characteristic flux at that time point
model(
    # determines the relative flux to put a species into the core.
    # A smaller value will result in a larger, more complex model
    # when running a new model, it is recommended to start with higher values and then decrease␣
↪to converge on the model
    toleranceMoveToCore=0.1,
    # comment out the next three terms to disable pruning
    # determines the relative flux needed to not remove species from the model.
    # Lower values will keep more species and utilize more memory
    toleranceKeepInEdge=0.01,
    # determines when to stop a ODE run to add a species.
    # Lower values will improve speed.
    # if it is too low, may never get to the end simulation to prune species.
    toleranceInterruptSimulation=1,
    # number of edge species needed to accumulate before pruning occurs
    # larger values require more memory and will prune less often
    maximumEdgeSpecies=100000,
    # minimum number of core species needed before pruning occurs.
    # this prevents pruning when kinetic model is far away from completeness
    minCoreSizeForPrune=50,
    # make sure that the pruned edge species have existed for a set number of RMG iterations.
    # the user can specify to increase it from the default value of 2
    minSpeciesExistIterationsForPrune=2,
    # filter the reactions during the enlarge step to omit species from reacting if their
    # concentration are deemed to be too low
    filterReactions=False,
    # for bimolecular reactions, will only allow them to react if
    # filterThreshold*C_A*C_B > toleranceMoveToCore*characteristic_rate
    # and if filterReactions=True
    filterThreshold=1e8,
)

options(
    # provides a name for the seed mechanism produced at the end of an rmg run default is 'Seed'
    name='SeedName',
    # if True every iteration it saves the current model as libraries/seeds
```

```python
    # (and deletes the old one)
    # Unlike HTML this is inexpensive time-wise
    # note a seed mechanism will be generated at the end of a completed run and some incomplete
    # runs even if this is set as False
    generateSeedEachIteration=True,
    # If True the mechanism will also be saved directly as kinetics and thermo libraries in the␣
↪database
    saveSeedToDatabase=False,
    # only option is 'si'
    units='si',
    # how often you want to save restart files.
    # takes significant amount of time. comment out if you don't want to save
    saveRestartPeriod=None,
    # Draws images of species and reactions and saves the model output to HTML.
    # May consume extra memory when running large models.
    generateOutputHTML=True,
    # generates plots of the RMG's performance statistics. Not helpful if you just want a model.
    generatePlots=False,
    # saves mole fraction of species in 'solver/' to help you create plots
    saveSimulationProfiles=False,
    # gets RMG to output comments on where kinetics were obtained in the chemkin file.
    # useful for debugging kinetics but increases memory usage of the chemkin output file
    verboseComments=False,
    # gets RMG to generate edge species chemkin files. Uses lots of memory in output.
    # Helpful for seeing why some reaction are not appearing in core model.
    saveEdgeSpecies=False,
    # Sets a time limit in the form DD:HH:MM:SS after which the RMG job will stop. Useful for␣
↪profiling on jobs that
    # do not converge.
    # wallTime = '00:00:00',
    # Forces RMG to import library reactions as reversible (default). Otherwise, if set to True,
↪ RMG will import library
    # reactions while keeping the reversibility as as.
    keepIrreversible=False,
    # Allows families with three products to react in the diverse direction (default).
    trimolecularProductReversible=True,
)

# optional module allows for correction to unimolecular reaction rates at low pressures and/or␣
↪temperatures.
pressureDependence(
    # two methods available: 'modified strong collision' is faster and less accurate than␣
↪'reservoir state'
    method='modified strong collision',
    # these two categories determine how fine energy is descretized.
    # more grains increases accuracy but takes longer
    maximumGrainSize=(0.5, 'kcal/mol'),
    minimumNumberOfGrains=250,
    # the conditions for the rate to be output over
    # parameter order is: low_value, high_value, units, internal points
    temperatures=(300, 2200, 'K', 2),
    pressures=(0.01, 100, 'bar', 3),
    # The two options for interpolation are 'PDepArrhenius' (no extra arguments) and
    # 'Chebyshev' which is followed by the number of basis sets in
    # Temperature and Pressure. These values must be less than the number of
    # internal points specified above
    interpolation=('Chebyshev', 6, 4),
```

```
    # turns off pressure dependence for molecules with number of atoms greater than the number␣
↪specified below
    # this is due to faster internal rate of energy transfer for larger molecules
    maximumAtoms=15,
)

# optional block adds constraints on what RMG can output.
# This is helpful for improving the efficiency of RMG, but wrong inputs can lead to many errors.
generatedSpeciesConstraints(
    # allows exceptions to the following restrictions
    allowed=['input species', 'seed mechanisms', 'reaction libraries'],
    # maximum number of each atom in a molecule
    maximumCarbonAtoms=4,
    maximumOxygenAtoms=7,
    maximumNitrogenAtoms=0,
    maximumSiliconAtoms=0,
    maximumSulfurAtoms=0,
    # max number of non-hydrogen atoms
    # maximumHeavyAtoms=20,
    # maximum radicals on a molecule
    maximumRadicalElectrons=1,
    # maximum number of singlet carbenes (lone pair on a carbon atom) in a molecule
    maximumSingletCarbenes=1,
    # maximum number of radicals on a molecule with a singlet carbene
    # should be lower than maximumRadicalElectrons in order to have an effect
    maximumCarbeneRadicals=0,
    # If this is false or missing, RMG will throw an error if the more less-stable form of O2␣
↪is entered
    # which doesn't react in the RMG system. normally input O2 as triplet with SMILES [O][O]
    # allowSingletO2=False,
    # maximum allowed number of non-normal isotope atoms:
    # maximumIsotopicAtoms=2,
)

# optional block allows thermo to be estimated through quantum calculations
# quantumMechanics(
#     # the software package for calculations...can use 'mopac' or 'gaussian' if installed
#     software='mopac',
#     # methods available for calculations. 'pm2' 'pm3' or 'pm7' (last for mopac only)
#     method='pm3',
#     # where to store calculations
#     fileStore='QMfiles',
#     # where to store temporary run files
#     scratchDirectory=None,
#     # onlyCyclics allows linear molecules to be calculated using bensen group addivity....␣
↪need to verify
#     onlyCyclics=True,
#     # how many radicals should be utilized in the calculation.
#     # If the amount of radicals is more than this, RMG will use hydrogen bond incrementation␣
↪method
#     maxRadicalNumber=0,
# )

# optional block allows thermo to be estimated through ML estimator
# mlEstimator(
#     thermo=True,
#     # Name of folder containing ML architecture and parameters in database
```

```
#     name='main',
#     # Limits on atom numbers
#     minHeavyAtoms=1,
#     maxHeavyAtoms=None,
#     minCarbonAtoms=0,
#     maxCarbonAtoms=None,
#     minOxygenAtoms=0,
#     maxOxygenAtoms=None,
#     minNitrogenAtoms=0,
#     maxNitrogenAtoms=None,
#     # Limits on cycles
#     onlyCyclics=False,
#     minCycleOverlap=0,  # specifies the minimum number of atoms that must be shared between␣
↪any two cycles
#     # If the estimated uncertainty of the thermo prediction is greater than
#     # any of these values, then don't use the ML estimate
#     H298UncertaintyCutoff=(3.0, 'kcal/mol'),
#     S298UncertaintyCutoff=(2.0, 'cal/(mol*K)'),
#     CpUncertaintyCutoff=(2.0, 'cal/(mol*K)')
# )
```

1.6 Running a Job

Running RMG job is easy and under different situations you might want add additional flag as the following examples.

Basic run:

```
python rmg.py input.py
```

Run with a restart file (restart file should be located in same folder as input.py):

```
python rmg.py input.py -r
```

Run with CPU profiling:

```
python rmg.py input.py -p
```

We recommend you make a job-specific directory for each RMG simulation. Some jobs can take quite a while to complete, so we also recommend using a job scheduler (if working in an linux environment).

The instructions below describe more special cases for running an RMG job.

1.6.1 Running RMG in parallel with SLURM

RMG has the capability to run using multiple cores. Here is an example job submission script for an RMG-Py job with a SLURM scheduler

The job named `min_par` reserves 24 CPUs on a single node (`-np 24`), but uses only 12 workers (= 12 CPUs) in parallel during the RMG-Py simulation.

Make sure that:

- the queue named `debug` exists on your SLURM scheduler.
- you modify the path to the parent folder of the RMG-Py installation folder

- you have an anaconda environment named `rmg_env` that contains RMG-Py's dependencies

- the working directory from which you launched the job contains the RMG-Py input file `input.py`

`-v` adds verbosity to the output log file.

```
#!/bin/bash
#SBATCH -p debug
#SBATCH -J min_par
#SBATCH -n 24

hosts=$(srun bash -c hostname)

WORKERS=12


RMG_WS=/path/to/RMG/parent/folder
export PYTHONPATH=$PYTHONPATH:$RMG_WS/RMG-Py/

source activate rmg_env
python -m scoop -n $WORKERS --host $hosts -v $RMG_WS/RMG-Py/rmg.py input.py
source deactivate
```

1.6.2 Running RMG in parallel with SGE

RMG has the capability to run using multiple cores. Here is an example using the SGE scheduler.

In order to help understand, the example job is also named `min_par` reserving 24 CPUs on a single node (`#$ -pe singlenode 24`), but uses only 12 workers (= 12 CPUs) in parallel during the RMG-Py simulation.

Make sure that:

- the queue named `normal` exists on your SGE scheduler

- you modify the path to the parent folder of the RMG-Py installation folder

- you have an anaconda environment named `rmg_env` that contains RMG-Py's dependencies

- the working directory from which you launched the job contains the RMG-Py input file `input.py`

`-v` adds verbosity to the output log file

```
#! /bin/bash

#$ -o job.log
#$ -l normal
#$ -N min_par
#$ -pe singlenode 24

WORKERS=12


RMG_WS=/path/to/RMG/parent/folder
export PYTHONPATH=$PYTHONPATH:$RMG_WS/RMG-Py/

source activate rmg_env
python -m scoop --tunnel -n $WORKERS -v $RMG_WS/RMG-Py/rmg.py input.py

source deactivate
```

1.7 Analyzing the Output Files

You will see that a sucessfully executed RMG job will create multiple output files and folders: `output.html` (if `generateOutputHTML=True` is specified) `/chemkin` `/pdep` `/plot` `/solver` `/species` `restart.pkl` `RMG.log`

1.7.1 The Chemkin Folder

The `/chemkin` folder will likely have a large number of chemkin formatted files. In general, these can be disregarded, as you will be mainly interested in `chem.inp`, the chemkin formatted input file with a species list, thermochemical database, and a list of elementary reactions. All of inp files appended with numbers are those that have been generated by RMG as it runs and the mechanism is still in progress of enlarging. `chem_annotated.inp` is provided as a means to help make sense of species syntax and information sources (i.e., how RMG estimated indivual kinetic and thermodynmic parameters). In addition, a species dictionary, `species_dictionary.txt`, is generated containing all the species in your mechanism in the format of an adjacency list. Either chemkin file, in addition to the dictionary, may be used as inputs in the tools section of this website to better visualize the species and reactions. Alternatively, if the input option `generateOutputHTML=` is set to True, you will be able to visualize 2D images of all species and reactions in your mechanism as image files and in an html file, `output.html`. Once you are able to visualize the mechanism, several useful tools exist. For example, in the *Reaction Details* section, you'll see the following with check-box fields beside them:

- Kinetics
- Comments
- Chemkin strings

If you check the last box, chemkin strings, you can then search for strings corresponding to seemingly nonsensical named species (e.g. S(1234)) that may show up in any analyses/simulations you perform (e.g., with Cantera or Chemkin). Further, under *Reaction Families*, you can selectively view the reactions that been generated based on a particular RMG reaction family or library.

1.7.2 The Species Folder

If `generateOutputHTML=True` is specified as an RMG input *option*, the species folder will be populated with png files with 2D pictures of each species in your final mechanism. Otherwise, it will contain no files, or files generated from pressure dependent jobs.

1.7.3 The Pdep Folder

The `/pdep` folder will contain files associated with the pressure-dependent reactions that RMG has generated, if you requested such a job. These files are formatted as input files for *Arkane*, which can be run independently. This can be useful if one wants to visualize the potential energy surface corresponding to any particular network.

1.7.4 The Solver Folder

RMG currently includes a solver for isothermal batch reactors. This is in fact a critical part of the model enlargement algorithm. If you have included simulations in your input file, the solutions will be located in `/solver`. You will probably only be interested in the files with the largest number tags.

1.8 Guidelines for Building a Model

RMG has been designed to build kinetic models for gas phase pyrolysis and combustion of organic molecules made of C, H, O and S. By kinetic model, we mean a set of reactions and associated kinetics that represent the chemical transformations occurring in the system of interest. These systems could be the combustion of fuels, pyrolysis of hydrocarbon feedstocks, etc. The total number of reactions and species typically required to describe some of these processes can run into the thousands making these models difficult and error-prone to build manually. This is the main motivation behind using software like RMG that build such models automatically in a systematic reproducible manner.

In RMG, the user is expected to provide an input file specifying the conditions (temperature, pressure, etc.) under which one desires to develop kinetic models. The following are some tips for setting up your input/condition file.

1.8.1 Start with a good seed mechanism

RMG is a useful tool in elucidating important pathways in a given process but may not capture certain special reaction types which may be specific to the system you are interested in. However, if you already have a good idea of these reactions that are important and are not available in the standard RMG library, you can create a 'seed mechanism' and include it in the input file to RMG. This will directly include these in the model core and add other reactions from the RMG library on top of it using our rate based algorithm. (Similarly, you can specify your own thermodynamic parameters for species using thermochemistry libraries which are similar in concept to seed mechanisms. In order to build these libraries, you will need to specify all species in the RMG adjacency list format.) In a combustion system, RMG tends to do a decent job filling in the termination and propagation steps of a mechanism if it is guided with the initiation and chain branching steps using a seed mechanism. Ideally, RMG should be able to find all the right chemistry through our kinetics database but holes in current kinetic databases can make this task difficult. A good seed mechanism can address this issue for the system of interest and also reduce the size, cost and time taken to arrive at a converged model.

1.8.2 Setting up the right termination criterion

Start with a relatively large tolerance (such as 0.1) when building your first model to make sure that RMG can converge the model to completion without any hiccups, then begin tightening the tolerance if you are able to converge the initial model. For large molecules such as tetradecane (C14), even a tolerance of 0.1 may be too tight for RMG to work with and lead to convergence problems. Note that a good seed mechanism allows for faster convergence.

1.8.3 Restricting the number of carbon atoms, oxygen atoms, and radical sites per species

Options to tune the maximum number of carbon or oxygen atoms, or number of radical sites per species can be specified at the beginning of the condition file. In most systems, we do not expect large contributions from species with more than 1 radical center (i.e. biradicals, etc.) to affect the overall chemistry, thus it may be useful to limit the maximum number of radicals to 2 (to allow for O2). The same applies for the maximum number of oxygens you want to allow per species. Restricting the number of carbon atoms in each species may also be worthwhile to prevent very large molecules from being generated if many such species appear in your model. Using any of these options requires some prior knowledge of the chemistry in your system. It is recommended that an initial model be generated without turning these options on. If many unlikely species show up in your model (or if your model has trouble converging and is generating many unlikely species on the edge), you can begin tuning these options to produce a better model.

1.8.4 Adding key species into the initial condition file

Sometimes, chain branching reactions like dissociation of ROOH species do not make it to the core directly because if their fluxes are very small and the tolerance is not tight enough. In these cases, seeding the condition file with these species (with zero concentration) is helpful. By adding these species to the initial set of species in the condition file, the reactions involving those species will be automatically added to the core. (Putting these reactions in the seed mechanism has the same effect.) Thus, if a species is known to be a part of your system and RMG is having trouble incorporating it within your model, it should be added to the condition file with 0.0 set as the concentration.

1.8.5 Starting with a single molecule when generating a model for a mixture

For modeling the combustion of fuel mixtures, you may want to start with determining their composition and starting with a kinetic study of the dominant compound. It is possible to model the combustion of fuel mixtures but they are more challenging as well as harder to converge in RMG because RMG will automatically generate all cross reactions between the reacting species and intermedites. Starting with single species is always a good idea and is also useful when thinking about fuel mixtures. In order to build a better background in chemical kinetic model development and validation, please look at a recent paper from our group on butanol combustion available here. This should give you some idea about how RMG can be put to use for the species of interest to you.

1.9 Standalone Modules

There are several standalone modules that can be run separately from RMG. These scripts can be found in the `RMG-Py/scripts` folder, unless mentioned otherwise.

1.9.1 HTML Chemkin Output

The script, `generateChemkinHTML.py`, will create a formatted HTML page displaying all of the species and reactions in a given Chemkin file. Thermo and kinetics parameters are also displayed, along with any comments if the Chemkin file was generated by RMG.

This script gives the same output as turning on `generateOutputHTML` in the *options* section of the RMG input file. However, having using that setting can increase the memory usage and computation time for large jobs, so this script provides an option for generating the HTML file after job completion.

To use this script, you need a Chemkin input file and an RMG species dictionary. The syntax is as follows:

```
python importChemkinLibrary.py [-h, -f] CHEMKIN DICTIONARY [OUTPUT]
```

Positional arguments:

```
CHEMKIN        The path to the Chemkin file
DICTIONARY     The path to the RMG dictionary file
OUTPUT         Location to save the output files, defaults to the current directory
```

Optional arguments:

```
-h, --help     Show help message and exit
-f, --foreign  Not an RMG generated Chemkin file
```

This method is also available to use with a web browser from the RMG website: Convert Chemkin File.

1.9.2 Model Comparison

The script `diffModels` compares two RMG generated models to determine their differences. To use this method you will need the chemkin and species dictionary outputs from RMG. These can be found in the `chemkin` folder from the directory of the `input.py` file used for the RMG run. The syntax is as follows:

```
python diffModels.py CHEMKIN1 SPECIESDICT1 --thermo1 THERMO1 CHEMKIN2 SPECIESDICT2 --thermo2
→THERMO2 --web
```

where `CHEMKIN` represents the chemkin input file (`chem00XX.inp`), `SPECIESDICT` is the species dictionary from RMG (`species_dictionary.txt`) and the optional `--thermo` flag can be used to add separate thermo CHEMKIN files `THERMO`. The numbers (`1` and `2`) represent which model to each file is from. The optional `--web` flag is used for running this script through the RMG-website.

Running the script without any optional flags looks like:

```
python diffModels.py CHEMKIN1 SPECIESDICT1 CHEMKIN2 SPECIESDICT2
```

Output of each comparison is printed, and the method then produces a html file (`diff.html`) for easy viewing of the comparison.

This method is also available to use with a web browser from the RMG website: Model Comparison Tool.

1.9.3 Merging Models

This script combines up to 5 RMG models together. The thermo and kinetics from common species and reactions is taken from the first model with the commonality. To better understand the difference in two models, use diffModels.py. To use this method type:

```
python mergeModels.py --model1 chemkin1 speciesdict1 --model2 chemkin2 speciesdict2
```

where `chemkin` specifies the chemkin input file from the RMG run and `speciesdict` represents the species dictionary from the RMG run. These can be found in the `chemkin` folder from the directory of the `input.py` file used for the RMG run. The numbers are for different models that you want to merge. To merge more than two files, you can add `--model3 chemkin3 speciesdict3`. Up to 5 models can be merged together this way

Running this method will create a new species dictionary (species_dictionary.txt) and chemkin input file (chem.inp) in the parent directory of the terminal.

This method is also available to use with a web browser from the RMG website: Model Merge Tool.

1.9.4 Generate Reactions

The script generateReactions.py generates reactions between all species mentioned in an input file. To call this method type:

```
python generateReactions.py Input_File
```

where `Input_File` is a file similar to a general RMG input file which contains all the species for RMG to generate reactions between. An example file is placed in `$RMGPy/examples/generateReactions/input.py`

```python
# Data sources for kinetics
database(
    thermoLibraries = ['BurkeH2O2','primaryThermoLibrary','DFT_QCI_thermo','CBS_QB3_1dHR'],
    reactionLibraries = [],
```

(continues on next page)

```python
    seedMechanisms = [],
    kineticsDepositories = 'default',
    #this section lists possible reaction families to find reactioons with
    kineticsFamilies = ['!Intra_Disproportionation','!Substitution_O'],
    kineticsEstimator = 'rate rules',
)

# List all species you want reactions between
species(
    label='ethane',
    reactive=True,
    structure=SMILES("CC"),
)

species(
    label='H',
    reactive=True,
    structure=SMILES("[H]"),
)

species(
    label='butane',
    reactive=True,
    structure=SMILES("CCCC"),
)


# you must list reactor conditions (though this may not effect the output)
simpleReactor(
    temperature=(650,'K'),
    pressure=(10.0,'bar'),
    initialMoleFractions={
        "ethane": 1,
    },
    terminationConversion={
        'butane': .99,
    },
    terminationTime=(40,'s'),
)

#optional module if you want to get pressure dependent kinetics.

#pressureDependence(
#     method='modified strong collision',
#     maximumGrainSize=(0.5,'kcal/mol'),
#     minimumNumberOfGrains=250,
#     temperatures=(300,2200,'K',2),
#     pressures=(0.01,100,'bar',3),
#     interpolation=('Chebyshev', 6, 4),
#     maximumAtoms=15,
#)

#optional module if you want to limit species produced in reactions.

#generatedSpeciesConstraints(
#     allowed=['input species','seed mechanisms','reaction libraries'],
#     maximumCarbonAtoms=4,
```

```
#    maximumOxygenAtoms=7,
#    maximumNitrogenAtoms=0,
#    maximumSiliconAtoms=0,
#    maximumSulfurAtoms=0,
#    maximumHeavyAtoms=20,
#    maximumRadicalElectrons=1,
#)
```

This method will produce an `output.html` file in the directory of `input.py` which contains the all the reactions produced between the species.

This method is also available to use with a web browser from the RMG website: Populate Reactions.

1.9.5 Simulation and Sensitivity Analysis

For sensitivity analysis, RMG-Py must be compiled with the DASPK solver, which is done by default but has some dependency restrictions. (See *License Restrictions on Dependencies* for more details.) Sensitivity analysis or a simulation (without sensitivity) can be conducted in a standalone system for an existing kinetics model in Chemkin format.

To run a simulation and/or sensitivity analysis, use the simulate module in RMG-Py/scripts:

```
python simulate.py input.py chem.inp species_dictionary.txt
```

where `chem.inp` is the CHEMKIN file and the `species_dictionary.txt` contains the dictionary of species associated with the CHEMKIN file. `input.py` is an input file similar to one used for an RMG job but does not generate a RMG job. See the following `input.py` example file found under the $RMGPy/examples/sensitivity/input.py folder

```
# Data sources
database(
    thermoLibraries = ['primaryThermoLibrary'],
    reactionLibraries = [],
    seedMechanisms = [],
    kineticsDepositories = ['training'],
    kineticsFamilies = ['!Intra_Disproportionation','!Substitution_O'],
    kineticsEstimator = 'rate rules',
)

# Constraints on generated species
generatedSpeciesConstraints(
    maximumRadicalElectrons = 2,
)

# List of species
species(
    label='ethane',
    reactive=True,
    structure=SMILES("CC"),
)

# Reaction systems
simpleReactor(
    temperature=(1350,'K'),
    pressure=(1.0,'bar'),
    initialMoleFractions={
```

```
        "ethane": 1.0,
    },
    terminationConversion={
        'ethane': 0.9,
    },
    terminationTime=(1e6,'s'),
    sensitivity=['ethane'],
    sensitivityThreshold=0.01,
)

simulator(
    atol=1e-16,
    rtol=1e-8,
    sens_atol=1e-6,
    sens_rtol=1e-4,
)

model(
    toleranceKeepInEdge=0.0,
    toleranceMoveToCore=0.1,
    toleranceInterruptSimulation=0.1,
    maximumEdgeSpecies=100000
)

options(
    units='si',
    saveRestartPeriod=None,
    saveSimulationProfiles=False,
    generateOutputHTML=False,
    generatePlots=False,
)
```

The names of species named in the input file must coincide with the names specified in the CHEMKIN file.

Other options that can be specified for the `simulate.py` scripts are:

```
--no-dlim      Turn off diffusion-limited rates for LiquidReactor
-f, --foreign  Not an RMG generated Chemkin file (will be checked for duplicates)
```

Sensitivity analysis is conducted for the list of species given for the `sensitivity` argument in the input file. The normalized concentration sensitivities with respect to the reaction rate coefficients dln(C_i)/dln(k_j) are saved to a csv file with the file name `sensitivity_1_SPC_1.csv` with the first index value indicating the reactor system and the second naming the index of the species the sensitivity analysis is conducted for. Sensitivities to thermo of individual species is also saved as semi normalized sensitivities dln(C_i)/d(G_j) where the units are given in 1/(kcal mol-1). The sensitivityThreshold is set to some value so that only sensitivities for dln(C_i)/dln(k_j) > sensitivityThreshold or dlnC_i/d(G_j) > sensitivityThreshold are saved to this file.

1.9.6 Generating Flux Diagrams

The script, `generateFluxDiagrams.py`, will create a movie out of a completed RMG model that shows interconnected arrows between species that represent fluxes.

To use this method, you just need a Chemkin input file and an RMG species dictionary. The syntax is as follows:

```
python generateFluxDiagram.py [-h] [--java] [--no-dlim] [-s SPECIES] [-f]
                              [-n N] [-e N] [-c TOL] [-r TOL] [-t S]
                              INPUT CHEMKIN DICTIONARY [CHEMKIN_OUTPUT]
```

Positional arguments:

```
INPUT               RMG input file
CHEMKIN             Chemkin file
DICTIONARY          RMG dictionary file
CHEMKIN_OUTPUT      Chemkin output file
```

Optional arguments:

```
-h, --help            show this help message and exit
--java                process RMG-Java model
--no-dlim             Turn off diffusion-limited rates
-s DIR, --species DIR Path to folder containing species images
-f, --foreign         Not an RMG generated Chemkin file (will be checked for duplicates)
-n N, --maxnode N     Maximum number of nodes to show in diagram
-e N, --maxedge N     Maximum number of edges to show in diagram
-c TOL, --conctol TOL Lowest fractional concentration to show
-r TOL, --ratetol TOL Lowest fractional species rate to show
-t S, --tstep S       Multiplicative factor to use between consecutive time points
```

This method is also available to use with a web browser from the RMG website: Generate Flux Diagram.

1.9.7 Thermo Estimation Module

The thermo estimation module can be run stand-alone. An example input file for this module is shown below:

```
database(
    thermoLibraries = ['primaryThermoLibrary', 'GRI-Mech3.0']
)

species(
    label='Cineole',
    structure=SMILES('CC12CCC(CC1)C(C)(C)O2'),
)

quantumMechanics(
    software='mopac',#mopac or gaussian
    method='pm3',#pm3, pm6, pm7
    fileStore='QMfiles', # defaults to inside the output folder.
    onlyCyclics = True,#True, False
    maxRadicalNumber = 0, # 0, 1
)
```

The `database` block is used to specify species thermochemistry libraries. Multiple libraries may be created, if so desired. The order in which the thermo libraries are specified is important: If a species appears in multiple thermo libraries, the first instance will be used.

Please see Section *Thermo Database* for details on editing the thermo library. In general, it is best to leave the ThermoLibrary set to its default value. In particular, the thermodynamic properties for H and H2 must be specified in one of the primary thermo libraries as they cannot be estimated by Benson's method.

For example, if you wish to use the GRI-Mech 3.0 mechanism [GRIMech3.0] as a ThermoLibrary in your model, the syntax will be:

```
thermoLibraries = ['primaryThermoLibrary','GRI-Mech3.0']
```

This library is located in the `RMG-database/input/thermo/libraries` directory. All "Locations" for the ThermoLibrary field must be with respect to the `RMG-database/input/thermo/libraries` directory.

The optional `quantumMechanics` block is used when quantum mechanical calculations are desired to determine thermodynamic parameters. These calculations are only run if the molecule is not included in a specified thermo library. The `software` option accepts either the `mopac` or `gaussian` string. The `method` option refers to the level-of-theory, which can either be `pm3`,``pm6``, or `pm7`. A folder can be specified to store the files used in these calculations, however if not specified this defaults to a *QMfiles* folder in the output folder. The `onlyCyclics` option, if `True`, only runs these calculations for cyclic species. In this case, group contribution estimates are used for all other species. The calculations are also only run on species with a maximum radical number set by the user. If a molecule has a higher radical number, the molecule is saturated with hydrogen atoms, then quantum mechanical calculations with subsequent hydrogen bond incrementation is used to determine the thermodynamic parameters.

Submitting a job is easy:

```
python thermoEstimator.py input.py
```

We recommend you make a job-specific directory for each thermoEstimator simulation.

You can also specify that an RMG-style thermo library be saved upon completion:

```
python thermoEstimator.py -l input.py
```

Note that the RMG website also provides thermo estimation through the Molecule Search.

1.9.8 Convert FAME to Arkane Input File

This module is utilized to convert FAME file types (used in RMG-Java) to Arkane (formerly called CanTherm) objects (used in RMG-Py) for pressure dependent calculations.

FAME is an early version of the pdep code in Arkane, and it is written in Fortran and used by RMG-Java. This script enables importing FAME input files into Arkane. Note that it is mostly designed to load the FAME input files generated automatically by RMG-Java, and may not load hand-crafted FAME input files. If you specify a *moleculeDict*, then this script will use it to associate the species with their structures.

```
python convertFAME.py fame_object
```

where `fame_object` is the FAME file used to be converted into the Arkane object.

Some additional options involve adding an RMG dictionary to process with the file. The syntax for this is

```
python convertFAME.py -d RMG_dictionary.txt fame_object
```

where `RMG_dictionary.txt` is the dictionary to process with the file.

A max energy cuttoff is also possible when converting the file formats.

```
python convertFAME.py -d RMG_dictionary.txt -x value units value units fame_object
```

where `value` represents the max energy amount and `units` represents its units

1.9.9 Database Scripts

This section details usage for scripts available in `RMG-database/scripts` folder.

evansPolanyi.py

This script will generate an Evans-Polanyi plot for a single kinetics depository.

Usage:

```
python evansPolanyi.py [-h] DEPOSITORY
```

Positional arguments:

```
DEPOSITORY    the depository to use
```

Optional arguments:

```
-h, --help    show help message and exit
```

exportKineticsLibraryToChemkin.py

This script exports an individual RMG-Py kinetics library to a chemkin and dictionary file. Thermo is taken from RMG's estimates and libraries. In order to use more specific thermo, you must tweak the thermoLibraries and estimators in use when loading the database. The script will save the chem.inp and species_dictionary.txt files in the local directory.

Usage:

```
python exportKineticsLibrarytoChemkin.py [-h] LIBRARYNAME
```

Positional arguments:

```
LIBRARYNAME    the libraryname of the RMG-Py format kinetics library
```

Optional arguments:

```
-h, --help    show help message and exit
```

exportOldDatabase.py

This script exports the database to the old RMG-Java format. The script requires two command-line arguments: the path to the database to import, and the path to save the old RMG-Java database to.

Usage:

```
python exportOldDatabase.py OUTPUT
```

Positional arguments:

```
OUTPUT    path to the directory where the RMG-Java database should be saved
```

importChemkinLibrary.py

This script imports a chemkin file (along with RMG dictionary) from a local directory and saves a set of RMG-Py kinetics library and thermo library files. These py files are automatically added to the input/kinetics/libraries and input/thermo/libraries folder under the user-specified *name* for the chemkin library.

Usage:

```
python importChemkinLibrary.py [-h] CHEMKIN DICTIONARY NAME
```

Positional arguments:

```
CHEMKIN      The path of the chemkin file
DICTIONARY   The path of the RMG dictionary file
NAME         Name of the chemkin library to be saved
```

Optional arguments:

```
-h, --help  show help message and exit
```

importJavaKineticsLibrary.py

This script imports an individual RMG-Java kinetics library from a local directory and saves the output kinetics library py file into a path of the user's choosing. This library will be automatically added to the 'libraryname' folder in the input/kinetics/libraries directory and can be used directly as an RMG-Py kinetics library.

Usage:

```
python importJavaKineticsLibrary.py [-h] INPUT LIBRARYNAME
```

Positional arguments:

```
INPUT         the input path of the RMG-Java kinetics library directory
LIBRARYNAME   the libraryname for the RMG-Py format kinetics library
```

Optional arguments:

```
-h, --help     show help message and exit
```

importJavaThermoLibrary.py

This script imports an individual RMG-Java themo library from a local directory and saves the output thermo library py file into a path of the user's choosing. This library will be automatically saved to libraryname.py in the input/thermo/libraries directory and can be used directly as an RMG-Py thermo library.

Usage:

```
python importJavaThermoLibrary.py [-h] INPUT LIBRARYNAME
```

Positional arguments:

```
INPUT         the input path of the RMG-Java thermo library directory
LIBRARYNAME   the libraryname for the RMG-Py format thermo library
```

Optional arguments:

```
-h, --help     show help message and exit
```

importOldDatabase.py

This script imports an RMG-Java database from the output directory and saves it in the input directory. Only recommended for use in extreme circumstances.

Usage:

```
python importOldDatabase.py [-h]  INPUT OUTPUT
```

Positional arguments:

```
INPUT          the input path of the RMG-Java database directory
LIBRARYNAME    output path for the desired RMG-Py database directory
```

Optional arguments:

```
-h, --help     show help message and exit
```

1.9.10 Standardize Model Species Names

This script enables the automatic renaming of species names of of two or more Chemkin files (and associated species dictionaries) so that they use consistent, matching names. Simply pass the paths of the Chemkin files and species dictionaries on the command-line, e.g.:

```
python standardizeModelSpeciesNames.py --model1 /path/to/chem1.inp /path/to/species_dictionary1.
↪txt --model2 /path/to/chem2.inp /path/to/species_dictionary2.txt
```

The resulting files are saved as `chem1.inp` and `species_dictionary1.txt`, `chem2.inp`, `species_dictionary2.txt` and so forth (depending on how many models you want to standardize) and will be saved in the execution directory.

1.9.11 Reaction Reduction in an RMG Job

This script is located at `RMG-Py/rmgpy/reduction/main.py` instead of the usual `RMG-Py/scripts` folder.

RMG's method of generating reactions between all species in a core mechanism and including them in the resulting model is a robust process to obtain all chemistry. However, the huge number of cross reactions lead to a non-sparse matrix, which can increase computational time when using the resulting models in other simulations.

To help reduce the complexity of RMG produced mechanisms, a mechanism reduction script was written that eliminates unimportant reactions up to a set threshold. Though this method will reduce number of reactions and guarantee target species concentrations at the given conditions are minimally affected, no guarantee is given that it will result in optimally reduced mechanism.

To reduce an RMG job, you will need an additional file `reduction_input.py`. This file contains two terms that tell the reduction algorithm what to do. The example file located in `rmgpy/reduction/test_data/minimal/chemkin` is written as followed.

```
targets = ['ethane', 'C']
tolerance = .05
```

`targets` is a list of species labels whose concentration change should be minimized, and `tolerance` is the percent change the user can tolerate at the end of simulation. In the above example, this would be 5%. Higher values of `tolerance` lead to fewer final reactions with more error in output rates.

To run a simulation, type

```
python $RMG/rmgpy/reduction/main.py input.py reduction_input.py chem_annotated.inp species_
↪dictionary.txt
```

A command line interface to the reduction driver script is contained in `rmgpy/reduction/main.py`. It accepts four files:

- `input.py`: RMG-Py input file containing the settings to evaluate state variables.

- `reduction_input.py`: Reduction input file containing the target variables and associated error tolerances to allow in the reduced model

- `chem_annotated.inp`: the reaction mechanism to reduce.

- `species_dictionary.txt`: the species dictionary associated with the reaction mechanism to reduce.

The algorithm will reduce the number of reactions until the tolerance is no longer met. If everything goes as planned, a `chem_reduced.inp` is generated containing the reduced mechanism. In addition, a number of files `chem_reduced_{i}.inp` are created and correspond to the intermediate reduced mechanisms. They can be used in place of the final reduced model, in case the reduction algorithm does not terminate normally.

You can go to `$RMG/examples/reduction` to try this module.

Background

The reduction algorithm computes the ratio of species reaction rate (r_{ij}) to the total rate of formation/consumption (R_i) of all species i, and compares this ratio to a tolerance (ϵ), with values of epsilon between 0 and 1. If the ratio of a reaction is greater than epsilon it is deemed *important* for the species in question. When a reaction is not important for a single species, at any given time between t=0 and the user-defined end time, then it is deemed unimportant for the given system. As a result, the reaction is removed from the mechanism.

The value of epsilon is determined by an optimization algorithm that attempts to reduce the model as much as possible given the constraints of the user-defined target variables. A logarithmic bisection optimization algorithm is used to provide guesses for the value of epsilon based on the two previous guesses that undershoot and overshoot the user-defined relative deviation of the target variables

A value of 5% for the relative deviation of the target variable implies that the mole fraction of the target variable at the end time of the batch reactor simulation as computed by the reduced mechanism may deviate up to 5% w.r.t. to the mole fraction of the target variable at the end time of the batch reactor simulation as computed by the full mechanism.

1.9.12 Isotopes

Describing isotopes in adjacency lists

Isotopic enrichment can be indicated in a molecular structure's adjacency list. The example below is methane with an isotopically labeled carbon of isotope number 13, which is indicated with `i13`:

```
1 C u0 p0 c0 i13 {2,S} {3,S} {4,S} {5,S}
2 H u0 p0 c0 {1,S}
3 H u0 p0 c0 {1,S}
4 H u0 p0 c0 {1,S}
5 H u0 p0 c0 {1,S}
```

Running the RMG isotopes algorithm

The isotopes script is located in the folder scripts. To run the algorithm, ensure the RMG packages are loaded and type:

```
python /path/to/rmg/scripts/isotopes.py /path/to/input/file.py
```

The input file is identical to a standard RMG input file and should contain the conditions you want to run (unless you are inputting an already completed RMG model). Without any options, the script will run the original RMG input file to generate a model. Once the RMG job is finished, it will create new species for all isotopologues of previously generated species and then generate all reactions between the isotopologues.

Some arguments can be used to alter the behavior of the script. If you already have a model (which includes atom mapping in RMG's format) which you would like to add isotope labels to, you can use the option `--original path/to/model/directory` with the desired model files stored within with structure `chemkin/chem_annotated.inp` and `chemkin/species_dictionary.txt`. With this option, the isotope script will use the specified model instead of re-running an RMG job.

If you only desire the reactions contained in the specific RMG job, you can add `--useOriginalReactions` in addition to `--original`. This will create a full set of isotopically labeled versions of the reactions you input and avoid a time-consuming generate reactions proceedure.

The arguement `--maximumIsotopicAtoms [integer]` limits the number of enriched atoms in any isotopologue in the model. This is beneficial for decreasing model size, runtime of model creation and runtime necessary for analysis.

Adding kinetic isotope effects which are described in this paper can be obtained through the argument `--kineticIsotopeEffect simple`. Currently this is the only supported method, though this can be extended to other effects.

If you have a desired output folder, `--output output_folder_name` can direct all output files to the specified folder. The default is to create a folder named 'iso'.

There are some limitations in what can be used in isotope models. In general, RMG Reaction libraries and other methods of kinetic estimation that do not involve atom mapping to reaction recipes are not compatible (though they can be functional if all isotopologues are included in the reaction library). The algorithm also does not function with pressure dependent mechanisms generated by RMG, and has only been tested for gas phase kinetics. This algorithm currently only works for Carbon-13 enrichments.

Following the generation, a number of diagnostics check model accuracy. Isotopologues are checked to ensure their symmetries are consistent. Then, the reaction path degeneracy among reactions differing only in isotope labeling is checked to ensure it is consistent with the symmetry values of reactions. If one of these checks throws a warning, the model will likely exhibit non-natural fluctuations in enrichment ten to one hundred times larger than from non-hydrogen kinetic isotope effects.

Output from script

The isotope generation script will output two files inside the nested folders `iso/chemkin`, unless `--output` is specified. The file `species_dictionary.txt` lists the structure of all isotopologue using the RMG adjacency list structure. The other file of importance `chem_annotated.inp` is a chemkin input file containing elements, species, thermo, and reactions of the entire system.

1.10 Species Representation

Species objects in RMG contain a variety of attributes, including user given names, thermochemistry, as well as structural isomers. See the rmgpy.species.Species class documentation for more information.

RMG considers each species to be unique, and comprised of a set of molecular structural isomers, including resonance isomers. RMG uses the list of resonance isomers to compare whether two species are the same. Each molecular structure is stored in RMG using graph representations, using vertices and edges, where the vertices are the atoms and the edges are the bonds. This form of representation is known as an adjacency list. For more information on adjacency lists, see the rmgpy.molecule.adjlist page.

Species objects in the input file can also be constructed using other common representations such as SMILES, SMARTS, and InChIs. The following can all be used to represent the methane species:

```
species(
    label='CH4',
    reactive=True,
    structure=SMILES("C"),
)
```

Replacing the structure with any of the following representations will also produce the same species:

```
structure=adjacencyList("
1 C u0 p0 c0 {2,S} {3,S} {4,S} {5,S}
2 H u0 p0 c0 {1,S}
3 H u0 p0 c0 {1,S}
4 H u0 p0 c0 {1,S}
5 H u0 p0 c0 {1,S}
"),

structure=SMARTS("[CH4]"),

structure=SMILES("C"),

structure=InChI("InChI=1S/CH4/h1H4"),
```

To quickly generate any adjacency list, or to generate an adjacency list from other types of molecular representations such as SMILES, InChI, or even common species names, use the Molecule Search tool found here: http://rmg.mit.edu/molecule_search

1.10.1 Representing Oxygen

Special care should be taken when constructing a mechanism that involves molecular oxygen. The ground electronic state of molecular oxygen, $^3\Sigma_g^-$, does *not* contain a double bond, but instead a single bond and two lone electrons. In RMG's adjaceny list notation the ground state of oxygen is represented as

```
1 O u1 p2 {2,S}
2 O u1 p2 {1,S}
```

You should use the above adjacency list to represent molecular oxygen in your condition files, seed mechanisms, etc. The triplet form is 22 kcal/mol more stable than the first singlet excited state, $^1\Delta_g$, which does contain a double bond. The adjacency list for singlet oxygen is

```
1 O u0 p2 {2,D}
2 O u0 p2 {1,D}
```

Selecting the correct structure for oxygen is important, as the reactions generated from a double bond are significantly different than those generated from a radical or diradical. For example, the reaction

$$CH_4 + O_2 \rightarrow CH_3 + HO_2$$
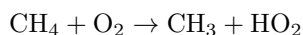
would occur for both triplet and singlet oxygen, but in entirely different families. For triplet oxygen the above represents a hydrogen abstraction, while for singlet oxygen it represents the reverse of a disproportionation reaction.

The RMG databases have been modified to make all of the oxygen-related chemistry that was present in RMG databases consistent with the single-bonded biradical representation.

Conversion between triplet and singlet forms is possible through the primary reaction library `OxygenSingTrip`; the reactions involved are very slow, however, and are likely to be absent from any mechanisms generated. At this point, no other reactions of singlet oxygen have been included in RMG.

In order to allow the singlet form of O2 to be used in RMG, please allow it explicitly by setting `allowSingletO2` to `True` in the `generateSpeciesConstraints` section of the RMG input file.

```
generatedSpeciesConstraints(
    allowSingletO2 = True,
)
```

1.11 Group Representation

Group representations are used to represent molecular substructures within RMG. These are commonly used for identifying functional groups for use in both the thermo and kinetic databases.

For syntax of how to define groups, see rmgpy.molecule.adjlist.

1.12 Databases

RMG has databases storing thermochemistry and kinetics data. These databases can be visualized on the RMG website here: http://rmg.mit.edu/database/

1.12.1 Introduction

This section describes some of the general characteristics of RMG's databases.

Group Definitions

The main section in many of RMG's databases are the 'group' definitions. Groups are adjacency lists that describe structures around the reacting atoms. Between the adjacency list's index number and atom type, a starred number is inserted if the atom is a reacting atom.

Because groups typically do not describe entire molecules, atoms may appear to be lacking full valency. When this occurs, the omitted bonds are allowed to be anything. An example of a primary carbon group from H-Abstraction is shown below. The adjacency list defined on the left matches any of the three drawn structures on the right (the numbers correspond to the index from the adjacency list).

```
label = "C/H3/C",
group =
"""
1 *1 Cs  0 {2,S} {3,S} {4,S} {5,S}
2 *2 H  0 {1,S}
3   H  0 {1,S}
4   H  0 {1,S}
5   {Cs, Cd, Ct} 0 {1,S}
"""
```
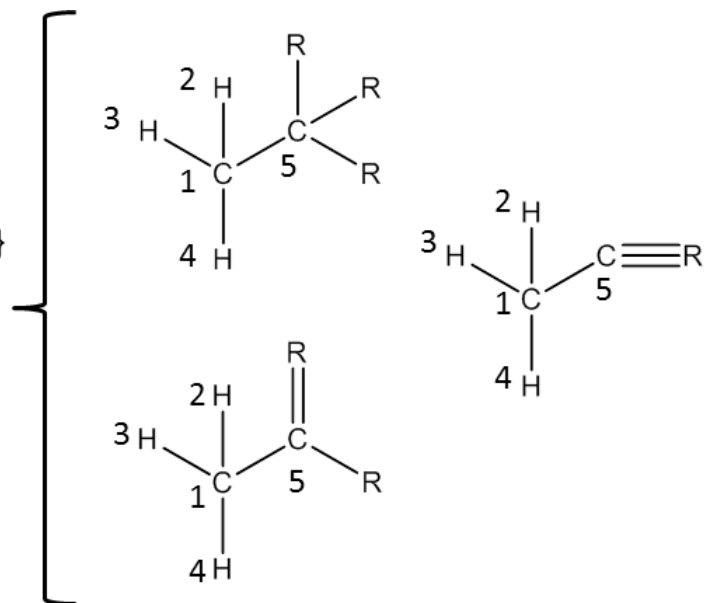
Atom types describe atoms in group definitions. The table below shows all atoms types in RMG.

Atom Type	Chemical Element	Localized electronic structure
R	Any	No requirements
R!H	Any except hydrogen	No requirements
H	Hydrogen	No requirements
C	Carbon	No requirements
Ca	Carbon	Atomic carbon with two lone pairs and no bonds
Cs	Carbon	Up to four single bonds
Csc	Carbon	Up to three single bonds, charged +1
Cd	Carbon	One double bond (to any atom other than O or S), up to two single bonds
Cdc	Carbon	One double bond, up to one single bond, charged +1
CO	Carbon	One double bond to an oxygen atom, up to two single bonds
CS	Carbon	One double bond to an sulfur atom, up to two single bonds
Cdd	Carbon	Two double bonds
Ct	Carbon	One triple bond, up to one single bond
Cb	Carbon	Two benzene bonds, up tp one single bond
Cbf	Carbon	Three benzene bonds (fused aromatics)
C2s	Carbon	One lone pair, up to two single bonds
C2sc	Carbon	One lone pair, up to three single bonds, charged -1
C2d	Carbon	One lone pair, one double bond
C2dc	Carbon	One lone pair, one double bond, up to one single bond, charge -1
C2tc	Carbon	One lone pair, one triple bond, charged -1
N	Nitrogen	No requirements
N0sc	Nitrogen	Three lone pairs, up to one single bond, charged -2
N1s	Nitrogen	Two lone pairs, up to one single bond
N1sc	Nitrogen	Two lone pairs, up to two single bonds, charged -1
N1dc	Nitrogen	Two lone pairs, one double bond, charged -1
N3s	Nitrogen	One lone pair, up to three single bonds
N3d	Nitrogen	One lone pair, one double bond, up to one single bond
N3t	Nitrogen	One lone pair, one triple bond
N3b	Nitrogen	One lone pair, two aromatic bonds

Continued on nex

Table 1 – continued from previous page

Atom Type	Chemical Element	Localized electronic structure
N5sc	Nitrogen	No lone pairs, up to four single bonds, charged +1
N5dc	Nitrogen	No lone pairs, one double bond, up to two single bonds, charged +1
N5ddc	Nitrogen	No lone pairs, two double bonds, charged +1
N5dddc	Nitrogen	No lone pairs, three double bonds, charged -1
N5tc	Nitrogen	No lone pairs, one triple bond, up to one single bond, charged +1
N5b	Nitrogen	No lone pairs, two aromatic bonds, up to one single bond
O	Oxygen	No requirements
Oa	Oxygen	Atomic oxygen with three lone pairs and no bonds
O0sc	Oxygen	Three lone pairs, up to one single bond, charged -1
O0dc	Oxygen	Three lone pairs, one double bond, charged -2
O2s	Oxygen	Two lone pairs, up to two single bonds
O2sc	Oxygen	Two lone pairs, up to one single bond, charged +1
O2d	Oxygen	Two lone pairs, one double bond
O4sc	Oxygen	One lone pair, up to three single bonds, charged +1
O4dc	Oxygen	One lone pair, one double bond, up to one single bond, charged +1
O4tc	Oxygen	One lone pair, one triple bond, charged +1
Si	Silicon	No requirements
Sis	Silicon	Up to four single bonds
Sid	Silicon	One double bond (not to O), up to two single bonds
SiO	Silicon	One double bond to an oxygen atom, up to two single bonds
Sidd	Silicon	Two double bonds
Sit	Silicon	One triple bond, up to one single bond
Sib	Silicon	Two benzene bonds, up tp one single bond
Sibf	Silicon	Three benzene bonds (fused aromatics)
S	Sulfur	No requirements
Sa	Sulfur	Atomic sulfur with three lone pairs and no bonds
S0sc	Sulfur	Three lone pairs, up to once single bond, charged -1
S2s	Sulfur	Two lone pairs, up to two single bonds
S2sc	Sulfur	Two lone pairs, up to three single bonds, charged -1/+1
S2d	Sulfur	Two lone pairs, one double bond
S2dc	Sulfur	Two lone pairs, one to two double bonds, up to one single bond, charged -1
S2tc	Sulfur	Two lone pairs, one triple bond, charged -1
S4s	Sulfur	One lone pair, up to four single bonds
S4sc	Sulfur	One lone pair, up to five single bonds, charged -1/+1
S4d	Sulfur	One lone pair, one double bond, up to two single bonds
S4dd	Sulfur	One lone pair, two double bonds
S4dc	Sulfur	One lone pair, one to three double bonds, up to three single bonds, charged -1/+1
S4b	Sulfur	One lone pair, two aromatic bonds
S4t	Sulfur	One lone pair, one triple bond, up to one single bond
S4tdc	Sulfur	One lone pair, one to two triple bonds, up to two double bonds, up to two single bonds, charged
S6s	Sulfur	No lone pairs, up to six single bonds
S6sc	Sulfur	No lone pairs, up to seven single bonds, charged -1/+1/+2
S6d	Sulfur	No lone pairs, one double bond, up to four single bonds
S6dd	Sulfur	No lone pairs, two double bonds, up to two single bonds
S6ddd	Sulfur	No lone pairs, up to three double bonds
S6dc	Sulfur	No lone pairs, one to three double bonds, up to five single bonds, charged -1/-1/+2
S6t	Sulfur	No lone pairs, one triple bond, up to three single bonds
S6td	Sulfur	No lone pairs, one triple bond, one double bond, up to one single bond
S6tt	Sulfur	No lone pairs, two triple bonds

Table 1 – continued from previous page

Atom Type	Chemical Element	Localized electronic structure
S6tdc	Sulfur	No lone pairs, one to two triple bonds, up to two double bonds, up to four single bonds, charge
Cl	Chlorine	No requirements
Cl1s	Chlorine	Three lone pairs, zero to one single bonds
I	Iodine	No requirements
I1s	Iodine	Three lone pairs, zero to one single bonds
He	Helium	No requirements, nonreactive
Ne	Neon	No requirements, nonreactive
Ar	Argon	No requirements, nonreactive

Additionally, groups can also be defined as unions of other groups. For example,:

```
label="X_H_or_Xrad_H",
group=OR{X_H, Xrad_H},
```

Forbidden Groups

Forbidden groups can be defined to ban structures globally in RMG or to ban pathways in a specific kinetic family.

Globally forbidden structures will ban all reactions containing either reactants or products that are forbidden. These groups are stored in in the file located at `RMG-database/input/forbiddenStructures.py`.

To ban certain specific pathways in the kinetics families, a *forbidden* group must be created, like the following group in the `intra_H_migration` family:

```
forbidden(
    label = "bridged56_1254",
    group =
"""
1 *1 C 1 {2,S} {6,S}
2 *4 C 0 {1,S} {3,S} {7,S}
3    C 0 {2,S} {4,S}
4 *2 C 0 {3,S} {5,S} {8,S}
5 *5 C 0 {4,S} {6,S} {7,S}
6    C 0 {1,S} {5,S}
7    C 0 {2,S} {5,S}
8 *3 H 0 {4,S}
""",
    shortDesc = u"""""",
    longDesc =
u"""

""",
)
```

Forbidden groups should be placed inside the groups.py file located inside the specific kinetics family's folder `RMG-database/input/kinetics/family_name/` alongside normal group entries. The starred atoms in the forbidden group ban the specified reaction recipe from occurring in matched products and reactants.

In addition for forbidding groups, there is the option of forbidding specific molecules or species. Forbidding a molecule will prevent that exact structure from being generated, while forbidding a species will prevent any of its resonance structures from being generated. To specify a forbidden molecule or species, simply replace the `group` keyword with `molecule` or `species`:

```
# This forbids a molecule
forbidden(
    label = "C_quintet",
    molecule =
"""
multiplicity 5
1 C u4 p0
""",
    shortDesc = u"""""",
    longDesc =
u"""

""",
)

# This forbids a species
forbidden(
    label = "C_quintet",
    species =
"""
multiplicity 5
1 C u4 p0
""",
    shortDesc = u"""""",
    longDesc =
u"""

""",
)
```

Hierarchical Trees

Groups are ordered into the nodes of a hierarchical trees which is written at the end of groups.py. The root node of each tree is the most general group with the reacting atoms required for the family. Descending from the root node are more specific groups. Each child node is a subset of the parent node above it.

A simplified example of the trees for H-abstraction is shown below. The indented text shows the syntax in groups.py and a schematic is given underneath.
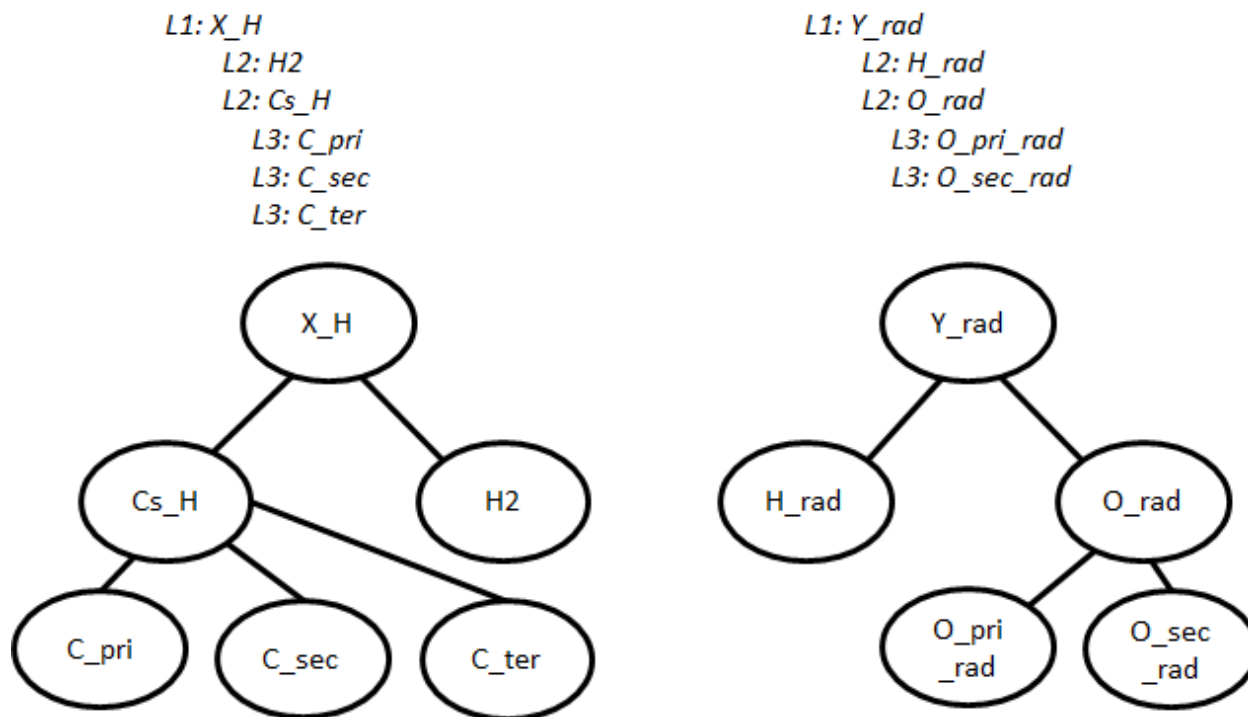
Individual groups only describe part of the reaction. To describe an entire reaction we need one group from each tree, which we call **node templates** or simply templates. (C_pri, O_pri_rad), (H2, O_sec_rad), and (X_H, Y_rad) are all valid examples of templates. Templates can be filled in with kinetic parameters from the training set or rules.

1.12.2 Thermo Database

This section describes the general usage of RMG's thermochemistry databases. Thermochemical data in RMG is reported using three different quantities:

1. Standard heat capacity data $C_p^o(T)$ as a function of temperature T

2. Standard enthalpy of formation at 298K $\Delta_f H^o(298K)$

3. Standard entropy at 298K $S^o(298K)$

A heat capacity model based on the Wilhoit equation is used for inter- and extrapolation of the heat capacity data as a function of temperature.

Libraries

Library types

Two types of thermo libraries are available in RMG: "gas phase" and "liquid thermo" libraries respectively identified thanks to the absence or presence of the keyword *solvent = "solvent_name"* in the header of a thermo library. Here is an example of a liquid thermo library header:

```
name = "example_liquid_thermo_library"
solvent = "octane"
shortDesc = u"test"
longDesc = u"""
```

In this example the library name is "example_liquid_thermo_library" and thermo data provided was obtained in *octane* solvent. The only difference between gas phase and liquid phase thermo libraries is made through this keyword, the rest of the library is similar to gas phase.

Note: You can only provide one solvent per library and users should pay attention to not mix thermo of species obtained in different solvent in a same library. RMG will raise an error if users try to load a liquid thermo library obtained in another solvent that the one provided in input file. (in the example provided here, this liquid thermo library can only be used in liquid phase simulation with octane as solvent. RMG will also raise an error if user try to use liquid phase thermo library in gas phase simulations.

Species thermochemistry libraries

The folder `RMG-database/input/thermo/libraries/` in RMG-database is the location to store species thermo-chemistry libraries. Each particularly library is stored in a file with the extension .py, e.g. 'DFT_QCI_thermo.py'.

An example of a species thermochemistry entry is shown here below:

```
entry(
    index = 1,
    label = "H2",
    molecule =
"""
1 H 0 0 {2,S}
2 H 0 0 {1,S}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([6.948,6.948,6.949,6.954,6.995,7.095,7.493],'cal/(mol*K)'),
        H298 = (0,'kcal/mol'),
        S298 = (31.095,'cal/(mol*K)'),
    ),
    shortDesc = u"""""",
    longDesc =
u"""

""",
)
```

The text above describes the first entry in the library (index = 1), labeled 'H2', through the adjacency list representation. Heat capacity data ('Cpdata') is described at 7 different temperatures, along with the standard enthalpy of formation at 298K ('H298'), and the standard entropy at 298K ('S298').

According to the *thermo classes* availble in RMG, you can provide different thermo data: NASA, thermodata (as shown above), wilhoit or NASAPolynomial.

Groups

The folder `RMG-database/input/thermo/groups/` in RMG-database is the location to store group contribution databases. Each particularly type of group contribution is stored in a file with the extension .py, e.g. 'groups.py':

file	Type of group contribution
gauche.py	1,4-gauche non-nearest neighbor interactions (NNIs)
group.py	group additive values (GAVs)
int15.py	1,5-repulsion non-nearest neighbor interactions (NNIs)
other.py	other non-nearest neighbor interactions (NNIs)
polycyclic.py	polycyclic ring corrections (RSCs)
radical.py	hydrogen bond increments (HBIs)
ring.py	monocyclic ring corrections (RSCs)

Like many other entities in RMG, the database of each type of group contribution is organized in a hierarchical tree, and is defined at the bottom of the database file. E.g.:

```
tree(
"""
L1: R
    L2: C
        L3: Cbf
            L4: Cbf-CbCbCbf
            L4: Cbf-CbCbfCbf
            L4: Cbf-CbfCbfCbf
        L3: Cb
            L4: Cb-H
            L4: Cb-Os
            L4: Cb-S2s
            L4: Cb-C
                L5: Cb-Cs
                L5: Cb-Cds
                    L6: Cb-(Cds-Od)
                    ...
```

More information on hierarchical tree structures in RMG can be found here: *Introduction*.

Group additive values (GAV)

An example of a GAV entry in group.py is shown here below:

```
entry(
    index = 3,
    label = "Cbf-CbCbCbf",
    group =
"""
1 * Cbf 0 {2,B} {3,B} {4,B}
2   Cb  0 {1,B}
3   Cb  0 {1,B}
4   Cbf 0 {1,B}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([3.01,3.68,4.2,4.61,5.2,5.7,6.2],'cal/(mol*K)',
                '+|-', [0.1,0.1,0.1,0.1,0.1,0.1,0.1]),
        H298 = (4.8,'kcal/mol','+|-',0.17),
        S298 = (-5,'cal/(mol*K)','+|-',0.1),
    ),
    shortDesc = u"""Cbf-CbfCbCb STEIN and FAHR; J. PHYS. CHEM. 1985, 89, 17, 3714""",
```

(continues on next page)

```
    longDesc =
u"""
Taken from STEIN and FAHR; J. PHYS. CHEM. 1985, 89, 17, 3714
""",
)
```

The text above describes a GAV "Cbf-CbCbCbf", with the central atom denoted by the asterisk in the adjacency list representation. Uncertainty margins are added in the data, after the unit specification. A short description 'shortDesc' specifies the origin of the data.

Ring Strain Corrections (RSC)

RMG distinguishes between monocyclic and polycyclic ring correction databases.

Monocyclic RSCs are used for molecules that contain one single ring. An example of a monocyclic RSC entry in ring.py is shown here below:

```
entry(
    index = 1,
    label = "Cyclopropane",
    group =
"""
1 * Cs 0 {2,S} {3,S}
2   Cs 0 {1,S} {3,S}
3   Cs 0 {1,S} {2,S}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([-3.227,-2.849,-2.536,-2.35,-2.191,-2.111,-1.76],'cal/(mol*K)'),
        H298 = (27.53,'kcal/mol'),
        S298 = (32.0088,'cal/(mol*K)'),
    ),
    shortDesc = u"""Cyclopropane ring BENSON""",
    longDesc =
u"""

""",
)
```

A molecule may have two or more fused rings that mutually interact. In that case, a polycyclic ring strain correction may be more adequate. RMG identifies molecules with fused ring systems and subsequently searches through polycyclic.py to identify an adequate RSC.

An example of a polycyclic RSC entry in polycyclic.py is shown here below:

```
entry(
    index = 2,
    label = "norbornane",
    group =
"""
1 * Cs 0 {3,S} {4,S} {7,S}
2   Cs 0 {3,S} {5,S} {6,S}
3   Cs 0 {1,S} {2,S}
4   Cs 0 {1,S} {5,S}
5   Cs 0 {2,S} {4,S}
```

```
6   Cs 0 {2,S} {7,S}
7   Cs 0 {1,S} {6,S}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([-4.5,-3.942,-3.291,-2.759,-2.08,-1.628,-0.898],'cal/(mol*K)'),
        H298 = (16.14,'kcal/mol'),
        S298 = (53.47,'cal/(mol*K)'),
    ),
    shortDesc = u"""""",
    longDesc =
u"""

""",
)
```

Hydrogen Bond Increments (HBI)

An example of a HBI entry in radical.py is shown here below:

```
entry(
    index = 4,
    label = "CH3",
    group =
"""
1 * C 1 {2,S} {3,S} {4,S}
2   H 0 {1,S}
3   H 0 {1,S}
4   H 0 {1,S}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([0.71,0.34,-0.33,-1.07,-2.43,-3.54,-5.43],'cal/(mol*K)'),
        H298 = (104.81,'kcal/mol','+|-',0.1),
        S298 = (0.52,'cal/(mol*K)'),
    ),
    shortDesc = u"""Calculated in relation to methane from NIST values""",
    longDesc =
u"""

""",
)
```

Non-nearest neighbor interactions

The majority of the NNIs groups pertain to small enthalpy of formation corrections. Only a very limited number
include entropy or heat capacity corrections. The database other.py contains cis-, ortho- and ketene-corrections.

An example of a NNI entry in gauche.py is shown here below:

```
entry(
    index = 11,
    label = "Cs(Cs(CsCsR)Cs(CsCsR)RR)",
```

```
    group =
"""
1  * Cs                        0 {2,S} {3,S} {4,S} {5,S}
2    Cs                        0 {1,S} {6,S} {7,S} {8,S}
3    Cs                        0 {1,S} {9,S} {10,S} {11,S}
4    {Cd,Cdd,Ct,Cb,Cbf,Os,CO,H} 0 {1,S}
5    {Cd,Cdd,Ct,Cb,Cbf,Os,CO,H} 0 {1,S}
6    Cs                        0 {2,S}
7    Cs                        0 {2,S}
8    {Cd,Cdd,Ct,Cb,Cbf,Os,CO,H} 0 {2,S}
9    Cs                        0 {3,S}
10   Cs                        0 {3,S}
11   {Cd,Cdd,Ct,Cb,Cbf,Os,CO,H} 0 {3,S}
""",
    thermo = ThermoData(
        Tdata = ([300,400,500,600,800,1000,1500],'K'),
        Cpdata = ([0,0,0,0,0,0,0],'cal/(mol*K)'),
        H298 = (0.8,'kcal/mol'),
        S298 = (0,'cal/(mol*K)'),
    ),
    shortDesc = u"""""",
    longDesc =
u"""

""",
)
```

1.12.3 Kinetics Database

This section describes the general usage of RMG's kinetic database. See *Modifying the Kinetics Database* for instructions on modifying the database.

Pressure independent reaction rates in RMG are calculated using a modified Arrhenius equation, designating the reaction coefficient as $k(T)$ at temperature T.

$$k(T) = A \left(\frac{T}{T_0} \right)^n e^{-(E_0 + \alpha \Delta H_{rxn})/(RT)}$$

R is the universal gas constant. The **kinetic parameters** determining the rate coefficient are:

- A: the pre-exponential A-factor
- T_0: the reference temperature
- n: the temperature exponent
- E_0: the activation energy for a thermoneutral reaction (a barrier height intrinsic to the kinetics family)
- α: the Evans-Polanyi coefficient (characterizes the position of the transition state along the reaction coordinate, $0 \leq \alpha \leq 1$)
- ΔH_{rxn}: the enthalpy of reaction

When Evans-Polanyi corrections are used, ΔH_{rxn} is calculated using RMG's thermo database, instead of being specified in the kinetic database. When Evans-Polanyi corrections are not used, ΔH_{rxn} and α are set to zero, and E_0 is the activation energy of the reaction.

Libraries

Kinetic libraries delineate kinetic parameters for specific reactions. RMG always chooses to use kinetics from libraries over families. If multiple libraries contain the same reaction, then precedence is given to whichever library is listed first in the input.py file.

For combustion mechanisms, you should always use at least one small-molecule combustion library, such as the pre-packaged *BurkeH2O2* and/or *FFCM1* for natural gas. The reactions contained in these libraries are poorly estimated by kinetic families and are universally important to combustion systems.

Kinetic libraries should also be used in the cases where:

- A set of reaction rates were optimized together

- You know the reaction rate is not generalizable to similar species (perhaps due to catalysis or aromatic structures)

- No family exists for the class of reaction

- You are not confident about the accuracy of kinetic parameters

Below is a list of pre-packaged kinetics library reactions in RMG:

Library	Description
1989_Stewart_2CH3_to_C2H5_H	Chemically Activated Methyl Recombination to Ethyl (2
2001_Tokmakov_H_Toluene_to_CH3_Benzene	H + Toluene = CH3 + Benzene
2005_Senosiain_OH_C2H2	pathways on the OH + acetylene surface
2006_Joshi_OH_CO	pathways on OH + CO = HOCO = H + CO2 surface
2009_Sharma_C5H5_CH3_highP	Cyclopentadienyl + CH3 in high-P limit
2015_Buras_C2H3_C4H6_highP	Vinyl + 1,3-Butadiene and other C6H9 reactions in high-
biCPD_H_shift	Sigmatropic 1,5-H shifts on biCPD PES
BurkeH2O2inArHe	Comprehensive H2/O2 kinetic model in Ar or He atmosp
BurkeH2O2inN2	Comprehensive H2/O2 kinetic model in N2 atmosphere
C2H4+O_Klipp2017	C2H4 + O intersystem crossing reactions, probably impo
C10H11	Cyclopentadiene pyrolysis in the presence of ethene
C3	Cyclopentadiene pyrolysis in the presence of ethene
C6H5_C4H4_Mebel	Formation Mechanism of Naphthalene and Indene
Chernov	Soot Formation with C1 and C2 Fuels (aromatic reaction
CurranPentane	Ignition of pentane isomers
Dooley	Methyl formate (contains several mechanisms)
ERC-FoundationFuelv0.9	Small molecule combustio (natural gas)
Ethylamine	Ethylamine pyrolysis and oxidation
FFCM1(-)	Foundational Fuel Chemistry Model Version 1.0 (excited
First_to_Second_Aromatic_Ring/2005_Ismail_C6H5_C4H6_highP	Phenyl + 1,3-Butadiene and other C10H11 reactions in h
First_to_Second_Aromatic_Ring/2012_Matsugi_C3H3_C7H7_highP	Propargyl + Benzyl and other C10H10 reactions in high-
First_to_Second_Aromatic_Ring/2016_Mebel_C9H9_highP	C9H9 reactions in high-P limit
First_to_Second_Aromatic_Ring/2016_Mebel_C10H9_highP	C10H9 reactions in high-P limit
First_to_Second_Aromatic_Ring/2016_Mebel_Indene_CH3_highP	CH3 + Indene in high-P limit
First_to_Second_Aromatic_Ring/2017_Buras_C6H5_C3H6_highP	Phenyl + Propene and other C9H11 reactions in high-P li
First_to_Second_Aromatic_Ring/2017_Mebel_C6H4C2H_C2H2_highP	C10H7 HACA reactions in high-P limit
First_to_Second_Aromatic_Ring/2017_Mebel_C6H5_C2H2_highP	C8H7 HACA reactions in high-P limit
First_to_Second_Aromatic_Ring/2017_Mebel_C6H5_C4H4_highP	Phenyl + Vinylacetylene and other C10H9 reactions in hi
First_to_Second_Aromatic_Ring/2017_Mebel_C6H5C2H2_C2H2_highP	C10H9 HACA reactions in high-P limit
First_to_Second_Aromatic_Ring/phenyl_diacetylene_effective	Effective Phenyl + Diacetylene rates to Benzofulvenyl an
Fulvene_H	Cyclopentadiene pyrolysis in the presence of ethene
GRI-HCO	The *HCO <=> H + CO* reaction

Table 2 – continued from previous page

Library	Description
GRI-Mech3.0	Gas Research Institute natural gas mechanism optimized
GRI-Mech3.0-N	GRI-Mech3.0 including nitrogen chemistry (NOx from N
Glarborg	Mechanisms by P. Glarborg, assorted by carbon number
JetSurF2.0	Jet Surrogate Fuel model up tp C12 (excited species rem
Klippenstein_Glarborg2016	Methane oxidation at high pressures and intermediate ter
Lai_Hexylbenzene	Alkylaromatic reactions for hexylbenzene
Mebel_C6H5_C2H2	Pathways from benzene to naphthalene
Mebel_Naphthyl	Reactions of naphthyl-1 and naphthyl-2
Methylformate	Methyl formate
Narayanaswamy	Oxidation of substituted aromatic species (aromatic react
Nitrogen_Dean_and_Bozzelli	Combustion Chemistry of Nitrogen
Nitrogen_Glarborg_Gimenez_et_al	High pressure C2H4 oxidation with nitrogen chemistry
Nitrogen_Glarborg_Lucassen_et_al	Fuel-nitrogen conversion in the combustion of small ami
Nitrogen_Glarborg_Zhang_et_al	Premixed nitroethane flames at low pressure
NOx	important NOx related reactions, e.g., thermal & prompt
NOx/LowT	Low temperature kinetics (~<1000K) for selected reactio
OxygenSingTrip	Reactions of singlet and triplet oxygen
SOx	important SOx related reactions, e.g., H-S, C-S, SOx
Sulfur/DMDS	Dimethyl disulfide (CH3SSCH3)
Sulfur/DMS	Dimethyl disulfide (CH3SSCH3)
Sulfur/DTBS	Di-tert-butyl Sulfide (C4H9SSC4H9)
Sulfur/GlarborgBozzelli	SO2 effect on moist CO oxidation with and without NO
Sulfur/GlarborgH2S	H2S oxidation at high pressures
Sulfur/GlarborgMarshall	OCS chemistry
Sulfur/GlarborgNS	Interactions between nitrogen and sulfur species in comb
Sulfur/Hexanethial_nr	Hexyl sulfide (C6H13SC6H13) + hexadecane (C16H34)
Sulfur/Sendt	Small sulfur molecule
Sulfur/TP_Song	Thiophene (C4H4S, aromatic)
Sulfur/Thial_Hydrolysis	Thioformaldehyde (CH2S) and thioacetaldehyde (C2H4S
TEOS	Organic oxidized silicone
c-C5H5_CH3_Sharma	Cyclopentadienyl + CH3
combustion_core	Leeds University natural gas mechanism (contains versio
fascella	Cyclopentadienyl + acetyl
kislovB	Formation of indene in combustion
naphthalene_H	Cyclopentadiene pyrolysis in the presence of ethene Part
vinylCPD_H	Cyclopentadiene pyrolysis in the presence of ethene Part

Families

Allowable reactions in RMG are divided up into classes called **reaction families**. All reactions not listed in a kinetic library have their kinetic parameters estimated from the reaction families.

Each reaction family contains the files:

- groups.py containing the recipe, group definitions, and hierarchical trees
- training.py containing a training set for the family
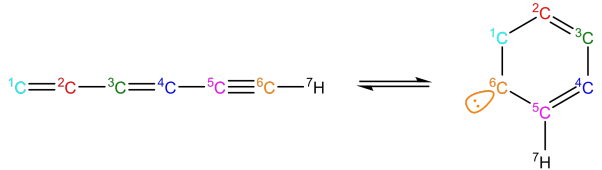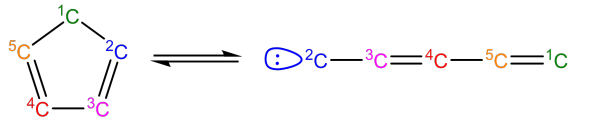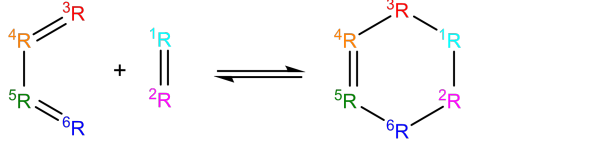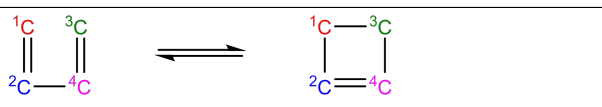- rules.py containing kinetic parameters for rules
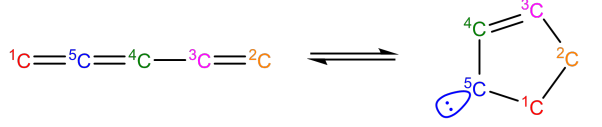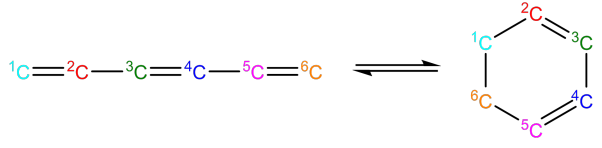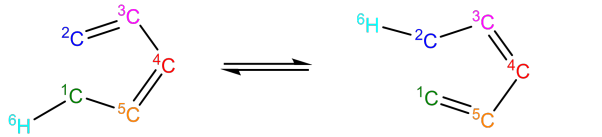
There are currently 58 reaction families in RMG:

1,2-Birad_to_alkene	
1,2_Insertion_carbene	
1,2_Insertion_CO	
1,2_shiftC	
1,2_shiftS	
1,3_Insertion_CO2	
1,3_Insertion_ROR	
1,3_Insertion_RSR	
1,4_Cyclic_birad_scission	
1,4_Linear_birad_scission	
1+2_Cycloaddition	
2+2_cycloaddition_CCO	
2+2_cycloaddition_Cd	
2+2_cycloaddition_CO	
2+2_cycloaddition_CS	
6_membered_central_C-C_shift	
Baeyer-Villiger_step1_cat	
Baeyer-Villiger_step2	
Baeyer-Villiger_step2_cat	

Table 3 – continued from previous page

Bimolec_Hydroperoxide_Decomposition	R—¹O—²O—H + R—O—¹O—³H ⇌ R—¹O· + H—²O—³H + R—O—¹O·
Birad_recombination	¹R∿∿²R ⇌ ¹R∿∿²R
Birad_R_Recombination	¹R· + ²R: ⇌ ¹R—²R·
CO_Disproportionation	¹R· + ²O=³C—⁴H ⇌ ¹R—⁴H + ²O≡³C
Concerted_Intra_Diels_alder_monocyclic_1,2_shiftH	¹C=²C—³C=⁴C—⁵C≡⁶C—⁷H ⇌ (ring structure with ¹C ²C ³C ⁴C ⁵C ⁶C ⁷H)
Cyclic_Ether_Formation	¹R∿∿²O—³OR ⇌ ¹R∿∿²O + ³OR·
Cyclic_Thioether_Formation	¹R·∿∿²S—³(O/S)R ⇌ ¹R∿∿²S + ³(O/S)R
Cyclopentadiene_scission	(ring with ¹C ²C ³C ⁴C ⁵C) ⇌ :²C—³C=⁴C—⁵C=¹C
Diels_alder_addition	(⁴R=³R, ⁵R=⁶R) + (¹R=²R) ⇌ (six-membered ring ¹R ²R ³R ⁴R ⁵R ⁶R)
Disproportionation	¹R· + ³R—²R—⁴H ⇌ ¹R—⁴H + ³R=²R
H_Abstraction	¹R—²H + ³R· ⇌ ¹R· + ²H—³R
HO2_Elimination_from_PeroxyRadical	⁵H—¹R—²R—³O—⁴O· ⇌ ³O·—⁴O—⁵H + ¹R=²R
Intra_2+2_cycloaddition_Cd	(¹C=²C, ³C=⁴C) ⇌ (ring ¹C ³C ²C ⁴C)
Intra_5_membered_conjugated_C=C_C=C_addition	¹C=⁵C=⁴C—³C=²C ⇌ (five-membered ring ¹C ²C ³C ⁴C ⁵C)
Intra_Diels_alder_monocyclic	¹C=²C—³C=⁴C—⁵C=⁶C ⇌ (six-membered ring ¹C ²C ³C ⁴C ⁵C ⁶C)
Intra_Disproportionation	¹R·∿∿³R—²R—⁴H ⇌ ⁴H—¹R∿∿³R=²R
Intra_ene_reaction	(ring ²C ³C ⁴C ¹C ⁵C, ⁶H) ⇌ (⁶H—²C ³C ⁴C ¹C=⁵C)
intra_H_migration	³H—²R∿∿¹R· ⇌ ²R∿∿¹R—³H

Table 3 – continued from previous page

intra_NO2_ONO_conversion	^1R—$^2N^+$—$^3O^-$ with O double bond ⇌ ^2N—^3O—^1R with O double bond
intra_OH_migration	^1R~~^2O—^3OH ⇌ ^3HO—^1R~~^2O
Intra_R_Add_Endocyclic	^1R~~^2R=^3R ⇌ ^1R~~^2R—^3R (cyclic)
Intra_R_Add_Exocyclic	^1R~~^2R=^3R ⇌ ^1R~~^2R—^3R (cyclic)
Intra_R_Add_Exo_scission	^1C—^3C—^2Cb ⇌ ^3C—^1C—^2Cb
Intra_R_Add_ExoTetCyclic	^1R~~^2R—^3R ⇌ ^1R~~^2R + ^3R (cyclic)
Intra_Retro_Diels_alder_bicyclic	ring(^1R,^2R,^3R,^4R,^5R,^6R) ⇌ ^1R=^2R—^3R=^4R~~^5R=^6R
Intra_RH_Add_Endocyclic	^4H—^1R~~^2R=^3R ⇌ ^4H...^1R~~^2R—^3R (cyclic)
Intra_RH_Add_Exocyclic	^4H—^1R~~^2R=^3R ⇌ ^1R~~^2R—^3R—^4H (cyclic)
intra_substitutionCS_cyclization	^3R~~^1C—^2S ⇌ ^3R~~^1C + ^2S (cyclic)
intra_substitutionCS_isomerization	^3R~~^2S—^1C ⇌ ^1C—^3R~~^2S
intra_substitutionS_cyclization	^3R~~^1S—^2R ⇌ ^3R~~^1S + ^2R (cyclic)
intra_substitutionS_isomerization	^3R~~^2R—^1S ⇌ ^1S—^3R~~^2R
ketoenol	^1R=^2R—^2O—^4R ⇌ ^4R—^1R—^2R=^2O
Korcek_step1	(ring structure) ⇌ (ring structure)
Korcek_step1_cat	(structure) + (structure) ⇌ (structure) + (structure)
Korcek_step2	(ring structure) ⇌ (structure) + (structure)

Continued on next page

Table 3 – continued from previous page

lone_electron_pair_bond	$R-{}^1N(R)(R) + {}^2O \rightleftharpoons R-{}^1N^+(R)(R)-{}^2O^-$
Peroxyl_Disproportionation	$R-{}^1O-{}^2O\cdot + R-{}^1O-{}^1O\cdot \rightleftharpoons R-{}^1O\cdot + R-{}^1O\cdot + {}^2O={}^1O$
Peroxyl_Termination	${}^4H-{}^1R-{}^2O-{}^1O\cdot + R-{}^1O-{}^4O\cdot \rightleftharpoons {}^1R={}^2O + R-{}^1O-{}^1H + {}^1O-{}^1O$
R_Addition_COm	${}^1C\equiv{}^3O^+ + {}^2R\cdot \rightleftharpoons {}^2R-{}^1C\cdot(={}^3O)$
R_Addition_CSm	${}^1C\equiv{}^3S^+ + {}^2R\cdot \rightleftharpoons {}^2R-{}^1C\cdot(={}^3S)$
R_Addition_MultipleBond	${}^2R={}^1R + {}^3R\cdot \rightleftharpoons {}^2R-{}^1R-{}^3R$
R_Recombination	${}^1R\cdot + {}^2R\cdot \rightleftharpoons {}^1R-{}^2R$
Singlet_Carbene_Intra_Disproportionation	$:{}^1C-{}^2C-{}^3H \rightleftharpoons {}^3H-{}^1C={}^2C$
Singlet_Val6_to_triplet	${}^1(O/S)={}^2(O/S) \rightarrow {}^1(O/S)-{}^2(O/S)$
Substitution_O	$R-{}^1O-{}^2R + {}^3R\cdot \rightleftharpoons R-{}^1O-{}^3R + {}^2R\cdot$
SubstitutionS	$R-{}^1S-{}^2R + {}^3R\cdot \rightleftharpoons R-{}^1S-{}^3R + {}^2R\cdot$

Recipe

The recipe can be found near the top of groups.py and describes the changes in bond order and radicals that occur during the reaction. Reacting atoms are labelled with a starred number. Shown below is the recipe for the H-abstraction family.
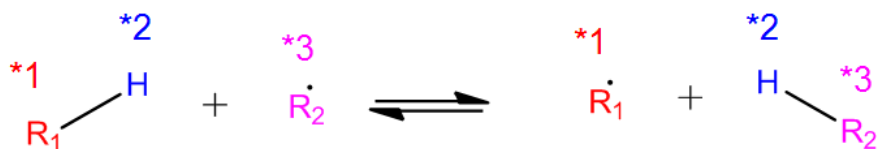
```
Recipe
(1) BREAK_BOND    {*1,S,*2}
(2) FORM_BOND     {*2,S,*3}
(3) GAIN_RADICAL  {*1,1}
(4) LOSE_RADICAL  {*3,1}
```

The table below shows the possible actions for recipes. The arguments are given in the curly braces as shown above. For the order of bond change in the Change_Bond action, a -1 could represent a triple bond changing to a double bond while a +1 could represent a single bond changing to a double bond.

Action	Argument1	Argument2	Argument3
Break_Bond	First bonded atom	Type of bond	Second bonded atom
Form_Bond	First bonded atom	Type of bond	Second bonded atom
Change_Bond	First bonded atom	Order of bond change	Second bonded atom
Gain_Radical	Specified atom	Number of radicals	
Lose_Radical	Specified atom	Number of radicals	

Change_Bond order cannot be directly used on benzene bonds. During generation, aromatic species are kekulized to alternating double and single bonds such that reaction families can be applied. However, RMG cannot properly handle benzene bonds written in the kinetic group definitions.

Training Set vs Rules

The training set and rules both contain trusted kinetics that are used to fill in templates in a family. The **training set** contains kinetics for specific reactions, which are then matched to a template. The kinetic **rules** contain kinetic parameters that do not necessarily correspond to a specific reaction, but have been generalized for a template.

When determining the kinetics for a reaction, a match for the template is searched for in the kinetic database. The three cases in order of decreasing reliability are:

1. Reaction match from training set

2. Node template exact match using either training set or rules

3. Node template estimate averaged from children nodes

Both training sets and reaction libraries use the observed rate, but rules must first be divided by the degeneracy of the reaction. For example, the reaction CH4 + OH –> H2O + CH3 has a reaction degeneracy of 4. If one performed an experiment or obtained this reaction rate using Arkane (applying the correct symmetry), the resultant rate parameters would be entered into libraries and training sets unmodified. However a kinetic rule created for this reaction must have its A-factor divided by 4 before being entered into the database.

The reaction match from training set is accurate within the documented uncertainty for that reaction. A template exact match is usually accurate within about one order of magnitude. When there is no kinetics available for for the template in either the training set or rules, the kinetics are averaged from the children nodes as an estimate. In these cases, the kinetic parameters are much less reliable. For more information on the estimation algorithm see *Kinetics Estimation*.

The training set can be modified in training.py and the rules can be modified in rules.py. For more information on modification see *Adding Training Reactions* and *Adding Kinetic Rules*.

1.12.4 Database Modification

Note that the RMG-Py database is written in Python code where line indentations determine the scope. When modifying the database, be sure to preserve all line indentations shown in the examples.

Modifying the Thermo Database

Creating Thermo Libraries

Adding Thermo Groups

Adding Thermo to the Depository

Modifying the Kinetics Database

For the casual user, it is recommended to use either a kinetic library or add to the training set instead of modifying the kinetic groups.

Put kinetic parameters into a kinetic library when:

- A set of reaction rates were optimized together
- You know the reaction rate is not generalizable to similar species (perhaps due to catalysis or aromatic structures)
- No family exists for the class of reaction
- You are not confident about the accuracy of kinetic parameters

Put kinetic parameters into the training set when:

- You are confident on the accurcy of the kinetic parameter
- You wish for the reaction to be generalized to similar reactions in your mechanism

Adding Reaction Family

There are several places in the RMG-database and RMG-Py source code where reaction family details are hard-coded. You should check all these when you create a new reaction family. Here are some of the places:

- **RMG-database/input/kinetics/families/[family name]**
    - add folder for your family name
    - create `groups.py`, `rules.py` and a template folder with species dictionary and `reactions.py`.
    - fill the files with rate data that you plan to use.
    - **Many tools exist to help with the conversion process:**
        * `convertKineticsLibraryToTrainingReactions.ipynb` in RMG-database/scripts
        * `importChemkinLibrary.py` in RMG-database/scripts
- **rmgpy.data.kinetics.family**
    - `applyRecipe`: swapping the atom labels (eg. *1 and *2) around
    - `getReactionPairs`: figuring out which species becomes which for flux analyses
    - `__generateReactions`: correcting degeneracy eg. dividing by 2 for radical recombination
- **rmgpy.data.kinetics.rules**
    - `processOldLibraryEntry`: determining units when importing RMG-Java database
    - `getAllRules`: for radical recombination add reverse templates
- **rmgpy.data.kinetics.groups**

> – getReactionTemplate: for radical recombination duplicate the template

- **RMG-database/input/kinetics/families/recommended.py**

  > – allows the usage of the database with the recommended families.

Creating Kinetics Libraries

To add a reaction library, simply create a folder bearing the library's name under `RMG-database/input/kinetics/libraries`. You'll need to create two files: `dictionary.txt` and `reactions.py`. The dictionary file contains the Adjacency lists for all relevant species (can be generated using the Molecule Search function of the rmg website, while the reactions file specifies the kinetics. To conform to RMG's format, simply copy and modify an existing library.

At the top of the reactions file fill in the name and short (one line) and long descriptions. The name must be identical to the folder's name. Then list the kinetics entries, each with a unique index number.

There are two flags relevant for pressure dependent library reactions that one should consider using:

1. **elementary_high_p**: Should be set to `True` for *elementary* unimolecular reactions (with only one reactant and/or product) with a kinetics entry that has information about the high pressure kinetics, i.e., Troe or Lindemann, PDepArrhenius or Chebyshev that are defined up to at least 100 bar, or Arrhenius that represents the high pressure limit (i.e., not the measured rate at some low or medium experimental pressure). If set to `True`, RMG will use the high pressure limit rate when constructing pressure-dependent networks. The kinetics entry of the original library reaction will only be updated if it is an Arrhenius type (will be replaced with either PDepArrhenius or Chebyshev, as specified in the pressureDependent block of the input file). If set to `False` (the default value), RMG will not use the high pressure limit rate in network exploration, and will not convert Arrhenius kinetics of library reactions that have no template (a corresponding reaction family) into a pressure-dependent form.

2. **allow_pdep_route**: If set to `True` and RMG discovers a pressure-dependent reaction with the same reactants and products, the latter *will* be considered in addition to the library reaction. This is useful for cases when more than one pathway connects the same reactants and products, and some of these pathways are well-skipping reactions. If set to `False` (the default value), similar network reactions will not be considered in the model generation.

The following formats are accepted as kinetics entries:

Arrhenius of the form $k(T) = A \left(\frac{T}{T_0} \right)^n \exp\left(-\frac{E_a}{RT} \right)$ (see Arrhenius Class for details):

```
entry(
index = 1,
label = "H + O2 <=> O + OH",
degeneracy = 1,
kinetics = Arrhenius(A=(9.841e+13, 'cm^3/(mol*s)'), n=0, Ea=(15310, 'cal/mol'), T0=(1, 'K')),
shortDesc = u"This is a short description limited to one line, e.g. 'CBS-QB3'",
longDesc = u"""This is a long description, unlimited by number of lines.
These descriptions can be added to every kinetics type.""")
```

MultiArrhenius is the sum of multiple Arrhenius expressions (all apply to the same temperature range) (see MultiArrhenius Class for details):

```
entry(
index = 2,
label = "O + H2 <=> H + OH",
degeneracy = 1,
        duplicate = True,
kinetics = MultiArrhenius(
        arrhenius = [Arrhenius(A=(3.848e+12, 'cm^3/(mol*s)'), n=0, Ea=(7950, 'cal/mol'), T0=(1,
↪'K')),
        Arrhenius(A=(6.687e+14, 'cm^3/(mol*s)'), n=0, Ea=(19180, 'cal/mol'), T0=(1, 'K'))]))
```

ThirdBody for pressure dependent reactions of the sort `H2 + M <=> H + H + M`. `efficiencies` are optional and specify the factor by which the rate is multiplies if the mentioned species is the third body collider. Note that for complex efficiency behaviour, an efficiency of `0` can be set, and a seperate specific reaction can be defined (see ThirdBody Class for details):

```
entry(
index = 3,
label = "H2 <=> H + H",
degeneracy = 1,
kinetics = ThirdBody(
        arrheniusLow = Arrhenius(A=(4.58e+19, 'cm^3/(mol*s)'), n=-1.4, Ea=(104390, 'cal/mol'),
→T0=(1, 'K')),
        efficiencies = {'[Ar]': 0, 'N#N': 1.01, '[H][H]': 2.55, 'O': 12.02, '[C-]#[O+]': 1.95,
→'O=C=O': 3.83, 'C': 2.00, 'C=O': 2.50, 'CO': 3.00, 'CC': 3.00}))

entry(
index = 4,
label = "H2 + Ar <=> H + H + Ar",
degeneracy = 1,
kinetics = Arrhenius(A=(5.176e+18, 'cm^3/(mol*s)'), n= 1.1, Ea=(104390, 'cal/mol'), T0=(1, 'K
→')))
```

Troe for pressure dependent reactions (see Troe Class for details):

```
entry(
index = 5,
label = "H + O2 <=> HO2",
degeneracy = 1,
kinetics = Troe(
        arrheniusHigh = Arrhenius(A=(4.565e+12, 'cm^3/(mol*s)'), n=0.44, Ea=(0, 'cal/mol'),
→T0=(1, 'K')),
        arrheniusLow = Arrhenius( A=(6.37e+20, 'cm^6/(mol^2*s)'), n = -1.72, Ea = (525, 'cal/mol
→'), T0 = (1, 'K')),
        alpha=0.5, T3=(30, 'K'), T1=(90000, 'K'), T2=(90000, 'K'),
        efficiencies = {'[Ar]': 0.6, '[He]': 0.71, 'N#N': 0.96, '[H][H]': 1.87, '[O][O]': 0.75,
→'O': 15.81, '[C-]#[O+]': 1.90, 'O=C=O': 3.45, 'C': 2.00, 'C=O': 2.50, 'CO': 3.00, 'CC': 3.00}
→))
```

Lindemann (see Lindemann Class for details):

```
entry(
    index = 6,
    label = "CO + O <=> CO2",
    degeneracy = 1,
    kinetics = Lindemann(
        arrheniusHigh = Arrhenius(A=(1.88e+11, 'cm^3/(mol*s)'), n=0, Ea=(2430, 'cal/mol'),
→T0=(1, 'K')),
        arrheniusLow = Arrhenius(A = (1.4e+21, 'cm^6/(mol^2*s)'), n = -2.1, Ea = (5500, 'cal/mol
→'), T0 = (1, 'K')),
        efficiencies = {'[Ar]': 0.87, '[He]': 2.50, 'O': 12.00, '[C-]#[O+]': 1.90, 'O=C=O': 3.
→80, 'C': 2.00, 'C=O': 2.50, 'CO': 3.00, 'CC': 3.00}))
```

PDepArrhenius where each Arrhenius expression corresponds to a different pressure, as specified. Allowed pressure units are `Pa`, `bar`, `atm`, `torr`, `psi`, `mbar` (see PDepArrhenius Class for details):

```
entry(
    index = 7,
    label = "HCO <=> H + CO",
```

```
    degeneracy = 1,
    kinetics = PDepArrhenius(
        pressures = ([1, 10, 20, 50, 100], 'atm'),
        arrhenius = [
            Arrhenius(A=(9.9e+11, 's^-1'), n=-0.865, Ea=(16755, 'cal/mol'), T0=(1, 'K')),
            Arrhenius(A=(7.2e+12, 's^-1'), n=-0.865, Ea=(16755, 'cal/mol'), T0=(1, 'K')),
            Arrhenius(A=(1.3e+13, 's^-1'), n=-0.865, Ea=(16755, 'cal/mol'), T0=(1, 'K')),
            Arrhenius(A=(2.9e+13, 's^-1'), n=-0.865, Ea=(16755, 'cal/mol'), T0=(1, 'K')),
            Arrhenius(A=(5.3e+13, 's^-1'), n=-0.865, Ea=(16755, 'cal/mol'), T0=(1, 'K'))]))
```

MultiPDepArrhenius (see MultiPDepArrhenius Class for details):

```
entry(
    index = 8,
    label = "N2H2 <=> NNH + H",
    degeneracy = 1,
    duplicate = True,
    kinetics = MultiPDepArrhenius(
        arrhenius = [
            PDepArrhenius(
                pressures = ([0.1, 1, 10], 'atm'),
                arrhenius = [
                    Arrhenius(A=(5.6e+36, '1/s'), n=-7.75, Ea=(70250.4, 'cal/mol'), T0=(1, 'K
→')),
                    Arrhenius(A=(1.8e+40, '1/s'), n=-8.41, Ea=(73390, 'cal/mol'), T0=(1, 'K')),
                    Arrhenius(A=(3.1e+41, '1/s'), n=-8.42, Ea=(76043, 'cal/mol'), T0=(1, 'K
→'))]),
            PDepArrhenius(
                pressures = ([0.1, 1, 10], 'atm'),
                arrhenius = [
                    Arrhenius(A=(1.6e+37, '1/s'), n=-7.94, Ea=(70757, 'cal/mol'), T0=(1, 'K')),
                    Arrhenius(A=(2.6e+40, '1/s'), n=-8.53, Ea=(72923, 'cal/mol'), T0=(1, 'K')),
                    Arrhenius(A=(1.3e+44, '1/s'), n=-9.22, Ea=(77076, 'cal/mol'), T0=(1, 'K
→'))])])
```

Chebyshev (see Chebyshev Class for details):

```
entry(
    index = 9,
    label = "CH3 + OH <=> CH2(S) + H2O",
    degeneracy = 1,
    kinetics = Chebyshev(
        coeffs = [
            [12.4209, -0.799241, -0.299133, -0.0143012],
            [0.236291, 0.856853, 0.246313, -0.0463755],
            [-0.0827561, 0.0457236, 0.105699, 0.057531],
            [-0.049145, -0.0760609, -0.0214574, 0.0247001],
            [-0.00664556, -0.0412733, -0.0308561, -0.00959838],
            [0.0111919, -0.00649914, -0.0106088, -0.0137528],
        ],
        kunits='cm^3/(mol*s)', Tmin=(300, 'K'), Tmax=(3000, 'K'), Pmin=(0.0013156, 'atm'),
→Pmax=(131.56, 'atm')))
```

Adding a specific collider

Only the `Troe` and `Lindemann` pressure dependent formats could be defined with a specific species as a third body collider, if needed. For example:

```
entry(
    index = 10,
    label = "SO2 + O <=> SO3",
    degeneracy = 1,
    kinetics = Troe(
        arrheniusHigh = Arrhenius(A=(3.7e+11, 'cm^3/(mol*s)'), n=0, Ea=(1689, 'cal/mol'), T0=(1,
→ 'K')),
        arrheniusLow = Arrhenius(A=(2.4e+27, 'cm^6/(mol^2*s)'), n=-3.6, Ea=(5186, 'cal/mol'),
→T0=(1, 'K')),
        alpha = 0.442, T3=(316, 'K'), T1=(7442, 'K'), efficiencies={'O=S=O': 10, 'O': 10, 'O=C=O
→': 2.5, 'N#N': 0}))

entry(
    index = 11,
    label = "SO2 + O (+N2) <=> SO3 (+N2)",
    degeneracy = 1,
    kinetics = Troe(
        arrheniusHigh = Arrhenius(A=(3.7e+11, 'cm^6/(mol^2*s)'), n=0, Ea=(1689, 'cal/mol'),
→T0=(1, 'K')),
        arrheniusLow = Arrhenius(A=(2.9e+27, 'cm^9/(mol^3*s)'), n=-3.58, Ea=(5206, 'cal/mol'),
→T0=(1, 'K')),
        alpha=0.43, T3=(371, 'K'), T1=(7442, 'K'), efficiencies={}))
```

Adding New Kinetic Groups and Rate Rules

Decide on a Template

First you need to know the template for your reaction to decide whether or not to create new groups:

1. Type your reaction into the kinetics search at http://rmg.mit.edu/database/kinetics/search/

2. Select the correct reaction

3. In the results search for "(RMG-Py rate rules)" and select that link. The kinetic family listed is the family of interest.

4. Scroll to the bottom and look at the end of the long description. There may be very long description of the averaging scheme, but the template for the reaction is the very last one listed:

Reference:

Reference link: http://dev.rmg.mit.edu/database/kinetics/families/H_Abstraction/rules

Reference type:

Short description: Estimated by RMG-Py rate rules

Long description: Average of (O_pri,C_methyl + Average of (O/H/NonDeC,C_methyl + O/H/OneDe,C_methyl). + Average of (O_pri,C_rad/H2/Cs). + Average of (O/H/NonDeC,C_rad/H2/Cs + Average of (H2O2,InChI=1/C4H9O/c1-2-3-4-5/h5H,1-4H2 + H2O2,InChI=1/C4H9O/c1-3-4(2)5/h4-5H,2-3H2,1H3 + H2O2,InChI=1/C4H9O/c1-3-4(2)5/h4-5H,1,3H2,2H3 + H2O2,InChI=1/C4H9O/c1-4(2,3)5/h5H,1H2,2-3H3). + Average of (H2O2,InChI=1/C3H5/c1-3-2/h3H,1-2H2). + Average of (ROOH_pri,C_rad/H2/CO + ROOH_sec,C_rad/H2/CO).). + Average of (O/H/NonDeC,C_rad/H/NonDeC + Average of (H2O2,InChI=1/C4H9O/c1-2-3-4-5/h2,5H,3-4H2,1H3 + H2O2,InChI=1/C4H9O/c1-2-3-4-5/h3,5H,2,4H2,1H3 + H2O2,InChI=1/C4H9O/c1-3-4(2)5/h3-5H,1-2H3). + Average of (Average of (H2O2,InChI=1/C4H9O/c1-2-3-4-5/h4-5H,2-3H2,1H3).). + Average of (Average of (ROOH_pri,C_rad/H/CO/Cs). + Average of (ROOH_sec,C_rad/H/CO/Cs).).). + Average of (Average of (O/H/NonDeC,C_rad/Cs3). + Average of (Average of (Average of (H2O2,InChI=1/C4H9O/c1-3-4(2)5/h5H,3H2,1-2H3).). + Average of (Average of (ROOH_pri,C_rad/OOH/Cs/Cs).). + Average of (Average of (ROOH_sec,C_rad/OOH/Cs/Cs).).).). Estimated using template (O_H,Cs_rad) for rate rule (O_pri,C_rad/Cs3)

Now you must determine whether the chosen template is appropriate. A good rule of thumb is to see if the all neighbours of the reacting atoms are as specified as possible. For example, assume your species is ethanol
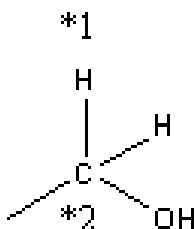
and RMG suggests the group:

```
label = "C_sec",
group =
"""
1 *1 Cs  0 {2,S} {3,S} {4,S}
2 *2 H   0 {1,S}
3    R!H 0 {1,S}
4    R!H 0 {1,S}
""",
```

If you use the suggested groups you will not capture the effect of the alcohol group. Therefore it is better to make a new group.

```
label = "C/H2/CsO",
group =
"""
1 *1 Cs   0 {2,S} {3,S} {4,S} {5,S}
2 *2 H   0 {1,S}
3    H   0 {1,S}
4    O   0 {1,S}
5    Cs  0 {1,S}
""",
```

If you have determined the suggested groups is appropriate, skip to *Adding Training Reactions* or *Adding Kinetic Rules*. Otherwise proceed to the next section for instructions on creating the new group.

Creating a New Group

In the family's groups.py, you will need to add an entry of the format:

```
entry(
        index = 61,
        label = "C_sec",
        group =
"""
1 *1 Cs   0 {2,S} {3,S} {4,S} {5,S}
2 *2 H   0 {1,S}
3    C   0 {1,S}
4    H   0 {1,S}
5    R!H 0 {1,S}
""",
        kinetics = None,
        reference = None,
        referenceType = "",
        shortDesc = u"""""",
        longDesc = u"""""",
)
```

- The index can be any number not already present in the set

- The label is the name of the group.

- The group is the group adjacency list with the starred reacting atoms.

- The other attributes do not need to be filled for a group

Next, you must enter your new group into the tree. At the bottom of groups.py you will find the trees. Place your group in the appropriate position. In the example given in the previous section, the new group would be added under the C_sec.

```
L1: X_H
        L2: H2
        L2: Cs_H
                L3: C_pri
                L3: C_sec
                        L4: C/H2/CsO
                L3: C_ter
```

Adding Kinetic Rules

Rules give generalized kinetic parameters for a specific node template. In most cases, your kinetic parameters describe a specific reaction in which case you will want to add your reaction to the training set.

The rule must be added into rules.py in the form:

```
entry(
        index = 150,
        label = "C/H/Cs3;O_rad/NonDeO",
        group1 =
"""
1 *1 Cs  0 {2,S} {3,S} {4,S} {5,S}
2 *2 H  0 {1,S}
3    Cs 0 {1,S}
4    Cs 0 {1,S}
5    Cs 0 {1,S}
""",
        group2 =
"""
1 *3 O 1 {2,S}
2    O 0 {1,S}
""",
        kinetics = ArrheniusEP(
                A = (2800000000000.0, 'cm^3/(mol*s)', '*|/', 5),
                n = 0,
                alpha = 0,
                E0 = (16.013, 'kcal/mol', '+|-', 1),
                Tmin = (300, 'K'),
                Tmax = (1500, 'K'),
        ),
        reference = None,
        referenceType = "",
        rank = 5,
        shortDesc = u"""Curran et al. [8] Rate expressions for H atom abstraction from fuels.""
↪",
        longDesc =
u"""
[8] Curran, H.J.; Gaffuri, P.; Pit z, W.J.; Westbrook, C.K. Combust. Flame 2002, 129, 253.
Rate expressions for H atom abstraction from fuels.

pg 257 A Comprehensive Modelling Study of iso-Octane Oxidation, Table 1. Radical:HO2, Site:␣
↪tertiary (c)

Verified by Karma James
""",
)
```

- The index can be any number not already used in rules.py.

- The label is the name of the rule.

- The groups must have the adjacency list of the respective groups. Between them they should have all starred atoms from the recipe.

- **The value and units of kinetic parameters must be given.**

    - Multiplicative uncertainty is given as `'*\|/,'` 5 meaning within a factor of 5

    - Additive uncertainty is given as `'+\|/-'`, 2 meaning plus or minus 2.

- Rank determines the priority of the rule when compared with other rules.

- The short description will appear in the annotated chemkin file.

- The long description only appears in the database.

Adding Training Reactions

If you know the kinetics of a specific reaction, rather than a rate rule for a template, you can add the kinetics to the database training set. By default, RMG creates new rate rules from this training set, which in turn benefits the kinetics of similar reactions. The new rate rules are formed by matching the reaction to the most most specific template nodes within the reaction's respective family. If you do not want the training depository reactions to create new rate rules in the database, set the option for `kineticsDepositories` within the `database` field in your input file to

```
kineticsDepositories = ['!training'],
```

Currently, RMG's rate rule estimates overrides all kinetics depository kinetics, including training reactions. Unless the training reaction's rate rule ranks higher than the existing node, it will not be used. If you want the training reaction to override the rate rule estimates, you should put the reaction into a reaction library or seed mechanism.

The easiest way to add training reactions to the database is via the RMG website. First, search for the reaction using http://rmg.mit.edu/database/kinetics/search/ . This will automatically search the existing RMG database for the reaction, as well as identify the reaction family template that this reaction matches. If the reaction does not match any family, then it cannot be added to the training reactions. Click the 'Create training rate from average' button underneath the kinetics plot for the reaction and edit the kinetics and reference descriptions for the reaction. The atom labels marking the reaction recipe actions (lose bond, add radical, etc.) will already be automatically labeled for you. After editing the reaction data, write a short message for the reaction added under the 'Summary of changes' field, then click 'Save.' You will need an account for the RMG website to make an entry.

Note: If you are entering the reaction in the reverse direction of the family, you must still label the reactants and products with the atomLabels of the original reaction template. Otherwise, RMG will not be able to locate the nodes in the group tree to match the reaction.

Entries added in the reverse direction of the original template will use the current RMG job's thermo database to estimate the kinetics in the forward direction. Therefore this value can differ depending on the order of thermo libraries used when running a job.

If adding the training reaction manually, first identify the reaction family of the reaction, then go to the family's folder in `RMG-database/input/kinetics/families/`. Create a new kinetics entry in the `training.py` file. Make sure to apply the reaction recipe labels properly for the reactants and products.

Pitfalls

Be careful with the specificity when naming neighbouring atoms. On upper nodes, you should try to be general so that you do not exclude reactions.

Sibling nodes must be exclusive from one another so that there is no question which group a molecule qualifies as. However, you do not need to be exhaustive and list out every possibility.

Be sure to give errors whenever adding rules. If you don't know the uncertainty, why do you trust the kinetics?

After you are done always check via populate reactions or the website, that your modifications are behaving the way you expect.

Caveat regarding how rate rules are used by RMG and the rate parameters you input: because tunneling is important for many chemical reactions, the rate of a reaction may not be easily represented by a bi-Arrhenius fit. 3-parameter fits are more common. However, the resulting fit may report an 'activation energy' that is much different (possibly by 10+ kcals) than the the true barrier height. When RMG is assembling pressure-dependent networks, it will use barrier heights from rate rules. This can lead to very inaccurate rate calculations. To avoid this issue, try to ensure that your fitted arrhenius activation energy truly does reflect the reaction barrier height.

1.13 Thermochemistry Estimation

This section gives in-depth descriptions of the methods used for determining thermochemistry of species.

Thermochemistry of species is obtained via three possible ways:

1. Species thermochemistry libraries

2. Group contribution methods

3. On-the-fly Quantum-chemical calculation of Thermochemical Properties (QMTP)

1.13.1 Species thermochemistry libraries

These databases contain thermochemical parameters for species. In these databases each entry contains an unambiguous definition of the species (through the adjacency list representation), along with a values for the thermochemistry in a format that allows the evaluation of each thermodynamic variable as a function of temperature.

RMG is shipped with a number of species thermochemistry libraries, located in the 'libraries' folder of RMG-database. More information on these species thermochemistry libraries can be found in *Thermo Database*.

1.13.2 Group contribution methods

When the thermochemistry of a species is not present in one of the available species thermochemistry libraries, RMG needs to estimate thermochemistry. One way to do so, is by using group contribution methods that estimate the thermochemistry of a molecule based on the sub-molecular fragments present in the molecule. The Benson group additivity framework is such an example of a group contribution method that has proven to provide accurate estimates of the ideal gas thermochemistry for a large range of molecules.

Benson's Group Additivity approach ([Benson1976]), divides a molecule into functional groups, and the contribution of each functional group to the overall thermochemistry is included. For example, the molecule 2-methylnonane consists of three types of groups:

• 1 tertiary carbon atom

• 6 secondary carbon atoms

• 3 primary carbon atoms
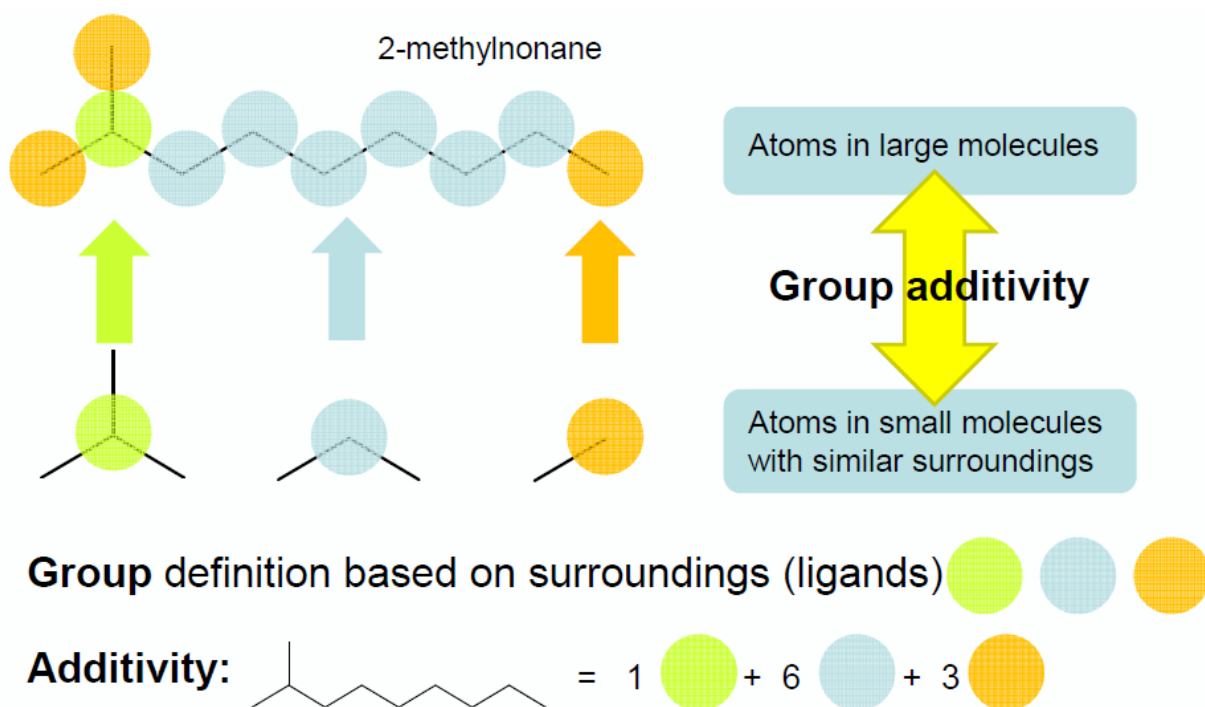
Thermochemistry for the molecule X is calculated by summing up the values for each of the contributions C_i. E.g.:

$$\Delta_f H_{298}^o(X) = \sum_i GAV(C_i)$$

The term 'group additive value' (GAV) denotes a polyvalent (ligancy > 1) monoatomic central atom C_i surrounded by its nearest-neighbor ligands.

Values for each central atomtype (e.g. "tertiary carbon atom") and its surrounding ligands can be found in the thermo group database, named group.py, of RMG. More information can be found here: *Thermo Database*.

NNIs

Besides the main group-centered (GAV) contributions, non-next-nearest neighbor interactions (NNI) may also be important to take into account. NNIs are interactions between atoms separated by at least 2 atoms, such as alkane 1,4-gauche, alkane 1,5 (cf. figure), alkene 1,4-gauche, alkene single and double cis, ene-yne cis and ortho interactions.
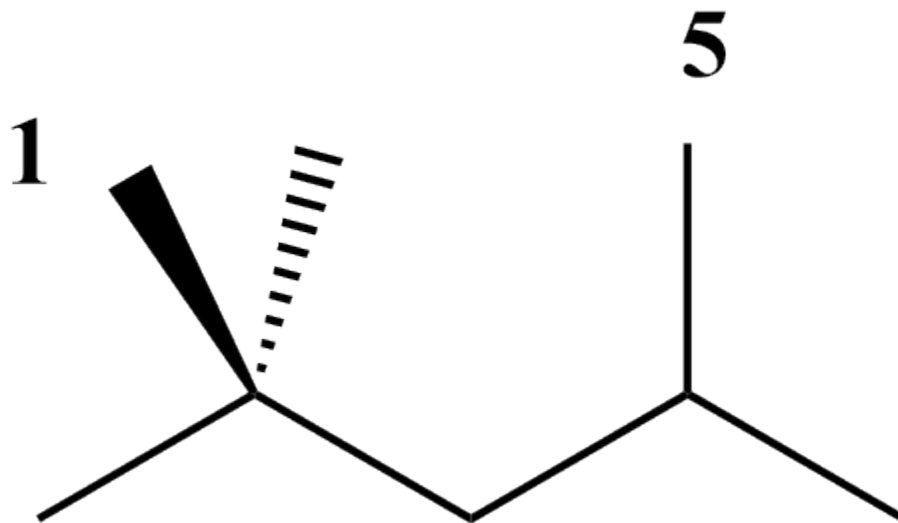
As a result, thermochemistry of the molecule X is determined as :

$$\Delta_f H^o_{298}(X) = \sum_i GAV(C_i) + \sum_j NNI_j$$

RMG contains a database with NNIs, named gauche.py and int15.py. More information on the nature on the available NNIs, and corresponding values can be found here: *Thermo Database*.

Ring Strain

To account for ring strain, ring strain corrections (RSC) were introduced. Because there is no obvious relation between the RSC and the ring structure, a specific RSC is required for every type of ring. For example, due to the significant ring strained induced in norbornane (cf. figure), a ring correction (RSC) needs to be added to the sum of the GAVs of the individual carbon atoms:
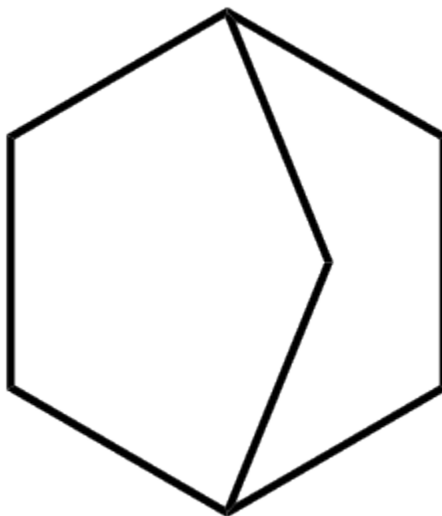
As a result, thermochemistry of the molecule X is determined as :

$$\Delta_f H^o_{298}(X) = \sum_i GAV(C_i) + RSC$$

RMG contains a database with single-ring corrections, 'ring.py' and polycyclic ring corrections, 'polycyclic.py'. More information on the nature on the available NNIs, and corresponding values can be found here: *Thermo Database*.

Hydrogen Bond Increment (HBI) method

Lay et al. [Lay] introduced the hydrogen bond increment (HBI) method to predict thermochemical properties of radicals. In contrast to Benson's method, the HBI method does not use the group-additivity concept. The HBI enthalpy of formation of a radical (R*) is calculated from the enthalpy of formation of the corresponding parent molecule (R-H) by adding a HBI to account for the loss of a hydrogen atom. Hence, for standard enthalpies of formation the HBI is defined as

$$HBI = \Delta_f H^o_{298}(R^*) - \Delta_f H^o_{298}(R - H) = BDE(R - H) - \Delta_f H^o_{298}(H*)$$

with BDE the bond dissociation enthalpy of the R-H bond at the radical position. Similar expressions are valid for the entropy and heat capacity.

As a result the thermochemistry of the radical is calculated as follows:

$$\Delta_f H^o_{298}(R^*) = HBI(\Delta_f H^o_{298}) + \Delta_f H^o_{298}(R - H)$$
$$C^o_p(R^*) = HBI(C^o_p) + C^o_p(R - H)$$
$$S^o_{298}(R^*) = HBI(S^o_{298}) + S^o_{298}(R - H)$$

The HBI method is the default method use to estimate thermochemistry of radicals. Thus, the effect of resonance stabilization on the enthalpy of the radical will be accounted for through the corresponding HBI. For example, the HBI labeled as "C=CC=CCJ" will account for the resonance present in 1,4-pentadien-3-yl radical.

The HBI method can be applied to a variety of saturated compound thermochemistry values. In RMG, library values for saturated compounds are prioritized over group additivity values for saturated compounds. Note that if QMTP is on, the QM saturated value will get priority over group additivty but library value will have priority over QM value. This ensures that there is a systematic HBI correction for values used in the final model: if the saturated molecule thermo uses a library as a source, the radical thermo applies the HBI correction to that same library value.

RMG contains a database for with HBIs, named radical.py. More information on the nature on the available HBIs, and corresponding values can be found here: *Thermo Database*.

1.13.3 On-the-fly Quantum-chemical calculation of Thermochemical Properties (QMTP)

An interface for performing on-the-fly quantum and force field calculations has been developed and integrated into RMG to complement the species thermochemistry databases and group contribution methods [Magoon and Green]_. This interface is particularly interesting for the estimation of thermochemistry of molecules that are not present in one of the species thermochemistry databases, and which cannot be estimated with sufficient accuracy using the Benson group additivity framework. This pertains specifically to polycyclic fused ring containing species, whose ring strain cannot be modeled using the available ring corrections in RMG's ring strain correction databases.

The QMTP interface involves a number of steps, summarized in the figure below.
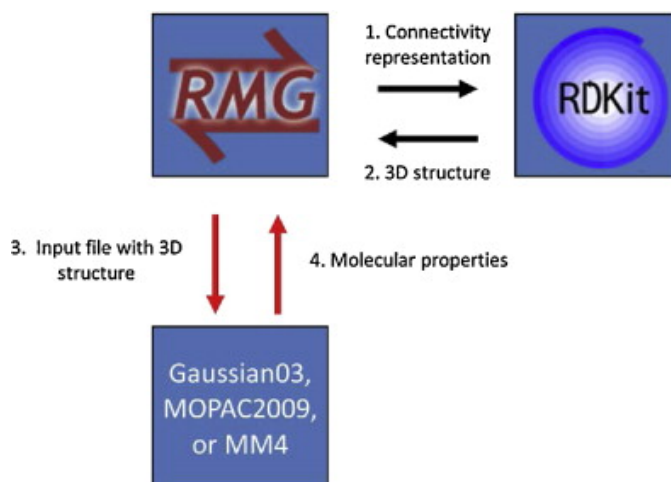
In a first step the connectivity representation is converted into a three-dimensional structure of the molecule through the generation of 3D coordinates for the atoms in the molecule. This is accomplished using a combination of a distance geometry method, followed by a optimization using the UFF force field available in RDKit [RDKit]. Next, an input file is created containing the 3D atomic coordinates along with a number of keywords. This file is sent to a computational chemistry package, either OpenMopac or Gaussian, that calculates the thermochemistry of the given molecule "on-the-fly". The keywords specify the type of calculation, and the level-of-theory. Finally, the calculated thermochemistry data is sent back to RMG.

The QMTP calculation creates a folder 'QMfiles' that contains a number of files that are created during the process. The filename of these files is a combination of the InChI key of the molecule, and a specific filename extension, e.g.

WEEGYLXZBRQIMU-UHFFFAOYSA.out is the output file produced by the QM package for the molecule cineole (SMILES: CC12CCC(CC1)C(C)(C)O2), represented by the InChI key WEEGYLXZBRQIMU-UHFFFAOYSA.

The table belows shows an overview of the used file extensions and their meaning.

File extension	Meaning
.mop	MOPAC input
.out	MOPAC output
.gjf	Gaussian input
.log	Gaussian output
.arc	MOPAC input created by MOPAC
.crude.mol	Mol file using crude, unrefined
.refined.mol	Mol file using UFF refined geometry
.symm	SYMMETRY input
.thermo	thermochemistry output file

For efficiency reasons, RMG minimizes the number of QMTP calculations. As a result, prior to initializing a QMTP routine, RMG checks whether the output files of a specific QMTP calculation are not already present in the QMfiles folder. It does so by comparing the InChI key of the given species to the filenames of the files in the QMfiles folder. If none of the InChI keys of the files correspond to the InChI key of the given species, RMG will initiate a new QMTP calculation.

Supported QM packages, and levels of theory

The following table shows an overview of the computational chemistry packages and levels of theory that are currently supported in the QMTP interface of RMG.

The MM4 force field software originates from Allinger and Lii. [Allinger].

QM Package	Supported Levels of Theory
OpenMopac	semi-empirical (PM3, PM6, PM7)
Gaussian03	semi-empirical (PM3)
MM4	molecular mechanics (MM4)

1.13.4 Symmetry and Chirality

Symmetry

The notion of symmetry is an essential part of molecules. Molecular symmetry refers to the indistinguishable orientations of a molecule and can be represented by molecular groups or a symmetry number. RMG uses a symmetry number which is the number of superimposible configurations, which includes external symmetry and internal free rotors, which is described by detail by [Benson]. This is macroscopically quantified as a decrease of the entropy S by a term $-R * ln(\sigma)$ with R the universal gas constant and σ the global symmetry number, corresponding to the number of indistinguishable orientations of the molecule.

In RMG, σ is calculated as the product of contributions of three symmetry center types : atoms, bonds and axes, cf. below.

$$\sigma = \prod_i \sigma_{atom,i} \cdot \prod_j \sigma_{bond,j} \cdot \prod_k \sigma_{axis,k}$$

More information can be found in the Ph.D Thesis of Joanna Yu [Yu].

For molecules whose thermochemistry is calculated through group contribution techniques, the rotational symmetry number is calculated through graph algorithms of RMG based on the above equation. If the thermochemistry is calculated through the QMTP process, the external, rotational symmetry number is calculatedµ using the open-source software SYMMETRY "Brute Force Symmetry Analyzer" [Patchkovskii]. This program uses the optimized three-dimensional geometry and calculates the corresponding point group.

Chirality

RMG does not take stereochemistry into account, effectively assuming a racemic mixture of mirror image enantiomers. As a result, a chirality contribution of $+R * ln(2)$ is included in the entropy of the molecule.

Chirality for molecules whose thermochemistry is determiend using group contribution techniques is detected using graph algorithms similar to those used for determining the symmetry number. If the thermochemistry is calculated through the QMTP process, chirality is detected using the point group information obtained via the software SYMMETRY.

Chiral molecules belong to point groups that lack a superposable mirror image (i.e. point groups lacking σ_h, σ_d, σ_v, and S_n symmetry elements).

In RMG, chirality is incorporated into the symmetry attribute by dividing the symmetry by two which will increase entropy by $+R * ln(2)$. RMG currently checks for each chiral center, defined by 4 different groups attached to a carbon, and halves the symmetry for each chiral center.

The effect of cis-trans isomers is currently not accounted for in RMG.

1.13.5 References

1.14 Kinetics Estimation

This section gives in-depth descriptions of algorithms used for determining kinetic parameters. For general usage of the kinetic database see *Kinetics Database*.

1.14.1 Priority of Kinetic Databases

When multiple sources are available for kinetic parameters, the following priority is followed:

1. Seed mechanisms (based on listed order in input.py)

2. Reaction libraries (based on listed order in input.py)

3. Matched training set reactions

4. Exact template matches from rules or matched training groups (based on rank)

5. Estimated averaged rules

In the case where multiple rules or training set reactions fall under the same template node, we use a user-defined rank to determine the priority of kinetic parameters

Rank	Example methods
Rank 1	Experiment/FCI
Rank 2	W4/HEAT with very good (2-d if necessary) rotors
Rank 3	CCSD(T)-F12/cc-PVnZ with n>2 or CCSD(T)-F12/CBS with good (2-d if necessary) rotors
Rank 4	CCSD(T)-F12/DZ, with good (2-d if necessary) rotors
Rank 5	CBS-QB3 with 1-d rotors
Rank 6	Double-hybrid DFT with 1-D rotors
Rank 7	Hybrid DFT (w/ dispersion) (rotors if necessary)
Rank 8	B3LYP & lower DFT (rotors if necessary)
Rank 9	Group Additivity
Rank 10	Direct Estimate/Guess
Rank 11	Average of Rates
Rank 0	General Estimate (Never used in generation)

The rank of 0 is assigned to kinetics that are generally default values for top level nodes that we have little faith in. It is never used in generation and its value will in fact be overriden by averages of its child nodes, which generates an averaged rate rule with rank 11.

Only non-zero rules are used in generation. A rank of 1 is assigned to the most trustworthy kinetics, while a rank of 10 is considered very poor. Thus, a rate rule of rank 3 will be given priority over a rate rule of rank 5.

Short Glossary:

FCI (Full Configuration Interaction): Exact solution to Schrodinger equation within the chosen basis set and Born-Oppenheimer approximation; possible for about 12 electrons with reasonably sized basis set (cost grows factorially with number of electrons).

Wn (Weizmann-n): Composite methods often with sub-kJ/mol accuracies; W1 is possible for about 9 heavy atoms; W1 aims to reproduce CCSD(T)/CBS; W4 aims to reproduce CCSDTQ5/CBS.

HEAT (High Accuracy Extrapolated ab inito thermochemistry): Sub-kJ/mol accuracies; essentially CCSDTQ with various corrections; similar in cost to Wn.

CBS (Complete Basis Set): Typically obtained by extrapolating to the complete basis set limit, i.e., successive cc-pVDZ, cc-pVTZ, cc-pVQZ, etc. calculations with some extrapolation formula.

CCSD(T)-F12: Coupled cluster with explicit electron correlation; chemical accuracy (1 kcal/mol) possible with double-zeta basis sets.

1.14.2 Kinetic Families

To show the algorithm used by kinetic families, the following H-abstraction will be used an example

First the reacting atoms will be identified. Then, the family's trees will be descended as far as possible to give the reaction's groups.
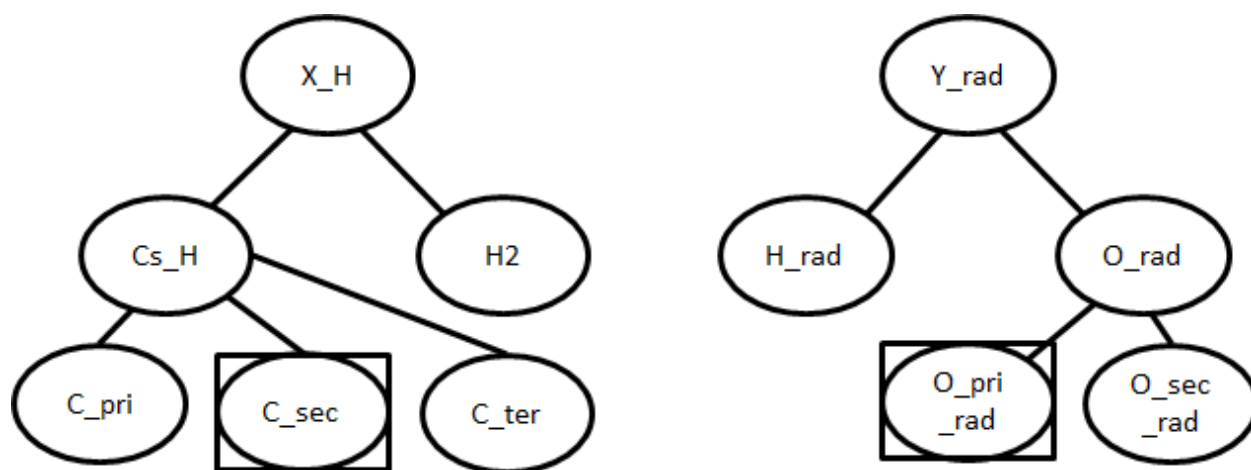
Using the sample tree shown above, the desired template is (C_sec, O_pri_rad). The algorithm will then search the database for parameters for the template. If they are present, an exact match will be returned using the kinetics of that template. Note that an exact match refers to the nodes (C_sec, O_pri_rad) and not the molecules (propane, OH).

There may not be an entry for (C_sec, O_pri_rad) in the database. In that case, the rule will attempt to "fall up" to more general nodes:
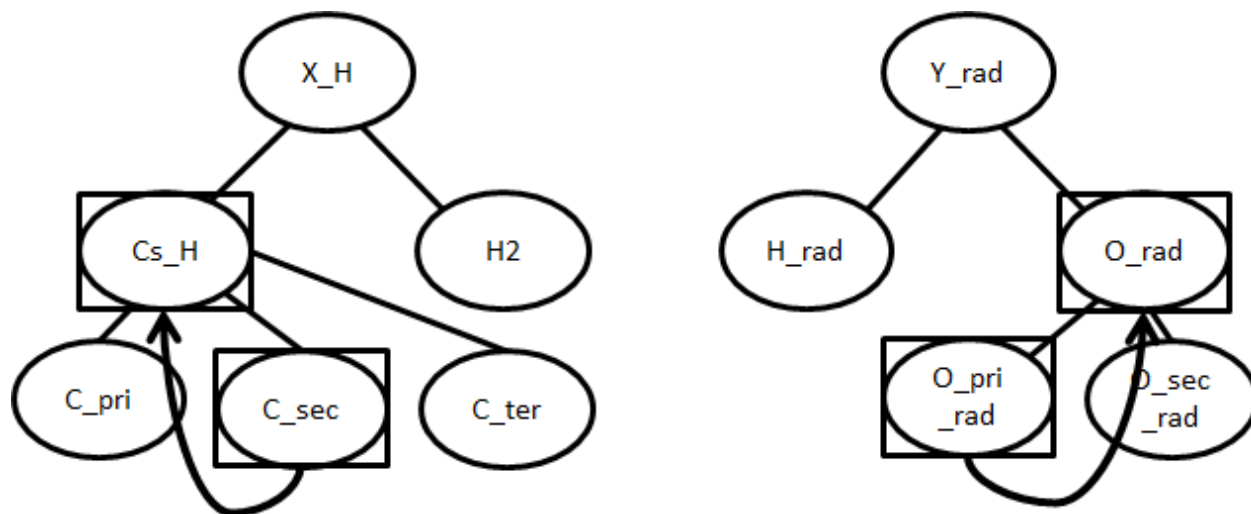
Now the preferred rule is (Cs_H, O_rad). If database contains parameters for this, those will be returned as an estimated match.

If there is still no kinetics for the template, the entire set of children for Cs_H and O_rad will be checked. For this example, this set would include every combination of {C_pri, C_sec, C_ter} with {O_pri_rad, O_sec_rad}. If any these templates have kinetics, an average of their parameters will be returned as an estimated match. The average for A is a geometric mean, while the average for n, E_a, and α are arithmetic means.

If there are still no "sibling" kinetics, then the groups will continue to fall up to more and more general nodes. In the worst case, the root nodes may be used.

A *Full List of the Kinetics Families* in RMG is available.

1.15 Liquid Phase Systems

To simulate liquids in RMG requires a module in your input file for liquid-phase:

```
solvation(
    solvent='octane'
)
```

Your reaction system will also be different (liquidReactor rather than simpleReactor):

```
liquidReactor(
    temperature=(500,'K'),
    initialConcentrations={
        "octane": (6.154e-3,'mol/cm^3'),
        "oxygen": (4.953e-6,'mol/cm^3')
    },
    terminationTime=(5,'s'),
    constantSpecies=['oxygen'],
    sensitivity=['octane','oxygen'],
    sensitivityThreshold=0.001,

)
```

To simulate the liquidReactor, one of the initial species / concentrations must be the solvent. If the solvent species does not appear as the initial species, RMG run will stop and raise error. The solvent can be either reactive, or nonreactive.

In order for RMG to recognize the species as the solvent, it is important to use the latest version of the RMG-database, whose solvent library contains solvent SMILES. If the latest database is used, RMG can determine whether the species is the solvent by looking at its molecular structure (SMILES or adjacency list). If the old version of RMG-database without the solvent SMILES is used, then RMG can recognize the species as the solvent only by its string name. This means that if the solvent is named "octane" in the solvation block and it is named "n-octane" in the species and initialConcentrations blocks, RMG will not be able to recognize them as the same solvent species and raise error because the solvent is not listed as one of the initial species.

For liquid phase generation, you can provide a list of species for which one concentration is held constant over time (Use the keyword `constantSpecies=[]` with species labels separated by `,`). To generate meaningful liquid phase oxidation mechanism, it is highly recommended to consider O2 as a constant species. To consider pyrolysis cases, it is still possible to obtain a mechanism without this option. Expected results with `Constant concentration` option can be summarized with those 3 cases respectively presenting a generation with 0, 1 (oxygen only) and 2 constant species (oxygen and decane):

output_0SPC_constant.png   output_1SPC_constant.png   output_2SPC_constant.png
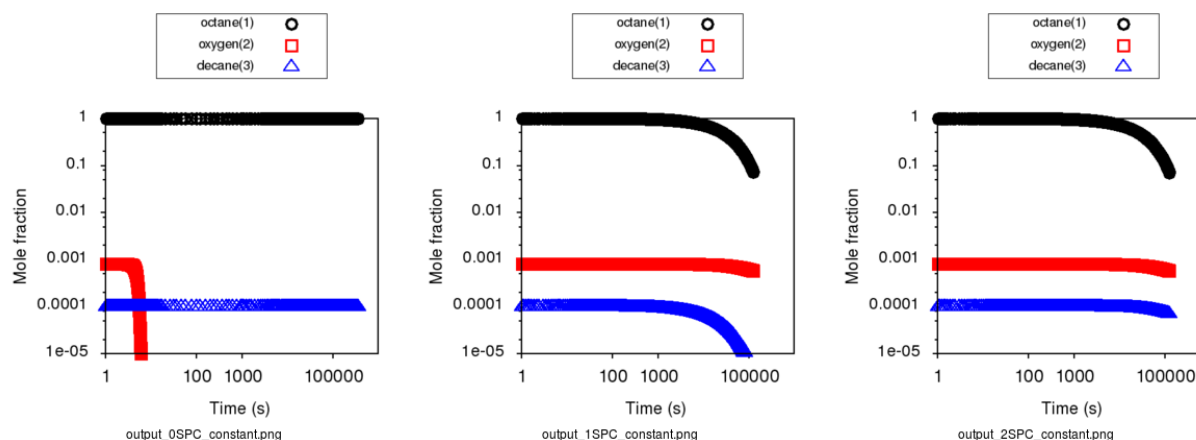
As it creates a mass lost, it is recommended to avoid to put any products as a constant species.

For sensitivity analysis, RMG-Py must be compiled with the DASPK solver, which is done by default but has some dependency restrictions. (See *License Restrictions on Dependencies* for more details.) Like for the simpleReactor, the sensitivity and sensitivityThrehold are optional arguments for when the user would like to conduct sensitivity analysis with respect to the reaction rate coefficients for the list of species given for `sensitivity`.

Sensitivity analysis is conducted for the list of species given for `sensitivity` argument in the input file. The normalized concentration sensitivities with respect to the reaction rate coefficients $dln(C_i)/dln(k_j)$ are saved to a csv file with the file name `sensitivity_1_SPC_1.csv` with the first index value indicating the reactor system and the second naming the index of the species the sensitivity analysis is conducted for. Sensitivities to thermo of individual species is also saved as semi normalized sensitivities $dln(C_i)/d(G_j)$ where the units are given in 1/(kcal mol-1). The sensitivityThreshold is set to some value so that only sensitivities for $dln(C_i)/dln(k_j)$ > sensitivityThreshold or $dlnC_i/d(G_j)$ > sensitivityThreshold are saved to this file.

Note that in the RMG job, after the model has been generated to completion, sensitivity analysis will be conducted in one final simulation (sensitivity is not performed in intermediate iterations of the job).

Notes: sensitivity, sensitivityThreshold and constantSpecies are optionnal keywords.

1.15.1 Equation of state

Specifying a liquidReactor will have two effects:

1. disable the ideal gas law renormalization and instead rely on the concentrations you specified in the input file to initialize the system.

2. prevent the volume from changing when there is a net stoichiometry change due to a chemical reaction (A = B + C).

1.15.2 Solvation thermochemistry

The next correction for liquids is solvation effects on the thermochemistry. By specifying a solvent in the input file, we load the solvent parameters to use.

The free energy change associated with the process of transferring a molecule from the gas phase to the solvent phase is defined as the free energy of solvation (ΔG). Many different methods have been developed for computing solvation energies among which continuum dielectric and force field based methods are popular. Not all of these methods are easy to automate, and many are not robust i.e. they either fail or give unreasonable results for certain solute-solvent pairs. CPU time and memory (RAM) requirements are also important considerations. A fairly accurate and fast method for computing ΔG, which is used in RMG, is the LSER approach described below.

Use of thermo libraries in liquid phase system

As it is for gas phase simulation, thermo libraries listed in the input files are checked first to find thermo for a given species and return the first match. As it exists two types of thermo libraries, (more details on *thermo libraries*), thermo of species matching a library in a liquid phase simulation is obtained following those two cases:

If library is a "liquid thermo library", thermo data are directly used without applying solvation on it.

If library is a "gas thermo library", thermo data are extracted and then corrections are applied on it using the *LSER method* for this specific species-solvent system.

Note: Gas phase libraries can be declared first, liquid thermo libraries will still be tested first but the order will be respected if several liquid libraries are provided.

Use of Abraham LSER to estimate thermochemistry

The Abraham LSER provides an estimate of the the partition coefficient (more specifically, the log (base 10) of the partition coefficient) of a solute between the vapor phase and a particular solvent (K_{vs}) (also known as gas-solvent partition coefficient) at 298 K:

$$\log K_{vs} = c + eE + sS + aA + bB + lL \tag{1.1}$$

The Abraham model is used in RMG to estimate ΔG which is related to the K_{vs} of a solute according to the following expression:

$$\begin{aligned} DeltaG &= -RT \ln K_{vs} \\ &= -2.303 RT \log K_{vs} \end{aligned} \tag{1.2}$$

The variables in the Abraham model represent solute (*E, S, A, B, V, L*) and solvent descriptors (*c, e, s, a, b, v, l*) for different interactions. The *sS* term is attributed to electrostatic interactions between the solute and the solvent (dipole-dipole interactions related to solvent dipolarity and the dipole-induced dipole interactions related to the polarizability of the solvent) [Vitha2006], [Abraham1999], [Jalan2010]. The *lL* term accounts for the contribution from cavity formation and dispersion (dispersion interactions are known to scale with solute volume [Vitha2006], [Abraham1999]. The *eE* term, like the *sS* term, accounts for residual contributions from dipolarity/polarizability related interactions for solutes whose blend of dipolarity/polarizability differs from that implicitly built into the *S* parameter [Vitha2006], [Abraham1999], [Jalan2010]. The *aA* and *bB* terms account for the contribution of hydrogen bonding between the solute and the surrounding solvent molecules. H-bonding interactions require two terms as the solute (or solvent) can act as acceptor (donor) and vice versa. The descriptor *A* is a measure of the solute's ability to donate a hydrogen bond (acidity) and the solvent descriptor *a* is a measure of the solvent's ability to accept a hydrogen bond. A similar explanation applies to the *bB* term [Vitha2006], [Abraham1999], [Poole2009].

The solvent descriptors (*c, e, s, a, b, l*) are largely treated as regressed empirical coefficients. Parameters are provided in RMG's database for the following solvents:

1. acetonitrile
2. benzene
3. butanol
4. carbontet
5. chloroform
6. cyclohexane

7. decane

8. dibutylether

9. dichloroethane

10. dimethylformamide

11. dimethylsulfoxide

12. dodecane

13. ethanol

14. ethylacetate

15. heptane

16. hexadecane

17. hexane

18. isooctane

19. nonane

20. octane

21. octanol

22. pentane

23. toluene

24. undecane

25. water

Group additivity method for solute descriptor estimation

Group additivity is a convenient way of estimating the thermochemistry for thousands of species sampled in a typical mechanism generation job. Use of the Abraham Model in RMG requires a similar approach to estimate the solute descriptors (*A, B, E, L,* and *S*). Platts et al. ([Platts1999]) proposed such a scheme employing a set of 81 molecular fragments for estimating *B, E, L, V* and *S* and another set of 51 fragments for the estimation of *A*. Only those fragments containing C, H and O are implemented in order to match RMG's existing capabilities. The value of a given descriptor for a molecule is obtained by summing the contributions from each fragment found in the molecule and the intercept associated with that descriptor.

Mintz model for enthalpy of solvation

For estimating ΔG at temperatures other than 298 K, the enthalpy change associated with solvation, ΔH must be calculated separately and, along with ΔS, assumed to be independent of temperature. Recently, Mintz et al. ([Mintz2007], [Mintz2007a], [Mintz2007b], [Mintz2007c], [Mintz2007d], [Mintz2008], [Mintz2008a], [Mintz2009]) have developed linear correlations similar to the Abraham model for estimating ΔH:

$$DeltaH(298K) = c' + a'A + b'B + e'E + s'S + l'L \tag{1.3}$$

where *A, B, E, S* and *L* are the same solute descriptors used in the Abraham model for the estimation of ΔG. The lowercase coefficients *c', a', b', e', s'* and *l'* depend only on the solvent and were obtained by fitting to experimental data. In RMG, this equation is implemented and together with ΔG(298 K) can be used to find ΔS(298 K). From this data, ΔG at other temperatures is found by extrapolation.

1.15.3 Diffusion-limited kinetics

The next correction for liquid-phase reactions is to ensure that bimolecular reactions do not exceed their diffusion limits. The theory behind diffusive limits in the solution phase for bimolecular reactions is well established ([Rice1985]) and has been extended to reactions of any order ([Flegg2016]). The effective rate constant of a diffusion-limited reaction is given by:

$$k_{\text{eff}} = \frac{k_{\text{diff}} k_{\text{int}}}{k_{\text{diff}} + k_{\text{int}}} \tag{1.4}$$

where k_{int} is the intrinsic reaction rate, and k_{diff} is the diffusion-limited rate, which is given by:

$$k_{\text{diff}} = \left[\prod_{i=2}^{N} \hat{D}_i^{3/2} \right] \frac{4\pi^{\alpha+1}}{\Gamma(\alpha)} \left(\frac{\sigma}{\sqrt{\Delta_N}} \right)^{2\alpha} \tag{1.5}$$

where $\alpha=(3N-5)/2$ and

$$\hat{D}_i = D_i + \frac{1}{\sum_m^{i-1} D_m^{-1}} \tag{1.6}$$

$$\Delta_N = \frac{\sum_{i=1}^{N} D_i^{-1}}{\sum_{i>m} (D_i D_m)^{-1}} \tag{1.7}$$

D_i are the individual diffusivities and σ is the Smoluchowski radius, which would usually be fitted to experiment, but RMG approximates it as the sum of molecular radii. RMG uses the McGowan method for estimating radii, and diffusivities are estimated with the Stokes-Einstein equation using experimental solvent viscosities (*eta* (T)). In a unimolecular to bimolecular reaction, for example, the forward rate constant (k_f) can be slowed down if the reverse rate ($k_{\text{r, eff}}$) is diffusion-limited since the equilibrium constant (K_{eq}) is not affected by diffusion limitations. In cases where both the forward and the reverse reaction rates are multimolecular, the forward rate coefficients limited in the forward and reverse directions are calculated and the limit with the smaller forward rate coefficient is used.

The viscosity of the solvent is calculated Pa.s using the solvent specified in the command line and a correlation for the viscosity using parameters *A, B, C, D, E*:

$$\ln \eta = A + \frac{B}{T} + C \log T + DT^E \tag{1.8}$$

To build accurate models of liquid phase chemical reactions you will also want to modify your kinetics libraries or correct gas-phase rates for intrinsic barrier solvation corrections (coming soon).

1.15.4 Example liquid-phase input file, no constant species

This is an example of an input file for a liquid-phase system:

```
# Data sources
database(
    thermoLibraries = ['primaryThermoLibrary'],
    reactionLibraries = [],
    seedMechanisms = [],
    kineticsDepositories = ['training'],
    kineticsFamilies = 'default',
    kineticsEstimator = 'rate rules',
)

# List of species
species(
```

```python
    label='octane',
    reactive=True,
    structure=SMILES("C(CCCCC)CC"),
)

species(
    label='oxygen',
    reactive=True,
    structure=SMILES("[O][O]"),
)

# Reaction systems
liquidReactor(
    temperature=(500,'K'),
    initialConcentrations={
        "octane": (6.154e-3,'mol/cm^3'),
        "oxygen": (4.953e-6,'mol/cm^3')
    },
    terminationTime=(5,'s'),
)

solvation(
    solvent='octane'
)

simulator(
    atol=1e-16,
    rtol=1e-8,
)

model(
    toleranceKeepInEdge=1E-9,
    toleranceMoveToCore=0.01,
    toleranceInterruptSimulation=0.1,
    maximumEdgeSpecies=100000
)

options(
    units='si',
    saveRestartPeriod=None,
    generateOutputHTML=False,
    generatePlots=False,
    saveSimulationProfiles=True,
)
```

1.15.5 Example liquid-phase input file, with constant species

This is an example of an input file for a liquid-phase system with constant species:

```python
# Data sources
database(
    thermoLibraries = ['primaryThermoLibrary'],
    reactionLibraries = [],
    seedMechanisms = [],
    kineticsDepositories = ['training'],
```

```python
    kineticsFamilies = 'default',
    kineticsEstimator = 'rate rules',
)

# List of species
species(
    label='octane',
    reactive=True,
    structure=SMILES("C(CCCCC)CC"),
)

species(
    label='oxygen',
    reactive=True,
    structure=SMILES("[O][O]"),
)

# Reaction systems
liquidReactor(
    temperature=(500,'K'),
    initialConcentrations={
        "octane": (6.154e-3,'mol/cm^3'),
        "oxygen": (4.953e-6,'mol/cm^3')
    },
    terminationTime=(5,'s'),
    constantSpecies=['oxygen'],
)

solvation(
    solvent='octane'
)

simulator(
    atol=1e-16,
    rtol=1e-8,
)

model(
    toleranceKeepInEdge=1E-9,
    toleranceMoveToCore=0.01,
    toleranceInterruptSimulation=0.1,
    maximumEdgeSpecies=100000
)

options(
    units='si',
    saveRestartPeriod=None,
    generateOutputHTML=False,
    generatePlots=False,
    saveSimulationProfiles=True,
)
```

1.16 Frequently Asked Questions

For any other questions related to RMG and its usage and installation, please post an issue at https://github.com/ReactionMechanismGenerator/RMG-Py/issues and the RMG developers will get back to you as soon as we can. You can also search for your problem on the issues page to see if there are already solutions in development. Alternatively, you can email us at rmg_dev@mit.edu.

1.16.1 Why can't my adjacency lists be read any more?

The adjacency list syntax changed in July 2014. The minimal requirement for most translations is to prefix the number of unpaired electrons with the letter *u*.

Example old syntax:

```
HXD13
1    C 0       {2,D}
2    C 0 {1,D} {3,S}
3    C 0 {2,S} {4,D}
4    C 0 {3,D} {5,S}
5 *1 C 0 {4,S} {6,S}
6 *2 C 0 {5,S}
```

Example new syntax:

```
HXD13
1    C u0       {2,D}
2    C u0 {1,D} {3,S}
3    C u0 {2,S} {4,D}
4    C u0 {3,D} {5,S}
5 *1 C u0 {4,S} {6,S}
6 *2 C u0 {5,S}
```

The new syntax, however, allows much greater flexibility, including definition of lone pairs, partial charges, wildcards, and molecule multiplicities, and was necessary to allow us to add Nitrogen chemistry. See rmgpy.molecule.adjlist for details of the new syntax.

1.17 Release Notes

1.17.1 RMG-Py Version 2.3.0

Date: Dec 20, 2018

- **Arkane (formerly CanTherm):**

    - CanTherm had been renamed to Arkane (Automated Reaction Kinetics And Network Exploration)

    - New network exploration functionality using RMG-database

    - Support for all elements has been added for reading quantum output files

    - New supporting information output file with rotational constants and frequencies

    - Known thermo and kinetics can be provided in addition to quantum information

    - Improve general user experience and error handling

- **New machine learning thermo estimator**

- Estimate species thermochemistry using a graph convolutional neural network

- Estimator trained on quantum calculations at B3LYP and CCSD(T)-F12 levels

- Currently supports C/H/O/N, with an emphasis on cyclic molecules

- **Resonance:**

  - New pathways added for lone-pair multiple-bond resonance, replacing two pathways which were more specific

  - New pathways added for aryne resonance

  - Aromatic resonance pathways simplified and refactored to use filtration

  - Kekule structures are now considered unreactive structures

- **Miscellaneous changes:**

  - Isotope support added for reading and writing InChI strings

  - New branching algorithm for picking up feedback loops implemented (beta)

  - Global forbidden structure checking is now only done for core species for efficiency, which may lead to forbidden species existing in the edge

  - Minor improvements to symmetry algorithm to fix a few incorrect cases

- **Bug fixes:**

  - Fixed issue where react flags were being reset when filterReactions was used with multiple reactors, resulting in no reactions generated

  - File paths for collision violators log changed to output directory

  - Fixed bug in local uncertainty introduced by ranged reactor changes

  - Fixed bug with diffusion limitation calculations for multi-molecular reactions

  - Various other minor fixes

1.17.2 RMG-database Version 2.3.0

Date: Dec 20, 2018

- **Kinetics rules to training reactions**

  - All kinetics rules have been converted into training reactions by converting each group to the smallest molecule that matches it

  - Training reactions are preferred over rules because they correspond to a specific reaction and are therefore easier to update

  - This conversion is in anticipation of upcoming changes to trees in kinetics families

- **Additions:**

  - R_Addition_MultipleBond training reactions

  - intra_NO2_ONO_conversion training reactions

  - SABIC_aromatics thermo library (CBS-QB3, RRHO)

  - McGowan volumes for noble gases

  - More entries added to Lai_Hexylbenzene libraries

  - Architecture and weights for neural network thermo estimator

1.17.3 RMG-Py Version 2.2.1

Date July 23, 2018

This release is minor patch which fixes a number of issues discovered after 2.2.0.

- **Collision limit checking:**

    - RMG will now output a list of collision limit violations for the generated model

- **Fixes:**

    - Ambiguous chemical formulas in SMILES lookup leading to incorrect SMILES generation

    - Fixed issue with reading geometries from QChem output files

    - React flags for reaction filter were not properly updated on each iteration

    - Fixed issue with inconsistent symmetry number calculation

1.17.4 RMG-Py Version 2.2.0

Date: July 5, 2018

- **New features:**

    - New ring membership attribute added to atoms. Can be specified in group adjacency lists in order to enforce ring membership of atoms during subgraph matching.

    - Reactors now support specification of T, P, X ranges. Different conditions are sampled on each iteration to optimally capture the full parameter space.

    - New termination type! Termination rate ratio stops the simulation when the characteristic rate falls to the specified fraction of the maximum characteristic rate. Currently not recommended for systems with two-stage ignition.

    - New resonance transitions implemented for species with lone pairs (particularly N and S containing species). A filtration algorithm was also added to select only the most representative structures.

    - Formal support for trimolecular reaction families.

    - New isotopes module allows post-processing of RMG mechanisms to generate a mechanism with isotopic labeling.

- **Changes:**

    - Library reactions can now be integrated into RMG pdep networks if the new elementary_high_p attribute is True

    - Library reactions may be duplicated by pdep reactions if the new allow_pdep_route attribute is True

    - Jupyter notebook for adding new training reactions has been revamped and is now located at ipython/kinetics_library_to_training.ipynb

    - Syntax for recommended families has changed to set notation instead of dictionaries, old style still compatible

    - Ranking system for database entries expanded to new 0-11 system from the old 0-5 system

    - Collision limit checking has been added for database entries

- **Cantherm:**

    - Improved support for MolPro output files

- – Added iodine support

- – Automatically read spin multiplicity from quantum output

- – Automatically assign frequency scale factor for supported model chemistries

- – Plot calculated rates and thermo by default

- – New sensitivity analysis feature analyzes sensitivity of reaction rates to isomer/TS energies in pdep networks

- **Fixes:**

  - – Properly update charges when creating product templates in reaction families

  - – Excessive duplicate reactions from different resonance structures has been fixed (bug introduced in 2.1.3)

  - – Fixed rate calculation for MultiPdepArrhenius objects when member rates have different plists

- **A more formal deprecation process is now being trialed. Deprecation warnings have been added to functions to be removed**

  - – All methods related to saving or reading RMG-Java databases and old-style adjacency lists

  - – The group additivity method for kinetics estimation (unrelated to thermo group additivity)

  - – The saveRestartPeriod option and the old method of saving restart files

1.17.5 RMG-database Version 2.2.0

Date: July 5, 2018

- **Additions:**

  - – New Intra_R_Add_Exo_Scission reaction family

  - – New 1,2_ShiftC reaction family

  - – **New reaction families for peroxide chemistry in liquid systems**

    - ∗ Korcek_step1_cat

    - ∗ Bimolec_Hydroperoxide_Decomposition

    - ∗ Peroxyl_Termination

    - ∗ Peroxyl_Disproportionation

    - ∗ Baeyer-Villiger_step1_cat

    - ∗ Baeyer-Villiger_step2

    - ∗ Baeyer-Villiger_step2_cat

  - – Numerous new training reactions added to many families

- **Changes:**

  - – New tree structure for Intra_R_Add_Endocyclic with consideration for cyclic species

  - – Multiple bond on ring is no longer allowed in Intra_R_Add_Exocyclic and should react in Intra_R_Add_Endocyclic instead

  - – Entry ranks rescaled to new 0-11 ranking system

  - – Global forbidden structures has been cleaned up, leading to significant performance improvement

- **Fixes:**

    - Corrected shape indices in NOx2018 transport library

    - Removed or corrected some kinetics entries based on collision limit check

1.17.6 RMG-Py Version 2.1.9

Date: May 1, 2018

- **Cantherm:**

    - Atom counts are no longer necessary in input files and are automatically determined from geometries

    - Custom atom energies can now be specified in input files

    - Removed atom energies for a few ambiguous model chemistries

    - Add atom energies for B3LYP/6-311+g(3df,2p)

- **Changes:**

    - Refactored molecule.parser and molecule.generator modules into molecule.converter and molecule.translator to improve code organization

    - SMILES generation now outputs canonical SMILES

    - Molecule.sortAtoms method restored for deterministic atom order

    - PDep reactions which match an existing library reaction are no longer added to the model

- **Fixes:**

    - Fix issue with reaction filter initiation when using seed mechanisms

1.17.7 RMG-database Version 2.1.9

Date: May 1, 2018

- **Chlorine:**

    - New Chlorinated_Hydrocarbons thermo library

    - Added group additivity values and long distance corrections for chlorinated species

    - Added chlorine groups and training reactions to H_Abstraction

- **Additions:**

    - New NOx2018 kinetics, thermo, and transport libraries

    - New N-S_interactions kinetics library

    - New SulfurHaynes thermo library

    - Added species to SOxNOx thermo library from quantum calculations

- **Other changes:**

    - Renamed NOx and SOx kinetics libraries to PrimaryNitrogenLibrary and PrimarySulfurLibrary

    - S2O2, SOO2, SO2O2, and N2SH were globally forbidden due to inability to optimize geometries

- **Fixes:**

    - Corrected some A-factor units in Nitrogen_Dean_and_Bozzelli kinetics library

1.17.8 RMG-Py Version 2.1.8

Date: March 22, 2018

- **New features:**

    - Chlorine and iodine atom types have been added, bringing support for these elements to RMG-database

    - Forbidden structures now support Molecule and Species definitions in addition to Group definitions

- **Changes:**

    - Reaction pair generation will now fall back to generic method instead of raising an exception

    - Removed sensitivity.py script since it was effectively a duplicate of simulate.py

    - Thermo jobs in Cantherm now output a species dictionary

    - Fitted atom energy corrections added for B3LYP/6-31g**

    - Initial framework added for hydrogen bonding

    - Renamed molepro module and associated classes to molpro (MolPro) to match actual spelling of the program

    - Chemkin module is now cythonized to improve performance

- **Fixes:**

    - Allow delocalization of triradicals to prevent hysteresis in resonance structure generation

    - Fix reaction comment parsing issue with uncertainty analysis

    - Fix numerical issue causing a number of pressure dependent RMG jobs to crash

    - Template reactions from seed mechanisms are now loaded as library reactions if the original family is not loaded

    - Fix issues with degeneracy calculation for identical reactants

1.17.9 RMG-database Version 2.1.8

Date: March 22, 2018

- **Changes:**

    - Corrected name of JetSurf2.0 kinetics and thermo libraries to JetSurf1.0

    - Added actual JetSurf2.0 kinetics and thermo libraries

    - Updated thermo groups for near-aromatic radicals, including radical and polycyclic corrections

1.17.10 RMG-Py Version 2.1.7

Date: February 12, 2018

- **Charged atom types:**

    - Atom types now have a charge attribute to cover a wider range of species

    - New atom types added for nitrogen and sulfur groups

    - Carbon and oxygen atom types renamed following new valence based naming scheme

- **Ring perception:**

    - Ring perception methods in the Graph class now use RingDecomposerLib

    - This includes the getSmallestSetOfSmallestRings methods and a newly added getRelevantCycles method

    - The set of relevant cycles is unique and generally more useful for chemical graphs

    - This also fixes inaccuracies with the original SSSR method

- **Other changes:**

    - Automatically load reaction libraries when using a seed mechanism

    - Default kinetics estimator has been changed to rate rules instead of group additivity

    - Kinetics families can now be set to be irreversible

    - Model enlargement now occurs after each reactor simulation rather than after all of them

    - Updated bond additivity corrections for CBS-QB3 in Cantherm

- **Fixes:**

    - Do not print SMILES when raising AtomTypeError to avoid further exceptions

    - Do not recalculate thermo if a species already has it

    - Fixes to parsing of family names in seed mechanisms

1.17.11 RMG-database Version 2.1.7

Date: February 12, 2018

- **Charged atom types:**

    - Update adjlists with new atom types across the entire database

    - Added sulfur groups to all relevant kinetics families

    - New thermo group additivity values for sulfur/oxygen species

- **Additions:**

    - Benzene bonds can now react in in R_Addition_MultipleBond

    - Many new training reactions and groups added in R_Addition_MultipleBond

    - New Singlet_Val6_to_triplet kinetics family

    - New Sulfur GlarborgBozzelli kinetics and thermo libraries

    - New Sulfur GlarborgMarshall kinetics and thermo libraries

    - New Sulfur GlarborgH2S kinetics and thermo libraries

    - New Sulfur GlarborgNS kinetics and thermo libraries

    - New NOx and NOx/LowT kinetics libraries

    - New SOx kinetics library

    - New BurcatNS thermo library

    - New SOxNOx thermo library

    - New 2+2_cycloaddition_CS kinetics family

- New Cyclic_Thioether_Formation kinetics family

- New Lai_Hexylbenzene kinetics and thermo libraries

- **Changes:**

    - 1,2-Birad_to_alkene family is now irreversible

    - OxygenSingTrip kinetics library removed (replaced by Singlet_Val6_to_triplet family)

    - Ozone is no longer forbidden

- **Fixes:**

    - Corrected adjlist for phenyl radical in JetSurf2.0 and USC-Mech-ii

    - Some singlet thermo groups relocated from radical.py to group.py

1.17.12 RMG-Py Version 2.1.6

Date: December 21, 2017

- **Model resurrection:**

    - Automatically attempts to save simulation after encountering a DASPK error

    - Adds species and reactions in order to modify model dynamics and fix the error

- **New features:**

    - Add functionality to read RCCSD(T)-F12 energies from MolPro log files

    - Add liquidReactor support to flux diagram generation

- **Other changes:**

    - Removed rmgpy.rmg.model.Species class and merged functionality into main rmgpy.species.Species class

    - Refactored parsing of RMG-generated kinetics comments from Chemkin files and fixed related issues

    - Refactored framework for generating reactions to reduce code duplication

    - Resonance methods renamed from generateResonanceIsomers to generate_resonance_structures across all modules

    - Raise CpInf to Cphigh for entropy calculations to prevent invalid results

- **Fixes:**

    - Update sensitivity analysis to use ModelSettings and SimulatorSettings classes introduced in v2.1.5

    - Fixed generate_reactions methods in KineticsDatabase to be directly usable again

    - Fixed issues with aromaticity perception and generation of aromatic resonance structures

1.17.13 RMG-database Version 2.1.6

Date: December 21, 2017

- **Additions:**

    - New training reactions added for [NH2] related H_Abstractions

    - 14 new kinetics libraries related to aromatics formation (see RMG-database #222 for details)

- **Other changes:**
    - Removed some global forbidden groups which are no longer needed
    - Forbid CO and CS biradicals
    - Updated lone_electron_pair_bond family and removed from recommended list

- **Fixes:**
    - Fixed unit errors in some H_Abstraction and R_Addition_MultipleBond depositories

1.17.14 RMG-Py Version 2.1.5

Date: October 18, 2017

- **New bicyclic formula:**
    - Estimates polycyclic corrections for unsaturated bicyclics by adjusting the correction for the saturated version
    - Can provide a decent estimate in many cases where there is not an exact match

- **Other changes:**
    - Refactored simulation algorithm to properly add multiple objects per iteration
    - Print equilibrium constant and reverse rate coefficient values when using Cantherm to calculate kinetics
    - Speed up degeneracy calculation by reducing unnecessary operations

- **Fixes:**
    - Loosen tolerance for bond order identification to account for floating point error
    - Fixed uncertainty analysis to allow floats as bond orders
    - Fixed some comment parsing issues in uncertainty analysis
    - Added product structure atom relabeling for families added in RMG-database v2.1.5
    - Fixed issue with automatic debugging of kinetics errors due to forbidden structures

1.17.15 RMG-database Version 2.1.5

Date: October 18, 2017

- **Additions:**
    - New thermo groups added for species relevant in cyclopentadiene and natural gas pyrolysis
    - Added C2H4+O_Klipp2017 kinetics library

- **Fixes:**
    - Prevent charged carbenes from reacting in Singlet_Carbene_Intra_Disproportionation
    - Updated H_Abstraction rates in ethylamine library and corresponding training reactions

1.17.16 RMG-Py Version 2.1.4

Date: September 08, 2017

- **Accelerator tools:**

    - Dynamics criterion provides another method to expand the mechanism by adding reactions to the core

    - Surface algorithm enables better control of species movement to the core when using the dynamics criterion

    - Multiple sets of model parameters can now be specified in a input file to allow different stages of model generation

    - A species number termination criterion can now be set to limit model size

    - Multiple items can now be added per iteration to speed up model construction

    - New ModelSettings and SimulatorSettings classes for storing input parameters

- **New features:**

    - Kinetics libraries can now be automatically generated during RMG runs to be used as seeds for subsequent runs

    - Loading automatically generated seed mechanisms recreates the original template reaction objects to allow restarting runs from the seed mechanism

    - Carbene constraints can now be set in the species constraint block using maxSingletCarbenes and maxCarbeneRadicals

    - Chirality is now considered for determining symmetry numbers

    - Thermodynamic pruning has been added to allow removal of edge species with unfavorable free energy (beta)

- **Other changes:**

    - RMG-Py exception classes have been consolidated in the rmgpy.exceptions module

    - Species labels will now inherit the label from a matched thermo library entry

    - Sensitivity analysis is now available for LiquidReactor

- **Fixes:**

    - Fixed sensitivity analysis following changes to the simulate method

    - Add memory handling when generating collision matrix for pressure dependence

    - Improved error checking for MOPAC

    - Prevent infinite loops when retrieving thermo groups

- **Known issues:**

    - Seed mechanisms cannot be loaded if the database settings are different from the original ones used to generate the seed

1.17.17 RMG-database Version 2.1.4

Date: September 08, 2017

- **New kinetics families for propargyl recombination route to benzene:**

    - Singlet_Carbene_Intra_Disproportionation

- Intra_5_membered_conjugated_C=C_C=C_addition

- Intra_Diels_alder_monocyclic

- Concerted_Intra_Diels_alder_monocyclic_1,2_shift

- Intra_2+2_cycloaddition_Cd

- Cyclopentadiene_scission

- 6_membered_central_C-C_shift

- **Renamed kinetics families:**

    - Intra_Diels_Alder –> Intra_Retro_Diels_alder_bicyclic

    - H_shift_cyclopentadiene –> Intra_ene_reaction

- **Other additions:**

    - Klippenstein_Glarborg2016 kinetics and thermo libraries

    - Group additivity values added for singlet carbenes, which are no longer forbidden

1.17.18 RMG-Py Version 2.1.3

Date: July 27, 2017

- **Thermo central database:**

    - Framework for tracking and submitting species to a central database have been added

    - Following species submission, the central database will queue and submit quantum chemistry jobs for thermochemistry calculation

    - This is an initial step towards self-improving thermochemistry prediction

- **Rotor handling in Cantherm:**

    - Free rotors can now be specified

    - Limit number of terms used when fitting hinder rotor scans

    - Fixed bug with ZPE calculation when using hindered rotors

- **New reaction degeneracy algorithm:**

    - Use atom ID's to distinguish degenerate reactions from duplicates due to other factors

    - Degeneracy calculation now operates across all families rather than within each separately

    - Multiple transition states are now identified based on template comparisons and kept as duplicate reactions

- **Nodal distances:**

    - Distances can now be assigned to trees in reaction families

    - This enables better rate averages with multiple trees

    - Fixed bug with finding the closest rate rule in the tree

- **New features:**

    - Added methods for automatically writing RMG-database files

    - New symmetry algorithm improves symmetry number calculations for resonant and cyclic species

    - Group additivity algorithm updated to apply new long distance corrections

- Specific colliders can now be specified for pressure-dependent rates

- Very short superminimal example added (hydrogen oxidation) for checking basic RMG operation

- Cantera now outputs a Chemkin file which can be directly imported into Chemkin

- **Fixes:**

    - Fixed bug with negative activation energies when using Evans-Polanyi rates

    - Fixed walltime specification from command line when running RMG

    - Fixes and unit tests added for diffusionLimited module

- **Known issues:**

    - The multiple transition state algorithm can result in undesired duplicate reactions for reactants with multiple resonance structures

1.17.19 RMG-database Version 2.1.3

Date: July 27, 2017

- **Long-distance interaction thermo corrections:**

    - The gauche and int15 group files have been replaced by longDistanceInteraction_noncyclic

    - New corrections for cyclic ortho/meta/para interactions are now available in longDistanceInteraction_cyclic

- **Changes:**

    - Oa_R_Recombination family renamed to Birad_R_Recombination

    - More training reactions added for sulfur species in H_Abstraction

    - RMG-database tests have been moved to RMG-Py

1.17.20 RMG-Py Version 2.1.2

Date: May 18, 2017

- **Improvements:**

    - New nitrogen atom types

    - Kinetics libraries can now be specified as a list of strings in the input file

    - New script to generate output HTML locally: generateChemkinHTML.py

    - New kekulization module replaces RDKit for generating Kekule structures

    - Benzene bonds can now be reacted in reaction families

    - Removed cantherm.geometry module due to redundancy with statmech.conformer

- **Fixes:**

    - Reaction direction is now more deterministic after accounting for floating point error

    - Multiple bugs with resonance structure generation for aromatics have been addressed

1.17.21 RMG-database Version 2.1.2

Date: May 18, 2017

- **Nitrogen improvements:**

    - Added ethylamine kinetics library

    - Updated group additivity values for nitrogen species

    - Added rate rules and training reactions for nitrogen species

- **Additions:**

    - New CO_Disproportionation family

    - Added CurranPentane kinetics and thermo libraries

- **Fixes:**

    - Corrected some rates in FFCM1(-) to use MultiArrhenius kinetics

    - Corrected a few adjlists in FFCM1(-)

1.17.22 RMG-Py Version 2.1.1

Date: April 07, 2017

- **Uncertainty analysis:**

    - Local and global uncertainty analysis now available for RMG-generated models

    - Global uncertainty analysis uses MIT Uncertainty Quantification library, currently only supported on Linux systems

    - Examples for each module are available in localUncertainty.ipynb and globalUncertainty.ipynb

- **Fixes:**

    - Clar structure generation no longer intercepts signals

    - Fixes to SMILES generation

    - Fix default spin state of [CH]

1.17.23 RMG-database Version 2.1.1

Date: April 07, 2017

- **Additions:**

    - More species added to FFCM1(-) thermo library

- **Changes:**

    - Improved handling of excited species in FFCM1(-) kinetics library

    - Replaced Klippenstein H2O2 kinetics and thermo libraries with BurkeH2O2inN2 and BurkeH2O2inArHe

- **Fixes:**

    - Corrected adjlists for some species in JetSurf2.0 kinetics and thermo libraries (also renamed from JetSurf0.2)

– Correct multiplicities for [C] and [CH] in multiple libraries ([C] from 5 to 3, [CH] from 4 to 2)

1.17.24 RMG-Py Version 2.1.0

Date: March 07, 2017

- **Clar structure generation**

    – optimizes the aromatic isomer representations in RMG

    – lays the foundations for future development of poly-aromatic kinetics reaction families

- **Flux pathway analysis**

    – introduces an ipython notebook for post-generatation pathway analysis (`ipython.mechanism_analyzer.ipynb`)

    – visualizes reactions and provides flux statistics in a more transparent way

- **Cantera mechanism**

    – automatically writes cantera version of RMG-generated mechanism at the end of RMG jobs

- **Fixes bugs**

    – upgrades `pruning` to fix new memory leaks introduced by recent functionalities

    – fixes the bug of duplicated species creation caused by `getThermoData` removing isomers unexpectedly

    – fixes restart file generation and parsing problems and users can choose restart mode again

    – upgrades bicyclic decomposition method such that more deterministic behaviors are ensured

    – change bond order type to float from string to improve RMG's symmetry calculation for species with multiple resonance structures

1.17.25 RMG-database Version 2.1.0

Date: March 07, 2017

- **Several new kinetics libraries added**

    – FFCM-1

    – JetSurF 0.2

    – Chernov_aromatic_only

    – Narayanaswamy_aromatic_only

    – 1989_Stewart_2CH3_to_C2H5_H

    – 2005_Senosiain_OH_C2H2

    – 2006_Joshi_OH_CO

    – C6H5_C4H4_Mebel

    – c-C5H5_CH3_Sharma

- **Several new thermochemistry libraries added**

    – FFCM-1

    – JetSurF 0.2

- Chernov_aromatic_only

- Narayanaswamy_aromatic_only

- **Improved kinetics tree accessibility**

  - adds database tests ensuring groups in the tree to be accessible

  - improves definitions of group structures in the kinetics trees to ensure accessibility

- New oxygenates thermo groups are added based Paraskeva et al.

- **Improved database tools**

  - `convertKineticsLibraryToTrainingReactions.ipynb` now can visualize groups of matched rate rules that training reactions hit

  - `exportKineticsLibrarytoChemkin.py` and `importChemkinLibrary.py` add more logging information on reaction sources

1.17.26 RMG-Py Version 2.0.0

Date: September 16, 2016

This release includes several milestones of RMG project:

- **Parallelization finally introduced in RMG:**

  - Generates reactions during `enlarge` step in parallel fashion (`rmgpy.rmg.react`)

  - Enables concurrent computing for QMTP thermochemistry calculations (`rmgpy.thermo.thermoengine`)

  - Instructions of running RMG parallel mode can be found here for SLURM scheduler and here for SGE scheduler.

- **Polycyclic thermochemistry estimation improved:**

  - Extends group additivity method for polycyclics and estimates polycyclics of any large sizes by a heuristic method (bicyclics decomposition)

- **New tree averaging for kinetics:**

  - Fixes previous issue of imcomplete generation of cross-level rate rules

  - Implements Euclidean distance algorithm for the selection of the best rate rules to use in `estimateKinetics`

  - Streamlines storage of kinetics comments for averaged rules, which can be analyzed by `extractSourceFromComments`

- **Database entry accessibility tests:**

  - Adds entry accessibility tests for future entries (`testing.databaseTest`)

- **Fixes bugs**

  - fluxdiagram generation is now fixed, one can use it to generate short video of fluxdigram evolution

  - mac environment yml file is introduced to make sure smooth RMG-Py installation and jobs on mac

  - fixes failure of `checkForExistingSpecies` for polyaromatics species

  - fixes execution failure when both pruning and pDep are turned on

  - fixes pDep irreversible reactions

– fixes issue of valency of `Cbf` atom by dynamic benzene bond order assignment

1.17.27 RMG-database Version 2.0.0

Date: September 16, 2016

In conjunction with the release of RMG-Py v2.0.0, an updated package for the RMG-database has also been released. This release brings some new additions and fixes:

- **Polycyclic thermochemistry estimation improved:**

    – polycyclic database reorganized and more entries added in systematic way (`input.thermo.groups.polycyclic`)

- **Database entry accessibility tests:**

    – Fixes existing inaccessible entries in solvation/statmech/thermo of RMG-database

1.17.28 RMG-Py Version 1.0.4

Date: March 28, 2016

- **Cantera support in RMG (`rmgpy.tools.canteraModel`):**

    – Provides functions to help simulate RMG models using Cantera.

    – Has capability to generate cantera conditions and convert CHEMKIN files to cantera models, or use RMG to directly convert species and reactions objects to Cantera objects.

    – Demonstrative example found in `ipython/canteraSimulation.ipynb`

- **Module for regression testing of models generated by RMG (`rmgpy.tools.observableRegression`):**

    – Helps identify differences between two versions of models generated by RMG, using the "observables" that the user cares about.

- **Automatic plotting of simulations and sensitivities when generating models (`rmgpy.tools.plot`):**

    – Contains plotting classes useful for plotting simulations, sensitivities, and other data

    – Automatic plotting of simulations in the job's `solver` folder when `saveSimulationProfiles` is set to `True` in the input file.

    – Sensitivities for top 10 most sensitivie reactions and thermo now plotted automatically and stored in the `solver` folder.

- **Improved thermochemistry estimation (mostly for cyclics and polycyclics)**

    – Add rank as an additional attribute in thermo database entries to determine trustworthiness

- **Bug fixes:**

    – Training reactions now load successfully regardless of `generateSpeciesConstraints` parameters

    – Transport data is now saved correctly to CHEMKIN `tran.dat` file and also imports successfully

    – Fixes appending of reactions to CHEMKIN file when reaction libraries are desired to be appended to output

    – Fixes writing of csv files for simulation and sensitivity results in Windows

    – Fixes `Reaction.draw()` function to draw the entire reaction rather than a single species

1.17.29 RMG-Py Version 1.0.3

Date: February 4, 2016

This mini release contains the following updates:

- Pdep convergence issues in RMG-Py v1.0.2 are now fixed.

- RMG-database version information and anaconda binary version information is now recorded in RMG log file.

1.17.30 RMG-Py Version 1.0.2

Date: January 29, 2016

This new release adds several new features and bug fixes.

- Windows users can rejoice: RMG is now available in binary format on the Anaconda platform. Building by source is also much easier now through the Anaconda managed python environment for dependencies. See the updated *Installation Page* for more details

- Reaction filtering for speeding up model generation has now been added. It has been shown to speed up model convergence by 7-10x. See more details about how to use it in your RMG job *here*. Learn more about the theory and algorithm on the *Rate-based Model Enlarging Algorithm* page.

- The RMG *native scripts* are now organized under the `rmgpy.tools` submodule for developer ease and better extensibility in external scripts.

- InChI conversion is now more robust for singlets and triplets, and augmented InChIs and InChI keys are now possible with new radical electron, lone pair, and multiplicity flags.

- Output HTML for visualizing models are now cleaned up and also more functional, including features to display thermo comments, display enthalpy, entropy, and free energy of reaction, as well as filter reactions by species. You can use this new visualization format either by running a job in RMG v1.0.2 or revisualizing your CHEMKIN file and species dictionary using the visualization web tool.

1.17.31 RMG-database Version 1.0.2

Date: January 29, 2016

In conjunction with the release of RMG-Py v1.0.2, an updated package for the RMG-database has also been released. This release brings some new additions and fixes:

- New group additivity values for oxitene, oxerene, oexpane, and furan ring groups

- **Improvements to sulfur chemistry:**

    - Restructuring of radical trees in the kinetics families `SubstitutionS` and `intra_substitutionCS_cyclization`

    - A reaction library for di-tert-butyl sulfide

- Improvements for the `R_Addition_Multiple_Bond` kinetics family through new rate rules for the addition of allyl radical to double bonds in ethene, propene, and butene-like compounds, based on CBS-QB3 estimates from K. Wang, S.M. Villano, A.M. Dean, "Reactions of allylic radicals that impact molecular weight growth kinetics", *PCCP*, 6255-6273 (2015).

- Several new thermodynamic and kinetics libraries for molecules associated with the pyrolysis of cyclopentadiene in the presence of ethene, based off of calculations from the paper A.G. Vandeputte, S.S. Merchant, M.R. Djokic, K.M. Van Geem, G.B. Marin, W. H. Green, "Detailed study of cyclopentadiene pyrolysis in the presence of ethene: realistic pathways from C5H5 to naphthalene" (2016)

1.18 Credits

RMG is based upon work supported by the Department of Energy, Office of Basic Energy Sciences through grant DE-FG02-98ER14914 and by the National Science Foundation under Grants No. 0312359 and 0535604.

Project Supervisors:

- Prof. William H. Green (whgreen@mit.edu)
- Prof. Richard H. West (r.west@northeastern.edu)

Current Developers: (rmg_dev@mit.edu)

- Dr. Alon G. Dana
- Mark Goldman
- Colin Grambow
- Matt Johnson
- Mengjie Liu
- Mark Payne

Previous Developers:

- Dr. Joshua W. Allen
- Jacob Barlow
- Dr. Pierre L. Bhoorasingh
- Dr. Beat A. Buesser
- Dr. Caleb A. Class
- Dr. Connie Gao
- Dr. Kehang Han
- Dr. Fariba S. Khanshan
- Victor Lambert
- Dr. Shamel S. Merchant
- Dr. Belinda Slakman
- Sean Troiano
- Dr. Aaron Vandeputte
- Dr. Nick M. Vandewiele
- Dr. Nathan Yee
- Peng Zhang

1.19 How to Cite

Connie W. Gao, Joshua W. Allen, William H. Green, Richard H. West, "Reaction Mechanism Generator: Automatic construction of chemical kinetic mechanisms." *Computer Physics Communications* 203 (2016) 212-225. https://doi.org/10.1016/j.cpc.2016.02.013

- genindex

- modindex

- search

ARKANE USER'S GUIDE

2.1 Introduction

Arkane (Automated Reaction Kinetics and Network Exploration) is a tool for computing the thermodynamic properties of chemical species and high-pressure-limit rate coefficients for chemical reactions using the results of a quantum chemistry calculation. Thermodynamic properties are computed using the rigid rotor-harmonic oscillator approximation with optional corrections for hindered internal rotors. Kinetic parameters are computed using canonical transition state theory with optional tunneling correction.

Arkane can also estimate pressure-dependent phenomenological rate coefficients $k(T, P)$ for unimolecular reaction networks of arbitrary complexity. The approach is to first generate a detailed model of the reaction network using the one-dimensional master equation, then apply one of several available model reduction methods of varying accuracy, speed, and robustness to simplify the detailed model into a set of phenomenological rate coefficients. The result is a set of $k(T, P)$ functions suitable for use in chemical reaction mechanisms. More information is available at Allen et al..

Arkane is developed and distributed as part of RMG-Py, but can be used as a stand-alone application for Thermo-chemistry, Transition State Theory, and Master Equation chemical kinetics calculations.

Arkane is written in the Python programming language to facilitate ease of development, installation, and use.

Additional theoretical background can be found at RMG's Theory Guide and Arkane's Manual as well as the manual's supplement information.

2.1.1 License

Arkane is provided as free, open source code under the terms of the MIT/X11 License. The full, official license is reproduced below

```
Copyright (c) 2002-2018 Prof. William H. Green (whgreen@mit.edu),
Prof. Richard H. West (r.west@neu.edu) and the RMG Team (rmg_dev@mit.edu)

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the 'Software'),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

(continues on next page)

```
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

2.2 Installation

2.2.1 Installing Arkane

Arkane can be obtained by installing the RMG-Py software, which includes all neccesary dependancies.

Instructions to install RMG-Py can be found at the *RMG-Py Installation page*.

Note that you'll need to choose between the Basic User or Developer installation instructions that are specific to your operating system. Modifying Arkane source code will require Developer installation. If you are only looking to run the code, the Basic User installation will work.

2.3 Creating Input Files for Thermodynamics and High-Pressure Limit Kinetics Computations

2.3.1 Syntax

The format of Arkane input files is based on Python syntax. In fact, Arkane input files are valid Python source code, and this is used to facilitate reading of the file.

Each section is made up of one or more function calls, where parameters are specified as text strings, numbers, or objects. Text strings must be wrapped in either single or double quotes.

The following is a list of all the components of a Arkane input file for thermodynamics and high-pressure limit kinetics computations:

Component	Description
modelChemistry	Level of theory from quantum chemical calculations
atomEnergies	Dictionary of atomic energies at modelChemistry level
frequencyScaleFactor	A factor by which to scale all frequencies
useHinderedRotors	True (by default) if hindered rotors are used, False if not
useAtomCorrections	True (by default) if atom corrections are used, False if not
useBondCorrections	True if bond corrections are used, False (by default) if not
species	Contains parameters for non-transition states
transitionState	Contains parameters for transition state(s)
reaction	Required for performing kinetic computations
statmech	Loads statistical mechanics parameters
thermo	Performs a thermodynamics computation
kinetics	Performs a high-pressure limit kinetic computation

2.3.2 Model Chemistry

The first item in the input file should be a `modelChemistry` assignment with a string describing the model chemistry.

Arkane uses this information to adjust the computed energies to the usual gas-phase reference states by applying atom, bond and spin-orbit coupling energy corrections. This is particularly important for `thermo()` calculations (see below). Note that the user must specify under the `species()` function the type and number of bonds for Arkane to apply these corrections. The example below specifies CBS-QB3 as the model chemistry:

```
modelChemistry = "CBS-QB3"
```

Alternatively, the atomic energies at the `modelChemistry` level of theory can be directly specified in the input file by providing a dictionary of these energies in the following format:

```
atomEnergies = {
    'H': -0.499818,
    'C': -37.78552,
    'N': -54.520543,
    'O': -74.987979,
    'S': -397.658253,
}
```

The table below shows which model chemistries have atomization energy corrections (AEC), bond corrections (BC), and spin orbit corrections (SOC). It also lists which elements are available for a given model chemistry.

Model Chemistry	AEC	BC	SOC	Freq Scale	Supported Elements
`'CBS-QB3'`	v	v	v	v (0.990)	H, C, N, O, P, S
`'CBS-QB3-Paraskevas'`	v	v	v	v (0.990)	H, C, N, O, P, S
`'G3'`	v		v		H, C, N, O, P, S
`'M08SO/MG3S*'`	v		v		H, C, N, O, P, S
`'M06-2X/cc-pVTZ'`	v		v	v (0.955)	H, C, N, O, P, S
`'Klip_1'`	v		v		H, C, N, O
`'Klip_2'` *uses QCI(tz,qz) values*	v		v		H, C, N, O
`'Klip_3'` *uses QCI(dz,qz) values*	v		v		H, C, N, O
`'Klip_2_cc'` *uses CCSD(T)(tz,qz) values*	v		v		H, C, O
`'CCSD-F12/cc-pVDZ-F12'`	v		v	v (0.947)	H, C, N, O
`'CCSD(T)-F12/cc-pVDZ-F12_H-TZ'`	v		v		H, C, N, O
`'CCSD(T)-F12/cc-pVDZ-F12_H-QZ'`	v		v		H, C, N, O
`'CCSD(T)-F12/cc-pVnZ-F12'`, $n = D,T,Q$	v	v	v	v	H, C, N, O, S
`'CCSD(T)-F12/cc-pVDZ-F12_noscale'`	v		v		H, C, N, O
`'CCSD(T)-F12/cc-pCVnZ-F12'`, $n = D,T,Q$	v		v	v	H, C, N, O
`'CCSD(T)-F12/aug-cc-pVnZ'`, $n = D,T,Q$	v		v	v	H, C, N, O, S
`'CCSD(T)-F12/cc-pVTZ-f12(-pp)'`,	v		v		H, C, N, O, S, I
`'CCSD(T)/aug-cc-pVTZ(-pp)'`,	v		v		H, C, O, S, I
`'B-CCSD(T)-F12/cc-pVnZ-F12'`, $n = D,T,Q$	v		v		H, C, N, O, S
`'B-CCSD(T)-F12/cc-pCVnZ-F12'`, $n = D,T,Q$	v		v		H, C, N, O
`'B-CCSD(T)-F12/aug-cc-pVnZ'`, $n = D,T,Q$	v		v		H, C, N, O
`'G03_PBEPBE_6-311++g_d_p'`	v		v		H, C, N, O
`'MP2_rmp2_pVnZ'`, $n = D,T,Q$	v		v	v	H, C, N, O
`'FCI/cc-pVnZ'`, $n = D,T,Q$	v		v		C
`'BMK/cbsb7'`	v	v	v		H, C, N, O, P, S
`'BMK/6-311G(2d,d,p)'`	v	v	v		H, C, N, O, P, S
`'B3LYP/6-311+G(3df,2p)'`	v	v	v	v (0.967)	H, C, N, O, P, S

Continued on next page

Table 1 – continued from previous page

Model Chemistry	AEC	BC	SOC	Freq Scale	Supported Elements
`'B3LYP/6-31G**'`	v	v		v (0.961)	H, C, O, S
`'MRCI+Davidson/aug-cc-pV(T+d)Z'`	v		v		H, C, N, O, S
`'wb97x-d/aug-cc-pvtz'`	v		v		H, C, N, O

Notes:

- The `'CBS-QB3-Paraskevas'` model chemistry is identical to `'CBS-QB3'` except for BCs for C/H/O bonds, which are from Paraskevas et al. (DOI: 10.1002/chem.201301381) instead of Petersson et al. (DOI: 10.1063/1.477794). Beware, combining BCs from different sources may lead to unforeseen results.

- In `'M08SO/MG3S*'` the grid size used in the [QChem] electronic structure calculation utilizes 75 radial points and 434 angular points.

- Refer to paper by Goldsmith et al. (*Goldsmith, C. F.; Magoon, G. R.; Green, W. H., Database of Small Molecule Thermochemistry for Combustion. J. Phys. Chem. A 2012, 116, 9033-9057*) for definition of `'Klip_2'` (*QCI(tz,qz)*) and `'Klip_3'` (*QCI(dz,qz)*).

If a model chemistry other than the ones in the above table is used, then the user should supply the corresponding atomic energies (using `atomEnergies`) to get meaningful results. Bond corrections would not be applied in this case.

If a model chemistry or atomic energies are not available, then a kinetics job can still be run by setting `useAtomCorrections` to `False`, in which case Arkane will not raise an error for unknown elements. The user should be aware that the resulting energies and thermodynamic quantities in the output file will not be meaningful, but kinetics and equilibrium constants will still be correct.

2.3.3 Frequency Scale Factor

Frequency scale factors are empirically fit to experiment for different `modelChemistry`. Refer to NIST website for values (http://cccbdb.nist.gov/vibscalejust.asp). For CBS-QB3, which is not included in the link above, `frequencyScaleFactor = 0.99` according to Montgomery et al. (*J. Chem. Phys. 1999, 110, 2822–2827*). The frequency scale factor is automatically assigned according to the supplied `modelChemistry`, if available (see above table). If not available automatically and not specified by the user, it will be assumed a unity value.

2.3.4 Species

Each species of interest must be specified using a `species()` function, which can be input in two different ways, discussed in the separate subsections below:

1. By pointing to the output files of quantum chemistry calculations, which Arkane will parse for the necessary molecular properties 2. By directly entering the molecular properties

Within a single input file, both Option #1 and #2 may be used for different species.

Option #1: Automatically Parse Quantum Chemistry Calculation Output

For this option, the `species()` function only requires two parameters, as in the example below:

```
species('C2H6', 'C2H6.py')
```

The first parameter (`'C2H6'` above) is the species label, which can be referenced later in the input file. The second parameter (`'C2H6.py'` above) points to the location of another python file containing details of the species. This file will be referred to as the species input file.

The species input file accepts the following parameters:

Parameter	Required?	Description
bonds	optional	Type and number of bonds in the species
linear	yes	`True` if the molecule is linear, `False` if not
externalSymmetry	yes	The external symmetry number for rotation
spinMultiplicity	yes	The ground-state spin multiplicity (degeneracy)
opticalIsomers	yes	The number of optical isomers of the species
energy	yes	The ground-state 0 K atomization energy in Hartree (without zero-point energy) **or** The path to the quantum chemistry output file containing the energy
geometry	yes	The path to the quantum chemistry output file containing the optimized geometry
frequencies	yes	The path to the quantum chemistry output file containing the computed frequencies
rotors	optional	A list of `HinderedRotor()` and/or `FreeRotor()` objects describing the hindered/free rotors

The types and number of atoms in the species are automatically inferred from the quantum chemistry output and are used to apply atomization energy corrections (AEC) and spin orbit corrections (SOC) for a given `modelChemistry` (see *Model Chemistry*). If not interested in accurate thermodynamics (e.g., if only using `kinetics()`), then atom corrections can be turned off by setting `useAtomCorrections` to `False`.

The `bond` parameter is used to apply bond corrections (BC) for a given `modelChemistry`.

Allowed bond types for the `bonds` parameter are, e.g., `'C-H'`, `'C-C'`, `'C=C'`, `'N-O'`, `'C=S'`, `'O=O'`, `'C#N'`...

`'O=S=O'` is also allowed.

The order of elements in the bond correction label is not important. Use `-/=/#` to denote a single/double/triple bond, respectively. For example, for formaldehyde we would write:

```
bonds = {'C=O': 1, 'C-H': 2}
```

The parameter `linear` only needs to be specified as either `True` or `False`. The parameters `externalSymmetry`, `spinMultiplicity` and `opticalIsomers` only accept integer values. Note that `externalSymmetry` corresponds to the number of unique ways in which the species may be rotated about an axis (or multiple axes) and still be indistinguishable from its starting orientation (reflection across a mirror plane does not count as rotation about an axis). For ethane, we would write:

```
linear = False

externalSymmetry = 6

spinMultiplicity = 1

opticalIsomers = 1
```

The `energy` parameter is a dictionary with entries for different `modelChemistry`. The entries can consist of either floating point numbers corresponding to the 0 K atomization energy in Hartree (without zero-point energy correction), or they can specify the path to a quantum chemistry calculation output file that contains the species's energy. For example:

```
energy = {
'CBS-QB3': Log('ethane_cbsqb3.log'),
'Klip_2': -79.64199436,
}
```

In this example, the `CBS-QB3` energy is obtained from a Gaussian log file, while the `Klip_2` energy is specified

directly. The energy used will depend on what `modelChemistry` was specified in the input file. Arkane can parse the energy from a Gaussian, Molpro, or QChem log file, all using the same `Log` class, as shown below.

The input to the remaining parameters, `geometry`, `frequencies` and `rotors`, will depend on if hindered/free rotors are included. Both cases are described below.

Without Hindered/Free Rotors

In this case, only `geometry` and `frequencies` need to be specified, and they can point to the same or different quantum chemistry calculation output files. The `geometry` file contains the optimized geometry, while the `frequencies` file contains the harmonic oscillator frequencies of the species in its optimized geometry. For example:

```
geometry = Log('ethane_cbsqb3.log')

frequencies = Log('ethane_freq.log')
```

In summary, in order to specify the molecular properties of a species by parsing the output of quantum chemistry calculations, without any hindered/free rotors, the `species()` function in the input file should look like the following example:

```
species('C2H6', 'C2H6.py')
```

and the species input file (`C2H6.py` in the example above) should look like the following:

```
bonds = {
    'C-C': 1,
    'C-H': 6,
}

linear = False

externalSymmetry = 6

spinMultiplicity = 1

opticalIsomers = 1

energy = {
    'CBS-QB3': Log('ethane_cbsqb3.log'),
    'Klip_2': -79.64199436,
}

geometry = Log('ethane_cbsqb3.log')

frequencies = Log('ethane_freq.log')
```

With Hindered/Free Rotors

In this case, `geometry`, `frequencies` and `rotors` need to be specified. The `geometry` and `frequencies` parameters must point to the **same** quantum chemistry calculation output file in this case. For example:

```
geometry = Log('ethane_freq.log')

frequencies = Log('ethane_freq.log')
```

The `geometry/frequencies` log file must contain both the optimized geometry and the Hessian (matrix of partial second derivatives of potential energy surface, also referred to as the force constant matrix), which is used to calculate the harmonic oscillator frequencies. If Gaussian is used to generate the `geometry/frequencies` log file, the Gaussian input file must contain the keyword `iop(7/33=1)`, which forces Gaussian to output the complete Hessian. Because the `iop(7/33=1)` option is only applied to the first part of the Gaussian job, the job must be a `freq` job only (as opposed to an `opt freq` job or a composite method job like `cbs-qb3`, which only do the `freq` calculation after the optimization). Therefore, the proper workflow for generating the `geometry/frequencies` log file using Gaussian is:

1. Perform a geometry optimization.

2. Take the optimized geometry from step 1, and use it as the input to a `freq` job with the following input keywords:
   `#method basis-set freq iop(7/33=1)`

The output of step 2 is the correct log file to use for `geometry/frequencies`.

`rotors` is a list of `HinderedRotor()` and/or `FreeRotor()` objects. Each `HinderedRotor()` object requires the following parameters:
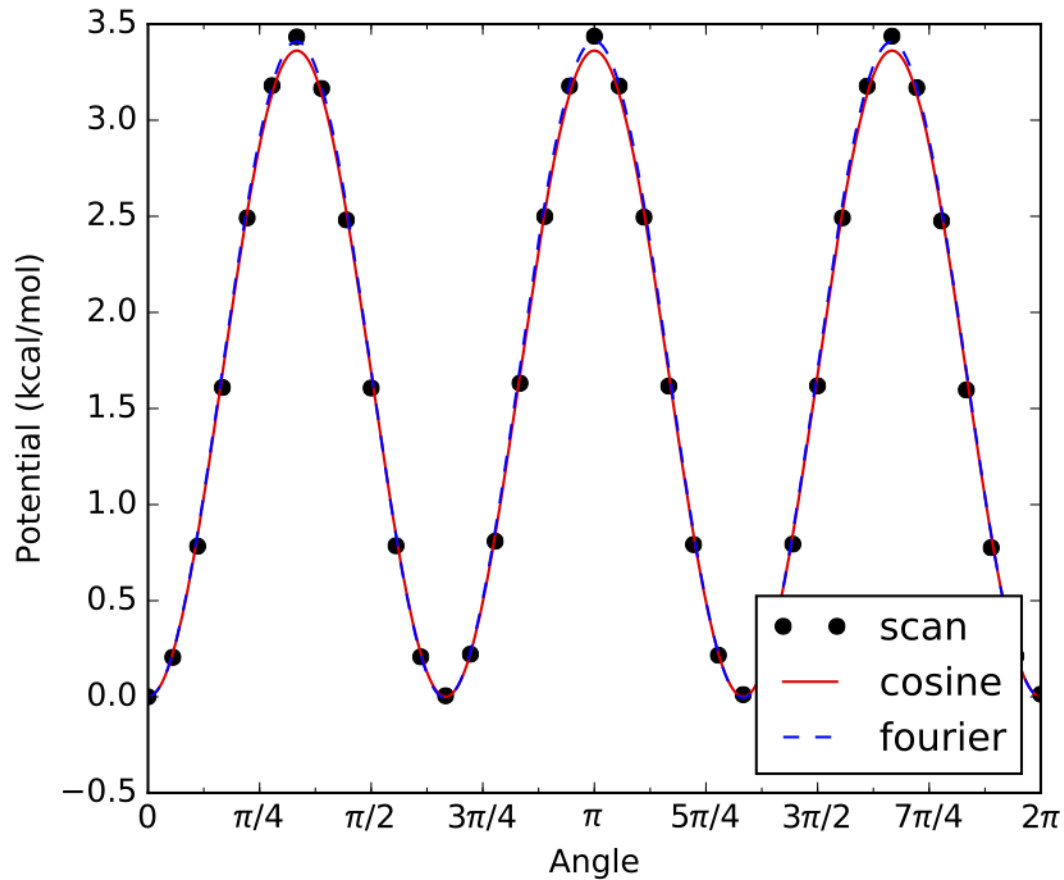
Parameter	Description
`scanLog`	The path to the Gaussian/Qchem log file, or a text file containing the scan energies
`pivots`	The indices of the atoms in the hindered rotor torsional bond
`top`	The indices of all atoms on one side of the torsional bond (including the pivot atom)
`symmetry`	The symmetry number for the torsional rotation (number of indistinguishable energy minima)
`fit`	Fit to the scan data. Can be either `fourier`, `cosine` or `best` (default).

`scanLog` can either point to a `Log` file, or simply a `ScanLog`, with the path to a text file summarizing the scan in the following format:

```
Angle (radians)          Energy (kJ/mol)
 0.0000000000             0.0147251160
 0.1745329252             0.7223109804
 0.3490658504             2.6856059517
        .                        .
        .                        .
        .                        .
 6.2831853072             0.0000000000
```

The `Energy` can be in units of `kJ/mol`, `J/mol`, `cal/mol`, `kcal/mol`, `cm^-1` or `hartree`.

The `symmetry` parameter will usually equal either 1, 2 or 3. Below are examples of internal rotor scans with these commonly encountered symmetry numbers. First, `symmetry = 3`:

Internal rotation of a methyl group is a common example of a hindered rotor with `symmetry = 3`, such as the one above. As shown, all three minima (and maxima) have identical energies, hence `symmetry = 3`.

Similarly, if there are only two minima along the internal rotor scan, and both have identical energy, then `symmetry = 2`, as in the example below:

If any of the energy minima in an internal rotor scan are not identical, then the rotor has no symmetry (`symmetry = 1`), as in the example below:

For the example above there are 3 local energy minima, 2 of which are identical to each other. However, the 3rd minima is different from the other 2, therefore this internal rotor has no symmetry.

For practical purposes, when determining the symmetry number for a given hindered rotor simply check if the internal rotor scan looks like the `symmetry = 2` or `3` examples above. If it doesn't, then most likely `symmetry = 1`.

Each `FreeRotor()` object requires the following parameters:

Parameter	Description
`pivots`	The indices of the atoms in the free rotor torsional bond
`top`	The indices of all atoms on one side of the torsional bond (including the pivot atom)
`symmetry`	The symmetry number for the torsional rotation (number of indistinguishable energy minima)

Note that a `scanLog` is not needed for `FreeRotor()` because it is assumed that there is no barrier to internal rotation. Modeling an internal rotation as a `FreeRotor()` puts an upper bound on the impact of that rotor on the species's overall partition function. Modeling the same internal rotation as a Harmonic Oscillator (default if it is not specifed as either a `FreeRotor()` or `HinderedRotor()`) puts a lower bound on the impact of that rotor on the species's overall partition function. Modeling the internal rotation as a `HinderedRotor()` should fall in between these two extremes.

To summarize, the species input file with hindered/free rotors should look like the following example (different options for specifying the same `rotors` entry are commented out):

```
bonds = {
    'C-C': 1,
```

```
    'C-H': 6,
}

linear = False

externalSymmetry = 6

spinMultiplicity = 1

opticalIsomers = 1

energy = {
    'CBS-QB3': Log('ethane_cbsqb3.log'),
    'Klip_2': -79.64199436,
}

geometry = Log('ethane_freq.log')

frequencies = Log('ethane_freq.log')

rotors = [
    HinderedRotor(scanLog=Log('ethane_scan_1.log'), pivots=[1,5], top=[1,2,3,4], symmetry=3,
→fit='best'),
    #HinderedRotor(scanLog=ScanLog('C2H6_rotor_1.txt'), pivots=[1,5], top=[1,2,3,4], symmetry=3,
→ fit='best'),
    #FreeRotor(pivots=[1,5], top=[1,2,3,4], symmetry=3),
]
```

Note that the atom labels identified within the rotor section should correspond to the indicated geometry.

Option #2: Directly Enter Molecular Properties

While it is usually more convenient to have Arkane parse molecular properties from the output of quantum chemistry calculations (see *Option #1: Automatically Parse Quantum Chemistry Calculation Output*) there are instances where an output file is not available and it is more convenient for the user to directly enter the molecular properties. This is the case, for example, if the user would like to use calculations from literature, where the final calculated molecular properties are often reported in a table (e.g., vibrational frequencies, rotational constants), but the actual output files of the underlying quantum chemistry calculations are rarely provided.

For this option, there are a number of required parameters associated with the `species()` function

Parameter	Required?	Description
`label`	yes	A unique string label used as an identifier
`E0`	yes	The ground-state 0 K enthalpy of formation (including zero-point energy)
`modes`	yes	The molecular degrees of freedom (see below)
`spinMultiplicity`	yes	The ground-state spin multiplicity (degeneracy), sets to 1 by default if not used
`opticalIsomers`	yes	The number of optical isomers of the species, sets to 1 by default if not used
`reactive`	only bath gases	Boolean indicating whether the molecule reacts, set to `False` for bath gases. default is `True`

The `label` parameter should be set to a string with the desired name for the species, which can be reference later in the input file.

```
label = 'C2H6'
```

The `E0` ground state 0 K enthalpy of formation (including zero-point energy) should be given in the quantity format
`(value, 'units')`, using units of either `kJ/mol`, `kcal/mol`, `J/mol`, or `cal/mol`:

```
E0 = (100.725, 'kJ/mol')
```

Note that if Arkane is being used to calculate the thermochemistry of the species, it is critical that the value of `E0` is
consistent with the definition above (0 K enthalpy of formation with zero-point energy). However, if the user is only
interested in kinetics, `E0` can be defined on any arbitrary absolute energy scale, as long as the correct relative energies
between various `species()` and `transitionState()` are maintained. For example, it is common in literature for
the energy of some reactant(s) to be arbitrarily defined as zero, and the energies of all transition states, intermediates
and products are reported relative to that.

Also note that the value of `E0` provided here will be used directly, i.e., no atom or bond corrections will be applied.

If you want Cantherm to correct for zero point energy, you can either just place the raw units in Hartree (as if it were
read directly from quantum):

> E0 = 547.6789753223456

Or you can add a third argument to the Quantity specified whether zero-point energy is included or not:

> E0 = (95.1, 'kJ/mol', 'E0') # when ZPE is not included E0 = (95.1, 'kJ/mol', 'E0-ZPE') # when ZPE is
> already included

When specifying the `modes` parameter, define a list with the following types of degrees of freedom. To understand
how to define these degrees of freedom, please click on the links below:

Translational degrees of freedom

Class	Description
IdealGasTranslation	A model of three-dimensional translation of an ideal gas

Rotational degrees of freedom

Class	Description
LinearRotor	A model of two-dimensional rigid rotation of a linear molecule
NonlinearRotor	A model of three-dimensional rigid rotation of a nonlinear molecule
KRotor	A model of one-dimensional rigid rotation of a K-rotor
SphericalTopRotor	A model of three-dimensional rigid rotation of a spherical top molecule

Vibrational degrees of freedom

Class	Description
HarmonicOscillator	A model of a set of one-dimensional harmonic oscillators

Note that the `frequencies` provided here will be used directly, i.e., the `frequencyScaleFactor` will not be ap-
plied.

Torsional degrees of freedom

Class	Description
HinderedRotor	A model of a one-dimensional hindered rotation
FreeRotor	A model of a one-dimensional free rotation

The `spinMultiplicity` is defined using an integer, and is set to 1 if not indicated in the `species()` function.

```
spinMultiplicity = 1
```

Similarly, the `opticalIsomers` is also defined using an integer, and is set to 1 if not used in the `species()` function.

```
opticalIsomers = 1
```

The following is an example of a typical `species()` function, based on ethane (different options for specifying the same internal rotation are commented out):

```
species(
    label = 'C2H6',
    E0 = (100.725, 'kJ/mol'),
    modes = [
        IdealGasTranslation(mass=(30.0469, 'amu')),
        NonlinearRotor(
            inertia = ([6.27071, 25.3832, 25.3833], 'amu*angstrom^2'),
            symmetry = 6,
        ),
        HarmonicOscillator(
            frequencies = ([818.917, 819.48, 987.099, 1206.81, 1207.06, 1396, 1411.35, 1489.78,
→1489.97, 1492.49, 1492.66, 2995.36, 2996.06, 3040.83, 3041, 3065.86, 3066.02], 'cm^-1'),
        ),
        HinderedRotor(
            inertia = (1.56768, 'amu*angstrom^2'),
            symmetry = 3,
            barrier = (11.2717, 'kJ/mol'),
        ),
        #HinderedRotor(
        #inertia = (1.56768, 'amu*angstrom^2'),
        #symmetry = 3,
        #fourier = (
        #    [
        #        [0.00458375, 0.000841648, -5.70271, 0.00602657, 0.0047446],
        #        [0.000726951, -0.000677255, 0.000207033, 0.000553307, -0.000503303],
        #    ],
        #    'kJ/mol',
        #),
        #),
        #FreeRotor(
        #    inertia = (1.56768, 'amu*angstrom^2'),
        #    symmetry = 3,
        #),
    ],
    spinMultiplicity = 1,
    opticalIsomers = 1,
)
```

Note that the format of the `species()` function above is identical to the `conformer()` function output by Arkane in `output.py`. Therefore, the user could directly copy the `conformer()` output of an Arkane job to another Arkane input file, change the name of the function to `species()` (or `transitionState()`, if appropriate, see next section) and run a new Arkane job in this manner. This can be useful if the user wants to easily switch a `species()` function from Option #1 (parsing quantum chemistry calculation output) to Option #2 (directly enter molecular properties).

2.3.5 Transition State

Transition state(s) are only required when performimg kinetics computations. Each transition state of interest must be specified using a `transitionState()` function, which is analogous to the `species()` function described above. Therefore, the `transitionState()` function may also be specified in two ways: *Option #1: Automatically Parse Quantum Chemistry Calculation Output* and *Option #2: Directly Enter Molecular Properties*

The following is an example of a typical `transitionState()` function using Option #1:

```
transitionState('TS', 'TS.py')
```

Just as for a `species()` function, the first parameter is the label for that transition state, and the second parameter points to the location of another python file containing details of the transition state. This file will be referred to as the transition state input file, and it accepts the same parameters as the species input file described in *Option #1: Automatically Parse Quantum Chemistry Calculation Output*.

The following is an example of a typical `transitionState()` function using Option #2:

```
transitionState(
    label = 'TS',
    E0 = (267.403, 'kJ/mol'),
    modes = [
        IdealGasTranslation(mass=(29.0391, 'amu')),
        NonlinearRotor(
            inertia = ([6.78512, 22.1437, 22.2114], 'amu*angstrom^2'),
            symmetry = 1,
        ),
        HarmonicOscillator(
            frequencies = ([412.75, 415.206, 821.495, 924.44, 982.714, 1024.16, 1224.21, 1326.
→36, 1455.06, 1600.35, 3101.46, 3110.55, 3175.34, 3201.88], 'cm^-1'),
        ),
    ],
    spinMultiplicity = 2,
    opticalIsomers = 1,
    frequency = (-750.232, 'cm^-1'),
)
```

The only additional parameter required for a `transitionState()` function as compared to a `species()` function is `frequency`, which is the imaginary frequency of the transition state needed to account for tunneling. Refer to *Option #2: Directly Enter Molecular Properties* for a more detailed description of the other parameters.

2.3.6 Reaction

This is only required if you wish to perform a kinetics computation. Each reaction of interest must be specified using a `reaction()` function, which accepts the following parameters:

Parameter	Description
`label`	A unique string label used as an identifier
`reactants`	A list of strings indicating the labels of the reactant species
`products`	A list of strings indicating the labels of the product species
`transitionState`	The string label of the transition state
`tunneling`	Method of estimating the quantum tunneling factor (optional)

The following is an example of a typical reaction function:

```
reaction(
    label = 'H + C2H4 <=> C2H5',
    reactants = ['H', 'C2H4'],
    products = ['C2H5'],
    transitionState = 'TS',
    tunneling='Eckart'
)
```

Note: the quantum tunneling factor method that may be assigned is either `'Eckart'` or `'Wigner'`.

2.3.7 Thermodynamics Computations

Use a `thermo()` function to make Arkane execute the thermodynamic parameters computatiom for a species. Pass the string label of the species you wish to compute the thermodynamic parameters for and the type of thermodynamics polynomial to generate (either `'Wilhoit'` or `'NASA'`). A table of relevant thermodynamic parameters will also be displayed in the output file.

Below is a typical `thermo()` execution function:

```
thermo('ethane', 'NASA')
```

2.3.8 Kinetics Computations

Use a `kinetics()` function to make Arkane execute the high-pressure limit kinetic parameters computation for a reaction. The `'label'` string must correspond to that of a defined `reaction()` function. If desired, define a temperature range and number of temperatures at which the high-pressure rate coefficient will be tabulated and saved to the outupt file. The 3-parameter modified Arrhenius coefficients will automatically be fit to the computed rate coefficients. The quantum tunneling factor will also be displayed.

Below is a typical `kinetics()` function:

```
kinetics(
label = 'H + C2H4 <=> C2H5',
Tmin = (400,'K'), Tmax = (1200,'K'), Tcount = 6,
)
```

If specific temperatures are desired, you may specify a list (`Tlist = ([400,500,700,900,1100,1200],'K')`) instead of Tmin, Tmax, and Tcount.

This is also acceptable:

```
kinetics('H + C2H4 <=> C2H5')
```

If a sensitivity analysis is desired, simply add the conditions at which to calculate sensitivity coefficients in the following format, e.g.:

```
kinetics(
    label = 'HSOO <=> HOOS',
    Tmin = (500,'K'), Tmax = (3000,'K'), Tcount = 15,
    sensitivity_conditions = [(1000, 'K'), (2000, 'K')]
)
```

The output of a sensitivity analysis is saved into a `sensitivity` folder in the output directory. A text file, named with the reaction label, delineates the semi-normalized sensitivity coefficients `dln(k)/dE0` in units of `mol/J` at all

requested conditions. A horizontal bar figure is automatically generated per reaction with subplots for both the forward and reverse direction at all conditions.

2.3.9 Examples

Perhaps the best way to learn the input file syntax is by example. To that end, a number of example input files and their corresponding output have been given in the examples directory.

2.3.10 Troubleshooting and FAQs

1) The network that Arkane generated and the resulting pdf file show abnormally large absolute values. What's going on?

> This can happen if the number of atoms and atom types is not properly defined or consistent in your input file(s).

2.3.11 Arkane User Checklist

Using Arkane, or any rate theory package for that matter, requires careful consideration and management of a large amount of data, files, and input parameters. As a result, it is easy to make a mistake somewhere. This checklist was made to minimize such mistakes for users:

- Do correct paths exist for pointing to the files containing the electronic energies, molecular geometries and vibrational frequencies?

For calculations involving pressure dependence:

- Does the network pdf look reasonable? That is, are the relative energies what you expect based on the input?

For calculations using internal hindered rotors:

- Did you check to make sure the rotor has a reasonable potential (e.g., visually inspect the automatically generated rotor pdf files)?

- Within your input files, do all specified rotors point to the correct files?

- Do all of the atom label indices correspond to those in the file that is read by Log?

- Why do the fourier fits look so much different than the results of the ab initio potential energy scan calculations? This is likely because the initial scan energy is not at a minimum. One solution is to simply shift the potential with respect to angle so that it starts at zero and, instead of having Arkane read a Qchem or Gaussian output file, have Arkane point to a 'ScanLog' file. Another problem can arise when the potential at 2*pi is also not [close] to zero.

2.4 Creating Input Files for Pressure Dependent Calculations

2.4.1 Syntax

There are four parts to a pressure-dependent calculation input file, giving the species, transition states, path reactions, reaction network, and algorithm parameters. The species section must come before the reaction section. Before discussing each of these sections, a brief word on the general input file syntax will be given.

The format of Arkane input files is based on Python syntax. In fact, Arkane input files are valid Python source code, and this is used to facilitate reading of the file.

Each section is made up of one or more function calls, where parameters are specified as text strings, numbers, or objects. Text strings must be wrapped in either single or double quotes.

The following is a list of all the components of a Arkane input file for pressure-dependent calculations:

Component	Description
modelChemistry	Level of theory from quantum chemical calculations
atomEnergies	Dictionary of atomic energies at modelChemistry level
frequencyScaleFactor	A factor by which to scale all frequencies
useHinderedRotors	True if hindered rotors are used, False if not
useAtomCorrections	True if atom corrections are used, False if not
useBondCorrections	True if bond corrections are used, False if not
species	Contains parameters for non-transition states
transitionState	Contains parameters for transition state(s)
reaction	Required for performing kinetic computations
network	Divides species into reactants, isomers, products and bath gases
pressureDependence	Defines parameters necessary for solving master equation
statmech	Loads statistical mechanics parameters
thermo	Performs a thermodynamics computation
kinetics	Performs a high-pressure limit kinetic computation

Note that many of the functions in the table above overlap with the functions available for thermodynamics and high-pressure limit kinetics computations. For most of these overlapping functions, the input is identical. Important differences are mentioned in the sections below.

2.4.2 Model Chemistry

The first item in the input file should be a modelChemistry assignment with a string describing the model chemistry.

Arkane uses this information to adjust the computed energies to the usual gas-phase reference states by applying atom, bond and spin-orbit coupling energy corrections. This is particularly important for thermo() calculations (see below). Note that the user must specify under the species() function the type and number of bonds for Arkane to apply these corrections. The example below specifies CBS-QB3 as the model chemistry:

```
modelChemistry = "CBS-QB3"
```

Alternatively, the atomic energies at the modelChemistry level of theory can be directly specified in the input file by providing a dictionary of these energies in the following format:

```
atomEnergies = {
    'H': -0.499818,
    'C': -37.78552,
    'N': -54.520543,
    'O': -74.987979,
    'S': -397.658253,
}
```

Whether or not atomization energy corrections (AEC), bond corrections (BC), and spin orbit corrections (SOC); and which elements are available for a given model chemistry is described under High-Pressure Limit: Model Chemistry

2.4.3 Frequency Scale Factor

Frequency scale factors are empirically fit to experiment for different modelChemistry. Refer to NIST website for values (http://cccbdb.nist.gov/vibscalejust.asp). For CBS-QB3, which is not included in the link above,

`frequencyScaleFactor = 0.99` according to Montgomery et al. (*J. Chem. Phys. 1999, 110, 2822–2827*).

2.4.4 Species Parameters

Each species in the network must be specified using a `species()` function. This includes all unimolecular isomers, bimolecular reactants and products, and the bath gas(es). A species that appears in multiple bimolecular channels need only be specified with a single `species()` function.

The input to the `species()` function for a pressure-dependent calculation is the same as for a thermodynamic orhigh-pressure limit kinetics calculation, with the addition of a few extra parameters needed to describe collisional energy transfer. There are two options for providing input to the `species()` function, which are described in the subsections below:

1. By pointing to the output files of quantum chemistry calculations, which Arkane will parse for the necessary molecular properties 2. By directly entering the molecular properties

Within a single input file, both Option #1 and #2 may be used.

Regardless of which option is used to specify molecular properties (e.g., vibrational frequencies, rotational constants) in the `species()` function, the four parameters listed below (mostly relating to the collisional energy transfer model) are always specified in the same way.

Parameter	Required?	Description
`structure`	all species except bath gases	A chemical structure for the species defined using either SMILES or InChI
`molecularWeight`	all species	The molecular weight, if not given it is calculated based on the structure
`collisionModel`	unimolecular isomers and bath gases	Transport data for the species
`energyTransferModel`	unimolecular isomers	Assigned with `SingleExponentialDown` model
`reactive`	bath gases	Booleanindicating whether the molecule reacts, set to `False` for bath gases. default is `True`

The `structure` parameter is defined by either SMILES or InChI. For instance, either representation is acceptable for the acetone molecule:

```
structure = SMILES('CC(C)=O')

structure = adjacencyList("""1  C u0 p0 c0 {2,S} {5,S} {6,S} {7,S}
                             2  C u0 p0 c0 {1,S} {3,S} {4,D}
                             3  C u0 p0 c0 {2,S} {8,S} {9,S} {10,S}
                             4  O u0 p2 c0 {2,D}
                             5  H u0 p0 c0 {1,S}
                             6  H u0 p0 c0 {1,S}
                             7  H u0 p0 c0 {1,S}
                             8  H u0 p0 c0 {3,S}
                             9  H u0 p0 c0 {3,S}
                             10 H u0 p0 c0 {3,S}""")

structure = InChI('InChI=1S/C3H6O/c1-3(2)4/h1-2H3')
```

The `molecularWeight` parameter should be defined in the quantity format `(value, 'units')`, for example:

```
molecularWeight = (44.04, 'g/mol')
```

If the `molecularWeight` parameter is not given, it is calculated by Arkane based on the chemical structure.

The `collisionModel` is defined for unimolecular isomers with the transport data using a `TransportData` object:

```
collisionModel = TransportData(sigma=(3.70,'angstrom'), epsilon=(94.9,'K'))
```

`sigma` and `epsilon` are Lennard-Jones parameters, which can be estimated using the Joback method on the RMG website.

The `energyTransferModel` model available is a `SingleExponentialDown`.

- `SingleExponentialDown` - Specify `alpha0`, `T0` and `n` for the average energy transferred in a deactiving collision

$$\langle \Delta E_{\mathrm{down}} \rangle = \alpha_0 \left(\frac{T}{T_0} \right)^n$$

An example of a typical `energyTransferModel` function is:

```
energyTransferModel = SingleExponentialDown(
        alpha0 = (0.5718,'kcal/mol'),
        T0 = (300,'K'),
        n = 0.85,
    )
```

Parameters for the single exponential down model of collisional energy transfer are usually obtained from analogous systems in literature. For example, if the user is interested in a pressure-dependent network with overall molecular formula C7H8, the single exponential down parameters for toluene in helium availabe from literature could be used for all unimolecular isomers in the network (assuming helium is the bath gas). One helpful literature source for calculated exponential down parameters is the following paper: http://www.sciencedirect.com/science/article/pii/S1540748914001084#s0060

The following subsections describe how the remaining molecular properties can be input to the `species()` function using either Option #1 or #2 mentioned above.

Option #1: Automatically Parse Quantum Chemistry Calculation Output

For this option, the `species()` function only requires two parameters in addition to the four parameters in the table above, as shown in the example below:

```
species('acetylperoxy',''acetylperoxy.py',
        structure = SMILES('CC(=O)O[O]'),
        molecularWeight = (75.04,"g/mol"),
        collisionModel = TransportData(sigma=(5.09,'angstrom'), epsilon=(473,'K')),
        energyTransferModel = SingleExponentialDown(
            alpha0 = (0.5718,'kcal/mol'),
            T0 = (300,'K'),
            n = 0.85,
        ),
)
```

The first parameter (`'acetylperoxy'` above) is the species label, which can be referenced later in the input file. The second parameter (`'acetylperoxy.py'` above) points to the location of another python file containing details of the species. This file will be referred to as the species input file.

The species input file accepts the following parameters:

Parameter	Required?	Description
bonds	optional	Type and number of bonds in the species
linear	yes	`True` if the molecule is linear, `False` if not
externalSymmetry	yes	The external symmetry number for rotation
spinMultiplicity	yes	The ground-state spin multiplicity (degeneracy)
opticalIsomers	yes	The number of optical isomers of the species
energy	yes	The ground-state 0 K atomization energy in Hartree (without zero-point energy) **or** The path to the quantum chemistry output file containing the energy
geometry	yes	The path to the quantum chemistry output file containing the optimized geometry
frequencies	yes	The path to the quantum chemistry output file containing the computed frequencies
rotors	optional	A list of `HinderedRotor()` and/or `FreeRotor()` objects describing the hindered/free rotors

The types and number of atoms in the species are automatically inferred from the quantum chemistry output and are used to apply atomization energy corrections (AEC) and spin orbit corrections (SOC) for a given `modelChemistry` (see *Model Chemistry*). If not interested in accurate thermodynamics (e.g., if only using `kinetics()`), then atom corrections can be turned off by setting `useAtomCorrections` to `False`.

The `bond` parameter is used to apply bond corrections (BC) for a given `modelChemistry`.

Allowed bond types for the `bonds` parameter are, e.g., `'C-H'`, `'C-C'`, `'C=C'`, `'N-O'`, `'C=S'`, `'O=O'`, `'C#N'`...

`'O=S=O'` is also allowed.

The order of elements in for the bond correction is not important. Use `-`/`=`/`#` to denote a single/double/triple bond, respectively. For example, for acetylperoxy radical we would write:

```
bonds = {'C-C': 1, 'C=O': 1, 'C-O': 1, 'O-O': 1, 'C-H': 3}
```

The parameter `linear` only needs to be specified as either `True` or `False`. The parameters `externalSymmetry`, `spinMultiplicity` and `opticalIsomers` only accept integer values. Note that `externalSymmetry` corresponds to the number of unique ways in which the species may be rotated about an axis (or multiple axes) and still be indistinguishable from its starting orientation (reflection across a mirror plane does not count as rotation about an axis). For acetylperoxy radical, we would write:

```
linear = False

externalSymmetry = 1

spinMultiplicity = 2

opticalIsomers = 1
```

The `energy` parameter is a dictionary with entries for different `modelChemistry`. The entries can consist of either floating point numbers corresponding to the 0 K atomization energy in Hartree (without zero-point energy correction), or they can specify the path to a quantum chemistry calculation output file that contains the species's energy. For example:

```
energy = {
'CBS-QB3': Log('acetylperoxy_cbsqb3.log'),
'Klip_2': -79.64199436,
}
```

In this example, the `CBS-QB3` energy is obtained from a Gaussian log file, while the `Klip_2` energy is specified

directly. The energy used will depend on what `modelChemistry` was specified in the input file. Arkane can parse the energy from a Gaussian, Molpro, or QChem log file, all using the same `Log` class, as shown below.

The input to the remaining parameters, `geometry`, `frequencies` and `rotors`, will depend on if hindered/free rotors are included. Both cases are described below.

Without Hindered/Free Rotors

In this case, only `geometry` and `frequencies` need to be specified, and they can point to the same or different quantum chemistry calculation output files. The `geometry` file contains the optimized geometry, while the `frequencies` file contains the harmonic oscillator frequencies of the species in its optimized geometry. For example:

```
geometry = Log('acetylperoxy_cbsqb3.log')

frequencies = Log('acetylperoxy_freq.log')
```

In summary, in order to specify the molecular properties of a species by parsing the output of quantum chemistry calculations, without any hindered/free rotors, the species input file should look like the following (using acetylperoxy as an example):

```
bonds = {
'C-C': 1,
'C=O': 1,
'C-O': 1,
'O-O': 1,
'C-H': 3,
}

linear = False

externalSymmetry = 1

spinMultiplicity = 2

opticalIsomers = 1

energy = {
    'CBS-QB3': Log('acetylperoxy_cbsqb3.log'),
    'Klip_2': -79.64199436,
}

geometry = Log('acetylperoxy_cbsqb3.log')

frequencies = Log('acetylperoxy_freq.log')
```

With Hindered/Free Rotors

In this case, `geometry`, `frequencies` and `rotors` need to be specified. Note that the `geometry` and `frequencies` parameters must point to the **same** quantum chemistry calculation output file. For example:

```
geometry = Log('acetylperoxy_freq.log')

frequencies = Log('acetylperoxy_freq.log')
```

The `geometry/frequencies` log file must contain both the optimized geometry and the Hessian (matrix of partial second derivatives of potential energy surface, also referred to as the force constant matrix), which is used to calculate the harmonic oscillator frequencies. If Gaussian is used to generate the `geometry/frequencies` log file, the Gaussian input file must contain the keyword `iop(7/33=1)`, which forces Gaussian to output the complete Hessian. Because the `iop(7/33=1)` option is only applied to the first part of the Gaussian job, the job must be a `freq` job only (as opposed to an `opt freq` job or a composite method job like `cbs-qb3`, which only do the `freq` calculation after the optimization). Therefore, the proper workflow for generating the `geometry/frequencies` log file using Gaussian is:

1. Perform a geometry optimization.

2. Take the optimized geometry from step 1, and use it as the input to a `freq` job with the following input keywords:
   `#method basis-set freq iop(7/33=1)`

The output of step 2 is the correct log file to use for `geometry/frequencies`.

`rotors` is a list of `HinderedRotor()` and/or `FreeRotor()` objects. Each `HinderedRotor()` object requires the following parameters:
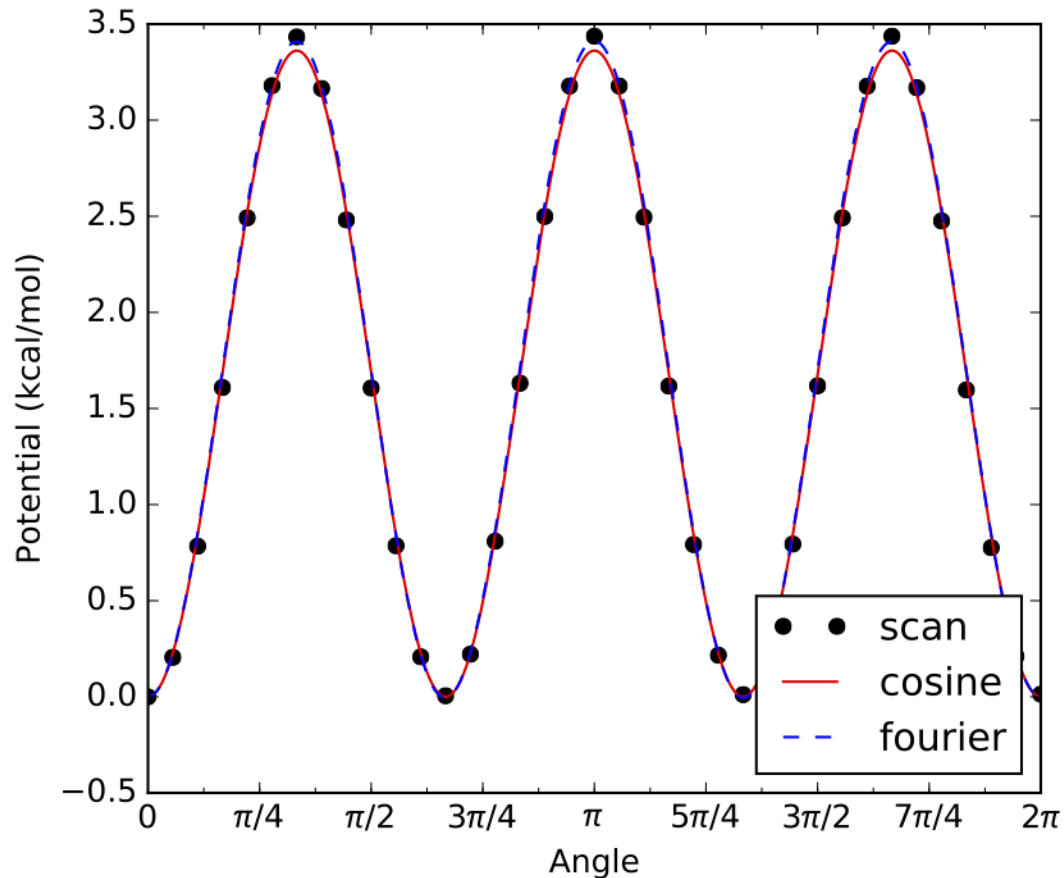
Parameter	Description
`scanLog`	The path to the Gaussian/Qchem log file or text file containing the scan
`pivots`	The indices of the atoms in the hindered rotor torsional bond
`top`	The indices of all atoms on one side of the torsional bond (including the pivot atom)
`symmetry`	The symmetry number for the torsional rotation (number of indistinguishable energy minima)
`fit`	Fit to the scan data. Can be either `fourier`, `cosine` or `best` (default).

`scanLog` can either point to a `Log` file, or simply a `ScanLog`, with the path to a text file summarizing the scan in the following format:

```
Angle (radians)          Energy (kJ/mol)
 0.0000000000             0.0147251160
 0.1745329252             0.7223109804
 0.3490658504             2.6856059517
       .                        .
       .                        .
       .                        .
 6.2831853072             0.0000000000
```
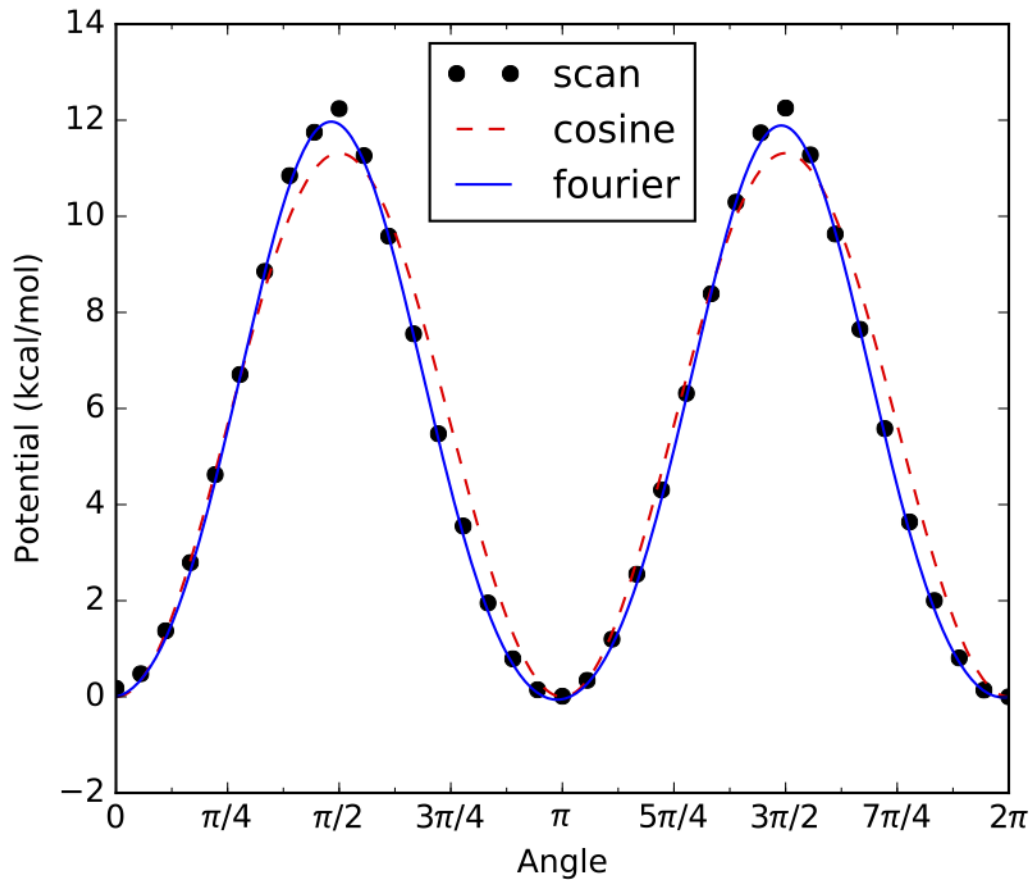
The `Energy` can be in units of `kJ/mol`, `J/mol`, `cal/mol`, `kcal/mol`, `cm^-1` or `hartree`.

The `symmetry` parameter will usually equal either 1, 2 or 3. Below are examples of internal rotor scans with these commonly encountered symmetry numbers. First, `symmetry = 3`:
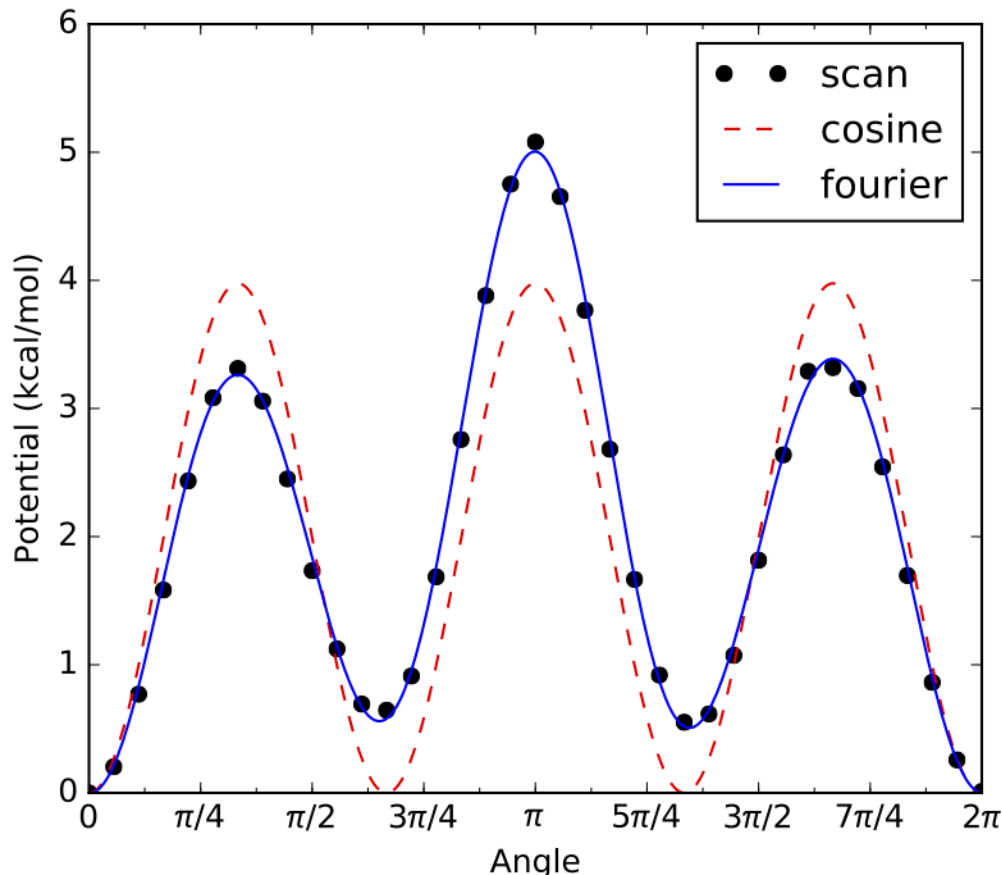
Internal rotation of a methyl group is a common example of a hindered rotor with `symmetry = 3`, such as the one above. As shown, all three minima (and maxima) have identical energies, hence `symmetry = 3`.

Similarly, if there are only two minima along the internal rotor scan, and both have identical energy, then `symmetry = 2`, as in the example below:

If any of the energy minima in an internal rotor scan are not identical, then the rotor has no symmetry (`symmetry = 1`), as in the example below:

For the example above there are 3 local energy minima, 2 of which are identical to each other. However, the 3rd minima is different from the other 2, therefore this internal rotor has no symmetry.

For practical purposes, when determining the symmetry number for a given hindered rotor simply check if the internal rotor scan looks like the `symmetry = 2` or `3` examples above. If it doesn't, then most likely `symmetry = 1`.

Each `FreeRotor()` object requires the following parameters:

Parameter	Description
`pivots`	The indices of the atoms in the free rotor torsional bond
`top`	The indices of all atoms on one side of the torsional bond (including the pivot atom)
`symmetry`	The symmetry number for the torsional rotation (number of indistinguishable energy minima)

Note that a `scanLog` is not needed for `FreeRotor()` because it is assumed that there is no barrier to internal rotation. Modeling an internal rotation as a `FreeRotor()` puts an upper bound on the impact of that rotor on the species's overall partition function. Modeling the same internal rotation as a Harmonic Oscillator (default if it is not specifed as either a `FreeRotor()` or `HinderedRotor()`) puts a lower bound on the impact of that rotor on the species's overall partition function. Modeling the internal rotation as a `HinderedRotor()` should fall in between these two extremes.

To summarize, the species input file with hindered/free rotors should look like the following example (different options for specifying the same `rotors` entry are commented out):

```
bonds = {
'C-C': 1,
```

```
'C=O': 1,
'C-O': 1,
'O-O': 1,
'C-H': 3,
}

linear = False

externalSymmetry = 1

spinMultiplicity = 2

opticalIsomers = 1

energy = {
    'CBS-QB3': Log('acetylperoxy_cbsqb3.log'),
    'Klip_2': -79.64199436,
}

geometry = Log('acetylperoxy_freq.log')

frequencies = Log('acetylperoxy_freq.log')

rotors = [
    HinderedRotor(scanLog=Log('acetylperoxy_scan_1.log'), pivots=[1,5], top=[1,2,3,4],
→symmetry=3, fit='best'),
    #HinderedRotor(scanLog=ScanLog('acetylperoxy_rotor_1.txt'), pivots=[1,5], top=[1,2,3,4],
→symmetry=3, fit='best'),
    #FreeRotor(pivots=[1,5], top=[1,2,3,4], symmetry=3),
]
```

Note that the atom labels identified within the rotor section should correspond to the indicated geometry.

Option #2: Directly Enter Molecular Properties

While it is usually more convenient to have Arkane parse molecular properties from the output of quantum chemistry calculations (see *Option #1: Automatically Parse Quantum Chemistry Calculation Output*) there are instances where an output file is not available and it is more convenient for the user to directly enter the molecular properties. This is the case, for example, if the user would like to use calculations from literature, where the final calculated molecular properties are often reported in a table (e.g., vibrational frequencies, rotational constants), but the actual output file of the underlying quantum chemistry calculation is rarely provided.

For this option, there are a number of required and optional parameters associated with a `species()` function:

Parameter	Required?	Description
`label`	all species	A unique string label used as an identifier
`structure`	all species except bath gas	A chemical structure for the species defined using either SMILES, adjacencyList, or InChI
`E0`	all species	The ground-state 0 K enthalpy of formation (including zero-point energy)
`modes`	all species	The molecular degrees of freedom (see below)
`spinMultiplicity`	all species	The ground-state spin multiplicity (degeneracy), sets to 1 by default if not used
`opticalIsomers`	all species	The number of optical isomers of the species, sets to 1 by default if not used
`molecularWeight`	all species	The molecular weight, if not given it is calculated based on the structure
`collisionModel`	unimolecular isomers and bath gases	Transport data for the species
`energyTransferModel`	unimolecular isomers	Assigned with `SingleExponentialDown` model
`thermo`	optional	Thermo data for the species

The parameters `structure`, `molecularWeight`, `collisionModel` and `energyTransferModel` were already discussed above in *Species Parameters*.

The `label` parameter should be set to a string with the desired name for the species, which can be reference later in the input file.

```
label = 'acetylperoxy'
```

The `E0` ground state 0 K enthalpy of formation (including zero-point energy) should be given in the quantity format (`value`, `'units'`), using units of either `kJ/mol`, `kcal/mol`, `J/mol`, or `cal/mol`:

```
E0 = (-34.6,'kcal/mol')
```

Note that if Arkane is being used to calculate the thermochemistry of the species, it is critical that the value of `E0` is consistent with the definition above (0 K enthalpy of formation with zero-point energy). However, if the user is only interested in kinetics, `E0` can be defined on any arbitrary absolute energy scale, as long as the correct relative energies between various `species()` and `transitionState()` are maintained. For example, it is common in literature for the energy of some reactant(s) to be arbitrarily defined as zero, and the energies of all transition states, intermediates and products are reported relative to that.

Also note that the value of `E0` provided here will be used directly, i.e., no atom or bond corrections will be applied.

When specifying the `modes` parameter, define a list with the following types of degrees of freedom. To understand how to define these degrees of freedom, please click on the links below:

Translational degrees of freedom

Class	Description
`IdealGasTranslation`	A model of three-dimensional translation of an ideal gas

Rotational degrees of freedom

Class	Description
`LinearRotor`	A model of two-dimensional rigid rotation of a linear molecule
`NonlinearRotor`	A model of three-dimensional rigid rotation of a nonlinear molecule
`KRotor`	A model of one-dimensional rigid rotation of a K-rotor
`SphericalTopRotor`	A model of three-dimensional rigid rotation of a spherical top molecule

Vibrational degrees of freedom

Class	Description
HarmonicOscillator	A model of a set of one-dimensional harmonic oscillators

Note that the `frequencies` provided here will be used directly, i.e., the `frequencyScaleFactor` will not be applied.

Torsional degrees of freedom

Class	Description
HinderedRotor	A model of a one-dimensional hindered rotation
FreeRotor	A model of a one-dimensional free rotation

The `spinMultiplicity` is defined using an integer, and is set to 1 if not indicated in the `species()` function.

```
spinMultiplicity = 2
```

Similarly, the `opticalIsomers` is also defined using an integer, and is set to 1 if not used in the `species()` function.

```
opticalIsomers = 1
```

The following is an example of a typical species item, based on the acetylperoxy radical $CH_3C(=O)OO \cdot$ (different options for specifying the same internal rotation are commented out):

```
species(
    label = 'acetylperoxy',
    structure = SMILES('CC(=O)O[O]'),
    E0 = (-34.6,'kcal/mol'),
    modes = [
        IdealGasTranslation(mass=(75.04,"g/mol")),
        NonlinearRotor(inertia=([54.2977,104.836,156.05],"amu*angstrom^2"), symmetry=1),
        HarmonicOscillator(frequencies=([319.695,500.474,536.674,543.894,727.156,973.365,1037.
→77,1119.72,1181.55,1391.11,1449.53,1454.72,1870.51,3037.12,3096.93,3136.39],"cm^-1")),
        HinderedRotor(inertia=(7.38359,"amu*angstrom^2"), symmetry=1, fourier=([[-1.95191,-11.
→8215,0.740041,-0.049118,-0.464522],[0.000227764,0.00410782,-0.000805364,-0.000548218,-0.
→000266277]],"kJ/mol")),
        HinderedRotor(inertia=(2.94723,"amu*angstrom^2"), symmetry=3, fourier=([[0.130647,0.
→0401507,-2.54582,-0.0436065,-0.120982],[-0.000701659,-0.000989654,0.00783349,-0.00140978,-0.
→00145843]],"kJ/mol")),
        #FreeRotor(inertia=(7.38359,"amu*angstrom^2"), symmetry=1),
        #FreeRotor(inertia=(2.94723,"amu*angstrom^2"), symmetry=3),
    ],
    spinMultiplicity = 2,
    opticalIsomers = 1,
    molecularWeight = (75.04,"g/mol"),
    collisionModel = TransportData(sigma=(5.09,'angstrom'), epsilon=(473,'K')),
    energyTransferModel = SingleExponentialDown(
        alpha0 = (0.5718,'kcal/mol'),
        T0 = (300,'K'),
        n = 0.85,
    ),
)
```

Note that the format of the `species()` function above is identical to the `conformer()` function output by Arkane in `output.py`. Therefore, the user could directly copy the `conformer()` output of an Arkane job to another Arkane

input file, change the name of the function to `species()` (or `transitionState()`, if appropriate, see next section) and run a new Arkane job in this manner. This can be useful if the user wants to easily switch a `species()` function from Option #1 (parsing quantum chemistry calculation output) to Option #2 (directly enter molecular properties).

2.4.5 Transition States

Transition states for reactions in the pressure dependent network should be defined very similarly to `species` using a `transitionState()` function, however it has less parameters (`structure`, `molecularWeight`, `collisionModel` and `energyTransferModel` aren't specified for a transition state). Like the `species()` function, the `transitionState()` function may also be specified in two ways: *Option #1: Automatically Parse Quantum Chemistry Calculation Output* and *Option #2: Directly Enter Molecular Properties*

The following is an example of a typical `transitionState()` function using Option #1:

```
transitionState('isom1', 'isom1.py')
```

Just as for a `species()` function, the first parameter is the label for that transition state, and the second parameter points to the location of another python file containing details of the transition state. This file will be referred to as the transition state input file, and it accepts the same parameters as the species input file described in *Option #1: Automatically Parse Quantum Chemistry Calculation Output*.

The following is an example of a typical `transitionState()` function using Option #2:

```
transitionState(
    label = 'isom1',
    E0 = (-5.8,'kcal/mol'),
    modes = [
        IdealGasTranslation(mass=(75.04,"g/mol")),
        NonlinearRotor(inertia=([49.3418,103.697,149.682],"u*angstrom**2"), symmetry=1,
→quantum=False),
        HarmonicOscillator(frequencies=([148.551,306.791,484.573,536.709,599.366,675.538,832.
→594,918.413,1022.28,1031.45,1101.01,1130.05,1401.51,1701.26,1844.17,3078.6,3163.07],"cm^-1"),
→quantum=True),
    ],
    spinMultiplicity = 2,
    opticalIsomers = 1,
    frequency = (-1679.04,'cm^-1'),
)
```

The only additional parameter required for a `transitionState()` function as compared to a `species()` function is `frequency`, which is the imaginary frequency of the transition state needed to account for tunneling. Refer to *Option #2: Directly Enter Molecular Properties* for a more detailed description of the other parameters.

2.4.6 Path Reactions

Each path reaction - a reaction directly connecting two molecular configurations in the network - is specified using a `reaction()` function. The following parameters are available:

Parameter	Required?	Description
`label`	All reactions	A name for the reaction
`reactants`	All reactions	A list of reactant species
`products`	All reactions	A list of product species
`transitionState`	All reactions	The transition state
`kinetics`	Optional	The high pressure-limit kinetics for the reaction
`tunneling`	Optional	The type of tunneling model (either 'Eckhart' or 'Wigner') to use for tunneling through the reaction barrier

If the optional `kinetics` parameter is specified, Arkane will perform an inverse Laplace transform (ILT) on the high pressure-limit kinetics provided to estimate the microcanonical rate coefficients, $k(E)$, needed for the master equation (refer to Theory manual for more detail). This feature is useful for barrierless reactions, such as radical recombinations, which don't have an obvious transition state. If the ILT approach to calculating $k(E)$ is taken, a placeholder `transitionState` must still be defined with an `E0` equal to the energy of the higher energy species it is connecting.

A typical `reaction()` function might look like this.

```
reaction(
    label = 'isom1',
    reactants = ['acetylperoxy'],
    products = ['hydroperoxylvinoxy'],
    transitionState = 'isom1',
    kinetics = Arrhenius(A=(2.65e6,'m^3/(mol*s)'), n=0.0, Ea=(0.0,'kcal/mol'), T0=(1,"K")),
    tunneling = 'Eckart',
)
```

Note that the reactants and products must have been previously declared using a `species()` function, using the same name labels. Transition states must also be previously declared using a `transitionState()` function.

If the optional `kinetics` entry is not specified, Arkane will calculate the required kinetic coefficients on its own. The `kinetics` entry is particularly useful to specify rates of barrierless reactions (for which Arkane cannot yet calculate high-pressure limit rates).

Currently, the `reaction()` function for a pressure-dependent job cannot connect bimolecular reactants to bimolecular products (e.g., as in a hydrogen abstraction or disproportionation reaction).

2.4.7 Network

A declaration for the overall network must be given using the `network` function.

This includes setting the following paramters:

Parameter	Description
`label`	A name for the network
`isomers`	A list of species participating in unimolecular reaction channels
`reactants`	A list of the species that participate in bimolecular reactant channels
`bathGas`	A dictionary of bath gases and their respective mole fractions, adding up to 1.0

Arkane is largely able to determine the molecular configurations that define the potential energy surface for your reaction network simply by inspecting the path reactions. However, you must indicate which unimolecular and bimolecular configurations you wish to include in the master equation formulation; all others will be treated as irreversible sinks.

Note that all species and bath gases used in the `network` function must have been previously declared with the same name labels in a previous `species` function in the input file.

You do not need to specify the product channels (infinite sinks) in this manner, as any configuration not marked as an isomer or reactant channel will be treated as a product channel.

An example of the `network` function is given below along with a scheme of the network:

```
network(
    label = 'acetyl + O2',
    isomers = [
        'acetylperoxy',
        'hydroperoxylvinoxy',
    ],
    reactants = [
        ('acetyl', 'oxygen'),
    ],
    bathGas = {
        'nitrogen': 0.4,
        'argon': 0.6,
    }
)
```
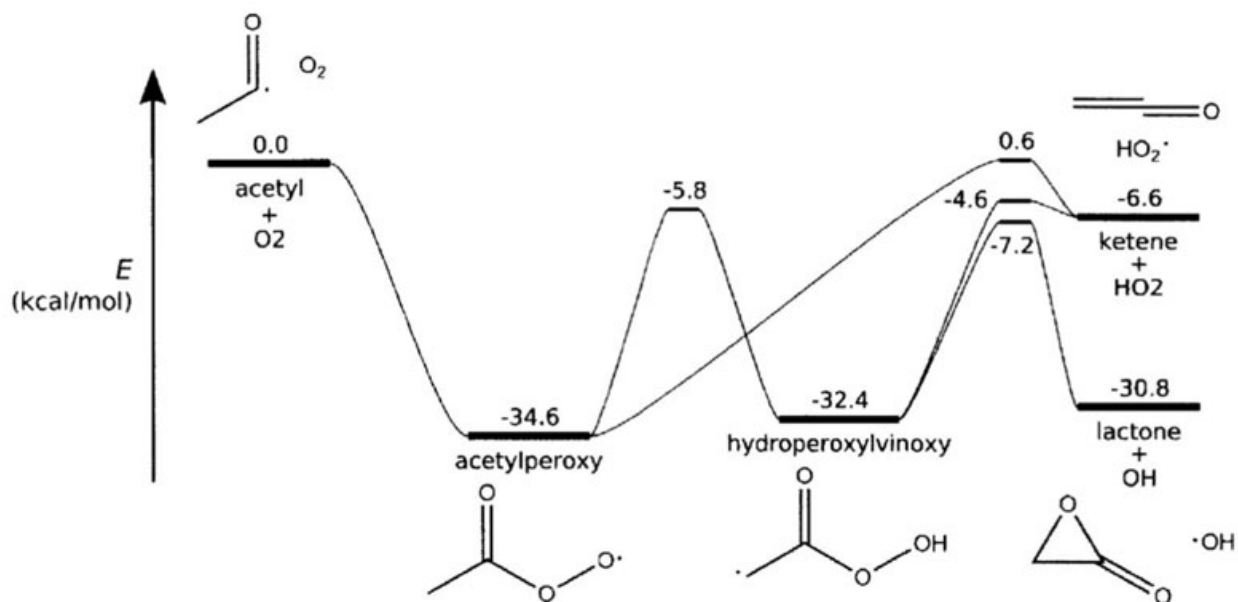

Image source: J.W. Allen, PhD dissertation, MIT 2013, calculated at the RQCISD(T)/CBS//B3LYP/6-311++G(d,p) level of theory

2.4.8 Algorithm Parameters

The overall parameters for the pressure-dependence calculation must be defined in a `pressureDependence` function at the end of the input file. The following parameters are necessary:

Parameter	Description
`label`	Use the name for the `network` declared previously
`method`	Method to use for calculating the pdep network. Use either `'modified strong collision'`, `'reservoir state'`, or `'chemically-significant eigenvalues'`
`interpolationModel`	Select the output type for the pdep kinetics, either in `'chebyshev'` or `'pdeparrhenius'` (plog) format
`activeKRotor`	A flag indicating whether to treat the K-rotor as active or adiabatic
`activeJRotor`	A flag indicating whether to treat the J-rotor as active or adiabatic
`Tmin/Tmax/Tcount` **or** `Tlist`	Define temperatures at which to compute (and output) $k(T, P)$
`Pmin/Pmax/Pcount` **or** `Plist`	Define pressures at which to compute (and output) $k(T, P)$
`maximumGrainSize` **and** `minimumGrainCount`	Defines fineness of energy grains used in master equation calculations.
`sensitivity_conditions`	Specifies the conditions at which to run a network sensitivity analysis.

Temperature and Pressure Ranges

Arkane will compute the $k(T, P)$ values on a grid of temperature and pressure points. `Tmin`, `Tmax`, and `Tcount` values, as well as `Pmin`, `Pmax`, and `Pcount` parameter values must be provided. Arkane will automatically choose the intermediate temperatures based on the interpolation model you wish to fit. This is the recommended approach.

Alternatively, the grid of temperature and pressure points can be specified explicitly using `Tlist` and/or `Plist`.

Energy Grains Determine the fineness of the energy grains to be used in the master equation calculations. Dictate the `maximumGrainSize`, and the `minimumGrainCount`.

An example of the algorithm parameters function for the acetyl + O2 network is shown below. This example also includes the `sensitivity_conditions` attribute which invokes a sensitivity analysis calculation:

```
pressureDependence(
    label='acetyl + O2',
    Tmin=(300.0,'K'), Tmax=(2000.0,'K'), Tcount=8,
    Pmin=(0.01,'bar'), Pmax=(100.0,'bar'), Pcount=5,
    #Tlist = ([300, 400, 600, 800, 1000, 1250, 1500, 1750, 2000],'K')
    #Plist = ([0.01, 0.1, 1.0, 10.0, 100.0],'bar')
    maximumGrainSize = (1.0,'kcal/mol'),
    minimumGrainCount = 250,
    method = 'modified strong collision',
    #method = 'reservoir state',
    #method = 'chemically-significant eigenvalues',
    interpolationModel = ('chebyshev', 6, 4),
    #interpolationModel = ('pdeparrhenius'),
    #activeKRotor = True,
    activeJRotor = True,
    sensitivity_conditions = [[(1000, 'K'), (1, 'bar')], [(1500, 'K'), (10, 'bar')]]
)
```

The output of a sensitivity analysis is saved into a `sensitivity` folder in the output directory. A text file, named with the network label, delineates the semi-normalized sensitivity coefficients `dln(k)/dE0` in units of `mol/J` for all network reactions (both directions if reversible) at all requested conditions. Horizontal bar figures are automatically generated per network reaction, showing the semi-normalized sensitivity coefficients at all conditions.

2.4.9 Thermodynamics Computations

The input to the `thermo()` function is identical to that of a non-pressure-dependent job.

Use a `thermo()` function to make Arkane execute the thermodynamic parameters computation for a species. Pass the string label of the species you wish to compute the thermodynamic parameters for and the type of thermodynamics polynomial to generate (either `'Wilhoit'` or `'NASA'`). A table of relevant thermodynamic parameters will also be displayed in the output file.

Below is a typical `thermo()` execution function:

```
thermo('ethane', 'NASA')
```

2.4.10 Kinetics Computations

The input to the `kinetics()` function is identical to that of a non-pressure-dependent job. Note, however, that because currently the `reaction()` function for a pressure-dependent job cannot connect bimolecular reactants to bimolecular products (e.g., as in a hydrogen abstraction or disproportionation reaction), it is also not possible to use `kinetics()` on such a reaction either. The kinetics of such a reaction can only be calculated as part of a non-pressure-dependent job.

Use a `kinetics()` function to make Arkane execute the high-pressure limit kinetic parameters computation for a reaction. The `'label'` string must correspond to that of a defined `reaction()` function. If desired, define a temperature range and number of temperatures at which the high-pressure rate coefficient will be tabulated and saved to the outupt file. The 3-parameter modified Arrhenius coefficients will automatically be fit to the computed rate coefficients. The quantum tunneling factor will also be displayed.

Below is a typical `kinetics()` function:

```
kinetics(
label = 'H + C2H4 <=> C2H5',
Tmin = (400,'K'), Tmax = (1200,'K'), Tcount = 6,
)
```

If specific temperatures are desired, you may specify a list (`Tlist = ([400,500,700,900,1100,1200],'K')`) instead of Tmin, Tmax, and Tcount.

This is also acceptable:

```
kinetics('H + C2H4 <=> C2H5')
```

2.4.11 Examples

Perhaps the best way to learn the input file syntax is by example. To that end, a number of example input files and their corresponding output have been given in the `examples/arkane/networks` directory, which includes several examples.

2.5 Creating Input Files for Automated Pressure Dependent Network Exploration

2.5.1 Syntax

Network exploration starts from a pressure dependent calculation job so naturally it requires an input file containing everything required for a pressure-dependent calculation input file. In addition all species blocks must have a structure input and there must be a database block in the job. At the end of the pressure-dependent calculation input file you append an explorer block. For example

```
explorer(
    source=['methoxy'],
    explore_tol=(1e-2,'s^-1'),
    energy_tol=1e4,
    flux_tol=1e-6,
)
```

The `source` is a list containing either the label for a single isomer or the labels corresponding to a bimolecular source channel. The network is expanded starting from this starting isomer/channel.

2.5.2 Network Exploration

The `explore_tol` is largest acceptable total rate at which isomers inside the network isomerize to become isomers that are not part of the network. Network expansion is done starting from just the network source using values from the rest of the Arkane job when available, otherwise from RMG. It cycles through all of the temperature and pressure points specified for fitting in the pressure dependence job and checks the total network leak rate at each one. Whenever this rate is greater than `explore_tol` the outside isomer with the most leak is added to the network and reacted and the loop is flagged to cycle through all of the temperatures and pressures again. Once this loop is finished a network_full.py file is generated in the pdep directory that has the full explored network.

2.5.3 Network Reduction

`energy_tol` and `flux_tol` control reduction of the full network. This is highly recommended as the full model will often contain many unimportant reactions. How strict you decide to set these parameters should be related to how much you trust the thermo and rate information being used in the job.

`energy_tol` is related to the maximum difference in E0 allowed between a reaction TS and the source channel/isomer. A given reaction is deemed filterable at a given condition if `energy_tol` * RT < E0_TS - E0_source. For example, if T = 1000 K and `energy_tol` is set to 100 and the source energy is `100 (kJ mol^-1)`, all reactions with a transition state energy more than `100 * 8.314 (J K^-1 mol^-1) * 1000 K + 100 (kJ mol^-1) = 931.4 (kJ mol^-1)` will not be considered in the final network.

`flux_tol` is related to the solution of a steady state problem. In this problem a constant flux of 1.0 (choice of units irrelevant) is applied to the source channel/isomer while all A => B + C reactions are assumed to be irreversible. Under these conditions steady state concentrations of each isomer and the rates of every unimolecular reaction can be calculated. Since the input flux to the system is 1.0 to some degree these rates represent the fraction of the input flux passing through that reaction. When the flux through a reaction is less than `flux_tol` it is deemed filterable at that condition.

Overall a reaction is removed from the network if it is deemed filterable by both methods (if both specified) at every temperature and pressure combination. After reduction a network_reduced.py file is generated in the pdep directory containing the reduced network.

2.6 Running Arkane

To execute an Arkane job, invoke the command

```
$ python arkane.py INPUTFILE
```

The absolute or relative paths to the arkane.py file as well as to the input file must be given.

The job will run and the results will be saved to `output.py` in the same directory as the input file. If you wish to save the output elsewhere, use the `-o/--output` option, e.g.

```
$ python arkane.py INPUTFILE -o OUTPUTFILE
```

2.6.1 Drawing Potential Energy Surface

Arkane contains functionality for automatically generating an image of the potential energy surface for a reaction network. This is done automatically and outputted in pdf format to a file called `network.pdf`.

2.6.2 Log Verbosity

You can manipulate the amount of information logged to the console window using the `-q/--quiet` flag (for quiet mode) or the `-v/--verbose` flag (for verbose mode). The former causes the amount of logging information shown to decrease; the latter causes it to increase.

2.6.3 Help

To view help information and all available options, use the `-h/--help` flag, e.g.

```
$ python arkane.py -h
```

2.7 Parsing Output Files

2.7.1 Thermodynamic and High-pressure Limit Kinetics Calculations

The syntax of Arkane output files closely mirrors that of the input files. For each `thermo()` function in the input file, there will be a corresponding `thermo()` function in the output file containing the computed thermodynamic model. Similarly, For each `kinetics()` function in the input file, there will be a corresponding `kinetics()` function in the output file containing the computed kinetics model.

2.7.2 Pressure-Dependent Calculations

The output file contains the entire contents of the input file. In addition, the output file contains a block of `pdepreaction()` calls. The parameters of each `pdepreaction()` block match those of the `reaction()` block from the input file, except that no transition state data is given and the `kinetics` are by definition pressure-dependent.

A `pdepreaction()` item is printed for each reaction pathway possible in the network. Each reaction is reversible. Reactions in the opposite direction are provided as commented out, so a user can choose to use them if she/he desires.

2.7.3 Chemkin Output File

In addition to the `output.py` which contains the thermodynamic, kinetic, and pressure dependent results from an Arkane run, a Chemkin input file, `chem.inp` is also returned. This file contains species and their thermodynamic parameters for each species that has the `thermo()` in the input file. The file also contains kinetics, both pressure dependent and high pressure limit, which have the `kinetics()` or `pressureDependence()` module called.

For the output file to function, all the names of species should be in valid chemkin format. The butanol and ethyl examples both show how to obtain a valid chemkin file.

The `chem.inp` file can be used in Chemkin software package or converted to a Cantera input file for use in Cantera software.

2.7.4 Log File

A log file containing similar information to that displayed on the console during Arkane execution is also automatically saved. This file has the name `Arkane.log` and is found in the same directory as the output file. The log file accepts logging messages at an equal or greater level of detail than the console; thus, it is often useful (and recommended) to examine both if something unexpected has occurred.

The `examples/arkane` directory contains both Arkane input files and the resulting output files.

2.7.5 Species Dictionary

Any species that had the `thermo()` method called and had the structure defined in the Arkane input file will also have an RMG style adjacency list representation in `species_dictionary.txt`. This allows the user to input the corresponding thermo and kinetics into RMG in various ways described in the RMG user guide.

2.8 Frequently Asked Questions

Are there other software packages for investigating pressure-dependent reaction networks?

Yes. The following is an illustrative list of such packages:

Name	Method(s)	Language	Author(s)
MultiWell	stochastic	Fortran	J. R. Barker *et al*
UNIMOL	CSE	Fortran	R. G. Gilbert, S. C. Smith
ChemRate	CSE	C++[1]	V. Mokrushin, W. Tsang
Variflex	CSE	Fortran	S. J. Klippenstein *et al*
MESMER	CSE (+ RS)	C++	S. H. Robertson *et al*
CHEMDIS[2]	MSC	Fortran	A. Y. Chang, J. W. Bozzelli, A. M. Dean

(MSC = modified strong collision, RS = reservoir state, CSE = chemically-significant eigenvalues)

Many of the above packages also provide additional functionality beyond the approximate solving of the master equation. For example, Variflex can be used for variational transition state theory calculations, while ChemRate provides a (Windows) graphical user interface for exploring a database of experimental data and physical quantities.

---

[1] Uses MFC for Windows graphical user interface

[2] No longer distributed

2.9 Credits

Author: Joshua W. Allen (joshua.w.allen@gmail.com)

P.I.: Prof. William H. Green (whgreen@mit.edu)

The author acknowledges the Green group for helping put the software through its paces and providing suggestions for its improvement.

Arkane is based upon work supported by the King Abdullah University of Science and Technology.
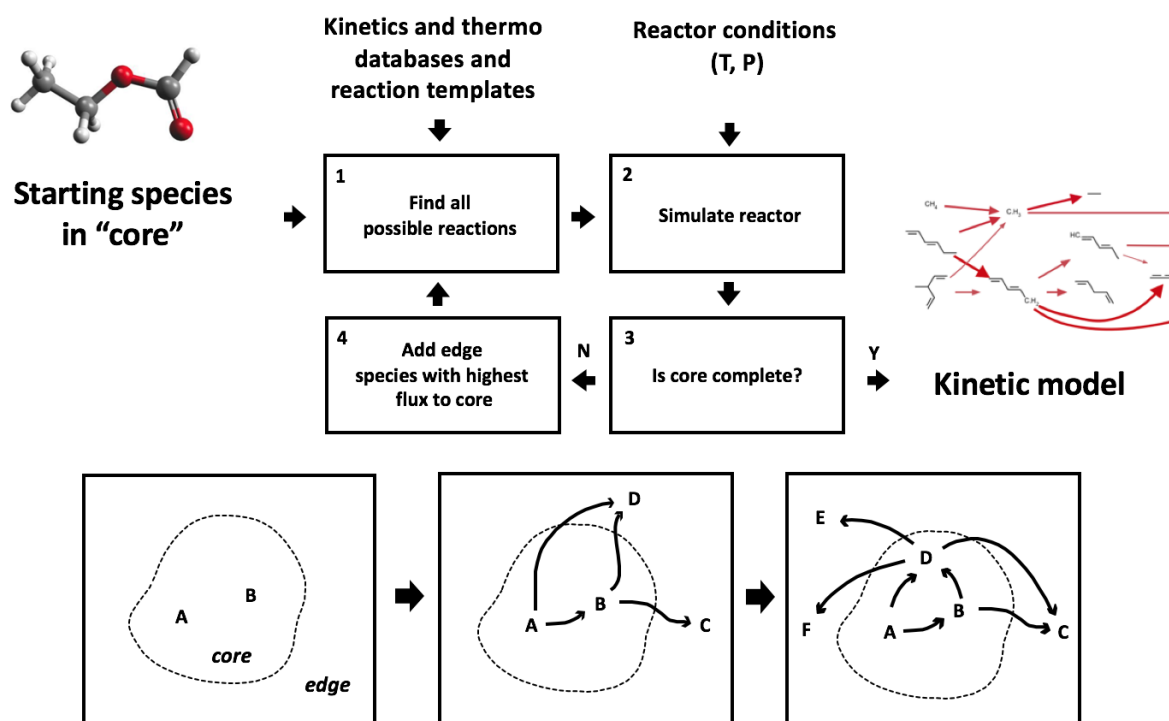
- genindex
- modindex
- search

THEORY GUIDE

The theoretical foundations to some of the parts of RMG and Arkane are described in greater detail in these sections.

3.1 RMG Theory Guide

3.1.1 Rate-based Model Enlarging Algorithm

To construct a mechanism, the user must specify an initial set of species and the initial conditions (temperature, pressure, species concentrations, etc.). RMG reacts the initial species in all possible ways according to its known reaction families, and it integrates the model in time. RMG tracks the rate (flux) at which each new "edge" species is produced, and species (and the reactions producing them) that are produced with significant fluxes are incorporated into the model (the "core"). These new core species are reacted with all other core species in the model to generate a new set of edge species and reactions. The time-integration restarts, and the expanded list of edge species is monitored for significant species to be included in the core. The process continues until all significant species and reactions have

been included in the model. The definition of a "significant" rate can be specified by the user by taking the following definition for a single species rate:

$$R_i = \frac{dC_i}{dt}$$

and the following definition for the reaction system's characteristic rate, which is the sum of all **core** species rates:

$$R_{char} = \sqrt{\sum_j R_j^2} \qquad \text{species } j \in \text{core}$$

When a species $i \in$ edge exceeds a "significant" rate equal to ϵR_{char}, it is added to the core. The parameter ϵ is the user-specified `toleranceMoveToCore` that can be adjusted under the *model tolerances* in the *RMG Input File*.

For more information on rate-based model enlargement, please refer to the papers [Gao2016] or [Susnow1997].

Filtering Reactions within the Rate-based Algorithm

Filtering reactions in the react step in the flux-based algorithm attempts to speed up model generation by attacking the pain point. RMG has trouble converging when generating models for large molecules because it searches for reactions on the order of $(n_{reaction\ sites})^{n_{species}}$.

The original algorithm performs in the following manner:

1. Reacts species together (slow)

2. Determines which reactions are negligible (fast)

By filtering reactions we add a pre-filtering step before step 1 which prevents species from reacting together when the reactions are expected to be negligible throughout the simulation.
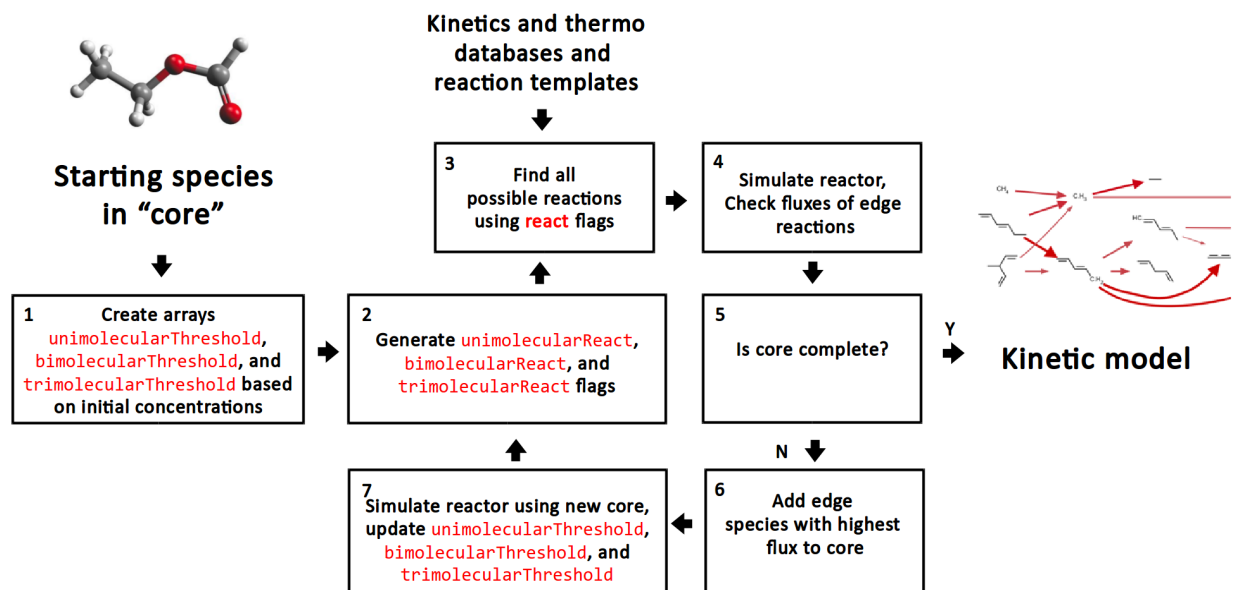
Here, `unimolecularThreshold`, `bimolecularThreshold`, and `trimolecularThreshold` are binary arrays storing flags for whether a species or a pair of species are above a reaction threshold. For a unimolecular rate, this threshold is set to `True` if the unimolecular rate of reaction k for a species A

$$R_{unimolecular} = k_{threshold} C_A > \epsilon R_{char}$$

at any given time t in the reaction system, where $k_{threshold} = \frac{k_B T}{h}$

For a bimolecular reaction occuring between species A and B, this threshold is set to `True` if the bimolecular rate

$$R_{bimolecular} = k_{threshold} C_A C_B > \epsilon R_{char}$$

where $k_{threshold} = filterThreshold$. `filterThreshold` is set by the user in the input file and its default value is $10^8 \frac{m^3}{mol \cdot s}$. This is on the same order of magnitude as the collision limit for two hydrogen atoms at 1000 K. In general, it is recommended to set `filterThreshold` such that $k_{threshold}$ is slightly greater than the maximum rate constants one expects to be present in the system of interest. This will ensure that very fast reactions are not accidentally filtered out.

Similarly, for a trimolecular reaction, the following expression is used:

$$R_{trimolecular} = k_{threshold} C_A C_B C_C > \epsilon R_{char}$$

where $k_{threshold} = 10^{-3} \cdot filterThreshold \frac{m^6}{mol^2 \cdot s}$. Based on extending Smoluchowski theory to multiple molecules, the diffusion limit rate constant for trimolecular reactions (in $\frac{m^6}{mol^2 \cdot s}$) is approximately three orders of magnitude smaller than the rate constant for bimolecular reactions (in $\frac{m^3}{mol \cdot s}$). It is assumed here that Smoluchowski theory gives a sufficient approximation to collision theory in the gas phase.

When the liquid-phase reactor is used, the diffusion limits are calculated using the Stokes-Einstein equation instead. For bimolecular reactions, this results in

$$k_{threshold}[m^3/mol/s] = 22.2 \frac{T[K]}{\mu[Pa \cdot s]}$$

and for trimolecular

$$k_{threshold}[m^6/mol^2/s] = 0.11 \frac{T[K]}{\mu[Pa \cdot s]}$$

where μ is the solvent viscosity. The coefficients in the above equations were obtained by using a representative value of the molecular radius of 2 Angstrom. More details on the calculation of diffusion limits in the liquid phase can be found in *the description of liquid-phase systems* under *diffusion-limited kinetics*.

Three additional binary arrays `unimolecularReact`, `bimolecularReact`, and `trimolecularReact` store flags for when the `unimolecularThreshold`, `bimolecularThreshold`, or `trimolecularThreshold` flag shifts from `False` to `True`. RMG reacts species when the flag is set to `True`.

3.1.2 Prune Edge Species

When dealing with complicated reaction systems, RMG calculation would easily hit the computer memory limitation. Memory profiling shows most memory especially during memory limitation stage is occupied by edge species. However, most edge species in fact wouldn't be included in the core (or final model). Thus, it's natural to get rid of some not "so useful" edge species during calculation in order to achieve both low memory consumption and mechanism accuracy. Pruning is such a way.

Key Parameters in Pruning

- toleranceKeepInEdge

  Any edge species to prune should have peak flux along the whole conversion course lower than toleranceKeepInEdge * characteristic flux. Thus, larger values will lead to smaller edge mechanisms.

- toleranceMoveToCore

  Any edge species to enter core model should have flux at some point larger than toleranceMoveToCore * characteristic flux Thus, in general, smaller values will lead to larger core mechanisms.

- toleranceInterrupSimulation

Once flux of any edge species exceeds toleranceInterruptSimulation ∗ characteristic flux, dynamic simulation will be stopped. Usually this tolerance will be set a very high value so that any flux's exceeding that means mechanism is too incomplete to continue dynamic simulation.

- maximumEdgeSpecies

If dynamic simulation isn't interrupted in half way and total number of the edge species whose peak fluxes are higher than toleranceKeepInEdge ∗ characteristic flux exceeds maximumEdgeSpecies, such excessive amount of edge species with lowest peak fluxes will be pruned.

- minCoreSizeForPrune

Ensures that a minimum number of species are in the core before pruning occurs, in order to avoid pruning the model when it is far away from completeness. The default value is set to 50 species.

- minSpeciesExistIterationsForPrune

Set the number of iterations an edge species must stay in the job before it can be pruned. The default value is 2 iterations.

How Pruning Works

3.1.3 Dynamics Criterion

When dealing with more complex chemical mechanisms the standard RMG flux criterion has trouble picking up key chain branching reactions and has limited guarantees that it accurately represents the concentrations of all species. The dynamics criterion is a measure of how much a given reaction affects core concentrations. This allows it to pick up key low-flux chain branching reactions and better represent species concentrations.

Calculating the Dynamics Criterion

Let us define rates of production $P_i(t)$ and consumption $L_i(t)$ for a given species $\frac{dc_i}{dt} = P_i(t) - L_i(t)$

Let us define a dimensionless concentration variable we will refer to as the accumulation number Ac for a given species

$Ac_{spc,i} = \frac{P_i}{L_i} \approx \frac{\bar{c}_i}{c_{i0}}$

where $\bar{c}_i$ is the steady state concentration or more specifically the concentration at which $P_i = L_i$ assuming L_i scales with c_i and c_{i0} is the current concentration.

This species accumulation number is a measure of how far species i is from steady state.

Since this number can only be calculated for core species, by itself it is only a measure of the behavior of species i within the reaction network.

However if we consider models with and without some edge reaction j we can define

$\Pi_{Ac,i,j} = \frac{Ac_{spc,i,withj}}{Ac_{spc,i,withoutj}}$

Which is a measure of how much the concentration of species i is impacted by reaction j.

In order to directly compare multiple reactions we can then sum over all core species involved in reaction j to get our criterion the dynamics number.
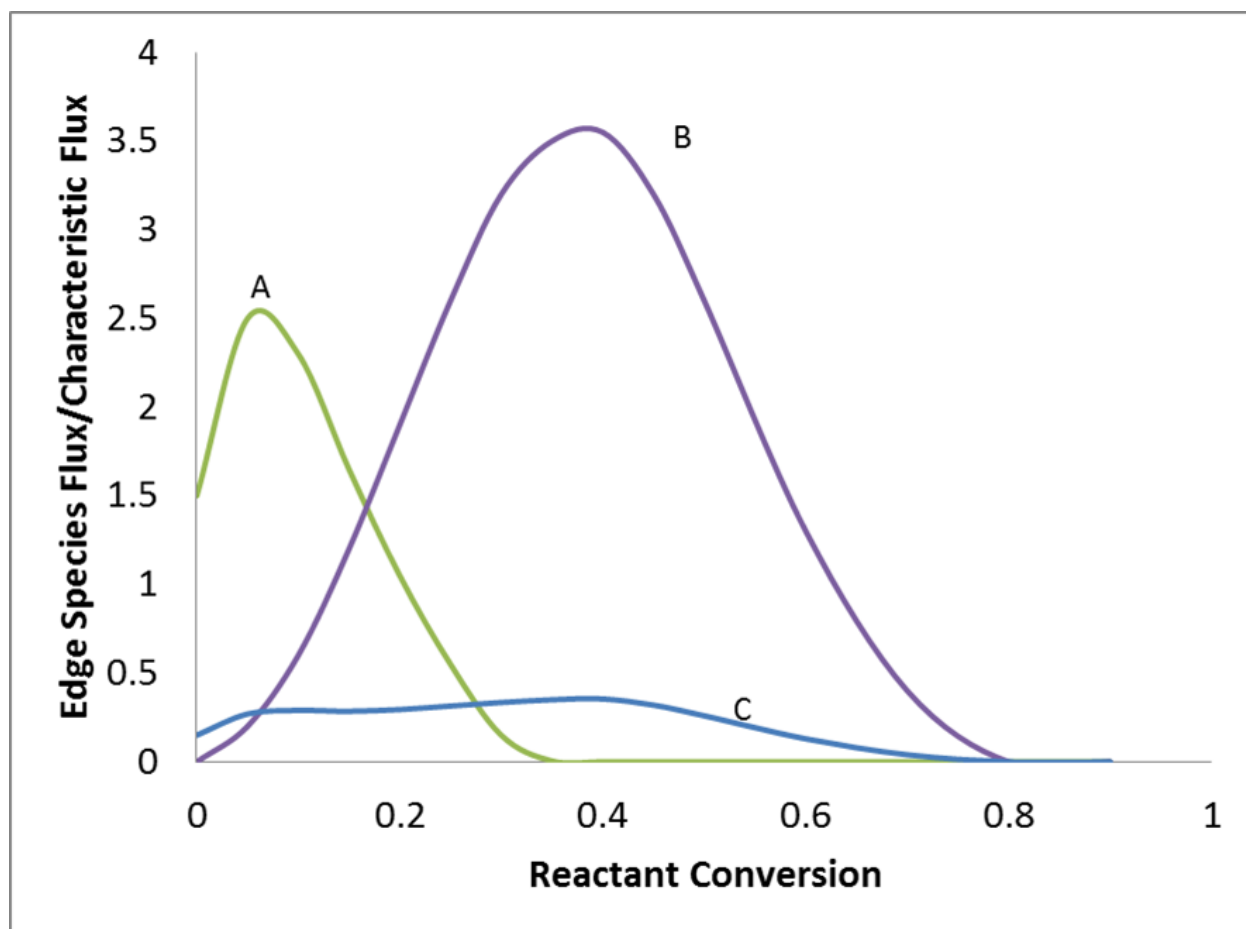
$\sum_{i \in core} |Ln(\Pi_{Ac,i,j})| = Dy > \epsilon$

Fig. 1: The goal of pruning is to delete those "useless" edge species. So "usefulness" should be defined and it's natural to have flux as a criterion for "usefulness". Since flux changes with reactant conversion, peak flux is chosen here to make decision of pruning or not. Every time pruning is triggered, edge species with peak flux lower than toleranceKeepInEdge $*$ characteristic flux will be deleted.
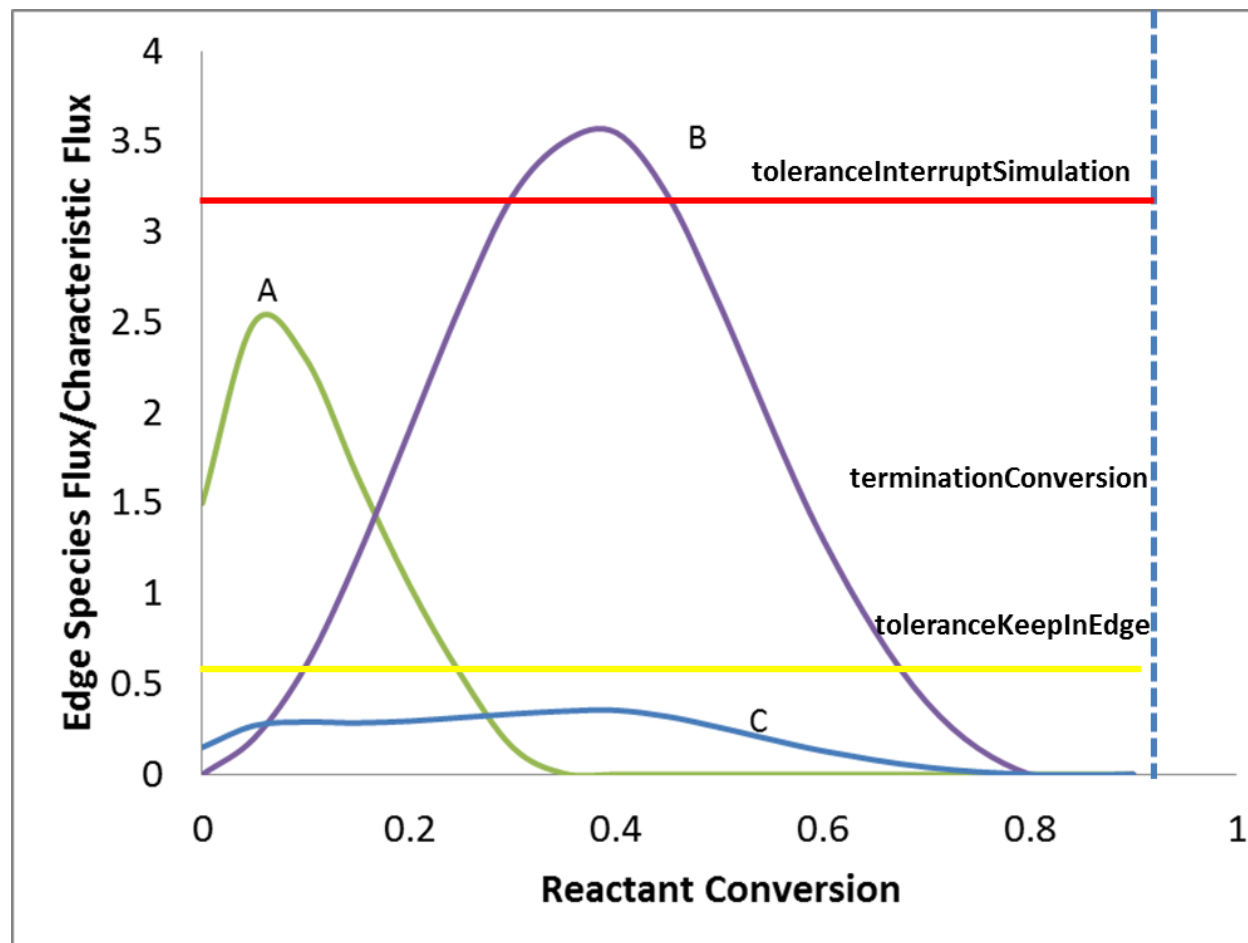
Fig. 2: However, pruning is not always triggered because of toleranceInterruptSimulation. As mentioned above, in order to prune, RMG needs to figure out the peak flux of each edge species, which requires dynamic simulation to complete. If some run of dynamic simulation is terminated in half way by toleranceInterruptSimulation, pruning is rejected although there might be some edge species with peak fluxes lower than toleranceKeepInEdge * characteristic flux. Since pruning requires to complete dynamic simulation, setting toleranceInterruptSimulation to be positive infinity, as an extreme case, means always enabling pruning. Another extreme case would be that it has same value as toleranceMoveToCore where no pruning occurs.

In summary, each run of dynamic simulation will proceed towards terminationConversion unless some flux exceeds toleranceInterruptSimulation * characteristic flux.Following complete simulation is the pruning of edge species whose flux is not high enough be kept in the edge, which is followed by pruning of excessive amount of edge species to make sure total edge species number is no greater than maximumEdgeSpecies.

Surface Algorithm

One common issue with the dynamics criterion is that it treats all core species equally. Because of this, if the dynamics criterion is set too low it enters a feedback loop where it adds species and then since it can't get those species' concentrations right it adds more species and so on. In order to avoid this feedback loop the surface algorithm was developed. It creates a new partition called the *surface* that is considered part of the core. We will refer to the part of the core that is not part of the surface as the *bulk core*. When operating without the dynamics criterion everything moves from edge to the bulk core as usual; however the dynamics criterion is managed differently. When using the surface algorithm most reactions pulled in by the dynamics criterion enter the surface instead of the bulk core. However, unlike movement to bulk core a constraint is placed on movement to the surface. Any reaction moved to the surface must have either both reactants or both products in the bulk core. This prevents the dynamics criterion from pulling in reactions to get the concentrations of species in the surface right avoiding the feedback loop. To avoid important species being trapped in the surface we also add criteria for movement from surface to bulk core based on flux or dynamics criterion. However, to avoid important species being trapped in the surface we also add criteria for movement from surface to bulk core based on flux or dynamics criterion.
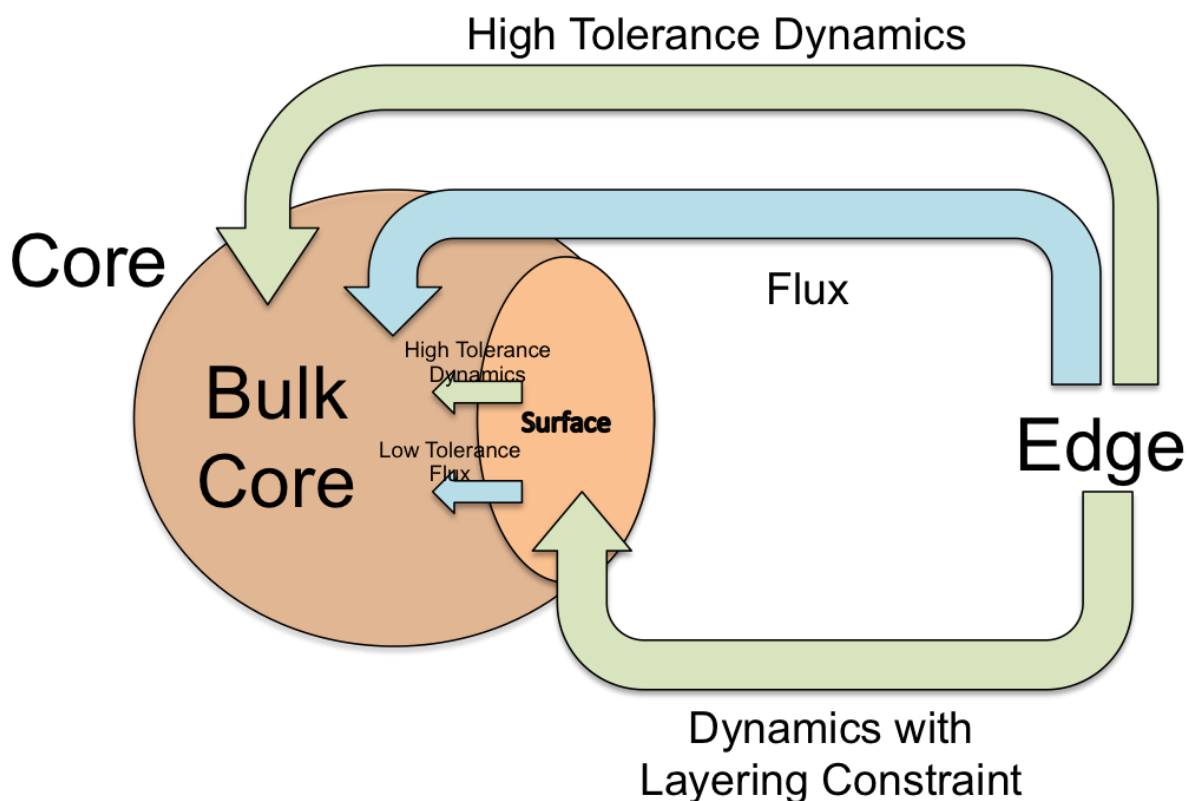
Key Parameters for Dynamics Criterion and Surface Algorithm

- **toleranceMoveEdgeReactionToCore**

  An edge reaction will be pulled directly into the bulk core if its dynamics number ever exceeds this value.

- **toleranceMoveEdgeReactionToSurface**

  An edge reaction will be pulled into the surface if its dynamics number ever exceeds this value.

- **toleranceMoveEdgeReactionToCoreInterrupt**

  When any reaction's dynamics number exceeds this value the simulation will be interrupted.

- **toleranceMoveEdgeReactionToSurfaceInterrupt**

  When the dynamics number of any reaction that would be valid for movement to the surface exceeds this value the simulation will be interrupted

- **toleranceMoveSurfaceReactionToCore**

  A surface reaction will be pulled into the bulk core if its dynamics number ever exceeds this value. Note this is done on the fly during simulation.

- **toleranceMoveSurfaceSpeciesToCore**

  A surface species will be pulled into the bulk core if it's rate ratio ever exceeds this value. Note this is done on the fly during simulation.
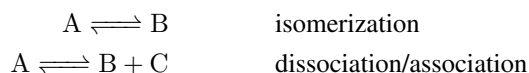
- genindex

- modindex

- search

## 3.2 Pressure-Dependence Theory Guide

3.2.1 Introduction

Unimolecular Reactions

Unimolecular reactions are those that involve a single reactant or product molecule, the union of isomerization and dissociation/association reactions:

$$A \rightleftharpoons B \qquad \text{isomerization}$$
$$A \rightleftharpoons B + C \qquad \text{dissociation/association}$$

Gas-phase chemical reactions occur as the result of bimolecular collisions between two reactant molecules. This presents a problem when there is only one participating reactant molecule! The conclusion is that the above reactions cannot be elementary as written; another step must be involved.

For a unimolecular reaction to proceed, the reactant molecule A must first be excited to an energy that exceeds the barrier for reaction. A molecule that is sufficiently excited to react is called an *activated species* and often labeled with an asterisk A^*. If we replace the stable species with the activated species in the reactions above, the reactions become elementary again:

$$A^* \rightleftharpoons B^*$$
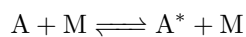$$A^* \rightleftharpoons B + C$$
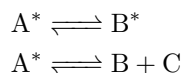
There are a number of ways that an activated species A^* can be produced:

- **Chemical activation**. A^* is produced as the adduct of an association reaction:

$$B + C \rightleftharpoons A^*$$
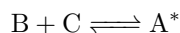
- **Thermal activation**. A^* is produced via transfer of energy from an otherwise inert species M via bimolecular collision:

$$A + M \rightleftharpoons A^* + M$$

- **Photoactivation**. A^* is produced as a result of absorption of a photon:

$$A + h\nu \longrightarrow A^*$$
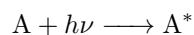
Once an activated molecule has been produced, multiple isomerization and dissociation reactions may become competitive with one another and with collisional stabilization (thermal deactivation); these combine to form a network of unimolecular reactions. The major pathway will depend on the relative rates of collision and reaction, which in turn is a function of both temperature and pressure. At high pressure the collision rate will be fast, and activated molecules will tend to be collisionally stabilized before reactive events can occur; this is called the *high-pressure limit*. At low pressures the collision rate will be slow, and activated molecules will tend to isomerize and dissociate, often traversing multiple reactive events before collisional stabilization can occur.

The onset of the pressure-dependent regime varies with both temperature and molecular size. The figure below shows the approximate pressure at which pressure-dependence becomes important as a function of temperature and molecular size. The parameter $m \equiv N_{\mathrm{vib}} + \frac{1}{2}N_{\mathrm{rot}}$ represents a count of the internal degrees of freedom (vibrations and hindered rotors, respectively). The ranges of the x-axis and y-axis suggest that pressure dependence is in fact important over a wide regime of conditions of practical interest, particularly in high-temperature processes such as pyrolysis and combustion [Wong2003].
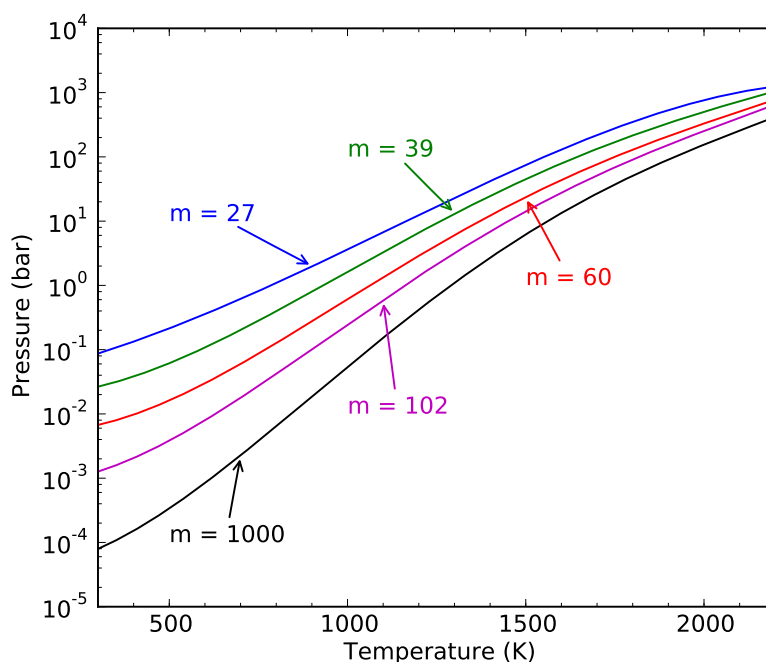

Fig. 3: Plot of the switchover pressure – indicating the onset of pressure dependence – as a function of temperature and molecular size. The value $m \equiv N_{\mathrm{vib}} + \frac{1}{2}N_{\mathrm{rot}}$ represents a count of the internal degrees of freedom. Over a wide variety of conditions of practical interest, even very large molecules exhibit significant pressure dependence. Figure adapted from Wong, Matheu and Green (2003).

Historical Context

The importance of bimolecular collisions in unimolecular reactions was first proposed by Lindemann in 1922 [Lindemann1922]. It was soon recognized by Hinshelwood and others that a rigorous treatment of these processes required consideration of molecular energy levels [Hinshelwood1926]. The RRKM expression for the microcanonical rate coefficient $k(E)$ was derived in the early 1950s [Rice1927] [Kassel1928] [Marcus1951]. In

the late 1950s master equation models of chemical systems began appearing [Siegert1949] [Bartholomay1958] [Montroll1958] [Krieger1960] [Gans1960], including an early linear integral-differential equation formulation by Widom [Widom1959]. Analytical solutions for a variety of simple models soon followed [Keck1965] [Troe1967] [Troe1973], as did the first numerical approaches [Tardy1966]. Numerical methods – which are required for complex unimolecular reaction networks – became much more attractive in the 1970s with the appearance of new algorithms, including Gear's method for solving stiff systems of ordinary differential equations [Gear1971] and efficient algorithms for calculating the density of states [Beyer1973] [Stein1973] [Astholz1979]. In the 1990s computing power had increased to the point where it was practical to solve them numerically by discretizing the integrals over energy.

3.2.2 The Master Equation

A full treatment of the energy states of each molecule is unfeasible for molecules larger than diatomics, as there are simply too many states. To simplify things we apply the RRKM approximation, which leaves the state of a molecule as a function of two quantities: the total energy E and total angular momentum quantum number J. Frequently we will find that even this is too difficult, and will only keep the total energy E as an independent variable.

Isomers, Reactants, and Products

Throughout this document we will utilize the following terminology:

- An **isomer** is a unimolecular configuration on the potential energy surface.

- A **reactant channel** is a bimolecular configuration that associates to form an isomer. Dissociation from the isomer back to reactants is allowed.

- A **product channel** is a bimolecular configuration that is formed by dissociation of an isomer. Reassociation of products to the isomer is *not* allowed.

The isomers are the configurations for which we must model the energy states. We designate $p_i(E, J, t)$ as the population of isomer i having total energy E and total angular momentum quantum number J at time t. At long times, statistical mechanics requires that the population of each isomer approach a Boltzmann distribution $b_i(E, J)$:

$$\lim_{t \to \infty} p_i(E, J, t) \propto b_i(E, J)$$

We can simplify by eliminating the angular momentum quantum number to get

$$p_i(E, t) = \sum_J p_i(E, J, t)$$

Let us also denote the (time-dependent) total population of isomer i by $x_i(t)$:

$$x_i(t) \equiv \sum_J \int_0^\infty p_i(E, J, t) \, dE$$

The two molecules of a reactant or product channel are free to move apart from one another and interact independently with other molecules in the system. Accordingly, we treat these channels as fully thermalized, leaving as the only variable the total concentrations $y_{nA}(t)$ and $y_{nB}(t)$ of the molecules A_n and B_n of reactant channel n. (Since the product channels act as infinite sinks, their populations do not need to be considered explicitly.)

Finally, we will use N_{isom}, N_{reac}, and N_{prod} as the numbers of isomers, reactant channels, and product channels, respectively, in the system.

Collision Models

Bimolecular collisions with an inert species M are the primary means by which an isomer molecule changes its energy. A reasonable estimate – although generally a bit of an underestimate – of the total rate of collisions $k_{\mathrm{coll},i}(T)$ for each isomer i comes from Lennard-Jones collision theory:

$$k_{\mathrm{coll},i}(T) = \sqrt{\frac{8k_{\mathrm{B}}T}{\pi\mu_i}}\pi d_i^2 \Omega_i^{(2,2)*}$$

Above, μ_i is the reduced mass, d_i is the collision diameter, and k_{B} is the Boltzmann constant. The collision diameter is generally taken as $d \approx \frac{1}{2}(\sigma_i + \sigma_{\mathrm{M}})$, the arithmetic average of the Lennard-Jones σ parameter for the isomer and the bath gas. The parameter $\Omega_i^{(2,2)*}$ represents a configurational integral, which is well-approximated by the expression

$$\Omega_i^{(2,2)*} = 1.16145\tilde{T}^{-0.14874} + 0.52487 e^{-0.7732\tilde{T}} + 2.16178 e^{-2.437887\tilde{T}}$$

where $\tilde{T} \equiv k_{\mathrm{B}}T/\sqrt{\epsilon_i \epsilon_{\mathrm{M}}}$ is a reduced temperature and ϵ_i is the Lennard-Jones ϵ parameter. Note that we have used a geometric average for the ϵ parameters of the isomer and the bath gas in this expression. Assuming the total gas concentration to be constant and that the gas is ideal, we obtain an expression for the collision frequency $\omega_i(T, P)$, which makes explicit the pressure dependence:

$$\omega_i(T, P) = k_{\mathrm{coll},i}(T)\frac{P}{k_{\mathrm{B}}T}$$

Now that we have an estimate for the total rate of collisions, we need to develop a model of the effect that these collisions have on the state of the isomer distribution. To this end, we define $P(E, J, E', J')$ as the probability of a collision resulting in a transfer of a molecule from state (E', J') to state (E, J). There are two mathematical constraints on $P(E, J, E', J')$. The first of these is normalization:

$$\sum_{J'}(2J' + 1)\int_0^\infty P(E, J, E', J')\, dE' = 1$$

The second of these is detailed balance, required in order to obtain the Boltzmann distribution at long times:

$$P(E, J, E', J')b(E', J') = P(E', J', E, J)b(E, J)$$

$$P_i(E', J', E, J) = \frac{\rho_i(E', J')}{\rho_i(E, J)}\exp\left(-\frac{E' - E}{k_{\mathrm{B}}T}\right)P_i(E, J, E', J') \qquad E < E'$$

Rather than define models directly for $P(E, J, E', J')$, we usually eliminate the angular momentum contribution and instead define $P(E, E')$. This can be related to $P(E, J, E', J')$ via

$$P(E, J, E', J') = P(E, E')\phi(E, J) = P(E, E')(2J + 1)\frac{\rho(E, J)}{\rho(E)}$$

where $\rho(E) \equiv \sum_J(2J + 1)\rho(E, J)$.

There are a variety of models used for $P(E, E')$. By far the most common is the single exponential down model

$$P(E, E') = C(E')\exp\left(-\frac{E' - E}{\alpha}\right) \qquad E < E'$$

where $C(E')$ is determined from the normalization constraint. Note that this function has been defined for the deactivating direction ($E < E'$) only, as the activating direction ($E > E'$) is then set from detailed balance. The parameter α corresponds to the average energy transferred in a deactivating collision $\langle \Delta E_{\mathrm{d}}\rangle$, which itself is a weak function of temperature.

Other models for $P(E, J, E', J')$ include the Gaussian down

$$P(E, E') = C(E')\exp\left[-\frac{(E' - E)^2}{\alpha^2}\right] \qquad E < E'$$

and the double exponential down

$$P(E, E') = C(E') \left[(1 - f) \exp\left(-\frac{E' - E}{\alpha_1} \right) + f \exp\left(-\frac{E' - E}{\alpha_2} \right) \right] \qquad E < E'$$

The parameters for these simple models generally contain so much uncertainty that more complex functional forms are generally not used.

Reaction Models

Chemical reaction events cause a change in molecular configuration at constant energy. The rate coefficient for this process must be determined as a function of energy rather than the usual temperature. Such a quantity is called a *microcanonical rate coefficient* and written as $k(E, J)$. In the master equation we will differentiate between microcanonical rate coefficients for isomerization, dissociation, and association by using different letters: $k_{ij}(E, J)$ for isomerization, $g_{nj}(E, J)$ for dissociation, and $f_{im}(E, J)$ for association. (By convention, we use indices i and j to refer to unimolecular isomers, m and n to refer to bimolecular reactant and product channels, and, later, r and s to refer to energy grains.)

As with collision models, the values of the microcanonical rate coefficients are constrained by detailed balance so that the proper equilibrium is obtained. The detailed balance expressions have the form

$$k_{ij}(E, J)\rho_j(E, J) = k_{ji}(E, J)\rho_i(E, J)$$

for isomerization and

$$f_{in}(E, J)\rho_n(E, J) = g_{ni}(E, J)\rho_i(E, J)$$

for association/dissociation, where $\rho_i(E, J)$ is the density of states of the appropriate unimolecular or bimolecular configuration.

An alternative formulation incorporates the macroscopic equilibrium coefficient $K_{\text{eq}}(T)$ and equilibrium distributions $b_i(E, J, T)$ at each temperature:

$$k_{ij}(E, J)b_j(E, J, T) = K_{\text{eq}}(T)k_{ji}(E, J)b_i(E, J, T)$$

for isomerization and

$$f_{in}(E, J)b_n(E, J, T) = K_{\text{eq}}(T)g_{ni}(E, J)b_i(E, J, T)$$

for association/dissociation. Note that these two formulations are equivalent if the molecular degrees of freedom are consistent with the macroscopic thermodynamic parameters. There are multiple reasons to use the latter formulation:

- Only the density of states of the unimolecular isomers need be computed. This is a result of the assumption of thermalized bimolecular channels, which means that we only need to compute the product $f_{in}b_n$, and not the individual values of f_{in} and b_n.

- Only the reactive rovibrational modes need be included in the density of states. Missing modes will not affect the observed equilibrium because we are imposing the macroscopic equilibrium via $K_{\text{eq}}(T)$.

- Constants of proportionality in the density of states become unimportant, as they cancel when taking the ratio $\rho(E, J)/Q(\beta)$. For example, if the external rotational constants are unknown then we will include an active K-rotor in the density of states; this property means that the rotational constant of this active K-rotor cancels and is therefore arbitrary.

There are two common ways of determining values for $k(E, J)$: the inverse Laplace transform method and RRKM theory. The latter requires detailed information about the transition state, while the former only requires the high-pressure limit rate coefficient $k_\infty(T)$.

Inverse Laplace Transform

The microcanonical rate coefficient $k(E)$ is related to the canonical high-pressure limit rate coefficient $k_\infty(T)$ via a Boltzmann averaging

$$k_\infty(T) = \frac{\sum_J \int_0^\infty k(E)\rho(E,J)e^{-\beta E}\, dE}{\sum_J \int_0^\infty \rho(E,J)e^{-\beta E}\, dE}$$

where $\rho(E, J)$ is the rovibrational density of states for the reactants and $\beta \equiv (k_\mathrm{B}T)^{-1}$. Neglecting the angular momentum dependence, the above can be written in terms of Laplace transforms as

$$k_\infty(T) = \frac{\mathcal{L}\left[k(E)\rho(E)\right]}{\mathcal{L}\left[\rho(E)\right]} = \frac{\mathcal{L}\left[k(E)\rho(E)\right]}{Q(\beta)}$$

where $Q(\beta)$ is the rovibrational partition function for the reactants. The above implies that E and β are the transform variables. We can take an inverse Laplace transform in order to solve for $k(E)$:

$$k(E) = \frac{\mathcal{L}^{-1}\left[k_\infty(\beta)Q(\beta)\right]}{\rho(E)}$$

Hidden in the above manipulation is the assumption that $k_\infty(\beta)$ is valid over a temperature range from zero to positive infinity.

The most common form of $k_\infty(T)$ is the modified Arrhenius expression

$$k(T) = AT^n \exp\left(-\frac{E_\mathrm{a}}{k_\mathrm{B}T}\right)$$

where A, n, and E_a are the Arrhenius preexpoential, temperature exponent, and activation energy, respectively. For $n = 0$ and $E_\mathrm{a} > 0$ the inverse Laplace transform can be easily evaluated to give

$$k(E) = A\frac{\rho(E - E_\mathrm{a})}{\rho(E)} \qquad E > E_\mathrm{a}$$

We can also determine an expression when $n > 0$ and $E_\mathrm{a} > 0$ using a convolution integral:

$$k(E) = A\frac{\phi(E - E_\mathrm{a})}{\rho(E)} \qquad E > E_\mathrm{a}$$

$$\phi(E) = \mathcal{L}^{-1}\left[T^n Q(\beta)\right] = \frac{1}{k_\mathrm{B}^n \Gamma(n)}\int_0^E (E - x)^{n-1}\rho(x)\, dx$$

Finally, for cases where $n < 0$ and/or $E_\mathrm{a} < 0$ we obtain a rough estimate by lumping these contributions into the preexponential at the temperature we are working at. By redoing this at each temperature being considered we minimize the error introduced, at the expense of not being able to identify a single $k(E)$.

RRKM Theory

RRKM theory – named for Rice, Ramsperger, Kassel, and Marcus – is a microcanonical transition state theory. Like canonical transition state theory, detailed information about the transition state and reactants are required, e.g. from a quantum chemistry calculation. If such information is available, then the microcanonical rate coefficient can be evaluated via the equation

$$k(E, J) = \frac{N^\ddagger(E, J)}{h\rho(E, J)}$$

where $N^{\ddagger}(E, J)$ is the sum of states of the transition state, $\rho(E, J)$ is the density of states of the reactant, and h is the Planck constant. Both the transition state and the reactants have been referenced to the same zero of energy. The sum of states is related to the density of states via

$$N(E, J) = \int_0^E \rho(x, J) \, dx$$

The angular momentum quantum number dependence can be removed via

$$k(E) = \sum_J (2J + 1)k(E, J)$$

The Full Master Equation

The governing equation for the population distributions $p_i(E, J, t)$ of each isomer i and the reactant concentrations $y_{nA}(t)$ and $y_{nB}(t)$ combines the collision and reaction models to give a linear integro-differential equation:

$$\frac{d}{dt}p_i(E, J, t) = \omega_i(T, P) \sum_{J'} \int_0^{\infty} P_i(E, J, E', J')p_i(E', J', t) \, dE' - \omega_i(T, P)p_i(E, J, t)$$

$$+ \sum_{j \neq i}^{N_{\text{isom}}} k_{ij}(E, J)p_j(E, J, t) - \sum_{j \neq i}^{N_{\text{isom}}} k_{ji}(E, J)p_i(E, J, t)$$

$$+ \sum_{n=1}^{N_{\text{reac}}} y_{nA}(t)y_{nB}(t)f_{in}(E, J)b_n(E, J, t) - \sum_{n=1}^{N_{\text{reac}}+N_{\text{prod}}} g_{ni}(E, J)p_i(E, J, t)$$

$$\frac{d}{dt}y_{nA}(t) = \frac{d}{dt}y_{nB}(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^{\infty} g_{ni}(E, J)p_i(E, J, t) \, dE$$

$$- \sum_{i=1}^{N_{\text{isom}}} y_{nA}(t)y_{nB}(t) \int_0^{\infty} f_{in}(E, J)b_n(E, J, t) \, dE$$

A summary of the variables is given below:

Variable	Meaning
$p_i(E, J, t)$	Population distribution of isomer i
$y_{nA}(t)$	Total population of species A_n in reactant channel n
$\omega_i(T, P)$	Collision frequency of isomer i
$P_i(E, J, E', J')$	Collisional transfer probability from (E', J') to (E, J) for isomer i
$k_{ij}(E, J)$	Microcanonical rate coefficient for isomerization from isomer j to isomer i
$f_{im}(E, J)$	Microcanonical rate coefficient for association from reactant channel m to isomer i
$g_{nj}(E, J)$	Microcanonical rate coefficient for dissociation from isomer j to reactant or product channel n
$b_n(E, J, t)$	Boltzmann distribution for reactant channel n
N_{isom}	Total number of isomers
N_{reac}	Total number of reactant channels
N_{prod}	Total number of product channels

The above is called the two-dimensional master equation because it contains two dimensions: total energy E and total angular momentum quantum number J. In the first equation (for isomers), the first pair of terms correspond to collision, the second pair to isomerization, and the final pair to association/dissociation. Similarly, in the second equation above (for reactant channels), the pair of terms refer to dissociation/association.

We can also simplify the above to the one-dimensional form, which only has E as a dimension:

$$\frac{d}{dt}p_i(E,t) = \omega_i(T,P) \int_0^\infty P_i(E,E') p_i(E',t) \, dE' - \omega_i(T,P) p_i(E,t)$$

$$+ \sum_{j\neq i}^{N_{\text{isom}}} k_{ij}(E) p_j(E,t) - \sum_{j\neq i}^{N_{\text{isom}}} k_{ji}(E) p_i(E,t)$$

$$+ \sum_{n=1}^{N_{\text{reac}}} y_{n\text{A}}(t) y_{n\text{B}}(t) f_{in}(E) b_n(E,t) - \sum_{n=1}^{N_{\text{reac}}+N_{\text{prod}}} g_{ni}(E) p_i(E,t)$$

$$\frac{d}{dt} y_{n\text{A}}(t) = \frac{d}{dt} y_{n\text{B}}(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^\infty g_{ni}(E) p_i(E,t) \, dE$$

$$- \sum_{i=1}^{N_{\text{isom}}} y_{n\text{A}}(t) y_{n\text{B}}(t) \int_0^\infty f_{in}(E) b_n(E,t) \, dE$$

The equations as given are nonlinear, both due to the presence of the bimolecular reactants and because both ω_i and $P_i(E,E')$ depend on the composition, which is changing with time. The rate coefficients can be derived from considering the pseudo-first-order situation where $y_{n\text{A}}(t) \ll y_{n\text{B}}(t)$, and all $y(t)$ are negligible compared to the bath gas M. From these assumptions the changes in ω_i, $P_i(E,E')$, and all $y_{n\text{B}}$ can be neglected, which yields a linear equation system.

The Energy-Grained Master Equation

Except for the simplest of unimolecular reaction networks, both the one-dimensional and two-dimensional master equation must be solved numerically. To do this we must discretize and truncate the energy domain into a finite number of discrete bins called *grains*. This converts the linear integro-differential equation into a system of first-order ordinary differential equations:

$$\frac{d}{dt}\begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ y_{1\text{A}} \\ y_{2\text{A}} \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{M}_1 & \mathbf{K}_{12} & \cdots & \mathbf{F}_{11}\mathbf{b}_1 y_{1\text{B}} & \mathbf{F}_{12}\mathbf{b}_2 y_{2\text{B}} & \cdots \\ \mathbf{K}_{21} & \mathbf{M}_2 & \cdots & \mathbf{F}_{21}\mathbf{b}_1 y_{1\text{B}} & \mathbf{F}_{22}\mathbf{b}_2 y_{2\text{B}} & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ (\mathbf{g}_{11})^T & (\mathbf{g}_{12})^T & \cdots & h_1 & 0 & \cdots \\ (\mathbf{g}_{21})^T & (\mathbf{g}_{22})^T & \cdots & 0 & h_2 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ y_{1\text{A}} \\ y_{2\text{A}} \\ \vdots \end{bmatrix}$$

The diagonal matrices $\mathbf{K}_{ij}$ and $\mathbf{F}_{in}$ and the vector $\mathbf{g}_{ni}$ contain the microcanonical rate coefficients for isomerization, association, and dissociation, respectively:

$$(\mathbf{K}_{ij})_{rs} = \begin{cases} \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} k_{ij}(E) \, dE & r = s \\ 0 & r \neq s \end{cases}$$

$$(\mathbf{F}_{in})_{rs} = \begin{cases} \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} f_{in}(E) \, dE & r = s \\ 0 & r \neq s \end{cases}$$

$$(\mathbf{g}_{ni})_r = \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} g_{ni}(E) \, dE$$

The matrices $\mathbf{M}_i$ represent the collisional transfer probabilities minus the rates of reactive loss to other isomers and to reactants and products:

$$(\mathbf{M}_i)_{rs} = \begin{cases} \omega_i \left[P_i(E_r, E_r) - 1 \right] - \sum_{j\neq i}^{N_{\text{isom}}} k_{ij}(E_r) - \sum_{n=1}^{N_{\text{reac}}+N_{\text{prod}}} g_{ni}(E_r) & r = s \\ \omega_i P_i(E_r, E_s) & r \neq s \end{cases}$$

The scalars h_n are simply the total rate coefficient for loss of reactant channel n due to chemical reactions:

$$h_n = - \sum_{i=1}^{N_{\text{isom}}} \sum_{r=1}^{N_{\text{grains}}} y_{n\text{B}} f_{in}(E_r) b_n(E_r)$$

Further Reading

The interested reader is referred to any of a variety of other sources for alternative presentations, of which an illustrative sampling is given here [Gilbert1990] [Baer1996] [Holbrook1996] [Forst2003] [Pilling2003].

3.2.3 Methods for Determining Phenomenological Rate Coefficients

Solving the energy-grained master equation is often prohibitively expensive for chemical reaction mechanisms of even modest size. Instead, we seek to reduce the master equation matrix down to a set of phenomenological rate coefficients $k(T, P)$. In particular, we need to replace the isomer population distributions $p_i(E, t)$ with the corresponding time-dependent total isomer populations $x_i(t)$.

Three methods of varying rigor, computational cost, and robustness will be discussed in the upcoming sections. The modified strong collision (MSC) method is the fastest and most robust, but utilizes the least realistic approximations. The reservoir state (RS) method uses better approximations, which leads to increased accuracy, but requires more computational effort. Finally, the chemically-significant eigenvalues (CSE) method is the most theoretically sound, but is very computationally expensive and not very robust. Your choice of method will depend on the particular balance between expense, robustness, and rigor that is required for your intended application.

A Common Formalism

All of the methods discussed here can be expressed in terms of a common formalism. Each method seeks to express the population distribution vector $p_i(E, t)$ for each unimolecular isomer i as a linear combination of the total populations $x_j(t)$ and $y_{m\text{A}}(t) y_{m\text{B}}$ of unimolecular isomers A_j and reactant channels $A_m + B_m$:

$$p_i(E, t) = \sum_{j=1}^{N_{\text{isom}}} x_j(t) u_{ij}(E) + \sum_{m=1}^{N_{\text{reac}}} y_{m\text{A}}(t) y_{m\text{B}} v_{im}(E)$$

The function $u_{ij}(E)$ represents the portion of the population distribution of unimolecular isomer i at energy E that tracks the population of isomer j. In the modified strong collision and reservoir state methods, this is because the energy levels of isomer i are in pseudo-steady-state relationships with isomer j. The interpretation is a bit different for the chemically-significant eigenvalues method, but the form of the equations is the same. Similarly, the function $v_{im}(E)$ represents the population distribution of unimolecular isomer i at energy E that tracks the population of reactant channel m. Both functions $u_{ij}(E)$ and $v_{im}(E)$ are functions of energy only, and not of time.

After discretizing the energy domain, the above becomes

$$\mathbf{p}_i(t) = \sum_{j=1}^{N_{\text{isom}}} x_j(t) \mathbf{u}_{ij} + \sum_{m=1}^{N_{\text{reac}}} y_{m\text{A}}(t) y_{m\text{B}} \mathbf{v}_{im}$$

The phenomenological rate coefficients can be constructed from the energy-grained master equation matrix and the

vectors $\mathbf{u}_{ij}$ and $\mathbf{v}_{im}$:

$$k_{ij}(T, P) = \sum_{s=1}^{N_{\text{grains}}} (\mathbf{M}_i \mathbf{u}_{ij})_s + \sum_{\ell \neq i}^{N_{\text{isom}}} \sum_{s=1}^{N_{\text{grains}}} (\mathbf{K}_{i\ell} \mathbf{u}_{\ell j})_s$$

$$k_{im}(T, P) = \sum_{s=1}^{N_{\text{grains}}} (\mathbf{M}_i \mathbf{v}_{im})_s + \sum_{\ell \neq i}^{N_{\text{isom}}} \sum_{s=1}^{N_{\text{grains}}} (\mathbf{K}_{i\ell} \mathbf{v}_{\ell m})_s + \sum_{s=1}^{N_{\text{grains}}} (\mathbf{F}_{im} \mathbf{b}_m)_s$$

$$k_{nj}(T, P) = \sum_{\ell=1}^{N_{\text{isom}}} \mathbf{g}_{n\ell} \cdot \mathbf{u}_{\ell j}$$

$$k_{nm}(T, P) = \sum_{\ell=1}^{N_{\text{isom}}} \mathbf{g}_{n\ell} \cdot \mathbf{v}_{\ell m}$$

Above, the indices i and j represent unimolecular isomers of the initial adduct, m represents bimolecular reactants, n represents bimolecular reactants and products, and s represents an energy grain. Thus, the rate coefficients above are for isomerization, association, dissociation, and bimolecular reactions, respectively.

The output from each of the three methods is a set of phenomenological rate coefficients $k(T, P)$ and the vectors $\mathbf{u}_{ij}$ and $\mathbf{v}_{im}$ which can be used to construct the approximate population distribution predicted by that method.

The Modified Strong Collision Method

The Reservoir State Method

The Chemically-Signficant Eigenvalues Method

- genindex
- modindex
- search

[GRIMech3.0] Gregory P. Smith, David M. Golden, Michael Frenklach, Nigel W. Moriarty, Boris Eiteneer, Mikhail Goldenberg, C. Thomas Bowman, Ronald K. Hanson, Soonho Song, William C. Gardiner, Jr., Vitali V. Lissianski, and Zhiwei Qin http://www.me.berkeley.edu/gri_mech/

[RDKit] RDKit: Open-source cheminformatics; http://www.rdkit.org

[Chang2000] A.Y. Chang, J.W. Bozzelli, and A. M. Dean. "Kinetic Analysis of Complex Chemical Activation and Unimolecular Dissociation Reactions using QRRK Theory and the Modified Strong Collision Approximation." *Z. Phys. Chem.* **214** (11), p. 1533-1568 (2000).

[Green2007] N.J.B. Green and Z.A. Bhatti. "Steady-State Master Equation Methods." *Phys. Chem. Chem. Phys.* **9**, p. 4275-4290 (2007).

[Benson1976] Benson, Sidney William. "Thermochemical kinetics." (1976)

[Benson] Benson, S.W. (1965), https://en.wikipedia.org/wiki/Benson_group_increment_theory

[Lay] Lay, T.; Bozzelli, J.; Dean, A.; Ritter, E. J. Phys. Chem. 1995, 99,14514-14527

[Allinger] Allinger, N. L., & Lii, J.-H. (2008). MM4(2008) and MM4(2003).

[Patchkovskii] Patchkovskii, S. (2003). SYMMETRY, http://www.cobalt.chem.ucalgary.ca/ps/symmetry/.

[Yu] "Estimation method for the thermochemical properties of polycyclic aromatic molecules" (Ph.D), Joanna Yu, M.I.T (2005)

[RDKit] Landrum, G. (2012). RDKit, http://rdkit.org.

[Vitha2006] M. Vitha and P.W. Carr. "The chemical interpretation and practice of linear solvation energy relationships in chromatography." *J. Chromatogr. A.* **1126(1-2)**, p. 143-194 (2006).

[Abraham1999] M.H. Abraham et al. "Correlation and estimation of gas-chloroform and water-chloroformpartition coefficients by a linear free energy relationship method." *J. Pharm. Sci.* **88(7)**, p. 670-679 (1999).

[Jalan2010] A. Jalan et al. "Predicting solvation energies for kinetic modeling." *Annu. Rep.Prog. Chem., Sect. C* **106**, p. 211-258 (2010).

[Poole2009] C.F. Poole et al. "Determination of solute descriptors by chromatographic methods." *Anal. Chim. Acta* **652(1-2)** p. 32-53 (2009).

[Platts1999] J. Platts and D. Butina. "Estimation of molecular linear free energy relation descriptorsusing a group contribution approach." *J. Chem. Inf. Comput. Sci.* **39**, p. 835-845 (1999).

[Mintz2007] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved inwater and in 1-octanol based on the Abraham model." *J. Chem. Inf. Model.* **47(1)**, p. 115-121 (2007).

[Mintz2007a] C. Mintz et al. "Enthalpy of solvation corrections for gaseous solutes dissolved in benzene and in alkane solvents based on the Abraham model." *QSAR Comb. Sci.* **26(8)**, p. 881-888 (2007).

[Mintz2007b] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved in toluene and carbon tetrachloride based on the Abraham model." *J. Sol. Chem.* **36(8)**, p. 947-966 (2007).

[Mintz2007c] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved indimethyl sulfoxide and propylene carbonate based on the Abraham model." *Thermochim. Acta* **459(1-2)**, p, 17-25 (2007).

[Mintz2007d] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved inchloroform and 1,2-dichloroethane based on the Abraham model." *Fluid Phase Equilib.* **258(2)**, p. 191-198 (2007).

[Mintz2008] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved inlinear alkanes (C5-C16) based on the Abraham model." *QSAR Comb. Sci.* **27(2)**, p. 179-186 (2008).

[Mintz2008a] C. Mintz et al. "Enthalpy of solvation correlations for gaseous solutes dissolved inalcohol solvents based on the Abraham model." *QSAR Comb. Sci.* **27(5)**, p. 627-635 (2008).

[Mintz2009] C. Mintz et al. "Enthalpy of solvation correlations for organic solutes and gasesdissolved in acetonitrile and acetone." *Thermochim. Acta* **484(1-2)**, p. 65-69 (2009).

[Rice1985] S.A. Rice. "Diffusion-limited reactions." In *Comprehensive Chemical Kinetics*, EditorsC.H. Bamford, C.F.H. Tipper and R.G. Compton. **25**, (1985).

[Flegg2016] M.B. Flegg. "Smoluchowski reaction kinetics for reactions of any order." *SIAM J. Appl. Math.* **76(4)**, p. 1403-1432 (2016).

[Gao2016] C. W. Gao, J. W. Allen, W. H. Green, R. H. West, "Reaction Mechanism Generator: automatic construction of chemical kinetic mechanisms." *Computer Physics Communications* (2016).

[Susnow1997] R. G. Susnow, A. M. Dean, W. H. Green, P. K. Peczak, and L. J. Broadbelt. "Rate-Based Construction of Kinetic Models for Complex Systems." *J. Phys. Chem. A* **101**, p. 3731-3740 (1997).

[Wong2003] B. M. Wong, D. M. Matheu, and W. H. Green. *J. Phys. Chem. A* **107**, p. 6206-6211 (2003). doi:10.1021/jp034165g

[Lindemann1922] F. A. Lindemann. *Trans. Faraday Soc.* **17**, p. 598-606 (1922).

[Hinshelwood1926] C. N. Hinshelwood. *Proc. Royal Soc. A* **17**, p. 230-233 (1926). JSTOR:94593

[Rice1927] O. K. Rice and H. C. Ramsperger. *J. Am. Chem. Soc.* **49**, p. 1617-1629 (1927). doi:10.1021/ja01406a001

[Kassel1928] L. S. Kassel. *J. Phys. Chem.* **32**, p. 1065-1079 (1928). doi:10.1021/j150289a011

[Marcus1951] R. A. Marcus and O. K. Rice. *J. Phys. Coll. Chem.* **55**, p. 894-908 (1951). doi:10.1021/j150489a013

[Siegert1949] A. J. F. Siegert. *Phys. Rev.* **76**, p. 1708-1714 (1949). doi:10.1103/PhysRev.76.1708

[Bartholomay1958] A. F. Bartholomay. *Bull. Math. Biophys.* **20**, p. 175-190 (1958). doi:10.1007/BF02478297

[Montroll1958] E. W. Montroll and K. E. Shuler. *Adv. Chem. Phys.* **1**, p. 361-399 (1958).

[Krieger1960] I. M. Krieger and P. J. Gans. *J. Chem. Phys.* **32**, p. 247-250 (1960). doi:10.1063/1.1700909

[Gans1960] P. J. Gans. *J. Chem. Phys.* **33**, p. 691-694 (1960). doi:10.1063/1.1731239

[Widom1959] B. Widom. *J. Chem. Phys.* **31**, p. 1387-1394 (1959). doi:10.1063/1.1730604

[Keck1965] J. Keck and G. Carrier. *J. Chem. Phys.* **43**, p. 2284-2298 (1965). doi:10.1063/1.1697125

[Troe1967] J. Troe and H. Gg. Wagner. *Ber. Bunsenges. Phys. Chem.* **71**, p. 937 (1967). doi:10.1002/bbpc.19670710904

[Troe1973] J. Troe. *Ber. Bunsenges. Phys. Chem.* **77**, p. 665 (1973). doi:10.1002/bbpc.19730770903

[Tardy1966] D. C. Tardy and B. S. Rabinovitch. *J. Chem. Phys.* **45**, p. 3720-3730 (1966). doi:10.1063/1.1727392

[Gear1971] C. W. Gear. *Commun. ACM* **14**, p. 176-179 (1971). doi:10.1145/362566.362571

[Beyer1973] T. Beyer and D. F. Swinehart. *Commun. ACM* **16**, p. 379 (1973). doi:10.1145/362248.362275

[Stein1973] S. E. Stein and B. S. Rabinovitch. *J. Chem. Phys.* **58**, p. 2438-2444 (1973). doi:10.1063/1.1679522

[Astholz1979] D. C. Astholz, J. Troe, and W. Wieters. *J. Chem. Phys.* **70**, p. 5107-5116 (1979). doi:10.1063/1.437352

[Gilbert1990] R. G. Gilbert and S. C. Smith. *Theory of Unimolecular and Recombination Reactions*. Blackwell Sci. (1990).

[Baer1996] T. Baer and W. L. Hase. *Unimolecular Reaction Dynamics*. Oxford University Press (1996).

[Holbrook1996] K. A. Holbrook, M. J. Pilling, and S. H. Robertson. *Unimolecular Reactions*. Second Edition. John Wiley and Sons (1996).

[Forst2003] W. Forst. *Unimolecular Reactions: A Concise Introduction*. Cambridge University Press (2003).

[Pilling2003] M. J. Pilling and S. H. Robertson. *Annu. Rev. Phys. Chem.* **54**, p. 245-275 (2003). doi:10.1146/annurev.physchem.54.011002.103822