RMG-Py API Reference
Release 2.0.0

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

Sep 16, 2016
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 is the API Reference guide for RMG. For instructions on how to use RMG, please refer to the User Guide.

For the latest documentation and source code, please visit http://reactionmechanismgenerator.github.io/RMG-Py/
This document provides the complete details of the application programming interface (API) for the Python version of the Reaction Mechanism Generator. The functionality of RMG-Py is divided into many modules and subpackages. An overview of these components is given in the table below. Click on the name of a component to learn more and view its API.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmgpy.cantherm</td>
<td>Computing chemical properties from quantum chemistry calculations</td>
</tr>
<tr>
<td>rmgpy.chemkin</td>
<td>Reading and writing models in Chemkin format</td>
</tr>
<tr>
<td>rmgpy.constants</td>
<td>Physical constants</td>
</tr>
<tr>
<td>rmgpy.data</td>
<td>Working with the RMG database</td>
</tr>
<tr>
<td>rmgpy.kinetics</td>
<td>Kinetics models of chemical reaction rates</td>
</tr>
<tr>
<td>rmgpy.molecule</td>
<td>Molecular representations using chemical graph theory</td>
</tr>
<tr>
<td>rmgpy.pdep</td>
<td>Pressure-dependent kinetics from master equation models</td>
</tr>
<tr>
<td>rmgpy.qm</td>
<td>On-the-fly quantum calculations</td>
</tr>
<tr>
<td>rmgpy.quantity</td>
<td>Physical quantities and unit conversions</td>
</tr>
<tr>
<td>rmgpy.reaction</td>
<td>Chemical reactions</td>
</tr>
<tr>
<td>rmgpy.rmg</td>
<td>Automatic reaction mechanism generation</td>
</tr>
<tr>
<td>rmgpy.solver</td>
<td>Modeling reaction systems</td>
</tr>
<tr>
<td>rmgpy.species</td>
<td>Chemical species</td>
</tr>
<tr>
<td>rmgpy.statmech</td>
<td>Statistical mechanics models of molecular degrees of freedom</td>
</tr>
<tr>
<td>rmgpy.thermo</td>
<td>Thermodynamics models of chemical species</td>
</tr>
</tbody>
</table>

### 1.1 CanTherm (rmgpy.cantherm)

The `rmgpy.cantherm` subpackage contains the main functionality for CanTherm, a tool for computing thermodynamic and kinetic properties of chemical species and reactions.

#### 1.1.1 Reading Gaussian log files

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GaussianLog</td>
<td>Extract chemical parameters from Gaussian log files</td>
</tr>
</tbody>
</table>

#### 1.1.2 Geometry

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>The three-dimensional geometry of a molecular conformation</td>
</tr>
</tbody>
</table>
1.1.3 Input

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadInputFile()</td>
<td>Load a CanTherm job input file</td>
</tr>
</tbody>
</table>

1.1.4 Job classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanTherm</td>
<td>Main class for CanTherm jobs</td>
</tr>
<tr>
<td>StatMechJob</td>
<td>Compute the molecular degrees of freedom for a molecular conformation</td>
</tr>
<tr>
<td>ThermoJob</td>
<td>Compute the thermodynamic properties of a species</td>
</tr>
<tr>
<td>KineticsJob</td>
<td>Compute the high pressure-limit rate coefficient for a reaction using transition state theory</td>
</tr>
<tr>
<td>PressureDependenceJob</td>
<td>Compute the phenomenological pressure-dependent rate coefficients $k(T, P)$ for a unimolecular reaction network</td>
</tr>
</tbody>
</table>

1.1.5 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GaussianError</td>
<td>Raised when an error occurs while working with a Gaussian log file</td>
</tr>
</tbody>
</table>

**rmgpy.cantherm.gaussian.GaussianLog**

class rmgpy.cantherm.gaussian.GaussianLog(path)

Represent a log file from Gaussian. The attribute *path* refers to the location on disk of the Gaussian log file of interest. Methods are provided to extract a variety of information into CanTherm classes and/or NumPy arrays.

getNumberOfAtoms()

Return the number of atoms in the molecular configuration used in the Gaussian log file.

loadConformer(symmetry=None, spinMultiplicity=None, opticalIsomers=1)

Load the molecular degree of freedom data from a log file created as the result of a Gaussian “Freq” quantum chemistry calculation. As Gaussian’s guess of the external symmetry number is not always correct, you can use the *symmetry* parameter to substitute your own value; if not provided, the value in the Gaussian log file will be adopted. In a log file with multiple Thermochemistry sections, only the last one will be kept.

loadEnergy(frequencyScaleFactor=1.0)

Load the energy in J/mol from a Gaussian log file. The file is checked for a complete basis set extrapolation; if found, that value is returned. Only the last energy in the file is returned. The zero-point energy is not included in the returned value; it is removed from the CBS-QB3 value.

loadForceConstantMatrix()

Return the force constant matrix from the Gaussian log file. The job that generated the log file must have the option iop(7/33=1) in order for the proper force constant matrix (in Cartesian coordinates) to be printed in the log file. If multiple such matrices are identified, only the last is returned. The units of the returned force constants are J/m^2. If no force constant matrix can be found in the log file, None is returned.

loadGeometry()

Return the optimum geometry of the molecular configuration from the Gaussian log file. If multiple such geometries are identified, only the last is returned.
loadNegativeFrequency()
Return the negative frequency from a transition state frequency calculation in cm^-1.

loadScanEnergies()
Extract the optimized energies in J/mol from a log file, e.g. the result of a Gaussian “Scan” quantum
chemistry calculation.

loadZeroPointEnergy()
Load the unscaled zero-point energy in J/mol from a Gaussian log file.

rmgpy.cantherm.geometry.Geometry

class rmgpy.cantherm.geometry.Geometry(coordinates, number, mass)
The three-dimensional geometry of a molecular configuration. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinates</td>
<td>numpy.ndarray</td>
<td>An N x 3 array containing the 3D coordinates of each atom</td>
</tr>
<tr>
<td>number</td>
<td>numpy.ndarray</td>
<td>An array containing the integer atomic number of each atom</td>
</tr>
<tr>
<td>mass</td>
<td>numpy.ndarray</td>
<td>An array containing the atomic mass in kg/mol of each atom</td>
</tr>
</tbody>
</table>

The integer index of each atom is consistent across all three attributes.

getCenterOfMass(Atoms=None)
Calculate and return the [three-dimensional] position of the center of mass of the current geometry. If a
list Atoms of atoms is specified, only those atoms will be used to calculate the center of mass. Otherwise,
all atoms will be used.

getInternalReducedMomentOfInertia(pivots, top1)
Calculate and return the reduced moment of inertia for an internal torsional rotation around the axis defined
by the two atoms in pivots. The list top1 contains the atoms that should be considered as part of the rotating
top; this list should contain the pivot atom connecting the top to the rest of the molecule. The procedure
used is that of Pitzer 1, which is described as $I^{(2,3)}$ by East and Radom 2. In this procedure, the molecule
is divided into two tops: those at either end of the hindered rotor bond. The moment of inertia of each top
is evaluated using an axis passing through the center of mass of both tops. Finally, the reduced moment of
inertia is evaluated from the moment of inertia of each top via the formula

$$\frac{1}{I^{(2,3)}} = \frac{1}{I_1} + \frac{1}{I_2}$$

getMomentOfInertiaTensor()
Calculate and return the moment of inertia tensor for the current geometry in kg*m^2. If the coordinates
are not at the center of mass, they are temporarily shifted there for the purposes of this calculation.

getPrincipalMomentsOfInertia()
Calculate and return the principal moments of inertia and corresponding principal axes for the current
geometry. The moments of inertia are in kg*m^2, while the principal axes have unit length.

g getTotalMass(Atoms=None)
Calculate and return the total mass of the atoms in the geometry in kg/mol. If a list Atoms of atoms is
specified, only those atoms will be used to calculate the center of mass. Otherwise, all atoms will be used.

CanTherm input files

rmgpy.cantherm.input.loadInputFile(path)
Load the CanTherm input file located at path on disk, and return a list of the jobs defined in that file.

**rmgpy.cantherm.KineticsJob**

class **rmgpy.cantherm.KineticsJob**(reaction, Tmin=None, Tmax=None, Tlist=None, Tcount=0)

A representation of a CanTherm kinetics job. This job is used to compute and save the high-pressure-limit kinetics information for a single reaction.

- **Tlist**
  The temperatures at which the k(T) values are computed.

- **Tmax**
  The maximum temperature at which the computed k(T) values are valid, or None if not defined.

- **Tmin**
  The minimum temperature at which the computed k(T) values are valid, or None if not defined.

**execute**(outputFile=None, plot=False)

Execute the kinetics job, saving the results to the given outputFile on disk.

**generateKinetics**(Tlist=None)

Generate the kinetics data for the reaction and fit it to a modified Arrhenius model.

**plot**(outputDirectory)

Plot both the raw kinetics data and the Arrhenius fit versus temperature. The plot is saved to the file kinetics.pdf in the output directory. The plot is not generated if matplotlib is not installed.

**save**(outputFile)

Save the results of the kinetics job to the file located at path on disk.

**rmgpy.cantherm.CanTherm**

class **rmgpy.cantherm.CanTherm**(inputFile=None, outputDirectory=None, verbose=20)

The CanTherm class represents an instance of CanTherm, a tool for computing properties of chemical species and reactions. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobList</td>
<td>A list of the jobs to execute</td>
</tr>
<tr>
<td>inputFile</td>
<td>The path of the input file defining the jobs to execute</td>
</tr>
<tr>
<td>outputDirectory</td>
<td>The directory in which to write the output files</td>
</tr>
<tr>
<td>verbose</td>
<td>The level of detail in the generated logging messages</td>
</tr>
</tbody>
</table>

The output directory defaults to the same directory as the input file if not explicitly specified.

To use this class programmatically, create an instance and set its attributes using either the __init__() method or by directly accessing the attributes, and then invoke the execute() method. You can also populate the attributes from the command line using the parseCommandLineArguments() method before running execute().

**execute()**

Execute, in order, the jobs found in input file specified by the inputFile attribute.

**initializeLog**(verbose=20, logFile=None)

Set up a logger for CanTherm to use to print output to stdout. The verbose parameter is an integer specifying the amount of log text seen at the console; the levels correspond to those of the logging module.

**loadInputFile**(inputFile)

Load a set of jobs from the given inputFile on disk. Returns the loaded set of jobs as a list.

**logFooter**(level=20)

Output a footer to the log.
**logHeader**(level=20)

Output a header containing identifying information about CanTherm to the log.

**parseCommandLineArguments()**

Parse the command-line arguments being passed to CanTherm. This uses the *argparse* module, which ensures that the command-line arguments are sensible, parses them, and returns them.

### Saving CanTherm output

**rmgpy.cantherm.output.prettify**(string, indent=4)

Return a “pretty” version of the given string, representing a snippet of Python code such as a representation of an object or function. This involves splitting of tuples, lists, and dicts (including parameter lists) onto multiple lines, indenting as appropriate for readability.

**class** **rmgpy.cantherm.output.PrettifyVisitor**(level=0, indent=4)

A class for traversing an abstract syntax tree to assemble a prettier version of the code used to create the tree. Used by the *prettify()* function.

**generic_visit**(node)

Called if no explicit visitor function exists for a node.

**visit**(node)

Visit a node.

**visit_Call**(node)

Return a pretty representation of the class or function call represented by node.

**visit_Dict**(node)

Return a pretty representation of the dict represented by node.

**visit_List**(node)

Return a pretty representation of the list represented by node.

**visit_Num**(node)

Return a pretty representation of the number represented by node.

**visit_Str**(node)

Return a pretty representation of the string represented by node.

**visit_Tuple**(node)

Return a pretty representation of the tuple represented by node.

**rmgpy.cantherm.PressureDependenceJob**

**class** **rmgpy.cantherm.PressureDependenceJob**(network, Tmin=None, Tmax=None, Tcount=0, Tlist=None, Pmin=None, Pmax=None, Pcount=0, Plist=None, maximumGrainSize=None, minimumGrainCount=0, method=None, interpolationModel=None, maximumAtoms=None, activeKRotor=True, activeJRotor=True, rmgmode=False)

A representation of a pressure dependence job. The attributes are:
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Tcount</td>
<td>The number of temperatures at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Pcount</td>
<td>The number of pressures at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Emin</td>
<td>The minimum energy to use to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Emax</td>
<td>The maximum energy to use to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>maximumGrainSize</td>
<td>The maximum energy grain size to use to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>minimumGrainCount</td>
<td>The minimum number of energy grains to use to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>method</td>
<td>The method to use to reduce the master equation to $k(T, P)$ values</td>
</tr>
<tr>
<td>interpolationModel</td>
<td>The interpolation model to fit to the computed $k(T, P)$ values</td>
</tr>
<tr>
<td>maximumAtoms</td>
<td>The maximum number of atoms to apply pressure dependence to (in RMG jobs)</td>
</tr>
<tr>
<td>activeKRotor</td>
<td>A flag indicating whether to treat the K-rotor as active or adiabatic</td>
</tr>
<tr>
<td>activeJRotor</td>
<td>A flag indicating whether to treat the J-rotor as active or adiabatic</td>
</tr>
<tr>
<td>rmgmode</td>
<td>A flag that toggles &quot;RMG mode&quot;, described below</td>
</tr>
<tr>
<td>network</td>
<td>The unimolecular reaction network</td>
</tr>
<tr>
<td>Tlist</td>
<td>An array of temperatures at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Plist</td>
<td>An array of pressures at which to compute $k(T, P)$ values</td>
</tr>
<tr>
<td>Elist</td>
<td>An array of energies to use to compute $k(T, P)$ values</td>
</tr>
</tbody>
</table>

In RMG mode, several alterations to the $k(T,P)$ algorithm are made both for speed and due to the nature of the approximations used:

- Densities of states are not computed for product channels
- Arbitrary rigid rotor moments of inertia are included in the active modes; these cancel in the ILT and equilibrium expressions
- $k(E)$ for each path reaction is computed in the direction A -> products, where A is always an explored isomer; the high-P kinetics are reversed if necessary for this purpose
- Thermodynamic parameters are always used to compute the reverse $k(E)$ from the forward $k(E)$ for each path reaction

RMG mode should be turned off by default except in RMG jobs.

**Plist**

The pressures at which the $k(T,P)$ values are computed.

**Pmax**

The maximum pressure at which the computed $k(T,P)$ values are valid, or `None` if not defined.

**Pmin**

The minimum pressure at which the computed $k(T,P)$ values are valid, or `None` if not defined.

**Tlist**

The temperatures at which the $k(T,P)$ values are computed.

**Tmax**

The maximum temperature at which the computed $k(T,P)$ values are valid, or `None` if not defined.

**Tmin**

The minimum temperature at which the computed $k(T,P)$ values are valid, or `None` if not defined.

**copy()**

Return a copy of the pressure dependence job.
**draw** (outputDirectory, format='pdf')
Generate a PDF drawing of the pressure-dependent reaction network. This requires that Cairo and its Python wrapper be available; if not, the drawing is not generated.

You may also generate different formats of drawings, by changing format to one of the following: pdf, svg, png.

**generatePressureList()**
Returns an array of pressures based on the interpolation model, minimum and maximum pressures \( P_{\text{min}} \) and \( P_{\text{max}} \) in Pa, and the number of pressures \( P_{\text{count}} \). For Chebyshev polynomials a Gauss-Chebyshev distribution is used; for all others a linear distribution on an logarithmic pressure domain is used. Note that the Gauss-Chebyshev grid does not place \( P_{\text{min}} \) and \( P_{\text{max}} \) at the endpoints, yet the interpolation is still valid up to these values.

**generateTemperatureList()**
Returns an array of temperatures based on the interpolation model, minimum and maximum temperatures \( T_{\text{min}} \) and \( T_{\text{max}} \) in K, and the number of temperatures \( T_{\text{count}} \). For Chebyshev polynomials a Gauss-Chebyshev distribution is used; for all others a linear distribution on an inverse temperature domain is used. Note that the Gauss-Chebyshev grid does not place \( T_{\text{min}} \) and \( T_{\text{max}} \) at the endpoints, yet the interpolation is still valid up to these values.

**maximumGrainSize**
The maximum allowed energy grain size, or None if not defined.

**saveInputFile** (path)
Save a CanTherm input file for the pressure dependence job to path on disk.

**rmgpy.cantherm.StatMechJob**

class rmgpy.cantherm.StatMechJob(species, path)
A representation of a CanTherm statistical mechanics job. This job is used to compute and save the statistical mechanics information for a single species or transition state.

**execute** (outputFile=None, plot=False)
Execute the statistical mechanics job, saving the results to the given outputFile on disk.

**load()**
Load the statistical mechanics parameters for each conformer from the associated files on disk. Creates Conformer objects for each conformer and appends them to the list of conformers on the species object.

**plotHinderedRotor** (angle, Vlist, cosineRotor, fourierRotor, rotor, rotorIndex, directory)
Plot the potential for the rotor, along with its cosine and Fourier series potential fits. The plot is saved to a set of files of the form hindered_rotor_1.pdf.

**save** (outputFile)
Save the results of the statistical mechanics job to the file located at path on disk.

**rmgpy.cantherm.ThermoJob**

class rmgpy.cantherm.ThermoJob(species, thermoClass)
A representation of a CanTherm thermodynamics job. This job is used to compute and save the thermodynamics information for a single species.

**execute** (outputFile=None, plot=False)
Execute the thermodynamics job, saving the results to the given outputFile on disk.
generateThermo()
Generate the thermodynamic data for the species and fit it to the desired heat capacity model (as specified in the thermoClass attribute).

plot(outputDirectory)
Plot the heat capacity, enthalpy, entropy, and Gibbs free energy of the fitted thermodynamics model, along with the same values from the statistical mechanics model that the thermodynamics model was fitted to. The plot is saved to the file thermo.pdf in the output directory. The plot is not generated if matplotlib is not installed.

save(outputFile)
Save the results of the thermodynamics job to the file located at path on disk.

1.2 Chemkin files (rmgpy.chemkin)
The rmgpy.chemkin module contains functions for reading and writing of Chemkin and Chemkin-like files.

1.2.1 Reading Chemkin files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadChemkinFile()</td>
<td>Load a reaction mechanism from a Chemkin file</td>
</tr>
<tr>
<td>loadSpeciesDictionary()</td>
<td>Load a species dictionary from a file</td>
</tr>
<tr>
<td>loadTransportFile()</td>
<td>Load a Chemkin transport properties file</td>
</tr>
<tr>
<td>readKineticsEntry()</td>
<td>Read a single reaction entry from a Chemkin file</td>
</tr>
<tr>
<td>readReactionComments()</td>
<td>Read the comments associated with a reaction entry</td>
</tr>
<tr>
<td>readReactionsBlock()</td>
<td>Read the reactions block of a Chemkin file</td>
</tr>
<tr>
<td>readThermoEntry()</td>
<td>Read a single thermodynamics entry from a Chemkin file</td>
</tr>
<tr>
<td>removeCommentFromLine()</td>
<td>Remove comment text from a line of a Chemkin file or species dictionary</td>
</tr>
</tbody>
</table>

1.2.2 Writing Chemkin files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>saveChemkinFile()</td>
<td>Save a reaction mechanism to a Chemkin file</td>
</tr>
<tr>
<td>saveSpeciesDictionary()</td>
<td>Save a species dictionary to a file</td>
</tr>
<tr>
<td>saveTransportFile()</td>
<td>Save a Chemkin transport properties file</td>
</tr>
<tr>
<td>saveHTMLFile()</td>
<td>Save an HTML file representing a Chemkin mechanism</td>
</tr>
<tr>
<td>saveJavaKineticsLibrary()</td>
<td>Save a mechanism to a (Chemkin-like) kinetics library for RMG-Java</td>
</tr>
<tr>
<td>getSpeciesIdentifier()</td>
<td>Return the Chemkin-valid identifier for a given species</td>
</tr>
<tr>
<td>markDuplicateReactions()</td>
<td>Find and mark all duplicate reactions in a mechanism</td>
</tr>
<tr>
<td>writeKineticsEntry()</td>
<td>Write a single reaction entry to a Chemkin file</td>
</tr>
<tr>
<td>writeThermoEntry()</td>
<td>Write a single thermodynamics entry to a Chemkin file</td>
</tr>
</tbody>
</table>

1.2.3 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChemkinError</td>
<td>Raised when an error occurs while working with a Chemkin file</td>
</tr>
</tbody>
</table>
Reading Chemkin files

Main functions

\texttt{rmgpy.chemkin.loadChemkinFile(path, dictionaryPath=None, transportPath=None, readComments=True, thermoPath=None, useChemkinNames=False)}

Load a Chemkin input file located at \texttt{path} on disk to \texttt{path}, returning lists of the species and reactions in the Chemkin file. The ‘thermoPath’ point to a separate thermo file, or, if ‘None’ is specified, the function will look for the thermo database within the chemkin mechanism file.

\texttt{rmgpy.chemkin.loadSpeciesDictionary(path)}

Load an RMG dictionary - containing species identifiers and the associated adjacency lists - from the file located at \texttt{path} on disk. Returns a dict mapping the species identifiers to the loaded species. Resonance isomers for each species are automatically generated.

\texttt{rmgpy.chemkin.loadTransportFile(path, speciesDict)}

Load a Chemkin transport properties file located at \texttt{path} and store the properties on the species in \texttt{speciesDict}.

Helper functions

\texttt{rmgpy.chemkin.readKineticsEntry(entry, speciesDict, Aunits, Eunits)}

Read a kinetics \texttt{entry} for a single reaction as loaded from a Chemkin file. The associated mapping of labels to species \texttt{speciesDict} should also be provided. Returns a \texttt{Reaction} object with the reaction and its associated kinetics.

\texttt{rmgpy.chemkin.readReactionComments(reaction, comments, read=True)}

Parse the \texttt{comments} associated with a given \texttt{reaction}. If the comments come from RMG (Py or Java), parse them and extract the useful information. Return the reaction object based on the information parsed from these comments. If \texttt{read} is False, the reaction is returned as an “Unclassified” \texttt{LibraryReaction}.

\texttt{rmgpy.chemkin.readReactionsBlock(f, speciesDict, readComments=True)}

Read a reactions block from a Chemkin file stream.

This function can also read the reactions.txt and pdepreactions.txt files from RMG-Java kinetics libraries, which have a similar syntax.

\texttt{rmgpy.chemkin.readThermoEntry(entry, Tmin=0, Tint=0, Tmax=0)}

Read a thermodynamics \texttt{entry} for one species in a Chemkin file. Returns the label of the species and the thermodynamics model as a \texttt{NASA} object.

Format specification at \url{http://www2.galcit.caltech.edu/EDL/public/formats/chemkin.html}

\texttt{rmgpy.chemkin.removeCommentFromLine(line)}

Remove a comment from a line of a Chemkin file or species dictionary file.

Returns the line and the comment. If the comment is encoded with latin-1, it is converted to utf-8.

Writing Chemkin files

Main functions

\texttt{rmgpy.chemkin.saveChemkinFile(path, species, reactions, verbose=True, checkForDuplicates=True)}

Save a Chemkin input file to \texttt{path} on disk containing the provided lists of \texttt{species} and \texttt{reactions}. If checkForDuplicates is False then we don’t check for unlabeled duplicate reactions, thus saving time (eg. if you are sure you’ve already labeled them as duplicate).
rmgpy.chemkin.saveSpeciesDictionary(path, species, oldStyle=False)

Save the given list of species as adjacency lists in a text file path on disk.

If oldStyle==True then it saves it in the old RMG-Java syntax.

rmgpy.chemkin.saveTransportFile(path, species)

Save a Chemkin transport properties file to path on disk containing the transport properties of the given list of species.

The syntax is from the Chemkin TRANSPORT manual. The first 16 columns in each line of the database are reserved for the species name (Presently CHEMKIN is programmed to allow no more than 16-character names.) Columns 17 through 80 are free-format, and they contain the molecular parameters for each species. They are, in order:

1. An index indicating whether the molecule has a monatomic, linear or nonlinear geometrical configuration. If the index is 0, the molecule is a single atom. If the index is 1 the molecule is linear, and if it is 2, the molecule is nonlinear.
2. The Lennard-Jones potential well depth $\epsilon/k_B$ in Kelvins.
3. The Lennard-Jones collision diameter $\sigma$ in Angstroms.
4. The dipole moment $\mu$ in Debye. Note: a Debye is $10^{-18} cm^3/2 erg^{1/2}$.
5. The polarizability $\alpha$ in cubic Angstroms.
6. The rotational relaxation collision number $Z_{rot}$ at 298K.
7. After the last number, a comment field can be enclosed in parenthesis.

rmgpy.chemkin.saveHTMLFile(path, readComments=True)

Save an output HTML file from the contents of a RMG-Java output folder.

rmgpy.chemkin.saveJavaKineticsLibrary(path, species, reactions)

Save the reaction files for a RMG-Java kinetics library: pdepreactions.txt and reactions.txt given a list of reactions, with species.txt containing the RMG-Java formatted dictionary.

**Helper functions**

rmgpy.chemkin.getSpeciesIdentifier(species)

Return a string identifier for the provided species that can be used in a Chemkin file. Although the Chemkin format allows up to 16 characters for a species identifier, this function uses a maximum of 10 to ensure that all reaction equations fit in the maximum limit of 52 characters.

rmgpy.chemkin.writeKineticsEntry(reaction, speciesList, verbose=True, javaLibrary=False)

Return a string representation of the reaction as used in a Chemkin file. Use verbose = True to turn on comments. Use javaLibrary = True in order to generate a kinetics entry suitable for an RMG-Java kinetics library.

rmgpy.chemkin.writeThermoEntry(species, verbose=True)

Return a string representation of the NASA model readable by Chemkin. To use this method you must have exactly two NASA polynomials in your model, and you must use the seven-coefficient forms for each.

rmgpy.chemkin.markDuplicateReactions(reactions)

For a given list of reactions, mark all of the duplicate reactions as understood by Chemkin.

This is pretty slow (quadratic in size of reactions list) so only call it if you’re really worried you may have undetected duplicate reactions.
1.3 Physical constants *(rmgpy.constants)*

The *rmgpy.constants* module contains module-level variables defining relevant physical constants relevant in chemistry applications. The recommended method of importing this module is

```python
import rmgpy.constants as constants
```

so as to not place the constants in the importing module’s global namespace.

The constants defined in this module are listed in the table below:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_h$</td>
<td>$E_h$</td>
<td>$4.39974434 \times 10^{-18}$ J</td>
<td>Hartree energy</td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>96485.3365 C/mol</td>
<td>Faraday constant</td>
</tr>
<tr>
<td>$G$</td>
<td>$G$</td>
<td>$6.67384 \times 10^{-11}$ m$^3$/kg$ \cdot$ s$^2$</td>
<td>Newtonian gravitational constant</td>
</tr>
<tr>
<td>$N_A$</td>
<td>$Na$</td>
<td>$6.02214179 \times 10^{23}$ mol$^{-1}$</td>
<td>Avogadro constant</td>
</tr>
<tr>
<td>$R$</td>
<td>$R$</td>
<td>8.314472 J/mol$ \cdot$ K</td>
<td>gas law constant</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$a0$</td>
<td>$5.2917721092 \times 10^{-11}$ m</td>
<td>Bohr radius</td>
</tr>
<tr>
<td>$c$</td>
<td>$c$</td>
<td>$299792458$ m/s</td>
<td>speed of light in a vacuum</td>
</tr>
<tr>
<td>$e$</td>
<td>$e$</td>
<td>$1.602176565 \times 10^{-19}$ C</td>
<td>elementary charge</td>
</tr>
<tr>
<td>$g$</td>
<td>$g$</td>
<td>$9.80665$ m/s$^2$</td>
<td>standard acceleration due to gravity</td>
</tr>
<tr>
<td>$h$</td>
<td>$h$</td>
<td>$6.62606896 \times 10^{-34}$ J$ \cdot$ s</td>
<td>Planck constant</td>
</tr>
<tr>
<td>$\hbar$</td>
<td>$\hbar$</td>
<td>$1.054571726 \times 10^{-34}$ J$ \cdot$ s</td>
<td>reduced Planck constant</td>
</tr>
<tr>
<td>$k_B$</td>
<td>$kB$</td>
<td>$1.3806504 \times 10^{-23}$ J/K</td>
<td>Boltzmann constant</td>
</tr>
<tr>
<td>$m_e$</td>
<td>$m_e$</td>
<td>$9.10938291 \times 10^{-31}$ kg</td>
<td>electron rest mass</td>
</tr>
<tr>
<td>$m_n$</td>
<td>$m_n$</td>
<td>$1.674927351 \times 10^{-27}$ kg</td>
<td>neutron rest mass</td>
</tr>
<tr>
<td>$m_p$</td>
<td>$m_p$</td>
<td>$1.672621777 \times 10^{-27}$ kg</td>
<td>proton rest mass</td>
</tr>
<tr>
<td>$m_u$</td>
<td>$amu$</td>
<td>$1.660538921 \times 10^{-27}$ kg</td>
<td>atomic mass unit</td>
</tr>
<tr>
<td>$\pi$</td>
<td>$\pi$</td>
<td>3.14159...</td>
<td></td>
</tr>
</tbody>
</table>

1.4 Database *(rmgpy.data)*

1.4.1 General classes

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Entry</em></td>
<td>An entry in a database</td>
</tr>
<tr>
<td><em>Database</em></td>
<td>A database of entries</td>
</tr>
<tr>
<td><em>LogicNode</em></td>
<td>A node in a database that represents a logical collection of entries</td>
</tr>
<tr>
<td><em>LogicAnd</em></td>
<td>A logical collection of entries, where all entries in the collection must match</td>
</tr>
<tr>
<td><em>LogicOr</em></td>
<td>A logical collection of entries, where any entry in the collection can match</td>
</tr>
<tr>
<td><em>makeLogicNode()</em></td>
<td>Create a <em>LogicNode</em> based on a string representation</td>
</tr>
</tbody>
</table>
1.4.2 Thermodynamics database

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThermoDepository</td>
<td>A depository of all thermodynamics parameters for one or more species</td>
</tr>
<tr>
<td>ThermoLibrary</td>
<td>A library of curated thermodynamics parameters for one or more species</td>
</tr>
<tr>
<td>ThermoGroups</td>
<td>A representation of a portion of a database for implementing the Benson group additivity method</td>
</tr>
<tr>
<td>ThermoDatabase</td>
<td>An entire thermodynamics database, including depositories, libraries, and groups</td>
</tr>
</tbody>
</table>

1.4.3 Kinetics database

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepositoryReaction</td>
<td>A reaction with kinetics determined from querying a kinetics depository</td>
</tr>
<tr>
<td>LibraryReaction</td>
<td>A reaction with kinetics determined from querying a kinetics library</td>
</tr>
<tr>
<td>TemplateReaction</td>
<td>A reaction with kinetics determined from querying a kinetics group additivity or rate rules method</td>
</tr>
<tr>
<td>ReactionRecipe</td>
<td>A sequence of actions that represent the process of a chemical reaction</td>
</tr>
<tr>
<td>KineticsDepository</td>
<td>A depository of all kinetics parameters for one or more reactions</td>
</tr>
<tr>
<td>KineticsLibrary</td>
<td>A library of curated kinetics parameters for one or more reactions</td>
</tr>
<tr>
<td>KineticsGroups</td>
<td>A set of group additivity values for a reaction family, organized in a tree</td>
</tr>
<tr>
<td>KineticsRules</td>
<td>A set of rate rules for a reaction family</td>
</tr>
<tr>
<td>KineticsFamily</td>
<td>A kinetics database for one reaction family, including depositories, libraries, groups, and rules</td>
</tr>
<tr>
<td>KineticsDatabase</td>
<td>A kinetics database for all reaction families, including depositories, libraries, groups, and rules</td>
</tr>
</tbody>
</table>

1.4.4 Statistical mechanics database

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupFrequencies</td>
<td>A set of characteristic frequencies for a group in the frequency database</td>
</tr>
<tr>
<td>StatmechDepository</td>
<td>A depository of all statistical mechanics parameters for one or more species</td>
</tr>
<tr>
<td>StatmechLibrary</td>
<td>A library of curated statistical mechanics parameters for one or more species</td>
</tr>
<tr>
<td>StatmechGroups</td>
<td>A set of characteristic frequencies for various functional groups, organized in a tree</td>
</tr>
<tr>
<td>StatmechDatabase</td>
<td>An entire statistical mechanics database, including depositories, libraries, and groups</td>
</tr>
</tbody>
</table>

1.4.5 Statistical mechanics fitting

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectFit</td>
<td>DQED class for fitting a small number of vibrational frequencies and hindered rotors</td>
</tr>
<tr>
<td>PseudoFit</td>
<td>DQED class for fitting a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both</td>
</tr>
<tr>
<td>PseudoRotorFit</td>
<td>DQED class for fitting a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only</td>
</tr>
<tr>
<td>fitStatmechDirect()</td>
<td>Directly fit a small number of vibrational frequencies and hindered rotors</td>
</tr>
<tr>
<td>fitStatmechPseudo()</td>
<td>Fit a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both</td>
</tr>
<tr>
<td>fitStatmechPseudoRotor()</td>
<td>Fit a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only</td>
</tr>
<tr>
<td>fitStatmechToHeatCapacity()</td>
<td>Fit vibrational and torsional degrees of freedom to heat capacity data</td>
</tr>
</tbody>
</table>
### 1.4.6 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseError</td>
<td>Raised when an error occurs while working with the database</td>
</tr>
<tr>
<td>InvalidActionError</td>
<td>Raised when an error occurs while applying a reaction recipe</td>
</tr>
<tr>
<td>UndeterminableKineticsError</td>
<td>Raised when the kinetics of a given reaction cannot be determined</td>
</tr>
<tr>
<td>StatmechFitError</td>
<td>Raised when an error occurs while fitting internal degrees of freedom to heat capacity data</td>
</tr>
</tbody>
</table>

#### rmgpy.data.base.Database

**class rmgpy.data.base.Database**

An RMG-style database, consisting of a dictionary of entries (associating items with data), and an optional tree for assigning a hierarchy to the entries. The use of the tree enables the database to be easily extensible as more parameters are available.

In constructing the tree, it is important to develop a hierarchy such that siblings are mutually exclusive, to ensure that there is a unique path of descent down a tree for each structure. If non-mutually exclusive siblings are encountered, a warning is raised and the parent of the siblings is returned.

There is no requirement that the children of a node span the range of more specific permutations of the parent. As the database gets more complex, attempting to maintain complete sets of children for each parent in each database rapidly becomes untenable, and is against the spirit of extensibility behind the database development.

You must derive from this class and implement the loadEntry(), saveEntry(), processOldLibraryEntry(), and generateOldLibraryEntry() methods in order to load and save from the new and old database formats.

- **ancestors(node)**
  - Returns all the ancestors of a node, climbing up the tree to the top.

- **areSiblings(node, nodeOther)**
  - Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

- **descendTree(structure, atoms, root=None, strict=False)**
  - Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
  - If root=None then uses the first matching top node.
  - Returns None if there is no matching root.
  - Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

- **descendants(node)**
  - Returns all the descendants of a node, climbing down the tree to the bottom.

- **generateOldTree(entries, level)**
  - Generate a multi-line string representation of the current tree using the old-style syntax.

- **getEntriesToSave()**
  - Return a sorted list of the entries in this database that should be saved to the output file.

- **getSpecies(path)**
  - Load the dictionary containing all of the species in a kinetics library or depository.
**load** *(path, local_context=None, global_context=None)*

Load an RMG-style database from the file at location `path` on disk. The parameters `local_context` and `global_context` are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

**loadOld** *(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)*

Load a dictionary-tree-library based database. The database is stored in three files: `dictstr` is the path to the dictionary, `treestr` to the tree, and `libstr` to the library. The tree is optional, and should be set to ‘’ if not desired.

**loadOldDictionary** *(path, pattern)*

Parse an old-style RMG database dictionary located at `path`. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of `pattern`.

**loadOldLibrary** *(path, numParameters, numLabels=1)*

Parse an RMG database library located at `path`.

**loadOldTree** *(path)*

Parse an old-style RMG database tree located at `path`. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

**matchNodeToChild** *(parentNode, childNode)*

Return True if `parentNode` is a parent of `childNode`. Otherwise, return False. Both `parentNode` and `childNode` must be Entry types with items containing Group or LogicNode types. If `parentNode` and `childNode` are identical, the function will also return False.

**matchNodeToNode** *(node, nodeOther)*

Return True if `node` and `nodeOther` are identical. Otherwise, return False. Both `node` and `nodeOther` must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure** *(node, structure, atoms, strict=False)*

Return True if the `structure` centered at `atom` matches the structure at `node` in the dictionary. The structure at `node` should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in `structure` may not have the correct labels, hence the `atoms` parameter. The `atoms` parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by `node` has an equivalent labeled atom in `structure`.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless `strict` is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary** *(path, numParameters, numLabels=1)*

Parse an RMG database library located at `path`, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**removeGroup** *(groupToRemove)*

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also
update the parent/child relationships

Returns the removed group

**save**(path)

Save the current database to the file at location *path* on disk.

**saveDictionary**(path)

Extract species from all entries associated with a kinetics library or depository and save them to the path given.

**saveOld**(dictstr, treestr, libstr)

Save the current database to a set of text files using the old-style syntax.

**saveOldDictionary**(path)

Save the current database dictionary to a text file using the old-style syntax.

**saveOldLibrary**(path)

Save the current database library to a text file using the old-style syntax.

**saveOldTree**(path)

Save the current database tree to a text file using the old-style syntax.

### rmgpy.data.kinetics.DepositoryReaction

class **rmgpy.data.kinetics.DepositoryReaction**(index=-1, reactants=None, products=None, kinetics=None, reversible=True, transitionState=None, duplicate=False, degeneracy=1, pairs=None, depository=None, family=None, entry=None)

A Reaction object generated from a reaction depository. In addition to the usual attributes, this class includes `depository` and `entry` attributes to store the library and the entry in that depository that it was created from.

**calculateMicrocanonicalRateCoefficient()**

Calculate the microcanonical rate coefficient \(k(E)\) for the reaction *reaction* at the energies *Elist* in J/mol. `reacDensStates` and `prodDensStates` are the densities of states of the reactant and product configurations for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.
- If the above is not possible but high-pressure limit kinetics \(k_\infty(T)\) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product `prodDensStates` and the temperature of interest *T* in K can also be provided. For isomerization and association reactions `prodDensStates` is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.

**calculateTSTRateCoefficient()**

Evaluate the forward rate coefficient for the reaction with corresponding transition state *TS* at temperature *T* in K using (canonical) transition state theory. The TST equation is

\[
k(T) = \kappa(T) \frac{k_B T}{\hbar} \frac{Q^T(T)}{Q^A(T)Q^B(T)} \exp \left( -\frac{E_0}{k_B T} \right)
\]

where \(Q^T\) is the partition function of the transition state, \(Q^A\) and \(Q^B\) are the partition function of the reactants, \(E_0\) is the ground-state energy difference from the transition state to the reactants, *T* is the
absolute temperature, $k_B$ is the Boltzmann constant, and $h$ is the Planck constant. $\kappa(T)$ is an optional tunneling correction.

canTST()
Return True if the necessary parameters are available for using transition state theory – or the microcanonical equivalent, RRKM theory – to compute the rate coefficient for this reaction, or False otherwise.

copy()
Create a deep copy of the current reaction.

draw()
Generate a pictorial representation of the chemical reaction using the draw module. Use path to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

fixBarrierHeight()
Turns the kinetics into Arrhenius (if they were ArrheniusEP) and ensures the activation energy is at least the endothermicity for endothermic reactions, and is not negative only as a result of using Evans Polanyi with an exothermic reaction. If forcePositive is True, then all reactions are forced to have a non-negative barrier.

fixDiffusionLimitedA()
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

generate3dTS()
Generate the 3D structure of the transition state. Called from model.generateKinetics().

self.reactants is a list of reactants self.products is a list of products

generatePairs()
Generate the reactant-product pairs to use for this reaction when performing flux analysis. The exact procedure for doing so depends on the reaction type:

<table>
<thead>
<tr>
<th>Reaction type</th>
<th>Template</th>
<th>Resulting pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isomerization</td>
<td>A -&gt; C</td>
<td>(A,C)</td>
</tr>
<tr>
<td>Dissociation</td>
<td>A -&gt; C + D</td>
<td>(A,C), (A,D)</td>
</tr>
<tr>
<td>Association</td>
<td>A + B -&gt; C</td>
<td>(A,C), (B,C)</td>
</tr>
<tr>
<td>Bimolecular</td>
<td>A + B -&gt; C + D</td>
<td>(A,C), (B,D) or (A,D), (B,C)</td>
</tr>
</tbody>
</table>

There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generateReverseRateCoefficient()
Generate and return a rate coefficient model for the reverse reaction. Currently this only works if the kinetics attribute is one of several (but not necessarily all) kinetics types.

getEnthalpiesOfReaction()
Return the enthalpies of reaction in J/mol evaluated at temperatures Tlist in K.

gGetEnthalpyOfReaction()
Return the enthalpy of reaction in J/mol evaluated at temperature T in K.

gGetEntropiesOfReaction()
Return the entropies of reaction in J/mol*K evaluated at temperatures Tlist in K.

gGetEntropyOfReaction()
Return the entropy of reaction in J/mol*K evaluated at temperature T in K.
getEquilibriumConstant()
Return the equilibrium constant for the reaction at the specified temperature $T$ in K. The type parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

generateEquilibriumConstants()
Return the equilibrium constants for the reaction at the specified temperatures $T_{list}$ in K. The type parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

generateFreeEnergiesOfReaction()
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures $T_{list}$ in K.

generateFreeEnergyOfReaction()
Return the Gibbs free energy of reaction in J/mol evaluated at temperature $T$ in K.

generateRateCoefficient()
Return the overall rate coefficient for the forward reaction at temperature $T$ in K and pressure $P$ in Pa, including any reaction path degeneracies.

If diffusionLimiter is enabled, the reaction is in the liquid phase and we use a diffusion limitation to correct the rate. If not, then use the intrinsic rate coefficient.

generateSource()
Return the database that was the source of this reaction. For a DepositoryReaction this should be a KineticsDepository object.

generateStoichiometricCoefficient()
Return the stoichiometric coefficient of species spec in the reaction. The stoichiometric coefficient is increased by one for each time spec appears as a product and decreased by one for each time spec appears as a reactant.

generateURL()
Get a URL to search for this reaction in the rmg website.

hasTemplate()
Return True if the reaction matches the template of reactants and products, which are both lists of Species objects, or False if not.

isAssociation()
Return True if the reaction represents an association reaction $A + B \leftrightarrow C$ or False if not.

isBalanced()
Return True if the reaction has the same number of each atom on each side of the reaction equation, or False if not.

isDissociation()
Return True if the reaction represents a dissociation reaction $A \leftrightarrow B + C$ or False if not.

isIsomerization()
Return True if the reaction represents an isomerization reaction $A \leftrightarrow B$ or False if not.

isIsomorphic()
Return True if this reaction is the same as the other reaction, or False if they are different. If eitherDirection=False then the directions must match.

isUnimolecular()
Return True if the reaction has a single molecule as either reactant or product (or both) $A \leftrightarrow B + C$ or $A + B \leftrightarrow C$ or $A \leftrightarrow B$, or False if not.
matchesMolecules()
Return True if the given reactants represent the total set of reactants or products for the current reaction, or False if not. The reactants should be Molecule objects.

reverseThisArrheniusRate()
Reverses the given kForward, which must be an Arrhenius type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

toCantera()
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

toChemkin()
Return the chemkin-formatted string for this reaction.
If kinetics is set to True, the chemkin format kinetics will also be returned (requires the speciesList to figure out third body colliders.) Otherwise, only the reaction string will be returned.

rmgpy.data.base.Entry
class rmgpy.data.base.Entry(index=-1, label='', item=None, parent=None, children=None, data=None, reference=None, referenceType='', shortDesc='', longDesc='', rank=None)
A class for representing individual records in an RMG database. Each entry in the database associates a chemical item (generally a species, functional group, or reaction) with a piece of data corresponding to that item. A significant amount of metadata can also be stored with each entry.

The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>A unique nonnegative integer index for the entry</td>
</tr>
<tr>
<td>label</td>
<td>A unique string identifier for the entry (or ‘’ if not used)</td>
</tr>
<tr>
<td>item</td>
<td>The item that this entry represents</td>
</tr>
<tr>
<td>parent</td>
<td>The parent of the entry in the hierarchy (or None if not used)</td>
</tr>
<tr>
<td>children</td>
<td>A list of the children of the entry in the hierarchy (or None if not used)</td>
</tr>
<tr>
<td>data</td>
<td>The data to associate with the item</td>
</tr>
<tr>
<td>reference</td>
<td>A Reference object containing bibliographic reference information to the source of the data</td>
</tr>
<tr>
<td>referenceType</td>
<td>The way the data was determined: ‘theoretical’, ‘experimental’, or ‘review’</td>
</tr>
<tr>
<td>shortDesc</td>
<td>A brief (one-line) description of the data</td>
</tr>
<tr>
<td>longDesc</td>
<td>A long, verbose description of the data</td>
</tr>
<tr>
<td>rank</td>
<td>An integer indicating the degree of confidence in the entry data, or None if not used</td>
</tr>
</tbody>
</table>

rmgpy.data.statmech.GroupFrequencies
class rmgpy.data.statmech.GroupFrequencies(frequencies=None, symmetry=1)
Represent a set of characteristic frequencies for a group in the frequency database. These frequencies are stored in the frequencies attribute, which is a list of tuples, where each tuple defines a lower bound, upper bound, and degeneracy. Each group also has a symmetry correction.

generateFrequencies(count=1)
Generate a set of frequencies. For each characteristic frequency group, the number of frequencies returned is degeneracy * count, and these are distributed linearly between the lower and upper bounds.
rmgpy.data.kinetics.KineticsDatabase

class rmgpy.data.kinetics.KineticsDatabase
A class for working with the RMG kinetics database.

extractSourceFromComments(reaction)

reaction: A reaction object containing kinetics data and kinetics data comments. Should be either a PDepReaction, LibraryReaction, or TemplateReaction object as loaded from the rmgpy.chemkin.loadChemkinFile function

 Parses the verbose string of comments from the thermo data of the species object, and extracts the thermo sources.

 Returns a dictionary with keys of either ‘Rate Rules’, ‘Training’, ‘Library’, or ‘PDep’. A reaction can only be estimated using one of these methods.

 source = {'RateRules': (Family_Label, OriginalTemplate, RateRules), ‘Library’: String_Name_of_Library_Used, ‘PDep’: Network_Index, ‘Training’: (Family_Label, Training_Reaction_Entry), }

generateReactions(reactants, products=None)

Generate all reactions between the provided list of one or two reactants, which should be Molecule objects. This method searches the depository, libraries, and groups, in that order.

generateReactionsFromFamilies(reactants, products, only_families=None)

Generate all reactions between the provided list of one or two reactants, which should be Molecule objects. This method applies the reaction family. If only_families is a list of strings, only families with those labels are used.

generateReactionsFromLibraries(reactants, products)

Generate all reactions between the provided list of one or two reactants, which should be Molecule objects. This method searches the depository.

generateReactionsFromLibrary(reactants, products, library)

Generate all reactions between the provided list of one or two reactants, which should be Molecule objects. This method searches the depository.

getForwardReactionForFamilyEntry(entry, family, thermoDatabase)

For a given entry for a reaction of the given reaction family (the string label of the family), return the reaction with kinetics and degeneracy for the “forward” direction as defined by the reaction family. For families that are their own reverse, the direction the kinetics is given in will be preserved. If the entry contains functional groups for the reactants, assume that it is given in the forward direction and do nothing. Returns the reaction in the direction consistent with the reaction family template, and the matching template. Note that the returned reaction will have its kinetics and degeneracy set appropriately.

In order to reverse the reactions that are given in the reverse of the direction the family is defined, we need to compute the thermodynamics of the reactants and products. For this reason you must also pass the thermoDatabase to use to generate the thermo data.

load(path, families=None, libraries=None, depositories=None)

Load the kinetics database from the given path on disk, where path points to the top-level folder of the families database.

loadFamilies(path, families=None, depositories=None)

Load the kinetics families from the given path on disk, where path points to the top-level folder of the kinetics families.

loadLibraries(path, libraries=None)

Load the listed kinetics libraries from the given path on disk.
Loads them all if libraries list is not specified or None. The path points to the folder of kinetics libraries in the database, and the libraries should be in files like <path>/<library>.py.

**loadOld**(path)

Load the old RMG kinetics database from the given path on disk, where path points to the top-level folder of the old RMG database.

**loadRecommendedFamiliesList**(filepath)

Load the list of recommended families from the given file. The file is usually ‘kinetics/families/recommended.py’. This is stored as a dictionary of True or False values (checked here), and should contain entries for every available family (checked in loadFamilies).

**reconstructKineticsFromSource**(reaction, source, fixBarrierHeight=False, forcePositiveBarrier=False)

Reaction is the original reaction with original kinetics. Note that for Library and PDep reactions this function does not do anything other than return the original kinetics...

You must enter source data in the appropriate format such as returned from returned from self.extractSourceFromComments, self-constructed. fixBarrierHeight and forcePositiveBarrier will change the kinetics based on the Reaction.fixBarrierHeight function. Return Arrhenius form kinetics if the source is from training reaction or rate rules.

**save**(path)

Save the kinetics database to the given path on disk, where path points to the top-level folder of the kinetics database.

**saveFamilies**(path)

Save the kinetics families to the given path on disk, where path points to the top-level folder of the kinetics families.

**saveLibraries**(path)

Save the kinetics libraries to the given path on disk, where path points to the top-level folder of the kinetics libraries.

**saveOld**(path)

Save the old RMG kinetics database to the given path on disk, where path points to the top-level folder of the old RMG database.

**saveRecommendedFamilies**(path)

Save the list of recommended families in a dictionary stored at path/recommended.py

---

**rmgpy.data.kinetics.KineticsDepository**

**class** rmgpy.data.kinetics.KineticsDepository *(label='', name='', shortDesc='', longDesc='')*

A class for working with an RMG kinetics depository. Each depository corresponds to a reaction family (a KineticsFamily object). Each entry in a kinetics depository involves a reaction defined either by a real reactant and product species (as in a kinetics library).

**ancestors**(node)

Returns all the ancestors of a node, climbing up the tree to the top.

**areSiblings**(node, nodeOther)

Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**descendTree**(structure, atoms, root=None, strict=False)

Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
If root=None then uses the first matching top node.
Returns None if there is no matching root.

Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)
Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldTree(entries, level)
Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()
Return a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)
Load the dictionary containing all of the species in a kinetics library or depository.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)
Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to '' if not desired.

loadOldDictionary(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

loadOldLibrary(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

loadOldTree(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

matchNodeToNode(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

matchNodeToStructure(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary(path, numParameters, numLabels=1)**

Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**removeGroup(groupToRemove)**

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships.

Returns the removed group

**save(path)**

Save the current database to the file at location path on disk.

**saveDictionary(path)**

Extract species from all entries associated with a kinetics library or depository and save them to the path given.

**saveEntry(f, entry)**

Write the given entry in the kinetics database to the file object f.

**saveOld(dicstr, treestr, libstr)**

Save the current database to a set of text files using the old-style syntax.

**saveOldDictionary(path)**

Save the current database dictionary to a text file using the old-style syntax.

**saveOldLibrary(path)**

Save the current database library to a text file using the old-style syntax.

**saveOldTree(path)**

Save the current database tree to a text file using the old-style syntax.

**rmgpy.data.kinetics.KineticsFamily**

A class for working with an RMG kinetics family: a set of reactions with similar chemistry, and therefore similar reaction rates. The attributes are:
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>string</td>
<td>The name of the reverse reaction family</td>
</tr>
<tr>
<td>forwardTemplate</td>
<td>Reaction</td>
<td>The forward reaction template</td>
</tr>
<tr>
<td>forwardRecipe</td>
<td>ReactionRecipe</td>
<td>The steps to take when applying the forward reaction to a set of reactants</td>
</tr>
<tr>
<td>reverseTemplate</td>
<td>Reaction</td>
<td>The reverse reaction template</td>
</tr>
<tr>
<td>reverseRecipe</td>
<td>ReactionRecipe</td>
<td>The steps to take when applying the reverse reaction to a set of reactants</td>
</tr>
<tr>
<td>forbidden</td>
<td>ForbiddenStructures</td>
<td>(Optional) Forbidden product structures in either direction</td>
</tr>
<tr>
<td>ownReverse</td>
<td>Boolean</td>
<td>It’s its own reverse?</td>
</tr>
<tr>
<td>groups</td>
<td>KineticsGroups</td>
<td>The set of kinetics group additivity values</td>
</tr>
<tr>
<td>rules</td>
<td>KineticsRules</td>
<td>The set of kinetics rate rules from RMG-Java</td>
</tr>
<tr>
<td>depositories</td>
<td>list</td>
<td>A set of additional depositories used to store kinetics data from various sources</td>
</tr>
</tbody>
</table>

There are a few reaction families that are their own reverse (hydrogen abstraction and intramolecular hydrogen migration); for these `reverseTemplate` and `reverseRecipe` will both be `None`.

### addAtomLabelsForReaction

```python
addAtomLabelsForReaction(reaction)
```

Apply atom labels on a reaction using the appropriate atom labels from this this reaction family. The reaction’s reactants and products must be lists of Molecule objects. The reaction is modified to use Species objects containing the labeled molecules after this function.

### addKineticsRulesFromTrainingSet

```python
addKineticsRulesFromTrainingSet(thermoDatabase=None)
```

For each reaction involving real reactants and products in the training set, add a rate rule for that reaction.

### ancestors

```python
ancestors(node)
```

Returns all the ancestors of a node, climbing up the tree to the top.

### applyRecipe

```python
applyRecipe(reactantStructures, forward=True, unique=True)
```

Apply the recipe for this reaction family to the list of Molecule objects `reactantStructures`. The atoms of the reactant structures must already be tagged with the appropriate labels. Returns a list of structures corresponding to the products after checking that the correct number of products was produced.

### areSiblings

```python
areSiblings(node, nodeOther)
```

Return `True` if `node` and `nodeOther` have the same parent node. Otherwise, return `False`. Both `node` and `nodeOther` must be Entry types with items containing Group or LogicNode types.

### calculateDegeneracy

```python
calculateDegeneracy(reaction)
```

For a `reaction` given in the direction in which the kinetics are defined, compute the reaction-path degeneracy.

### descendTree

```python
descendTree(structure, atoms, root=None, strict=False)
```

Descend the tree in search of the functional group node that best matches the local structure around `atoms` in `structure`.

If `root=None` then uses the first matching top node.

Returns `None` if there is no matching root.

Set `strict` to `True` if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

### descendants

```python
descendants(node)
```

Returns all the descendants of a node, climbing down the tree to the bottom.
**estimateKineticsUsingGroupAdditivity**(template, degeneracy=1)

Determine the appropriate kinetics for a reaction with the given template using group additivity.

**estimateKineticsUsingRateRules**(template, degeneracy=1)

Determine the appropriate kinetics for a reaction with the given template using rate rules.

**extractSourceFromComments**(reaction)

Returns the rate rule associated with the kinetics of a reaction by parsing the comments. Will return the template associated with the matched rate rule. Returns a tuple containing (Boolean_Is_Kinetics_From_Training_reaction, Source_Data)

For a training reaction, the Source_Data returns [Family_Label, Training_Reaction_Entry, Kinetics_In.Reverse?]

For a reaction from rate rules, the Source_Data is a tuple containing [Family_Label, {'template': original_Template, 'degeneracy': degeneracy, 'exact': boolean_exact?, 'rules': a list of (original rate rule entry, weight in average) 'training': a list of (original rate rule entry associated with training entry, original training entry, weight in average)]

Block quote ends without a blank line; unexpected unindent.

where TrainingReactions are ones that have created rules used in the estimate.

where Exact is a boolean of whether the rate is an exact match, Template is the reaction template used, and RateRules is a list of the rate rule entries containing the kinetics used

**fillKineticsRulesByAveragingUp**(verbose=False)

Fill in gaps in the kinetics rate rules by averaging child nodes recursively starting from the top level root template.

**generateOldTree**(entries, level)

Generate a multi-line string representation of the current tree using the old-style syntax.

**generateProductTemplate**(reactants0)

Generate the product structures by applying the reaction template to the top-level nodes. For reactants defined by multiple structures, only the first is used here; it is assumed to be the most generic.

**generateReactions**(reactants)

Generate all reactions between the provided list of one or two reactants, which should be either single Molecule objects or lists of same. Does not estimate the kinetics of these reactions at this time. Returns a list of TemplateReaction objects using Molecule objects for both reactants and products The reactions are constructed such that the forward direction is consistent with the template of this reaction family.

**getEntriesToSave**()

Return a sorted list of the entries in this database that should be saved to the output file.

**getKinetics**(reaction, template, degeneracy=1, estimator='', returnAllKinetics=True)

Return the kinetics for the given reaction by searching the various depositories as well as generating a result using the user-specified estimator of either ‘group additivity’ or ‘rate rules.’ Unlike the regular getKinetics() method, this returns a list of results, with each result comprising the kinetics, the source, and the entry. If it came from a template estimate, the source and entry will both be None. If returnAllKinetics==False, only the first (best?) matching kinetics is returned.

**getKineticsForTemplate**(template, degeneracy=1, method='rate rules')

Return an estimate of the kinetics for a reaction with the given template and reaction-path degeneracy. There are two possible methods to use: ‘group additivity’ (new RMG-Py behavior) and ‘rate rules’ (old RMG-Java behavior).
getKineticsFromDepository(depository, reaction, template, degeneracy)
Search the given depository in this kinetics family for kinetics for the given reaction. Returns a list of all of the matching kinetics, the corresponding entries, and True if the kinetics match the forward direction or False if they match the reverse direction.

getLabeledReactantsAndProducts(reactants, products)
Given reactants, a list of Molecule objects, and products, a list of Molecule objects, return two new lists of Molecule objects with atoms labeled: one for reactants, one for products. Returned molecules are totally new entities in memory so input molecules reactants and products won’t be affected. If RMG cannot find appropriate labels, (None, None) will be returned.

getRateRule(template)
Return the rate rule with the given template. Raises a ValueError if no corresponding entry exists.

getReactionPairs(reaction)
For a given reaction with properly-labeled Molecule objects as the reactants, return the reactant-product pairs to use when performing flux analysis.

getReactionTemplate(reaction)
For a given reaction with properly-labeled Molecule objects as the reactants, determine the most specific nodes in the tree that describe the reaction.

getReactionTemplateLabels(reaction)
Retrieve the template for the reaction and return the corresponding labels for each of the groups in the template.

getRootTemplate()
Return the root template for the reaction family. Most of the time this is the top-level nodes of the tree (as stored in the KineticsGroups object), but there are a few exceptions (e.g. R_Recombination).

getSourcesForTemplate(template)
Returns the set of rate rules and training reactions used to average this template. Note that the tree must be averaged with verbose=True for this to work.

Returns a tuple of rules, training
where rules are a list of tuples containing the [(original_entry, weight_used_in_average), ... ]
and training is a list of tuples containing the [(rate_rule_entry, training_reaction_entry, weight_used_in_average),...]

getAddress(path)
Load the dictionary containing all of the species in a kinetics library or depository.

getAddressDepository()
Returns the training depository from self.depositories

hasRateRule(template)
Return True if a rate rule with the given template currently exists, or False otherwise.

isMoleculeForbidden(molecule)
Return True if the molecule is forbidden in this family, or False otherwise.

load(path, local_context=None, global_context=None, depositoryLabels=None)
Load a kinetics database from a file located at path on disk.

If depositoryLabels is a list, eg. ['training', 'PrIMe'], then only those depositories are loaded, and they are searched in that order when generating kinetics.

If depositoryLabels is None then load ‘training’ first then everything else. If depositoryLabels is not None then load in the order specified in depositoryLabels.
loadForbidden(\texttt{(label, group, shortDesc=\textquotesingle\textquotesingle, longDesc=\textquotesingle\textquotesingle)})
Load information about a forbidden structure.

loadOld(\texttt{(path)})
Load an old-style RMG kinetics group additivity database from the location \texttt{path}.

loadOldDictionary(\texttt{(path, pattern)})
Parse an old-style RMG database dictionary located at \texttt{path}. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a \texttt{dict} object with the values converted to \texttt{Molecule} or \texttt{Group} objects depending on the value of \texttt{pattern}.

loadOldLibrary(\texttt{(path, numParameters, numLabels=1)})
Parse an RMG database library located at \texttt{path}.

loadOldTemplate(\texttt{(path)})
Load an old-style RMG reaction family template from the location \texttt{path}.

loadOldTree(\texttt{(path)})
Parse an old-style RMG database tree located at \texttt{path}. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

loadRecipe(\texttt{(actions)})
Load information about the reaction recipe.

loadTemplate(\texttt{(reactants, products, ownReverse=False)})
Load information about the reaction template.

\textbf{matchNodeToChild(\texttt{(parentNode, childNode)})}
Return \texttt{True} if \texttt{parentNode} is a parent of \texttt{childNode}. Otherwise, return \texttt{False}. Both \texttt{parentNode} and \texttt{childNode} must be Entry types with items containing Group or LogicNode types. If \texttt{parentNode} and \texttt{childNode} are identical, the function will also return \texttt{False}.

\textbf{matchNodeToNode(\texttt{(node, nodeOther)})}
Return \texttt{True} if \texttt{node} and \texttt{nodeOther} are identical. Otherwise, return \texttt{False}. Both \texttt{node} and \texttt{nodeOther} must be Entry types with items containing Group or LogicNode types.

\textbf{matchNodeToStructure(\texttt{(node, structure, atoms, strict=False)})}
Return \texttt{True} if the \texttt{structure} centered at \texttt{atom} matches the structure at \texttt{node} in the dictionary. The structure at \texttt{node} should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in \texttt{structure} may not have the correct labels, hence the \texttt{atoms} parameter. The \texttt{atoms} parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by \texttt{node} has an equivalent labeled atom in \texttt{structure}.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless \texttt{strict} is set to \texttt{True}.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to \texttt{True}, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

\textbf{parseOldLibrary(\texttt{(path, numParameters, numLabels=1)})}
Parse an RMG database library located at \texttt{path}, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.
removeGroup(\texttt{groupToRemove})

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group

retrieveOriginalEntry(\texttt{templateLabel})

Retrieves the original entry, be it a rule or training reaction, given the template label in the form ‘group1;group2’ or ‘group1:group2:group3’

Returns tuple in the form (RateRuleEntry, TrainingReactionEntry)

Where the TrainingReactionEntry is only present if it comes from a training reaction

retrieveTemplate(\texttt{templateLabels})

Reconstruct the groups associated with the labels of the reaction template and return a list.

save(\texttt{path})

Save the current database to the file at location \texttt{path} on disk.

saveDepository(\texttt{depository, path})

Save the given kinetics family \texttt{depository} to the location \texttt{path} on disk.

saveDictionary(\texttt{path})

Extract species from all entries associated with a kinetics library or depository and save them to the path given.

saveEntry(\texttt{f, entry})

Write the given \texttt{entry} in the thermo database to the file object \texttt{f}.

saveGroups(\texttt{path})

Save the current database to the file at location \texttt{path} on disk.

saveOld(\texttt{path})

Save the old RMG kinetics groups to the given \texttt{path} on disk.

saveOldDictionary(\texttt{path})

Save the current database dictionary to a text file using the old-style syntax.

saveOldLibrary(\texttt{path})

Save the current database library to a text file using the old-style syntax.

saveOldTemplate(\texttt{path})

Save an old-style RMG reaction family template from the location \texttt{path}.

saveOldTree(\texttt{path})

Save the current database tree to a text file using the old-style syntax.

saveTrainingReactions(\texttt{reactions, reference=None, referenceType='', shortDesc='', rank=3})

This function takes a list of reactions and appends it to the training reactions file. It ignores the existence of duplicate reactions.

The rank for each new reaction’s kinetics is set to a default value of 3 unless the user specifies differently for those reactions.

For each entry, the long description is imported from the kinetics comment.
rmgpy.data.kinetics.KineticsGroups

class rmgpy.data.kinetics.KineticsGroups(entries=None, top=None, label='', name='', shortDesc='', longDesc='', forwardRecipe=None, reverseRecipe=None, forwardTemplate=None, reverseTemplate=None, forbidden=None)

A class for working with an RMG kinetics family group additivity values.

ancestors(node)

Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)

Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

descendTree(structure, atoms, root=None, strict=False)

Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.

If root=None then uses the first matching top node.

Returns None if there is no matching root.

Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)

Returns all the descendants of a node, climbing down the tree to the bottom.

estimateKineticsUsingGroupAdditivity(template, referenceKinetics, degeneracy=1)

Determine the appropriate kinetics for a reaction with the given template using group additivity.

generateGroupAdditivityValues(trainingSet, kunits, method='Arrhenius')

Generate the group additivity values using the given trainingSet, a list of 2-tuples of the form (template, kinetics). You must also specify the kunits for the family and the method to use when generating the group values. Returns True if the group values have changed significantly since the last time they were fitted, or False otherwise.

generateOldTree(entries, level)

Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()

Return a sorted list of the entries in this database that should be saved to the output file.

getReactionTemplate(react)

For a given reaction with properly-labeled Molecule objects as the reactants, determine the most specific nodes in the tree that describe the reaction.

getSpecies(path)

Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)

Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)

Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to ‘’ if not desired.
**loadOldDictionary**(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

**loadOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

**loadOldTree**(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

**matchNodeToChild**(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

**matchNodeToNode**(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure**(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**removeGroup**(groupToRemove)
Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group.

**save**(path)
Save the current database to the file at location path on disk.

**saveDictionary**(path)
Extract species from all entries associated with a kinetics library or depository and save them to the path given.
saveOld(dictstr, treestr, libstr)
Save the current database to a set of text files using the old-style syntax.

saveOldDictionary(path)
Save the current database dictionary to a text file using the old-style syntax.

saveOldLibrary(path)
Save the current database library to a text file using the old-style syntax.

saveOldTree(path)
Save the current database tree to a text file using the old-style syntax.

rmgpy.data.kinetics.KineticsLibrary

class rmgpy.data.kinetics.KineticsLibrary(label='', name='', solvent=None, shortDesc='', longDesc='')
A class for working with an RMG kinetics library.

ancestors(node)
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)
Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

checkForDuplicates(markDuplicates=False)
Check that all duplicate reactions in the kinetics library are properly marked (i.e. with their duplicate attribute set to True). If markDuplicates is set to True, then ignore and mark all duplicate reactions as duplicate.

convertDuplicatesToMulti()
Merge all marked duplicate reactions in the kinetics library into single reactions with multiple kinetics.

descendTree(structure, atoms, root=None, strict=False)
Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
If root=None then uses the first matching top node.
Returns None if there is no matching root.
Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)
Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldTree(entries, level)
Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()
Return a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)
Load the dictionary containing all of the species in a kinetics library or depository.

loadOld(path)
Load an old-style RMG kinetics library from the location path.
**loadOldDictionary**(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

**loadOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

**loadOldTree**(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

**markValidDuplicates**(reactions1, reactions2)
Check for reactions that appear in both lists, and mark them as (valid) duplicates.

**matchNodeToChild**(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

**matchNodeToNode**(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure**(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**removeGroup**(groupToRemove)
Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group

**save**(path)
Save the current database to the file at location path on disk.

**saveDictionary**(path)
Extract species from all entries associated with a kinetics library or depository and save them to the path
saveEntry\(f, entry\)
Write the given \(entry\) in the kinetics library to the file object \(f\).

saveOld\(path\)
Save an old-style reaction library to \(path\). This creates files named \(\text{species.txt}\), \(\text{reactions.txt}\), and \(\text{pdep\_reactions.txt}\) in the given directory; these contain the species dictionary, high-pressure limit reactions and kinetics, and pressure-dependent reactions and kinetics, respectively.

saveOldDictionary\(path\)
Save the current database dictionary to a text file using the old-style syntax.

saveOldLibrary\(path\)
Save the current database library to a text file using the old-style syntax.

saveOldTree\(path\)
Save the current database tree to a text file using the old-style syntax.

rmgpy.data.kinetics.KineticsRules

class rmgpy.data.kinetics.KineticsRules(\(label='', name='', shortDesc='', longDesc='\))
A class for working with a set of “rate rules” for a RMG kinetics family.

ancestors\(node\)
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings\(node, nodeOther\)
Return \(True\) if \(node\) and \(nodeOther\) have the same parent node. Otherwise, return \(False\). Both \(node\) and \(nodeOther\) must be Entry types with items containing Group or LogicNode types.

calculateNormDistance\(template, otherTemplate\)
Calculate the norm distance squared between two rate rules with \(template\) and \(otherTemplate\). The norm distance is \(a^2 + b^2 + c^2 \ldots\) when \(a\) is the distance between the nodes in the first tree, \(b\) is the distance between the nodes in the second tree, etc.

descendTree\(structure, atoms, root=None, strict=False\)
Descend the tree in search of the functional group node that best matches the local structure around \(atoms\) in \(structure\).

If root=None then uses the first matching top node.

Returns None if there is no matching root.

Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants\(node\)
Returns all the descendants of a node, climbing down the tree to the bottom.

estimateKinetics\(template, degeneracy=1\)
Determine the appropriate kinetics for a reaction with the given \(template\) using rate rules.

fillRulesByAveragingUp\(rootTemplate, alreadyDone, verbose=False\)
Fill in gaps in the kinetics rate rules by averaging child nodes. If verbose is set to True, then exact sources of kinetics are saved in the kinetics comments (warning: this uses up a lot of memory due to the extensively long comments)

generateOldTree\(entries, level\)
Generate a multi-line string representation of the current tree using the old-style syntax.
**getAllRules**(template)
Return all of the exact rate rules with the given template. Raises a `ValueError` if no corresponding entry exists.

**getEntries**()
Return a list of all of the entries in the rate rules database, sorted by index.

**getEntriesToSave**()
Return a sorted list of all of the entries in the rate rules database to save.

**getRule**(template)
Return the exact rate rule with the given template, or `None` if no corresponding entry exists.

**getSpecies**(path)
Load the dictionary containing all of the species in a kinetics library or depository.

**hasRule**(template)
Return `True` if a rate rule with the given template currently exists, or `False` otherwise.

**load**(path, local_context=None, global_context=None)
Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

**loadOld**(path, groups, numLabels)
Load a set of old rate rules for kinetics groups into this depository.

**loadOldDictionary**(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to `Molecule` or `Group` objects depending on the value of pattern.

**loadOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

**loadOldTree**(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

**matchNodeToChild**(parentNode, childNode)
Return `True` if parentNode is a parent of childNode. Otherwise, return `False`. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return `False`.

**matchNodeToNode**(node, nodeOther)
Return `True` if node and nodeOther are identical. Otherwise, return `False`. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure**(node, structure, atoms, strict=False)
Return `True` if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary**(path, numParameters, numLabels=1)

Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**processOldLibraryEntry**(data)

Process a list of parameters data as read from an old-style RMG thermo database, returning the corresponding kinetics object.

**removeGroup**(groupToRemove)

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group

**save**(path)

Save the current database to the file at location path on disk.

**saveDictionary**(path)

Extract species from all entries associated with a kinetics library or depository and save them to the path given.

**saveEntry**(f, entry)

Write the given entry in the thermo database to the file object f.

**saveOld**(path, groups)

Save a set of old rate rules for kinetics groups from this depository.

**saveOldDictionary**(path)

Save the current database dictionary to a text file using the old-style syntax.

**saveOldLibrary**(path)

Save the current database library to a text file using the old-style syntax.

**saveOldTree**(path)

Save the current database tree to a text file using the old-style syntax.

**rmgpy.data.kinetics.LibraryReaction**

**class rmgpy.data.kinetics.LibraryReaction**(index=-1, reactants=None, products=None, kinetics=None, reversible=True, transitionState=None, duplicate=False, degeneracy=1, pairs=None, library=None, entry=None)

A Reaction object generated from a reaction library. In addition to the usual attributes, this class includes library and entry attributes to store the library and the entry in that library that it was created from.

**calculateMicrocanonicalRateCoefficient()**

Calculate the microcanonical rate coefficient \( k(E) \) for the reaction reaction at the energies Elist in J/mol. reacDensStates and prodDensStates are the densities of states of the reactant and product configurations for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction
is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.
- If the above is not possible but high-pressure limit kinetics \( k_\infty(T) \) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product \( \text{prodDensStates} \) and the temperature of interest \( T \) in K can also be provided. For isomerization and association reactions \( \text{prodDensStates} \) is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.

**calculateTSTRateCoefficient()**
Evaluate the forward rate coefficient for the reaction with corresponding transition state \( TS \) at temperature \( T \) in K using (canonical) transition state theory. The TST equation is

\[
k(T) = \kappa(T) \frac{k_B T}{h} \frac{Q^\dagger(T)}{Q^A(T)Q^B(T)} \exp \left( -\frac{E_0}{k_B T} \right)
\]

where \( Q^\dagger \) is the partition function of the transition state, \( Q^A \) and \( Q^B \) are the partition function of the reactants, \( E_0 \) is the ground-state energy difference from the transition state to the reactants, \( T \) is the absolute temperature, \( k_B \) is the Boltzmann constant, and \( h \) is the Planck constant. \( \kappa(T) \) is an optional tunneling correction.

**canTST()**
Return True if the necessary parameters are available for using transition state theory – or the microcanonical equivalent, RRKM theory – to compute the rate coefficient for this reaction, or False otherwise.

**copy()**
Create a deep copy of the current reaction.

**draw()**
Generate a pictorial representation of the chemical reaction using the draw module. Use path to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

**fixBarrierHeight()**
Turns the kinetics into Arrhenius (if they were ArrheniusEP) and ensures the activation energy is at least the endothermicity for endothermic reactions, and is not negative only as a result of using Evans Polanyi with an exothermic reaction. If forcePositive is True, then all reactions are forced to have a non-negative barrier.

**fixDiffusionLimitedA()**
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

**generate3dTS()**
Generate the 3D structure of the transition state. Called from model.generateKinetics().

self.reactants is a list of reactants self.products is a list of products

**generatePairs()**
Generate the reactant-product pairs to use for this reaction when performing flux analysis. The exact procedure for doing so depends on the reaction type.
### Reaction type | Template | Resulting pairs
---|---|---
Isomerization | A → C | (A,C)
Dissociation | A → C + D | (A,C), (A,D)
Association | A + B → C | (A,C), (B,C)
Bimolecular | A + B → C + D | (A,C), (B,D) or (A,D), (B,C)

There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

#### generateReverseRateCoefficient()
Generate and return a rate coefficient model for the reverse reaction. Currently this only works if the `kinetics` attribute is one of several (but not necessarily all) kinetics types.

#### getEnthalpiesOfReaction()
Return the enthalpies of reaction in J/mol evaluated at temperatures $T_{list}$ in K.

#### getEnthalpyOfReaction()
Return the enthalpy of reaction in J/mol evaluated at temperature $T$ in K.

#### getEntropiesOfReaction()
Return the entropies of reaction in J/mol*K evaluated at temperatures $T_{list}$ in K.

#### getEntropyOfReaction()
Return the entropy of reaction in J/mol*K evaluated at temperature $T$ in K.

#### getEquilibriumConstant()
Return the equilibrium constant for the reaction at the specified temperature $T$ in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

#### getEquilibriumConstants()
Return the equilibrium constants for the reaction at the specified temperatures $T_{list}$ in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

#### getFreeEnergiesOfReaction()
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures $T_{list}$ in K.

#### getFreeEnergyOfReaction()
Return the Gibbs free energy of reaction in J/mol evaluated at temperature $T$ in K.

#### getRateCoefficient()
Return the overall rate coefficient for the forward reaction at temperature $T$ in K and pressure $P$ in Pa, including any reaction path degeneracies.

If diffusionLimiter is enabled, the reaction is in the liquid phase and we use a diffusion limitation to correct the rate. If not, then use the intrinsic rate coefficient.

#### getSource()
Return the database that was the source of this reaction. For a LibraryReaction this should be a KineticsLibrary object.

#### getStoichiometricCoefficient()
Return the stoichiometric coefficient of species `spec` in the reaction. The stoichiometric coefficient is increased by one for each time `spec` appears as a product and decreased by one for each time `spec` appears as a reactant.

#### getURL()
Get a URL to search for this reaction in the rmg website.
hasTemplate()
Return True if the reaction matches the template of reactants and products, which are both lists of Species objects, or False if not.

isAssociation()
Return True if the reaction represents an association reaction $A + B \leftrightarrow C$ or False if not.

isBalanced()
Return True if the reaction has the same number of each atom on each side of the reaction equation, or False if not.

isDissociation()
Return True if the reaction represents a dissociation reaction $A \leftrightarrow B + C$ or False if not.

isIsomerization()
Return True if the reaction represents an isomerization reaction $A \leftrightarrow B$ or False if not.

isIsomorphic()
Return True if this reaction is the same as the other reaction, or False if they are different. If eitherDirection=False then the directions must match.

isUnimolecular()
Return True if the reaction has a single molecule as either reactant or product (or both) $A \leftrightarrow B + C$ or $A + B \leftrightarrow C$ or $A \leftrightarrow B$, or False if not.

matchesMolecules()
Return True if the given reactants represent the total set of reactants or products for the current reaction, or False if not. The reactants should be Molecule objects.

reverseThisArrheniusRate()
Reverses the given kForward, which must be an Arrhenius type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

toCantera()
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

toChemkin()
Return the chemkin-formatted string for this reaction.

If kinetics is set to True, the chemkin format kinetics will also be returned (requires the speciesList to figure out third body colliders.) Otherwise, only the reaction string will be returned.

rmgpy.data.base.LogicNode

class rmgpy.data.base.LogicNode(items, invert)
A base class for AND and OR logic nodes.

class rmgpy.data.base.LogicAnd(items, invert)
A logical AND node. Structure must match all components.

matchToStructure(database, structure, atoms, strict=False)
Does this node in the given database match the given structure with the labeled atoms?

Setting strict to True makes enforces matching of atomLabels in the structure to every atomLabel in the node.

class rmgpy.data.base.LogicOr(items, invert)
A logical OR node. Structure can match any component.

Initialize with a list of component items and a boolean instruction to invert the answer.
**getPossibleStructures**(*entries*)

Return a list of the possible structures below this node.

**matchLogicOr**(*other*)

Is other the same LogicOr group as self?

**matchToStructure**(*database, structure, atoms, strict=False*)

Does this node in the given database match the given structure with the labeled atoms?

Setting strict to True makes enforces matching of atomLabels in the structure to every atomLabel in the node.

**rmgpy.data.base.makeLogicNode**(*string*)

Creates and returns a node in the tree which is a logic node.

String should be of the form:

- OR{}
- AND{}
- NOT OR{}
- NOT AND{}

And the returned object will be of class LogicOr or LogicAnd

**rmgpy.data.kinetics.ReactionRecipe**

**class** **rmgpy.data.kinetics.ReactionRecipe**(*actions=None*)

Represent a list of actions that, when executed, result in the conversion of a set of reactants to a set of products. There are currently five such actions:

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE_BOND</td>
<td>center1, order, center2</td>
<td>change the bond order of the bond between center1 and center2 by order; do not break or form bonds</td>
</tr>
<tr>
<td>FORM_BOND</td>
<td>center1, order, center2</td>
<td>form a new bond between center1 and center2 of type order</td>
</tr>
<tr>
<td>BREAK_BOND</td>
<td>center1, order, center2</td>
<td>break the bond between center1 and center2, which should be of type order</td>
</tr>
<tr>
<td>GAIN_RADICAL</td>
<td>center, radical</td>
<td>increase the number of free electrons on center by radical</td>
</tr>
<tr>
<td>LOSE_RADICAL</td>
<td>center, radical</td>
<td>decrease the number of free electrons on center by radical</td>
</tr>
<tr>
<td>GAIN_PAIR</td>
<td>center, pair</td>
<td>increase the number of lone electron pairs on center by pair</td>
</tr>
<tr>
<td>LOSE_PAIR</td>
<td>center, pair</td>
<td>decrease the number of lone electron pairs on center by pair</td>
</tr>
</tbody>
</table>

The actions are stored as a list in the **actions** attribute. Each action is a list of items; the first is the action name, while the rest are the action parameters as indicated above.

**addAction**(*action*)

Add an action to the reaction recipe, where action is a list containing the action name and the required parameters, as indicated in the table above.

**applyForward**(*struct, unique=True*)

Apply the forward reaction recipe to molecule, a single Molecule object.

**applyReverse**(*struct, unique=True*)

Apply the reverse reaction recipe to molecule, a single Molecule object.
getReverse()
Generate a reaction recipe that, when applied, does the opposite of what the current recipe does, i.e., it is the recipe for the reverse of the reaction that this is the recipe for.

rmgpy.data.statmech.StatmechDatabase

class rmgpy.data.statmech.StatmechDatabase
A class for working with the RMG statistical mechanics (frequencies) database.

getStatmechData(molecule, thermoModel=None)
Return the thermodynamic parameters for a given Molecule object molecule. This function first searches the loaded libraries in order, returning the first match found, before falling back to estimation via group additivity.

getStatmechDataFromDepository(molecule)
Return statmech data for the given Molecule object molecule by searching the entries in the depository. Returns a list of tuples (statmechData, depository, entry).

getStatmechDataFromGroups(molecule, thermoModel)
Return statmech data for the given Molecule object molecule by estimating using characteristic group frequencies and fitting the remaining internal modes to heat capacity data from the given thermo model thermoModel. This always returns valid degrees of freedom data.

getStatmechDataFromLibrary(molecule, library)
Return statmech data for the given Molecule object molecule by searching the entries in the specified StatmechLibrary object library. Returns None if no data was found.

load(path, libraries=None, depository=True)
Load the statmech database from the given path on disk, where path points to the top-level folder of the thermo database.

loadDepository(path)
Load the statmech database from the given path on disk, where path points to the top-level folder of the thermo database.

loadGroups(path)
Load the statmech database from the given path on disk, where path points to the top-level folder of the thermo database.

loadLibraries(path, libraries=None)
Load the statmech database from the given path on disk, where path points to the top-level folder of the thermo database.

loadOld(path)
Load the old RMG thermo database from the given path on disk, where path points to the top-level folder of the old RMG database.

save(path)
Save the statmech database to the given path on disk, where path points to the top-level folder of the statmech database.

saveDepository(path)
Save the statmech depository to the given path on disk, where path points to the top-level folder of the statmech depository.

saveGroups(path)
Save the statmech groups to the given path on disk, where path points to the top-level folder of the statmech groups.
saveLibraries(path)
    Save the statmech libraries to the given path on disk, where path points to the top-level folder of the statmech libraries.

saveOld(path)
    Save the old RMG thermo database to the given path on disk, where path points to the top-level folder of the old RMG database.

rmgpy.data.statmech.StatmechDepository

class rmgpy.data.statmech.StatmechDepository(label='', name='', shortDesc='', longDesc='')
    A class for working with the RMG statistical mechanics (frequencies) depository.

ancestors(node)
    Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)
    Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

descendTree(structure, atoms, root=None, strict=False)
    Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
    If root=None then uses the first matching top node.
    Returns None if there is no matching root.
    Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)
    Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldTree(entries, level)
    Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()
    Return a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)
    Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)
    Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)
    Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to '' if not desired.

loadOldDictionary(path, pattern)
    Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.
loadOldLibrary\((path, numParameters, numLabels=1)\)
Parse an RMG database library located at \(path\).

loadOldTree\((path)\)
Parse an old-style RMG database tree located at \(path\). An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild\((parentNode, childNode)\)
Return True if \(parentNode\) is a parent of \(childNode\). Otherwise, return False. Both \(parentNode\) and \(childNode\) must be Entry types with items containing Group or LogicNode types. If \(parentNode\) and \(childNode\) are identical, the function will also return False.

matchNodeToNode\((node, nodeOther)\)
Return True if \(node\) and \(nodeOther\) are identical. Otherwise, return False. Both \(node\) and \(nodeOther\) must be Entry types with items containing Group or LogicNode types.

matchNodeToStructure\((node, structure, atoms, strict=False)\)
Return True if the \(structure\) centered at \(atom\) matches the structure at \(node\) in the dictionary. The structure at \(node\) should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in \(structure\) may not have the correct labels, hence the \(atoms\) parameter. The \(atoms\) parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by \(node\) has an equivalent labeled atom in \(structure\).

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless \(strict\) is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

parseOldLibrary\((path, numParameters, numLabels=1)\)
Parse an RMG database library located at \(path\), returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

removeGroup\((groupToRemove)\)
Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships.

Returns the removed group

save\((path)\)
Save the current database to the file at location \(path\) on disk.

saveDictionary\((path)\)
Extract species from all entries associated with a kinetics library or depository and save them to the path given.

saveEntry\((f, entry)\)
Write the given \(entry\) in the thermo database to the file object \(f\).

saveOld\(\text{dictstr}, \text{treestr}, \text{libstr}\)
Save the current database to a set of text files using the old-style syntax.

saveOldDictionary\((path)\)
Save the current database dictionary to a text file using the old-style syntax.
saveOldLibrary(path)
Save the current database library to a text file using the old-style syntax.

saveOldTree(path)
Save the current database tree to a text file using the old-style syntax.

rmgpy.data.statmechfit

Fitting functions

rmgpy.data.statmechfit.fitStatmechToHeatCapacity(Tlist, Cvlist, Nvib, Nrot, molecule=None)
For a given set of dimensionless heat capacity data Cvlist corresponding to temperature list Tlist in K, fit Nvib harmonic oscillator and Nrot hindered internal rotor modes. External and other previously-known modes should have already been removed from Cvlist prior to calling this function. You must provide at least 7 values for Cvlist.

This function returns a list containing the fitted vibrational frequencies in a HarmonicOscillator object and the fitted 1D hindered rotors in HinderedRotor objects.

rmgpy.data.statmechfit.fitStatmechDirect(Tlist, Cvlist, Nvib, Nrot, molecule=None)
Fit Nvib harmonic oscillator and Nrot hindered internal rotor modes to the provided dimensionless heat capacities Cvlist at temperatures Tlist in K. This method assumes that there are enough heat capacity points provided that the vibrational frequencies and hindered rotation frequency-barrier pairs can be fit directly.

rmgpy.data.statmechfit.fitStatmechPseudoRotors(Tlist, Cvlist, Nvib, Nrot, molecule=None)
Fit Nvib harmonic oscillator and Nrot hindered internal rotor modes to the provided dimensionless heat capacities Cvlist at temperatures Tlist in K. This method assumes that there are enough heat capacity points provided that the vibrational frequencies can be fit directly, but the hindered rotors must be combined into a single “pseudo-rotor”.

rmgpy.data.statmechfit.fitStatmechPseudo(Tlist, Cvlist, Nvib, Nrot, molecule=None)
Fit Nvib harmonic oscillator and Nrot hindered internal rotor modes to the provided dimensionless heat capacities Cvlist at temperatures Tlist in K. This method assumes that there are relatively few heat capacity points provided, so the vibrations must be combined into one real vibration and two “pseudo-vibrations” and the hindered rotors must be combined into a single “pseudo-rotor”.

Helper functions

rmgpy.data.statmechfit.harmonicOscillator_heatCapacity(T, freq)
Return the heat capacity in J/mol*K at the given set of temperatures Tlist in K for the harmonic oscillator with a frequency freq in cm^-1.

rmgpy.data.statmechfit.harmonicOscillator_d_heatCapacity_d_freq(T, freq)
Return the first derivative of the heat capacity with respect to the harmonic oscillator frequency in J/mol*K/cm^-1 at the given set of temperatures Tlist in K, evaluated at the frequency freq in cm^-1.

rmgpy.data.statmechfit.hinderedRotor_heatCapacity(T, freq, barr)
Return the heat capacity in J/mol*K at the given set of temperatures Tlist in K for the 1D hindered rotor with a frequency freq in cm^-1 and a barrier height barr in cm^-1.

rmgpy.data.statmechfit.hinderedRotor_d_heatCapacity_d_freq(T, freq, barr)
Return the first derivative of the heat capacity with respect to the hindered rotor frequency in J/mol*K/cm^-1 at the given set of temperatures Tlist in K, evaluated at the frequency freq in cm^-1 and a barrier height barr in cm^-1.
rmgpy.data.statmechfit.hinderedRotor_d_heatCapacity_d_barr(T, freq, barr)

Return the first derivative of the heat capacity with respect to the hindered rotor frequency in J/mol*K/cm^-1 at the given set of temperatures Tlist in K, evaluated at the frequency freq in cm^-1 and a barrier height barr in cm^-1.

Helper classes

class rmgpy.data.statmechfit.DirectFit(Tdata, Cvdata, Nvib, Nrot)

Class for fitting vibrational frequencies and hindered rotor frequency-barrier pairs for the case when there are few enough oscillators and rotors that their values can be fit directly.

initialize()

Initialize the DQED solver. The required parameters are:

• Neq - The number of algebraic equations.
• Nvars - The number of unknown variables.
• Ncons - The number of constraint equations.

The optional parameters are:

• bounds - A list of 2-tuples giving the lower and upper bound for each unknown variable. Use None if there is no bound in one or either direction. If provided, you must give bounds for every unknown variable.
• tolf - The tolerance used for stopping when the norm of the residual has absolute length less than tolf, i.e. \( \| \vec{f} \| \leq \epsilon_f \).
• told - The tolerance used for stopping when changes to the unknown variables has absolute length less than told, i.e. \( \| \Delta \vec{x} \| \leq \epsilon_d \).
• tolx - The tolerance used for stopping when changes to the unknown variables has relative length less than tolx, i.e. \( \| \Delta \vec{x} \| \leq \epsilon_x : \| \vec{x} \| \).
• maxIter - The maximum number of iterations to use
• verbose - True to have DQED print extra information about the solve, False to only see printed output when the solver has an error.

solve()

Using the initial guess \( \vec{x}_0 \), return the least-squares solution to the set of nonlinear algebraic equations defined by the evaluate() method of the derived class. This is the method that actually conducts the call to DQED. Returns the solution vector and a flag indicating the status of the solve. The possible output values of the flag are:
### Class: PseudoRotorFit

Class for fitting vibrational frequencies and hindered rotor frequency-barrier pairs for the case when there are too many oscillators and rotors for their values can be fit directly, and where collapsing the rotors into a single pseudo-rotor allows for fitting the vibrational frequencies directly.

#### initialize()

Initialize the DQED solver. The required parameters are:

- **Neq** - The number of algebraic equations.
- **Nvars** - The number of unknown variables.
- **Ncons** - The number of constraint equations.

The optional parameters are:

- **bounds** - A list of 2-tuples giving the lower and upper bound for each unknown variable. Use `None` if there is no bound in one or either direction. If provided, you must give bounds for every unknown variable.

- **tolf** - The tolerance used for stopping when the norm of the residual has absolute length less than `tolf`, i.e. \( \| \mathbf{f} \| \leq \epsilon_f \).

- **told** - The tolerance used for stopping when changes to the unknown variables has absolute length less than `told`, i.e. \( \| \Delta \mathbf{x} \| \leq \epsilon_d \).

- **tolx** - The tolerance used for stopping when changes to the unknown variables has relative length less than `tolx`, i.e. \( \| \Delta \mathbf{x} \| \leq \epsilon_x \cdot \| \mathbf{x} \| \).

- **maxIter** - The maximum number of iterations to use

- **verbose** - `True` to have DQED print extra information about the solve, `False` to only see printed output when the solver has an error.

#### solve()

Using the initial guess \( x_0 \), return the least-squares solution to the set of nonlinear algebraic equations defined by the `evaluate()` method of the derived class. This is the method that actually conducts the call to DQED. Returns the solution vector and a flag indicating the status of the solve. The possible output values of the flag are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>The norm of the residual is zero; the solution vector is a root of the system</td>
</tr>
<tr>
<td>3</td>
<td>The bounds on the trust region are being encountered on each step; the solution vector may or may not be a local minimum</td>
</tr>
<tr>
<td>4</td>
<td>The solution vector is a local minimum</td>
</tr>
<tr>
<td>5</td>
<td>A significant amount of noise or uncertainty has been observed in the residual; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>6</td>
<td>The solution vector is only changing by small absolute amounts; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>7</td>
<td>The solution vector is only changing by small relative amounts; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>8</td>
<td>The maximum number of iterations has been reached; the solution is the best found, but may or may not be a local minimum</td>
</tr>
<tr>
<td>9-18</td>
<td>An error occurred during the solve operation; the solution is not a local minimum</td>
</tr>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>2</td>
<td>The norm of the residual is zero; the solution vector is a root of the system</td>
</tr>
<tr>
<td>3</td>
<td>The bounds on the trust region are being encountered on each step; the solution vector may or may not be a local minimum</td>
</tr>
<tr>
<td>4</td>
<td>The solution vector is a local minimum</td>
</tr>
<tr>
<td>5</td>
<td>A significant amount of noise or uncertainty has been observed in the residual; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>6</td>
<td>The solution vector is only changing by small absolute amounts; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>7</td>
<td>The solution vector is only changing by small relative amounts; the solution may or may not be a local minimum</td>
</tr>
<tr>
<td>8</td>
<td>The maximum number of iterations has been reached; the solution is the best found, but may or may not be a local minimum</td>
</tr>
<tr>
<td>9-18</td>
<td>An error occurred during the solve operation; the solution is not a local minimum</td>
</tr>
</tbody>
</table>

class rmgpy.data.statmechfit.PseudoFit(Tdata, Cvdata, Nvib, Nrot)
Class for fitting vibrational frequencies and hindered rotor frequency-barrier pairs for the case when there are too many oscillators and rotors for their values can be fit directly, and where we must collapse both the vibrations and hindered rotations into “pseudo-oscillators” and “pseudo-rotors”.

initialize()
Initialize the DQED solver. The required parameters are:
- Neq - The number of algebraic equations.
- Nvars - The number of unknown variables.
- Ncons - The number of constraint equations.

The optional parameters are:
- bounds - A list of 2-tuples giving the lower and upper bound for each unknown variable. Use None if there is no bound in one or either direction. If provided, you must give bounds for every unknown variable.
- tolf - The tolerance used for stopping when the norm of the residual has absolute length less than tolf, i.e. \( \| \vec{f} \| \leq \epsilon_f \).
- told - The tolerance used for stopping when changes to the unknown variables has absolute length less than told, i.e. \( \| \Delta \vec{x} \| \leq \epsilon_d \).
- tolx - The tolerance used for stopping when changes to the unknown variables has relative length less than tolx, i.e. \( \| \Delta \vec{x} \| \leq \epsilon_x \cdot \| \vec{x} \| \).
- maxIter - The maximum number of iterations to use
- verbose - True to have DQED print extra information about the solve, False to only see printed output when the solver has an error.

solve()
Using the initial guess \( \vec{x}_0 \), return the least-squares solution to the set of nonlinear algebraic equations defined by the evaluate() method of the derived class. This is the method that actually conducts the call to DQED. Returns the solution vector and a flag indicating the status of the solve. The possible output values of the flag are:
### Value | Meaning
--- | ---
2 | The norm of the residual is zero; the solution vector is a root of the system
3 | The bounds on the trust region are being encountered on each step; the solution vector may or may not be a local minimum
4 | The solution vector is a local minimum
5 | A significant amount of noise or uncertainty has been observed in the residual; the solution may or may not be a local minimum
6 | The solution vector is only changing by small absolute amounts; the solution may or may not be a local minimum
7 | The solution vector is only changing by small relative amounts; the solution may or may not be a local minimum
8 | The maximum number of iterations has been reached; the solution is the best found, but may or may not be a local minimum
9-18 | An error occurred during the solve operation; the solution is not a local minimum

#### rmgpy.data.statmech.Stat mechGroups

class rmgpy.data.statmech.Stat mechGroups(
    label='',
    name='',
    shortDesc='',
    longDesc=''
)
A class for working with an RMG statistical mechanics (frequencies) group database.

- **ancestors**(node)
  Returns all the ancestors of a node, climbing up the tree to the top.

- **areSiblings**(node, nodeOther)
  Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

- **descendTree**(structure, atoms, root=None, strict=False)
  Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
  If root=None then uses the first matching top node.
  Returns None if there is no matching root.
  Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

- **descendants**(node)
  Returns all the descendants of a node, climbing down the tree to the bottom.

- **generateOldLibraryEntry**(data)
  Return a list of values used to save entries to the old-style RMG thermo database based on the thermodynamics object data.

- **generateOldTree**(entries, level)
  Generate a multi-line string representation of the current tree using the old-style syntax.

- **getEntriesToSave**()
  Return a sorted list of the entries in this database that should be saved to the output file.

- **getFrequencyGroups**(molecule)
  Return the set of characteristic group frequencies corresponding to the specified molecule. This is done by searching the molecule for certain functional groups for which characteristic frequencies are known, and using those frequencies.

- **getSpecies**(path)
  Load the dictionary containing all of the species in a kinetics library or depository.
getStatmechData(molecule, thermoModel)
Use the previously-loaded frequency database to generate a set of characteristic group frequencies corresponding to the specified molecule. The provided thermo data in thermoModel is used to fit some frequencies and all hindered rotors to heat capacity data.

load(path, local_context=None, global_context=None)
Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)
Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to '' if not desired.

loadOldDictionary(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

loadOldLibrary(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

loadOldTree(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

matchNodeToNode(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

matchNodeToStructure(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>
parseOldLibrary(path, numParameters, numLabels=1)
Parse an RMG database library located at path, returning the loaded entries (rather than storing them in
the database). This method does not discard duplicate entries.

processOldLibraryEntry(data)
Process a list of parameters data as read from an old-style RMG statmech database, returning the corre-
sponding thermodynamics object.

removeGroup(groupToRemove)
Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also
update the parent/child relationships

Returns the removed group

save(path)
Save the current database to the file at location path on disk.

saveDictionary(path)
Extract species from all entries associated with a kinetics library or depository and save them to the path
given.

saveEntry(f, entry)
Write the given entry in the thermo database to the file object f.

saveOld(dictstr, treestr, libstr)
Save the current database to a set of text files using the old-style syntax.

saveOldDictionary(path)
Save the current database dictionary to a text file using the old-style syntax.

saveOldLibrary(path)
Save the current database library to a text file using the old-style syntax.

saveOldTree(path)
Save the current database tree to a text file using the old-style syntax.

rmgpy.data.statmech.StatmechLibrary

class rmgpy.data.statmech.StatmechLibrary(label='', name='', shortDesc='', longDesc='')
A class for working with a RMG statistical mechanics (frequencies) library.

ancestors(node)
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)
Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and
nodeOther must be Entry types with items containing Group or LogicNode types.

descendTree(structure, atoms, root=None, strict=False)
Descend the tree in search of the functional group node that best matches the local structure around atoms
in structure.

If root=None then uses the first matching top node.

Returns None if there is no matching root.

Set strict to True if all labels in final matched node must match that of the structure. This is used in
kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species
generally not being prelabeled.

descendants(node)
Returns all the descendants of a node, climbing down the tree to the bottom.
generateOldLibraryEntry(data)
Return a list of values used to save entries to the old-style RMG thermo database based on the thermodynamics object data.

generateOldTree(entries, level)
Generate a multi-line string representation of the current tree using the old-style syntax.

generateEntriesToSave()
Return a sorted list of the entries in this database that should be saved to the output file.

generateSpecies(path)
Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)
Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)
Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to None if not desired.

loadOldDictionary(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

loadOldLibrary(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

loadOldTree(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

matchNodeToNode(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

matchNodeToStructure(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

### parseOldLibrary(path, numParameters, numLabels=1)

Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

### processOldLibraryEntry(data)

Process a list of parameters data as read from an old-style RMG thermo database, returning the corresponding thermodynamics object.

### removeGroup(groupToRemove)

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group

### save(path)

Save the current database to the file at location path on disk.

### saveDictionary(path)

Extract species from all entries associated with a kinetics library or depository and save them to the path given.

### saveEntry(f, entry)

Write the given entry in the thermo database to the file object f.

### saveOld(dictstr, treestr, libstr)

Save the current database to a set of text files using the old-style syntax.

### saveOldDictionary(path)

Save the current database dictionary to a text file using the old-style syntax.

### saveOldLibrary(path)

Save the current database library to a text file using the old-style syntax.

### saveOldTree(path)

Save the current database tree to a text file using the old-style syntax.

### rmgpy.data.kinetics.TemplateReaction

#### class rmgpy.data.kinetics.TemplateReaction(index=-1, reactants=None, products=None, kinetics=None, reversible=True, transitionState=None, duplicate=False, degeneracy=1, pairs=None, family=None, template=None, estimator=None, reverse=None)

A Reaction object generated from a reaction family template. In addition to the usual attributes, this class includes a family attribute to store the family that it was created from, as well as a estimator attribute to indicate whether it came from a rate rules or a group additivity estimate.

#### calculateMicrocanonicalRateCoefficient()

Calculate the microcanonical rate coefficient \( k(E) \) for the reaction \( reaction \) at the energies \( Elist \) in J/mol.
*reacDensStates* and *prodDensStates* are the densities of states of the reactant and product configurations for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.
- If the above is not possible but high-pressure limit kinetics \( k_\infty(T) \) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product *prodDensStates* and the temperature of interest \( T \) in K can also be provided. For isomerization and association reactions *prodDensStates* is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.

**calculateTSTRateCoefficient()**
Evaluate the forward rate coefficient for the reaction with corresponding transition state \( TS \) at temperature \( T \) in K using (canonical) transition state theory. The TST equation is

\[
k(T) = \kappa(T) k_B T \frac{Q^T(T)}{Q^A(T)Q^B(T)} \exp \left( -\frac{E_0}{k_B T} \right)
\]

where \( Q^T \) is the partition function of the transition state, \( Q^A \) and \( Q^B \) are the partition function of the reactants, \( E_0 \) is the ground-state energy difference from the transition state to the reactants, \( T \) is the absolute temperature, \( k_B \) is the Boltzmann constant, and \( h \) is the Planck constant. \( \kappa(T) \) is an optional tunneling correction.

**canTST()**
Return True if the necessary parameters are available for using transition state theory – or the microcanonical equivalent, RRKM theory – to compute the rate coefficient for this reaction, or False otherwise.

**copy()**
Create a deep copy of the current reaction.

**draw()**
Generate a pictorial representation of the chemical reaction using the draw module. Use *path* to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

**fixBarrierHeight()**
Turns the kinetics into Arrhenius (if they were ArrheniusEP) and ensures the activation energy is at least the endothermicity for endothermic reactions, and is not negative only as a result of using Evans Polanyi with an exothermic reaction. If *forcePositive* is True, then all reactions are forced to have a non-negative barrier.

**fixDiffusionLimitedA()**
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

**generate3dTS()**
Generate the 3D structure of the transition state. Called from model.generateKinetics().

def reactants is a list of reactants def products is a list of products

**generatePairs()**
Generate the reactant-product pairs to use for this reaction when performing flux analysis. The exact procedure for doing so depends on the reaction type:

1.4. Database (**rmgpy.data**)
<table>
<thead>
<tr>
<th>Reaction type</th>
<th>Template</th>
<th>Resulting pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isomerization</td>
<td>$A \rightarrow C$</td>
<td>$(A,C)$</td>
</tr>
<tr>
<td>Dissociation</td>
<td>$A \rightarrow C + D$</td>
<td>$(A,C), (A,D)$</td>
</tr>
<tr>
<td>Association</td>
<td>$A + B \rightarrow C$</td>
<td>$(A,C), (B,C)$</td>
</tr>
<tr>
<td>Bimolecular</td>
<td>$A + B \rightarrow C + D$</td>
<td>$(A,C), (B,D)$ or $(A,D), (B,C)$</td>
</tr>
</tbody>
</table>

There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

**generateReverseRateCoefficient()**
Generate and return a rate coefficient model for the reverse reaction. Currently this only works if the `kinetics` attribute is one of several (but not necessarily all) kinetics types.

**getEnthalpiesOfReaction()**
Return the enthalpies of reaction in J/mol evaluated at temperatures $T_{list}$ in K.

**getEnthalpyOfReaction()**
Return the enthalpy of reaction in J/mol evaluated at temperature $T$ in K.

**getEntropiesOfReaction()**
Return the entropies of reaction in J/mol*K evaluated at temperatures $T_{list}$ in K.

**getEntropyOfReaction()**
Return the entropy of reaction in J/mol*K evaluated at temperature $T$ in K.

**getEquilibriumConstant()**
Return the equilibrium constant for the reaction at the specified temperature $T$ in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

**getEquilibriumConstants()**
Return the equilibrium constants for the reaction at the specified temperatures $T_{list}$ in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

**getFreeEnergiesOfReaction()**
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures $T_{list}$ in K.

**getFreeEnergyOfReaction()**
Return the Gibbs free energy of reaction in J/mol evaluated at temperature $T$ in K.

**getRateCoefficient()**
Return the overall rate coefficient for the forward reaction at temperature $T$ in K and pressure $P$ in Pa, including any reaction path degeneracies.
If diffusionLimiter is enabled, the reaction is in the liquid phase and we use a diffusion limitation to correct the rate. If not, then use the intrinsic rate coefficient.

**getSource()**
Return the database that was the source of this reaction. For a TemplateReaction this should be a KineticsGroups object.

**getStoichiometricCoefficient()**
Return the stoichiometric coefficient of species $spec$ in the reaction. The stoichiometric coefficient is increased by one for each time $spec$ appears as a product and decreased by one for each time $spec$ appears as a reactant.

**getURL()**
Get a URL to search for this reaction in the rmg website.
hasTemplate()
  Return True if the reaction matches the template of reactants and products, which are both lists of Species objects, or False if not.

isAssociation()
  Return True if the reaction represents an association reaction $A + B \leftrightarrow C$ or False if not.

isBalanced()
  Return True if the reaction has the same number of each atom on each side of the reaction equation, or False if not.

isDissociation()
  Return True if the reaction represents a dissociation reaction $A \leftrightarrow B + C$ or False if not.

isIsomerization()
  Return True if the reaction represents an isomerization reaction $A \leftrightarrow B$ or False if not.

isIsomorphic()
  Return True if this reaction is the same as the other reaction, or False if they are different. If eitherDirection=False then the directions must match.

isUnimolecular()
  Return True if the reaction has a single molecule as either reactant or product (or both) $A \leftrightarrow B + C$ or $A + B \leftrightarrow C$ or $A \leftrightarrow B$, or False if not.

matchesMolecules()
  Return True if the given reactants represent the total set of reactants or products for the current reaction, or False if not. The reactants should be Molecule objects.

reverseThisArrheniusRate()
  Reverses the given kForward, which must be an Arrhenius type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

toCantera()
  Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

toChemkin()
  Return the chemkin-formatted string for this reaction.

  If kinetics is set to True, the chemkin format kinetics will also be returned (requires the speciesList to figure out third body colliders.) Otherwise, only the reaction string will be returned.

rmgpy.data.thermo.ThermoDatabase

class rmgpy.data.thermo.ThermoDatabase
  A class for working with the RMG thermodynamics database.

  computeGroupAdditivityThermo(molecule)
  Return the set of thermodynamic parameters corresponding to a given Molecule object molecule by estimation using the group additivity values. If no group additivity values are loaded, a DatabaseError is raised.

  The entropy is not corrected for the symmetry of the molecule. This should be done later by the calling function.

  estimateRadicalThermoViaHBI(molecule, stableThermoEstimator)
  Estimate the thermodynamics of a radical by saturating it, applying the provided stableThermoEstimator method on the saturated species, then applying hydrogen bond increment corrections for the radical site(s) and correcting for the symmetry.
estimateThermoViaGroupAdditivity(molecule)
Return the set of thermodynamic parameters corresponding to a given Molecule object molecule by estimation using the group additivity values. If no group additivity values are loaded, a DatabaseError is raised.

extractSourceFromComments(species)
species: A species object containing thermo data and thermo data comments
Parses the verbose string of comments from the thermo data of the species object, and extracts the thermo sources.
Returns a dictionary with keys of either ‘Library’, ‘QM’, and/or ‘GAV’. Commonly, species thermo are estimated using only one of these sources. However, a radical can be estimated with more than one type of source, for instance a saturated library value and a GAV HBI correction, or a QM saturated value and a GAV HBI correction.
The Dictionary_of_Groups_Used looks like {'groupType':[List of tuples containing (Entry, Weight)]

getAllThermoData(species)
Return all possible sets of thermodynamic parameters for a given Species object species. The hits from the depository come first, then the libraries (in order), and then the group additivity estimate. This method is useful for a generic search job.
Returns: a list of tuples (ThermoData, source, entry) (Source is a library or depository, or None)

getRingGroupsFromComments(thermoData)
Takes a string of comments from group additivity estimation, and extracts the ring and polycyclic ring groups from them, returning them as lists.

getThermoData(species, trainingSet=None)
Return the thermodynamic parameters for a given Species object species. This function first searches the loaded libraries in order, returning the first match found, before falling back to estimation via group additivity.
Returns: ThermoData

getThermoDataFromDepository(species)
Return all possible sets of thermodynamic parameters for a given Species object species from the depository. If no depository is loaded, a DatabaseError is raised.
Returns: a list of tuples (thermoData, depository, entry) without any Cp0 or CpInf data.

getThermoDataFromGroups(species)
Return the set of thermodynamic parameters corresponding to a given Species object species by estimation using the group additivity values. If no group additivity values are loaded, a DatabaseError is raised.
The resonance isomer (molecule) with the lowest H298 is used, and as a side-effect the resonance isomers (items in species.molecule list) are sorted in ascending order.
Returns: ThermoData

getThermoDataFromLibraries(species, trainingSet=None)
Return the thermodynamic parameters for a given Species object species. This function first searches the loaded libraries in order, returning the first match found, before failing and returning None. trainingSet is used to identify if function is called during training set or not. During training set calculation we want to use gas phase thermo to not affect reverse rate calculation.
Returns: ThermoData or None
**getThermoDataFromLibrary**(*species, library*)

Return the set of thermodynamic parameters corresponding to a given `Species` object `species` from the specified thermodynamics `library`. If `library` is a string, the list of libraries is searched for a library with that name. If no match is found in that library, `None` is returned. If no corresponding library is found, a `DatabaseError` is raised.

Returns a tuple: (ThermoData, library, entry) or None.

**load**(*path, libraries=None, depository=True*)

Load the thermo database from the given `path` on disk, where `path` points to the top-level folder of the thermo database.

**loadDepository**(*path*)

Load the thermo database from the given `path` on disk, where `path` points to the top-level folder of the thermo database.

**loadGroups**(*path*)

Load the thermo database from the given `path` on disk, where `path` points to the top-level folder of the thermo database.

**loadLibraries**(*path, libraries=None*)

Load the thermo database from the given `path` on disk, where `path` points to the top-level folder of the thermo database.

**loadOld**(*path*)

Load the old RMG thermo database from the given `path` on disk, where `path` points to the top-level folder of the old RMG database.

**prioritizeThermo**(*species, thermoDataList*)

Use some metrics to reorder a list of thermo data from best to worst. Return a list of indices with the desired order associated with the index of thermo from the data list.

**pruneHeteroatoms**(*allowed=['C', 'H', 'O', 'S']*)

Remove all species from thermo libraries that contain atoms other than those allowed.

This is useful before saving the database for use in RMG-Java

**save**(*path*)

Save the thermo database to the given `path` on disk, where `path` points to the top-level folder of the thermo database.

**saveDepository**(*path*)

Save the thermo depository to the given `path` on disk, where `path` points to the top-level folder of the thermo depository.

**saveGroups**(*path*)

Save the thermo groups to the given `path` on disk, where `path` points to the top-level folder of the thermo groups.

**saveLibraries**(*path*)

Save the thermo libraries to the given `path` on disk, where `path` points to the top-level folder of the thermo libraries.

**saveOld**(*path*)

Save the old RMG thermo database to the given `path` on disk, where `path` points to the top-level folder of the old RMG database.
rmgpy.data.thermo.ThermoDepository

class rmgpy.data.thermo.ThermoDepository(label='', name='', shortDesc='', longDesc='')  
A class for working with the RMG thermodynamics depository.

ancestors(node)  
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)  
Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

descendTree(structure, atoms, root=None, strict=False)  
Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.

If root=None then uses the first matching top node.

Returns None if there is no matching root.

Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)  
Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldTree(entries, level)  
Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()  
Return a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)  
Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)  
Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)  
Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to ‘’ if not desired.

loadOldDictionary(path, pattern)  
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

loadOldLibrary(path, numParameters, numLabels=1)  
Parse an RMG database library located at path.

loadOldTree(path)  
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild(parentNode, childNode)  
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and
childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

**matchNodeToNode**(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure**(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**removeGroup**(groupToRemove)
Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group

**save**(path)
Save the current database to the file at location path on disk.

**saveDictionary**(path)
Extract species from all entries associated with a kinetics library or depository and save them to the path given.

**saveEntry**(f, entry)
Write the given entry in the thermo database to the file object f.

**saveOld**(dictstr, treestr, libstr)
Save the current database to a set of text files using the old-style syntax.

**saveOldDictionary**(path)
Save the current database dictionary to a text file using the old-style syntax.

**saveOldLibrary**(path)
Save the current database library to a text file using the old-style syntax.

**saveOldTree**(path)
Save the current database tree to a text file using the old-style syntax.
rmgpy.data.thermo.ThermoGroups

class rmgpy.data.thermo.ThermoGroups(label='', name='', shortDesc='', longDesc='')
A class for working with an RMG thermodynamics group additivity database.

ancestors(node)
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)
Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and
nodeOther must be Entry types with items containing Group or LogicNode types.

copyData(source, destination)
This method copies the ThermoData object and all meta data from source to destination Args:
source: The entry for which data is being copied destination: The entry for which data is being
overwritten

descendTree(structure, atoms, root=None, strict=False)
Descend the tree in search of the functional group node that best matches the local structure around
atoms in structure.
If root=None then uses the first matching top node.
Returns None if there is no matching root.
Set strict to True if all labels in final matched node must match that of the structure. This is used in
kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species
generally not being prelabeled.

descendants(node)
Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldLibraryEntry(data)
Return a list of values used to save entries to the old-style RMG thermo database based on the thermody-
namics object data.

generateOldTree(entries, level)
Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()
Return a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)
Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)
Load an RMG-style database from the file at location path on disk. The parameters local_context and
global_context are used to provide specialized mapping of identifiers in the input file to corresponding
functions to evaluate. This method will automatically add a few identifiers required by all data entries, so
you don’t need to provide these.

loadOld(dictstr, treestr, libstr, numParameters, numLabels=1, pattern=True)
Load a dictionary-tree-library based database. The database is stored in three files: dictstr is the path to
the dictionary, treestr to the tree, and libstr to the library. The tree is optional, and should be set to ‘’ if not
desired.

loadOldDictionary(path, pattern)
Parse an old-style RMG database dictionary located at path. An RMG dictionary is a list of key-value
pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of pattern.

**loadOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path.

**loadOldTree**(path)
Parse an old-style RMG database tree located at path. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

**matchNodeToChild**(parentNode, childNode)
Return True if parentNode is a parent of childNode. Otherwise, return False. Both parentNode and childNode must be Entry types with items containing Group or LogicNode types. If parentNode and childNode are identical, the function will also return False.

**matchNodeToNode**(node, nodeOther)
Return True if node and nodeOther are identical. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

**matchNodeToStructure**(node, structure, atoms, strict=False)
Return True if the structure centered at atom matches the structure at node in the dictionary. The structure at node should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in structure may not have the correct labels, hence the atoms parameter. The atoms parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by node has an equivalent labeled atom in structure.

Matching to structure is more strict than to node. All labels in structure must be found in node. However the reverse is not true, unless strict is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>structure</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>atoms</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()</td>
</tr>
<tr>
<td>strict</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

**parseOldLibrary**(path, numParameters, numLabels=1)
Parse an RMG database library located at path, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

**processOldLibraryEntry**(data)
Process a list of parameters data as read from an old-style RMG thermo database, returning the corresponding thermodynamics object.

**removeGroup**(groupToRemove)
Removes a group that is in a tree from the database. For thermo groups we also, need to re-point any unicode thermoData that may have pointed to the entry.

Returns the removed group

**save**(path)
Save the current database to the file at location path on disk.

**saveDictionary**(path)
Extract species from all entries associated with a kinetics library or depository and save them to the path given.
saveEntry(f, entry)
Write the given entry in the thermo database to the file object f.

saveOld(dictstr, treestr, libstr)
Save the current database to a set of text files using the old-style syntax.

saveOldDictionary(path)
Save the current database dictionary to a text file using the old-style syntax.

saveOldLibrary(path)
Save the current database library to a text file using the old-style syntax.

saveOldTree(path)
Save the current database tree to a text file using the old-style syntax.

rmgpy.data.thermo.ThermoLibrary

class rmgpy.data.thermo.ThermoLibrary(label='', name='', solvent=None, shortDesc='', longDesc='')
A class for working with a RMG thermodynamics library.

ancestors(node)
Returns all the ancestors of a node, climbing up the tree to the top.

areSiblings(node, nodeOther)
Return True if node and nodeOther have the same parent node. Otherwise, return False. Both node and nodeOther must be Entry types with items containing Group or LogicNode types.

descendTree(structure, atoms, root=None, strict=False)
Descend the tree in search of the functional group node that best matches the local structure around atoms in structure.
If root=None then uses the first matching top node.

Returns None if there is no matching root.
Set strict to True if all labels in final matched node must match that of the structure. This is used in kinetics groups to find the correct reaction template, but not generally used in other GAVs due to species generally not being prelabeled.

descendants(node)
Returns all the descendants of a node, climbing down the tree to the bottom.

generateOldLibraryEntry(data)
Return a list of values used to save entries to the old-style RMG thermo database based on the thermodynamics object data.

generateOldTree(entries, level)
Generate a multi-line string representation of the current tree using the old-style syntax.

getEntriesToSave()
Returns a sorted list of the entries in this database that should be saved to the output file.

getSpecies(path)
Load the dictionary containing all of the species in a kinetics library or depository.

load(path, local_context=None, global_context=None)
Load an RMG-style database from the file at location path on disk. The parameters local_context and global_context are used to provide specialized mapping of identifiers in the input file to corresponding functions to evaluate. This method will automatically add a few identifiers required by all data entries, so you don’t need to provide these.
loadOld\((\text{dictstr}, \text{treestr}, \text{libstr}, \text{numParameters}, \text{numLabels}=1, \text{pattern}=\text{True})\)

Load a dictionary-tree-library based database. The database is stored in three files: \text{dictstr} is the path to the dictionary, \text{treestr} to the tree, and \text{libstr} to the library. The tree is optional, and should be set to empty if not desired.

loadOldDictionary\((\text{path}, \text{pattern})\)

Parse an old-style RMG database dictionary located at \text{path}. An RMG dictionary is a list of key-value pairs of a one-line string key and a multi-line string value. Each record is separated by at least one empty line. Returns a dict object with the values converted to Molecule or Group objects depending on the value of \text{pattern}.

loadOldLibrary\((\text{path}, \text{numParameters}, \text{numLabels}=1)\)

Parse an RMG database library located at \text{path}.

loadOldTree\((\text{path})\)

Parse an old-style RMG database tree located at \text{path}. An RMG tree is an n-ary tree representing the hierarchy of items in the dictionary.

matchNodeToChild\((\text{parentNode}, \text{childNode})\)

Return True if \text{parentNode} is a parent of \text{childNode}. Otherwise, return False. Both \text{parentNode} and \text{childNode} must be Entry types with items containing Group or LogicNode types. If \text{parentNode} and \text{childNode} are identical, the function will also return False.

matchNodeToNode\((\text{node}, \text{nodeOther})\)

Return True if \text{node} and \text{nodeOther} are identical. Otherwise, return False. Both \text{node} and \text{nodeOther} must be Entry types with items containing Group or LogicNode types.

matchNodeToStructure\((\text{node}, \text{structure}, \text{atoms}, \text{strict}=\text{False})\)

Return True if the structure centered at \text{atom} matches the structure at \text{node} in the dictionary. The structure at \text{node} should have atoms with the appropriate labels because they are set on loading and never change. However, the atoms in \text{structure} may not have the correct labels, hence the \text{atoms} parameter. The \text{atoms} parameter may include extra labels, and so we only require that every labeled atom in the functional group represented by \text{node} has an equivalent labeled atom in \text{structure}.

Matching to structure is more strict than to node. All labels in structure must be found in node. However, the reverse is not true, unless \text{strict} is set to True.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{node}</td>
<td>Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item</td>
</tr>
<tr>
<td>\text{structure}</td>
<td>A Group or a Molecule</td>
</tr>
<tr>
<td>\text{atoms}</td>
<td>Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by \text{structure.getLabeledAtoms()}</td>
</tr>
<tr>
<td>\text{strict}</td>
<td>If set to True, ensures that all the node’s atomLabels are matched by in the structure</td>
</tr>
</tbody>
</table>

parseOldLibrary\((\text{path}, \text{numParameters}, \text{numLabels}=1)\)

Parse an RMG database library located at \text{path}, returning the loaded entries (rather than storing them in the database). This method does not discard duplicate entries.

processOldLibraryEntry\((\text{data})\)

Process a list of parameters \text{data} as read from an old-style RMG thermo database, returning the corresponding thermodynamics object.

removeGroup\((\text{groupToRemove})\)

Removes a group that is in a tree from the database. In addition to deleting from self.entries, it must also update the parent/child relationships

Returns the removed group
**save**(*path*)
Save the current database to the file at location *path* on disk.

**saveDictionary**(*path*)
Extract species from all entries associated with a kinetics library or depository and save them to the path given.

**saveEntry**(*f, entry*)
Write the given *entry* in the thermo database to the file object *f*.

**saveOld**(*dictstr, treestr, libstr*)
Save the current database to a set of text files using the old-style syntax.

**saveOldDictionary**(*path*)
Save the current database dictionary to a text file using the old-style syntax.

**saveOldLibrary**(*path*)
Save the current database library to a text file using the old-style syntax.

**saveOldTree**(*path*)
Save the current database tree to a text file using the old-style syntax.

### 1.5 Kinetics (**rmgpy.kinetics**)

The **rmgpy.kinetics** subpackage contains classes that represent various kinetics models of chemical reaction rates and models of quantum mechanical tunneling through an activation barrier.

#### 1.5.1 Pressure-independent kinetics models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KineticsData</td>
<td>A kinetics model based on a set of discrete rate coefficient points in temperature</td>
</tr>
<tr>
<td>Arrhenius</td>
<td>A kinetics model based on the (modified) Arrhenius expression</td>
</tr>
<tr>
<td>MultiArrhenius</td>
<td>A kinetics model based on a sum of Arrhenius expressions</td>
</tr>
</tbody>
</table>

#### 1.5.2 Pressure-dependent kinetics models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDepKineticsData</td>
<td>A kinetics model based on a set of discrete rate coefficient points in temperature and pressure</td>
</tr>
<tr>
<td>PDepArrhenius</td>
<td>A kinetics model based on a set of Arrhenius expressions for a range of pressures</td>
</tr>
<tr>
<td>MultiPDepArrhenius</td>
<td>A kinetics model based on a sum of <em>PDepArrhenius</em> expressions</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>A kinetics model based on a Chebyshev polynomial representation</td>
</tr>
<tr>
<td>ThirdBody</td>
<td>A low pressure-limit kinetics model based on the (modified) Arrhenius expression, with a third body</td>
</tr>
<tr>
<td>Lindemann</td>
<td>A kinetics model of pressure-dependent falloff based on the Lindemann model</td>
</tr>
<tr>
<td>Troe</td>
<td>A kinetics model of pressure-dependent falloff based on the Lindemann model with the Troe falloff factor</td>
</tr>
</tbody>
</table>
### 1.5.3 Tunneling models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wigner</td>
<td>A one-dimensional tunneling model based on the Wigner expression</td>
</tr>
<tr>
<td>Eckart</td>
<td>A one-dimensional tunneling model based on the (asymmetric) Eckart expression</td>
</tr>
</tbody>
</table>

#### `rmgpy.kinetics.KineticsData`

A kinetics model based on an array of rate coefficient data vs. temperature. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tdata</td>
<td>An array of temperatures at which rate coefficient values are known</td>
</tr>
<tr>
<td>kdata</td>
<td>An array of rate coefficient values</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

Pmax
The maximum pressure at which the model is valid, or `None` if not defined.

Pmin
The minimum pressure at which the model is valid, or `None` if not defined.

Tdata
An array of temperatures at which rate coefficient values are known.

Tmax
The maximum temperature at which the model is valid, or `None` if not defined.

Tmin
The minimum temperature at which the model is valid, or `None` if not defined.

comment
comment: str

discrepancy(self, KineticsModel otherKinetics) → double
Returns some measure of the discrepancy based on two different reaction models.

getRateCoefficient(self, double T, double P=0.0) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature T in K.

isIdenticalTo(self, KineticsModel otherKinetics) → bool
Returns `True` if the kdata and Tdata match. Returns `False` otherwise.

isPressureDependent(self) → bool
Return `False` since, by default, all objects derived from KineticsModel represent pressure-independent kinetics.

isSimilarTo(self, KineticsModel otherKinetics) → bool
Returns `True` if rates of reaction at temperatures 300,1000,1500,2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(self, double T) → bool
Return `True` if the temperature T in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.
**kdata**

An array of rate coefficient values.

**setCanteraKinetics(self, ctReaction, speciesList)**

Sets the kinetics for a cantera reaction object.

**toHTML(self)**

Return an HTML rendering.

---

**rmgpy.kinetics.Arrhenius**

```python
class rmgpy.kinetics.Arrhenius(A=None, n=0.0, Ea=None, T0=(1.0, 'K'), Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')
```

A kinetics model based on the (modified) Arrhenius equation. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>The preexponential factor</td>
</tr>
<tr>
<td>T0</td>
<td>The reference temperature</td>
</tr>
<tr>
<td>n</td>
<td>The temperature exponent</td>
</tr>
<tr>
<td>Ea</td>
<td>The activation energy</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The Arrhenius equation, given below, accurately reproduces the kinetics of many reaction families:

\[
k(T) = A \left( \frac{T}{T_0} \right)^n \exp \left( -\frac{E_a}{RT} \right)
\]

Above, \(A\) is the preexponential factor, \(T_0\) is the reference temperature, \(n\) is the temperature exponent, and \(E_a\) is the activation energy.

**A**

The preexponential factor.

**Ea**

The activation energy.

**Pmax**

The maximum pressure at which the model is valid, or None if not defined.

**Pmin**

The minimum pressure at which the model is valid, or None if not defined.

**T0**

The reference temperature.

**Tmax**

The maximum temperature at which the model is valid, or None if not defined.

**Tmin**

The minimum temperature at which the model is valid, or None if not defined.

**changeRate(self, double factor)**

Changes \(A\) factor in Arrhenius expression by multiplying it by a factor.

**changeT0(self, double T0)**

Changes the reference temperature used in the exponent to \(T0\) in K, and adjusts the preexponential factor accordingly.
discrepancy(self, KineticsModel otherKinetics) → double
Returns some measure of the discrepancy based on two different reaction models.

fitToData(self, ndarray Tlist, ndarray klist, str kunits, double T0=1, ndarray weights=None, bool three-Params=True)
Fit the Arrhenius parameters to a set of rate coefficient data $klist$ in units of $kunits$ corresponding to a set of temperatures $Tlist$ in K. A linear least-squares fit is used, which guarantees that the resulting parameters provide the best possible approximation to the data.

getRateCoefficient(self, double T, double $P=0.0$) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature $T$ in K.

isIdenticalTo(self, KineticsModel otherKinetics) → bool
Returns True if kinetics matches that of another kinetics model. Must match temperature and pressure range of kinetics model, as well as parameters: A, $n$, Ea, T0. (Shouldn’t have pressure range if it’s Arrhenius.) Otherwise returns False.

isPressureDependent(self) → bool
Return False since, by default, all objects derived from KineticsModel represent pressure-independent kinetics.

isSimilarTo(self, KineticsModel otherKinetics) → bool
Returns True if rates of reaction at temperatures 500, 1000, 1500, 2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(self, double T) → bool
Return True if the temperature $T$ in K is within the valid temperature range of the kinetic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

$n$
The temperature exponent.

setCanteraKinetics(self, ctReaction, speciesList)
Passes in a cantera ElementaryReaction() object and sets its rate to a Cantera Arrhenius() object.

toCanteraKinetics(self)
Converts the Arrhenius object to a cantera Arrhenius object $Arrhenius(A,b,E)$ where A is in units of m^3/kmol/s, b is dimensionless, and E is in J/kmol.

toHTML(self)
Return an HTML rendering.

rmgpy.kinetics.MultiArrhenius

class rmgpy.kinetics.MultiArrhenius(arrhenius=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')
A kinetics model based on a set of (modified) Arrhenius equations, which are summed to obtain the overall rate. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrhenius</td>
<td>A list of the Arrhenius kinetics</td>
</tr>
<tr>
<td>$T_{min}$</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>$T_{max}$</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>$P_{min}$</td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>$P_{max}$</td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>
Pmax
   The maximum pressure at which the model is valid, or None if not defined.

Pmin
   The minimum pressure at which the model is valid, or None if not defined.

Tmax
   The maximum temperature at which the model is valid, or None if not defined.

 Tmin
   The minimum temperature at which the model is valid, or None if not defined.

arrhenius
   arrhenius: list

changeRate(self, double factor)
   Change kinetics rate by a multiple factor.

comment
   comment: str

discrepancy(self, KineticsModel otherKinetics) → double
   Returns some measure of the discrepancy based on two different reaction models.

generateRateCoefficient(self, double T, double P=0.0) → double
   Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature T in K.

isIdenticalTo(self, KineticsModel otherKinetics) → bool
   Returns True if kinetics matches that of another kinetics model. Each duplicate reaction must be matched
   and equal to that in the other MultiArrhenius model in the same order. Otherwise returns False.

isPressureDependent(self) → bool
   Return False since, by default, all objects derived from KineticsModel represent pressure-independent
   kinetics.

isSimilarTo(self, KineticsModel otherKinetics) → bool
   Returns True if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/- .5
   for log(k), in other words, within a factor of 3.

isTemperatureValid(self, double T) → bool
   Return True if the temperature T in K is within the valid temperature range of the kinetic data, or False
   if not. If the minimum and maximum temperature are not defined, True is returned.

setCanteraKinetics(self, ctReaction, speciesList)
   Sets the kinetic rates for a list of cantera Reaction objects Here, ctReaction must be a list rather than a
   single cantera reaction.

toArrhenius(self, double Tmin=-1, double Tmax=-1) → Arrhenius
   Return an Arrhenius instance of the kinetics model
   Fit the Arrhenius parameters to a set of rate coefficient data generated from the MultiArrhenius kinetics,
   over the temperature range Tmin to Tmax, in Kelvin. If Tmin or Tmax are unspecified (or -1) then the
   MultiArrhenius's Tmin and Tmax are used. A linear least-squares fit is used, which guarantees that the
   resulting parameters provide the best possible approximation to the data.

toHTML(self)
   Return an HTML rendering.
rmgpy.kinetics.PDepKineticsData

class rmgpy.kinetics.PDepKineticsData(Tdata=None, Pdata=None, kdata=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')
A kinetics model based on an array of rate coefficient data vs. temperature and pressure. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tdata</td>
<td>An array of temperatures at which rate coefficient values are known</td>
</tr>
<tr>
<td>Pdata</td>
<td>An array of pressures at which rate coefficient values are known</td>
</tr>
<tr>
<td>kdata</td>
<td>An array of rate coefficient values at each temperature and pressure</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which the model is valid, or None if not defined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which the model is valid, or None if not defined</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

Pdata
An array of pressures at which rate coefficient values are known.

Pmax
The maximum pressure at which the model is valid, or None if not defined.

Pmin
The minimum pressure at which the model is valid, or None if not defined.

Tdata
An array of temperatures at which rate coefficient values are known.

Tmax
The maximum temperature at which the model is valid, or None if not defined.

Tmin
The minimum temperature at which the model is valid, or None if not defined.

comment
comment: str

discrepancy(self, KineticsModel otherKinetics) → double
Returns some measure of the discrepancy based on two different reaction models.

efficiencies
efficiencies: dict

getchanteraEfficiencies(self, speciesList)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction

getEffectiveColliderEfficiencies(self, list species) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(self, double P, list species, ndarray fractions) → double
Return the effective pressure in Pa for a system at a given pressure P in Pa composed of the given list of species (Species or Molecule objects) with the given fractions.

getRateCoefficient(self, double T, double P=0.0) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature T in K and pressure P in Pa.
**highPlimit**

`highPlimit: rmgpy.kinetics.model.KineticsModel`

**isIdenticalTo** *(self, KineticsModel otherKinetics) → bool*

Returns `True` if the kdata and Tdata match. Returns `False` otherwise.

**isPressureDependent** *(self) → bool*

Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

**isPressureValid** *(self, double P) → bool*

Return `True` if the pressure `P` in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

**isSimilarTo** *(self, KineticsModel otherKinetics) → bool*

Returns `True` if rates of reaction at temperatures 500, 1000, 1500, 2000 K and 1 and 10 bar are within +/-.5 for log(k), in other words, within a factor of 3.

**isTemperatureValid** *(self, double T) → bool*

Return `True` if the temperature `T` in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

**kdata**

An array of rate coefficient values at each temperature and pressure.

**setCanteraKinetics** *(self, ctReaction, speciesList)*

Sets the kinetics for a cantera reaction object.

**toHTML** *(self)*

Return an HTML rendering.

**rmgpy.kinetics.PDepArrhenius**

**class** `rmgpy.kinetics.PDepArrhenius(presses=None, arrhenius=None, highPlimit=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')`

A kinetic model of a phenomenological rate coefficient `k(T, P)` where a set of Arrhenius kinetics are stored at a variety of pressures and interpolated between on a logarithmic scale. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pressures</td>
<td>The list of pressures</td>
</tr>
<tr>
<td>arrhenius</td>
<td>The list of Arrhenius objects at each pressure</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure in bar at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure in bar at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>efficiencies</td>
<td>A dict associating chemical species with associated efficiencies</td>
</tr>
<tr>
<td>order</td>
<td>The reaction order (1 = first, 2 = second, etc.)</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The pressure-dependent Arrhenius formulation is sometimes used to extend the Arrhenius expression to handle pressure-dependent kinetics. The formulation simply parameterizes `A`, `n`, and `Ea` to be dependent on pressure:

\[
k(T, P) = A(P) \left(\frac{T}{T_0}\right)^n(P) \exp\left(-\frac{E_a(P)}{RT}\right)
\]

Although this suggests some physical insight, the `k(T, P)` data is often highly complex and non-Arrhenius, limiting the usefulness of this formulation to simple systems.
**Pmax**
The maximum pressure at which the model is valid, or *None* if not defined.

**Pmin**
The minimum pressure at which the model is valid, or *None* if not defined.

**Tmax**
The maximum temperature at which the model is valid, or *None* if not defined.

**Tmin**
The minimum temperature at which the model is valid, or *None* if not defined.

**arrhenius**
arrhenius: list

**changeRate** *(self, double factor)*
Changes kinetics rate by a multiple *factor*.

**comment**
comment: str

**discrepancy** *(self, KineticsModel otherKinetics) → double*
Returns some measure of the discrepancy based on two different reaction models.

**efficiencies**
efficiencies: dict

**fitToData** *(self, ndarray Tlist, ndarray Plist, ndarray K, str kunits, double T0=1)*
Fit the pressure-dependent Arrhenius model to a matrix of rate coefficient data *K* with units of *kunits* corresponding to a set of temperatures *Tlist* in K and pressures *Plist* in Pa. An Arrhenius model is fit cpdef changeRate(self, double factor) at each pressure.

**getCanteraEfficiencies** *(self, speciesList)*
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction*.

**getEffectiveColliderEfficiencies** *(self, list species) → ndarray*
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

**getEffectivePressure** *(self, double P, list species, ndarray fractions) → double*
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

**getRateCoefficient** *(self, double T, double P=0) → double*
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K and pressure *P* in Pa.

**highPlimit**
highPlimit: rmgpy.kinetics.model.KineticsModel

**isIdenticalTo** *(self, KineticsModel otherKinetics) → bool*
Returns *True* if kinetics matches that of another kinetics model. Each duplicate reaction must be matched and equal to that in the other PDepArrhenius model in the same order. Otherwise returns *False*.

**isPressureDependent** *(self) → bool*
Return *True* since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

**isPressureValid** *(self, double P) → bool*
Return *True* if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or *False* if not. If the minimum and maximum pressure are not defined, *True* is returned.
**isSimilarTo** *(self, KineticsModel otherKinetics) → bool*

Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

**isTemperatureValid** *(self, double T) → bool*

Return `True` if the temperature `T` in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

**pressures**

The list of pressures.

**setCanteraKinetics** *(self, ctReaction, speciesList)*

Sets a Cantera PlogReaction()'s `rates` attribute with a list of tuples containing [(pressure in Pa, cantera arrhenius object), (..)]

**toHTML** *(self)*

Return an HTML rendering.

### `rmgpy.kinetics.MultiPDepArrhenius`

#### class `rmgpy.kinetics.MultiPDepArrhenius` *(arrhenius=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')*

A kinetic model of a phenomenological rate coefficient \( k(T, P) \) where sets of Arrhenius kinetics are stored at a variety of pressures and interpolated between on a logarithmic scale. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrhenius</td>
<td>A list of the <code>PDepArrhenius</code> kinetics at each temperature</td>
</tr>
<tr>
<td><code>Tmin</code></td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Tmax</code></td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Pmin</code></td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Pmax</code></td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>comment</code></td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

**Pmax**

The maximum pressure at which the model is valid, or `None` if not defined.

**Pmin**

The minimum pressure at which the model is valid, or `None` if not defined.

**Tmax**

The maximum temperature at which the model is valid, or `None` if not defined.

**Tmin**

The minimum temperature at which the model is valid, or `None` if not defined.

**arrhenius**

arrhenius: list

**changeRate** *(self, double factor)*

Change kinetic rate by a multiple `factor`.

**comment**

comment: str

**discrepancy** *(self, KineticsModel otherKinetics) → double*

Returns some measure of the discrepancy based on two different reaction models.

**efficiencies**

efficiencies: dict
**getCanteraEfficiencies** *(self, speciesList)*  
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction*.

**getEffectiveColliderEfficiencies** *(self, list species) → ndarray*  
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

**getEffectivePressure** *(self, double P, list species, ndarray fractions) → double*  
Return the effective pressure in Pa for a system at a given pressure P in Pa composed of the given list of species (Species or Molecule objects) with the given fractions.

**getRateCoefficient** *(self, double T, double P=0.0) → double*  
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature T in K and pressure P in Pa.

**highPlimit**  
*highPlimit: rmgpy.kinetics.model.KineticsModel*  

**isIdenticalTo** *(self, KineticsModel otherKinetics) → bool*  
Returns *True* if kinetics matches that of another kinetics model. Each duplicate reaction must be matched and equal to that in the other MultiArrhenius model in the same order. Otherwise returns *False*.

**isPressureDependent** *(self) → bool*  
Return *True* since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

**isPressureValid** *(self, double P) → bool*  
Return *True* if the pressure P in Pa is within the valid pressure range of the kinetic data, or *False* if not. If the minimum and maximum pressure are not defined, *True* is returned.

**isSimilarTo** *(self, KineticsModel otherKinetics) → bool*  
Returns *True* if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/-0.5 for log(k), in other words, within a factor of 3.

**isTemperatureValid** *(self, double T) → bool*  
Return *True* if the temperature T in K is within the valid temperature range of the kinetic data, or *False* if not. If the minimum and maximum temperature are not defined, *True* is returned.

**setCanteraKinetics** *(self, ctReaction, speciesList)*  
Sets the PLOG kinetics for multiple cantera *Reaction* objects, provided in a list. ctReaction is a list of cantera reaction objects.

**toHTML** *(self)*  
Return an HTML rendering.

---

**rmgpy.kinetics.Chebyshev**

**class rmgpy.kinetics.Chebyshev** *(coeffs=None, kunits='', highPlimit=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment='')*  
A model of a phenomenological rate coefficient $k(T, P)$ using a set of Chebyshev polynomials in temperature and pressure. The attributes are:
The Chebyshev polynomial formulation is a means of fitting a wide range of complex \( k(T, P) \) behavior. However, there is no meaningful physical interpretation of the polynomial-based fit, and one must take care to minimize the magnitude of Runge’s phenomenon. The formulation is as follows:

\[
\log k(T, P) = \sum_{t=1}^{N_T} \sum_{p=1}^{N_P} \alpha_{tp} \phi_t(\tilde{T}) \phi_p(\tilde{P})
\]

Above, \( \alpha_{tp} \) is a constant, \( \phi_n(x) \) is the Chebyshev polynomial of degree \( n \) evaluated at \( x \), and

\[
\tilde{T} = \frac{2T^{-1} - T_{\text{min}}^{-1} - T_{\text{max}}^{-1}}{T_{\text{max}}^{-1} - T_{\text{min}}^{-1}}
\]

\[
\tilde{P} = \frac{2 \log P - \log P_{\text{min}} - \log P_{\text{max}}}{\log P_{\text{max}} - \log P_{\text{min}}}
\]

are reduced temperature and reduced pressure designed to map the ranges \((T_{\text{min}}, T_{\text{max}})\) and \((P_{\text{min}}, P_{\text{max}})\) to \((-1, 1)\).

**Pmax**

The maximum pressure at which the model is valid, or None if not defined.

**Pmin**

The minimum pressure at which the model is valid, or None if not defined.

**Tmax**

The maximum temperature at which the model is valid, or None if not defined.

**Tmin**

The minimum temperature at which the model is valid, or None if not defined.

**changeRate**(self, double factor)

Changes kinetics rates by a multiple factor.

**coeffs**

The Chebyshev coefficients.

**comment**

comment: str

**degreeP**

degreeP: int

**degreeT**

degreeT: int

**discrepancy**(self, KineticsModel otherKinetics) \rightarrow double

Returns some measure of the discrepancy based on two different reaction models.
efficiencies
efficiencies: dict

fitToData(self, ndarray Tlist, ndarray Plist, ndarray K, str kunits, int degreeT, int degreeP, double Tmin, double Tmax, double Pmin, double Pmax)
Fit a Chebyshev kinetic model to a set of rate coefficients K, which is a matrix corresponding to the temperatures Tlist in K and pressures Plist in Pa. degreeT and degreeP are the degree of the polynomials in temperature and pressure, while Tmin, Tmax, Pmin, and Pmax set the edges of the valid temperature and pressure ranges in K and bar, respectively.

getCanteraEfficiencies(self, speciesList)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction

getEffectiveColliderEfficiencies(self, list species) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(self, double P, list species, ndarray fractions) → double
Return the effective pressure in Pa for a system at a given pressure P in Pa composed of the given list of species (Species or Molecule objects) with the given fractions.

getRateCoefficient(self, double T, double P=0) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature T in K and pressure P in Pa by evaluating the Chebyshev expression.

highPlimit
highPlimit: rmgpy.kinetics.model.KineticsModel

isIdenticalTo(self, KineticsModel otherKinetics) → bool
Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,

isPressureDependent(self) → bool
Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

isPressureValid(self, double P) → bool
Return True if the pressure P in Pa is within the valid pressure range of the kinetic data, or False if not. If the minimum and maximum pressure are not defined, True is returned.

isSimilarTo(self, KineticsModel otherKinetics) → bool
Returns True if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(self, double T) → bool
Return True if the temperature T in K is within the valid temperature range of the kinetic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

kunits
kunits: str

setCanteraKinetics(self, ctReaction, speciesList)
Sets the kinetics parameters for a Cantera ChebyshevReaction() object. Uses set_parameters(self,Tmin,Tmax,Pmin,Pmax,coeffs) where T's are in units of K, P's in units of Pa, and coeffs is 2D array of (nTemperature, nPressure).

toHTML(self)
Return an HTML rendering.
**rmgpy.kinetics.ThirdBody**

class **rmgpy.kinetics.ThirdBody**(*arrheniusLow=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, efficiencies=None, comment='')

A kinetic model of a phenomenological rate coefficient $k(T, P)$ using third-body kinetics. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrheniusLow</td>
<td>The Arrhenius kinetics at the low-pressure limit</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>efficiencies</td>
<td>A dict associating chemical species with associated efficiencies</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

Third-body kinetics simply introduce an inert third body to the rate expression:

$$k(T, P) = k_0(T)[M]$$

Above, $[M] \approx P/RT$ is the concentration of the bath gas. This formulation is equivalent to stating that the kinetics are always in the low-pressure limit.

- **Pmax**
  The maximum pressure at which the model is valid, or None if not defined.

- **Pmin**
  The minimum pressure at which the model is valid, or None if not defined.

- **Tmax**
  The maximum temperature at which the model is valid, or None if not defined.

- **Tmin**
  The minimum temperature at which the model is valid, or None if not defined.

- **arrheniusLow**
  arrheniusLow: rmgpy.kinetics.arrhenius.Arrhenius

- **changeRate**(self, double factor)
  Changes kinetics rate by a multiple factor.

- **comment**
  comment: str

- **discrepancy**(self, KineticsModel otherKinetics) → double
  Returns some measure of the discrepancy based on two different reaction models.

- **efficiencies**
  efficiencies: dict

- **getCanteraEfficiencies**(self, speciesList)
  Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: ThreeBodyReaction, FalloffReaction, ‘ChemicallyActivatedReaction’

- **getEffectiveColliderEfficiencies**(self, list species) → ndarray
  Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

- **getEffectivePressure**(self, double P, list species, ndarray fractions) → double
  Return the effective pressure in Pa for a system at a given pressure $P$ in Pa composed of the given list of species (Species or Molecule objects) with the given fractions.
getRateCoefficient(self, double T, double P=0.0) → double
Return the value of the rate coefficient \(k(T)\) in units of m^3, mol, and s at the specified temperature \(T\) in K and pressure \(P\) in Pa. If you wish to consider collision efficiencies, then you should first use getEffectivePressure() to compute the effective pressure, and pass that value as the pressure to this method.

highPlimit
highPlimit: rmgpy.kinetics.model.KineticsModel
isIdenticalTo(self, KineticsModel otherKinetics) → bool
Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,
isPressureDependent(self) → bool
Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.
isPressureValid(self, double P) → bool
Return True if the pressure \(P\) in Pa is within the valid pressure range of the kinetic data, or False if not. If the minimum and maximum pressure are not defined, True is returned.
isSimilarTo(self, KineticsModel otherKinetics) → bool
Returns True if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.
isTemperatureValid(self, double T) → bool
Return True if the temperature \(T\) in K is within the valid temperature range of the kinetic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.
setCanteraKinetics(self, ctReaction, speciesList)
Sets the kinetics and efficiencies for a cantera ThreeBodyReaction object
toHTML(self)
Return an HTML rendering.

rmgpy.kinetics.Lindemann
class rmgpy.kinetics.Lindemann(arrheniusHigh=None, arrheniusLow=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, efficiencies=None, comment='')
A kinetic model of a phenomenological rate coefficient \(k(T, P)\) using the Lindemann formulation. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrheniusHigh</td>
<td>The Arrhenius kinetics at the high-pressure limit</td>
</tr>
<tr>
<td>arrheniusLow</td>
<td>The Arrhenius kinetics at the low-pressure limit</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmin</td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Pmax</td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>efficiencies</td>
<td>A dict associating chemical species with associated efficiencies</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The Lindemann model qualitatively predicts the falloff of some simple pressure-dependent reaction kinetics. The formulation is as follows:

\[
k(T, P) = k_\infty(T) \left( \frac{P}{1 + P_r} \right)
\]
where

\[
P_r = \frac{k_0(T)}{k_\infty(T)} [M]
\]

\[
k_0(T) = A_0 T^{n_0} \exp\left(-\frac{E_0}{RT}\right)
\]

\[
k_\infty(T) = A_\infty T^{n_\infty} \exp\left(-\frac{E_\infty}{RT}\right)
\]

and \([M] \approx P/RT\) is the concentration of the bath gas. The Arrhenius expressions \(k_0(T)\) and \(k_\infty(T)\) represent the low-pressure and high-pressure limit kinetics, respectively.

**Pmax**
The maximum pressure at which the model is valid, or `None` if not defined.

**Pmin**
The minimum pressure at which the model is valid, or `None` if not defined.

**Tmax**
The maximum temperature at which the model is valid, or `None` if not defined.

**Tmin**
The minimum temperature at which the model is valid, or `None` if not defined.

**arrheniusHigh**

arrheniusHigh: rmgpy.kinetics.arrhenius.Arrhenius

**arrheniusLow**

arrheniusLow: rmgpy.kinetics.arrhenius.Arrhenius

**changeRate** *(self, double factor)*

Changes kinetics rate by a multiple `factor`.

**comment**

comment: str

**discrepancy** *(self, KineticsModel otherKinetics) → double*

Returns some measure of the discrepancy based on two different reaction models.

**efficiencies**

efficiencies: dict

**getCanteraEfficiencies** *(self, speciesList)*

Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction*

**getEffectiveColliderEfficiencies** *(self, list species) → ndarray*

Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

**getEffectivePressure** *(self, double P, list species, ndarray fractions) → double*

Return the effective pressure in Pa for a system at a given pressure \(P\) in Pa composed of the given list of species (Species or Molecule objects) with the given fractions.

**getRateCoefficient** *(self, double T, double P=0.0) → double*

Return the value of the rate coefficient \(k(T)\) in units of \(m^3, \text{mol, and s}\) at the specified temperature \(T\) in K and pressure \(P\) in Pa. If you wish to consider collision efficiencies, then you should first use `getEffectivePressure()` to compute the effective pressure, and pass that value as the pressure to this method.
**highPlimit**

`highPlimit: rmgpy.kinetics.model.KineticsModel`

**isIdenticalTo** *(self, KineticsModel otherKinetics) → bool*

Checks to see if kinetics matches that of other kinetics and returns `True` if coeffs, kunits, Tmin, Tmax are identical.

**isPressureDependent** *(self) → bool*

Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

**isPressureValid** *(self, double P) → bool*

Return `True` if the pressure `P` in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

**isSimilarTo** *(self, KineticsModel otherKinetics) → bool*

Returns `True` if rates of reaction at temperatures 500, 1000, 1500, 2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

**isTemperatureValid** *(self, double T) → bool*

Return `True` if the temperature `T` in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

**setCanteraKinetics** *(self, ctReaction, speciesList)*

Sets the efficiencies and kinetics for a cantera reaction.

**toHTML** *(self)*

Return an HTML rendering.

### rmgpy.kinetics.Troe

**class** `rmgpy.kinetics.Troe` *(arrheniusHigh=None, arrheniusLow=None, alpha=0.0, T3=None, T1=None, T2=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, efficiencies=None, comment='')*

A kinetic model of a phenomenological rate coefficient \( k(T, P) \) using the Troe formulation. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arrheniusHigh</code></td>
<td>The Arrhenius kinetics at the high-pressure limit</td>
</tr>
<tr>
<td><code>arrheniusLow</code></td>
<td>The Arrhenius kinetics at the low-pressure limit</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>The ( \alpha ) parameter</td>
</tr>
<tr>
<td><code>T1</code></td>
<td>The ( T_1 ) parameter</td>
</tr>
<tr>
<td><code>T2</code></td>
<td>The ( T_2 ) parameter</td>
</tr>
<tr>
<td><code>T3</code></td>
<td>The ( T_3 ) parameter</td>
</tr>
<tr>
<td><code>Tmin</code></td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Tmax</code></td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Pmin</code></td>
<td>The minimum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Pmax</code></td>
<td>The maximum pressure at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>efficiencies</code></td>
<td>A dict associating chemical species with associated efficiencies</td>
</tr>
<tr>
<td><code>comment</code></td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The Troe model attempts to make the Lindemann model quantitative by introducing a broadening factor \( F \). The formulation is as follows:

\[
k(T, P) = k_\infty(T) \left[ \frac{P_r}{1 + P_r} \right] F
\]
where
\[ P_r = \frac{k_0(T)}{k_\infty(T)}[M] \]
\[ k_0(T) = A_0 T^{n_0} \exp \left( -\frac{E_0}{RT} \right) \]
\[ k_\infty(T) = A_\infty T^{n_\infty} \exp \left( -\frac{E_\infty}{RT} \right) \]
and \([M] \approx P/RT\) is the concentration of the bath gas. The Arrhenius expressions \(k_0(T)\) and \(k_\infty(T)\) represent the low-pressure and high-pressure limit kinetics, respectively. The broadening factor \(F\) is computed via
\[ \log F = \left\{ 1 + \left[ \frac{\log P_r + c}{n - d(\log P_r + c)} \right]^2 \right\}^{-1} \log F_{\text{cent}} \]
\[ c = -0.4 - 0.67 \log F_{\text{cent}} \]
\[ n = 0.75 - 1.27 \log F_{\text{cent}} \]
\[ d = 0.14 \]
\[ F_{\text{cent}} = (1 - \alpha) \exp \left( -\frac{T}{T_3} \right) + \alpha \exp \left( -\frac{T}{T_1} \right) + \exp \left( -\frac{T_2}{T} \right) \]

\(P_{\text{max}}\)
The maximum pressure at which the model is valid, or \(\text{None}\) if not defined.

\(P_{\text{min}}\)
The minimum pressure at which the model is valid, or \(\text{None}\) if not defined.

\(T_1\)
The Troe \(T_1\) parameter.

\(T_2\)
The Troe \(T_2\) parameter.

\(T_3\)
The Troe \(T_3\) parameter.

\(T_{\text{max}}\)
The maximum temperature at which the model is valid, or \(\text{None}\) if not defined.

\(T_{\text{min}}\)
The minimum temperature at which the model is valid, or \(\text{None}\) if not defined.

\(\alpha\)
\(\alpha: \text{‘double’} \)

\(\text{arrheniusHigh}\)
arrheniusHigh: rmgpy.kinetics.arrhenius.Arrhenius

\(\text{arrheniusLow}\)
arrheniusLow: rmgpy.kinetics.arrhenius.Arrhenius

\(\text{changeRate(self, double factor)}\)
Changes kinetics rate by a multiple \(\text{factor}\).

\(\text{comment}\)
comment: str

\(\text{discrepancy(self, KineticsModel otherKinetics) \rightarrow double}\)
Returns some measure of the discrepancy based on two different reaction models.
**efficiencies**
- efficiencies: dict

**getCanteraEfficiencies** *(self, speciesList)*
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction, FalloffReaction, ChemicallyActivatedReaction*

**getEffectiveColliderEfficiencies** *(self, list species) → ndarray*
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

**getEffectivePressure** *(self, double P, list species, ndarray fractions) → double*
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

**getRateCoefficient** *(self, double T, double P=0.0) → double*
Return the value of the rate coefficient *k(T)* in units of m^3, mol, and s at the specified temperature *T* in K and pressure *P* in Pa. If you wish to consider collision efficiencies, then you should first use *getEffectivePressure()* to compute the effective pressure, and pass that value as the pressure to this method.

**highPlimit**
- highPlimit: rmgpy.kinetics.model.KineticsModel

**isIdenticalTo** *(self, KineticsModel otherKinetics) → bool*
Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin, isPressureDependent *(self) → bool*
Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

**isPressureValid** *(self, double P) → bool*
Return True if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or False if not. If the minimum and maximum pressure are not defined, True is returned.

**isSimilarTo** *(self, KineticsModel otherKinetics) → bool*
Returns True if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/- .5 for log(k), in other words, within a factor of 3.

**isTemperatureValid** *(self, double T) → bool*
Return True if the temperature *T* in K is within the valid temperature range of the kinetic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

**setCanteraKinetics** *(self, ctReaction, speciesList)*
Sets the efficiencies, kinetics, and troe falloff parameters for a cantera FalloffReaction.

**toHTML** *(self)*
Return an HTML rendering.

**rmgpy.kinetics.Wigner**

**class rmgpy.kinetics.Wigner** *(frequency)*
- A tunneling model based on the Wigner formula. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>The imaginary frequency of the transition state</td>
</tr>
</tbody>
</table>
An early formulation for incorporating the effect of tunneling is that of Wigner [1932Wigner]:

\[
\kappa(T) = 1 + \frac{1}{24} \left( \frac{h |\nu_{TS}|}{k_B T} \right)^2
\]

where \( h \) is the Planck constant, \( \nu_{TS} \) is the negative frequency, \( k_B \) is the Boltzmann constant, and \( T \) is the absolute temperature.

The Wigner formula represents the first correction term in a perturbative expansion for a parabolic barrier [1959Bell], and is therefore only accurate in the limit of a small tunneling correction. There are many cases for which the tunneling correction is very large; for these cases the Wigner model is inappropriate.

```python
calculateTunnelingFactor(self, double T) → double
Calculate and return the value of the Wigner tunneling correction for the reaction at the temperature \( T \) in K.
```

```python
calculateTunnelingFunction(self, ndarray Elist) → ndarray
Raises NotImplementedError, as the Wigner tunneling model does not have a well-defined energy-dependent tunneling function.
```

**frequency**
The negative frequency along the reaction coordinate.

---

**rmgpy.kinetics.Eckart**

```python
class rmgpy.kinetics.Eckart(frequency, E0_reac, E0_TS, E0_prod=None)
```

A tunneling model based on the Eckart model. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>The imaginary frequency of the transition state</td>
</tr>
<tr>
<td>( E_0 _reac )</td>
<td>The ground-state energy of the reactants</td>
</tr>
<tr>
<td>( E_0 _TS )</td>
<td>The ground-state energy of the transition state</td>
</tr>
<tr>
<td>( E_0 _prod )</td>
<td>The ground-state energy of the products</td>
</tr>
</tbody>
</table>

If \( E0\_prod \) is not given, it is assumed to be the same as the reactants; this results in the so-called “symmetric” Eckart model. Providing \( E0\_prod \), and thereby using the “asymmetric” Eckart model, is the recommended approach.

The Eckart tunneling model is based around a potential of the form

\[
V(x) = \frac{\hbar^2}{2m} \left[ \frac{Ax}{1 + e^x} + \frac{B e^x}{(1 + e^x)^2} \right]
\]

where \( x \) represents the reaction coordinate and \( A \) and \( B \) are parameters. The potential is symmetric if \( A = 0 \) and asymmetric if \( A \neq 0 \). If we add the constraint \(|B| > |A|\) then the potential has a maximum at

\[
x_{\text{max}} = \ln \left( \frac{B + A}{B - A} \right)
\]

\[
V(x_{\text{max}}) = \frac{\hbar^2 (A + B)^2}{4B}
\]

The one-dimensional Schrodinger equation with the Eckart potential is analytically solvable. The resulting microcanonical tunneling factor \( \kappa(E) \) is a function of the total energy of the molecular system:

\[
\kappa(E) = 1 - \frac{\cosh(2\pi a - 2\pi b) + \cosh(2\pi d)}{\cosh(2\pi a + 2\pi b) + \cosh(2\pi d)}
\]
where

\[ 2\pi a = \frac{2\sqrt{\alpha_1 \xi}}{\alpha_1^{-1/2} + \alpha_2^{-1/2}} \]

\[ 2\pi b = \frac{2\sqrt{|(\xi - 1)\alpha_1 + \alpha_2|}}{\alpha_1^{-1/2} + \alpha_2^{-1/2}} \]

\[ 2\pi d = 2\sqrt{\alpha_1 \alpha_2 - 4\pi^2/16} \]

\[ \alpha_1 = 2\pi \frac{\Delta V_1}{h|\nu_{TS}|} \]

\[ \alpha_2 = 2\pi \frac{\Delta V_2}{h|\nu_{TS}|} \]

\[ \xi = \frac{E}{\Delta V_1} \]

\( \Delta V_1 \) and \( \Delta V_2 \) are the thermal energy difference between the transition state and the reactants and products, respectively; \( \nu_{TS} \) is the negative frequency, \( h \) is the Planck constant.

Applying a Laplace transform gives the canonical tunneling factor as a function of temperature \( T \) (expressed as \( \beta \equiv 1/k_B T \)):

\[ \kappa(T) = e^{\beta \Delta V_1} \int_0^\infty \kappa(E)e^{-\beta E} dE \]

If product data is not available, then it is assumed that \( \alpha_2 \approx \alpha_1 \).

The Eckart correction requires information about the reactants as well as the transition state. For best results, information about the products should also be given. (The former is called the symmetric Eckart correction, the latter the asymmetric Eckart correction.) This extra information allows the Eckart correction to generally give a better result than the Wigner correction.

**E0_TS**
- The ground-state energy of the transition state.

**E0_prod**
- The ground-state energy of the products.

**E0_reac**
- The ground-state energy of the reactants.

**calculateTunnelingFactor**(self, double T) \to double
- Calculate and return the value of the Eckart tunneling correction for the reaction at the temperature \( T \) in K.

**calculateTunnelingFunction**(self, ndarray Elist) \to ndarray
- Calculate and return the value of the Eckart tunneling function for the reaction at the energies \( Elist \) in J/mol.

**frequency**
- The negative frequency along the reaction coordinate.
1.6 Molecular representations (rmgpy.molecule)

The `rmgpy.molecule` subpackage contains classes and functions for working with molecular representations, particularly using chemical graph theory.

1.6.1 Graphs

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>A generic vertex (node) in a graph</td>
</tr>
<tr>
<td>Edge</td>
<td>A generic edge (arc) in a graph</td>
</tr>
<tr>
<td>Graph</td>
<td>A generic graph data type</td>
</tr>
</tbody>
</table>

1.6.2 Graph isomorphism

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VF2</td>
<td>Graph isomorphism using the VF2 algorithm</td>
</tr>
</tbody>
</table>

1.6.3 Elements and atom types

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>A model of a chemical element</td>
</tr>
<tr>
<td>getElement()</td>
<td>Return the <code>Element</code> object for a given atomic number or symbol</td>
</tr>
<tr>
<td>AtomType</td>
<td>A model of an atom type: an element and local bond structure</td>
</tr>
<tr>
<td>getAtomType()</td>
<td>Return the <code>AtomType</code> object for a given atom in a molecule</td>
</tr>
</tbody>
</table>

1.6.4 Molecules

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>An atom in a molecule</td>
</tr>
<tr>
<td>Bond</td>
<td>A bond in a molecule</td>
</tr>
<tr>
<td>Molecule</td>
<td>A molecular structure represented using a chemical graph</td>
</tr>
</tbody>
</table>

1.6.5 Functional groups

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupAtom</td>
<td>An atom in a functional group</td>
</tr>
<tr>
<td>GroupBond</td>
<td>A bond in a functional group</td>
</tr>
<tr>
<td>Group</td>
<td>A functional group structure represented using a chemical graph</td>
</tr>
</tbody>
</table>

1.6.6 Adjacency lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromAdjacencyList()</td>
<td>Convert an adjacency list to a set of atoms and bonds</td>
</tr>
<tr>
<td>toAdjacencyList()</td>
<td>Convert a set of atoms and bonds to an adjacency list</td>
</tr>
</tbody>
</table>
1.6.7 Symmetry numbers

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculateAtomSymmetryNumber()</td>
<td>Calculate the atom-centered symmetry number for an atom in a molecule</td>
</tr>
<tr>
<td>calculateBondSymmetryNumber()</td>
<td>Calculate the bond-centered symmetry number for a bond in a molecule</td>
</tr>
<tr>
<td>calculateAxisSymmetryNumber()</td>
<td>Calculate the axis-centered symmetry number for a double bond axis in a molecule</td>
</tr>
<tr>
<td>calculateCyclicSymmetryNumber()</td>
<td>Calculate the ring-centered symmetry number for a ring in a molecule</td>
</tr>
<tr>
<td>calculateSymmetryNumber()</td>
<td>Calculate the total internal + external symmetry number for a molecule</td>
</tr>
</tbody>
</table>

1.6.8 Molecule and reaction drawing

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoleculeDrawer</td>
<td>Draw the skeletal formula of a molecule</td>
</tr>
<tr>
<td>ReactionDrawer</td>
<td>Draw a chemical reaction</td>
</tr>
</tbody>
</table>

1.6.9 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementError</td>
<td>Raised when an error occurs while working with chemical elements</td>
</tr>
<tr>
<td>AtomTypeError</td>
<td>Raised when an error occurs while working with atom types</td>
</tr>
<tr>
<td>InvalidAdjacencyListError</td>
<td>Raised when an invalid adjacency list is encountered</td>
</tr>
<tr>
<td>ActionError</td>
<td>Raised when an error occurs while working with a reaction recipe action</td>
</tr>
</tbody>
</table>

### rmgpy.molecule.graph.Vertex

**class rmgpy.molecule.graph.Vertex**

A base class for vertices in a graph. Contains several connectivity values useful for accelerating isomorphism searches, as proposed by Morgan (1965).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectivity1</td>
<td>int</td>
<td>The number of nearest neighbors</td>
</tr>
<tr>
<td>connectivity2</td>
<td>int</td>
<td>The sum of the neighbors' connectivity1 values</td>
</tr>
<tr>
<td>connectivity3</td>
<td>int</td>
<td>The sum of the neighbors' connectivity2 values</td>
</tr>
<tr>
<td>sortingLabel</td>
<td>int</td>
<td>An integer label used to sort the vertices</td>
</tr>
</tbody>
</table>

**copy()**

Return a copy of the vertex. The default implementation assumes that no semantic information is associated with each vertex, and therefore simply returns a new Vertex object.

**equivalent()**

Return True if two vertices self and other are semantically equivalent, or False if not. You should reimplement this function in a derived class if your vertices have semantic information.

**isSpecificCaseOf()**

Return True if self is semantically more specific than other, or False if not. You should reimplement this function in a derived class if your edges have semantic information.

**resetConnectivityValues()**

Reset the cached structure information for this vertex.

1.6. Molecular representations (rmgpy.molecule)
rmgpy.molecule.graph.Edge

class rmgpy.molecule.graph.Edge

A base class for edges in a graph. This class does not store the vertex pair that comprises the edge; that functionality would need to be included in the derived class.

copy()

Return a copy of the edge. The default implementation assumes that no semantic information is associated with each edge, and therefore simply returns a new Edge object. Note that the vertices are not copied in this implementation.

equivalent()

Return True if two edges self and other are semantically equivalent, or False if not. You should reimplement this function in a derived class if your edges have semantic information.

gotherVertex()

Given a vertex that makes up part of the edge, return the other vertex. Raise a ValueError if the given vertex is not part of the edge.

isSpecificCaseOf()

Return True if self is semantically more specific than other, or False if not. You should reimplement this function in a derived class if your edges have semantic information.

rmgpy.molecule.graph.Graph

class rmgpy.molecule.graph.Graph

A graph data type. The vertices of the graph are stored in a list vertices; this provides a consistent traversal order. The edges of the graph are stored in a dictionary of dictionaries edges. A single edge can be accessed using graph.edges[vertex1][vertex2] or the getEdge() method; in either case, an exception will be raised if the edge does not exist. All edges of a vertex can be accessed using graph.edges[vertex] or the getEdges() method.

addEdge()

Add an edge to the graph. The two vertices in the edge must already exist in the graph, or a ValueError is raised.

addVertex()

Add a vertex to the graph. The vertex is initialized with no edges.

copy()

Create a copy of the current graph. If deep is True, a deep copy is made: copies of the vertices and edges are used in the new graph. If deep is False or not specified, a shallow copy is made: the original vertices and edges are used in the new graph.

findIsomorphism()

Returns True if other is subgraph isomorphic and False otherwise, and the matching mapping. Uses the VF2 algorithm of Vento and Foggia.

findSubgraphIsomorphisms()

Returns True if other is subgraph isomorphic and False otherwise. Also returns the lists all of valid mappings.

Uses the VF2 algorithm of Vento and Foggia.

getAllCycles()

Given a starting vertex, returns a list of all the cycles containing that vertex.

getAllCyclicVertices()

Returns all vertices belonging to one or more cycles.
getAllPolycyclicVertices()
Return all vertices belonging to two or more cycles, fused or spirocyclic.

getDisparateRings()
Return a list of distinct polycyclic and monocyclic rings within the graph. There is some code duplication in this function in order to maximize speed up so as to call self.getSmallestSetOfSmallestRings() only once.
Returns: monocyclicRingsList, polycyclicRingsList

getEdge()
Returns the edge connecting vertices vertex1 and vertex2.

getEdges()
Return a list of the edges involving the specified vertex.

getMonocyclicRings()
Return a list of cycles that are monocyclic.

getPolycyclicRings()
Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

getSmallestSetOfSmallestRings()
Return a list of the smallest set of smallest rings in the graph. The algorithm implements was adapted from a description by Fan, Panaye, Doucet, and Barbu (doi: 10.1021/ci00015a002)

hasEdge()
Returns True if vertices vertex1 and vertex2 are connected by an edge, or False if not.

hasVertex()
Returns True if vertex is a vertex in the graph, or False if not.

isCyclic()
Return True if one or more cycles are present in the graph or False otherwise.

isEdgeInCycle()
Return True if the edge between vertices vertex1 and vertex2 is in one or more cycles in the graph, or False if not.

isIsomorphic()
Returns True if two graphs are isomorphic and False otherwise. Uses the VF2 algorithm of Vento and Foggia.

isMappingValid()
Check that a proposed mapping of vertices from self to other is valid by checking that the vertices and edges involved in the mapping are mutually equivalent.

isSubgraphIsomorphic()
Returns True if other is subgraph isomorphic and False otherwise. Uses the VF2 algorithm of Vento and Foggia.

isVertexInCycle()
Return True if the given vertex is contained in one or more cycles in the graph, or False if not.

merge()
Merge two graphs so as to store them in a single Graph object.

1.6. Molecular representations (rmgpy.molecule)
removeEdge()  
Remove the specified edge from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

removeVertex()  
Remove vertex and all edges associated with it from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

resetConnectivityValues()  
Reset any cached connectivity information. Call this method when you have modified the graph.

sortVertices()  
Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

split()  
Convert a single Graph object containing two or more unconnected graphs into separate graphs.

updateConnectivityValues()  
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

rmgpy.molecule.vf2.VF2

class rmgpy.molecule.vf2.VF2  
An implementation of the second version of the Vento-Foggia (VF2) algorithm for graph and subgraph isomorphism.

findIsomorphism()  
Return a list of dicts of all valid isomorphism mappings from graph graph1 to graph graph2 with the optional initial mapping initialMapping. If no valid isomorphisms are found, an empty list is returned.

findSubgraphIsomorphisms()  
Return a list of dicts of all valid subgraph isomorphism mappings from graph graph1 to subgraph graph2 with the optional initial mapping initialMapping. If no valid subgraph isomorphisms are found, an empty list is returned.

isIsomorphic()  
Return True if graph graph1 is isomorphic to graph graph2 with the optional initial mapping initialMapping, or False otherwise.

isSubgraphIsomorphic()  
Return True if graph graph1 is subgraph isomorphic to subgraph graph2 with the optional initial mapping initialMapping, or False otherwise.

rmgpy.molecule.Element

class rmgpy.molecule.Element  
A chemical element. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>int</td>
<td>The atomic number of the element</td>
</tr>
<tr>
<td>symbol</td>
<td>str</td>
<td>The symbol used for the element</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>The IUPAC name of the element</td>
</tr>
<tr>
<td>mass</td>
<td>float</td>
<td>The mass of the element in kg/mol</td>
</tr>
<tr>
<td>covRadius</td>
<td>float</td>
<td>Covalent bond radius in Angstrom</td>
</tr>
<tr>
<td>isotope</td>
<td>int</td>
<td>The isotope integer of the element</td>
</tr>
</tbody>
</table>
This class is specifically for properties that all atoms of the same element share. Ideally there is only one instance of this class for each element.

```python
rmgpy.molecule.getElement()
```

Return the `Element` object corresponding to the given parameter `value`. If an integer is provided, the value is treated as the atomic number. If a string is provided, the value is treated as the symbol. An `ElementError` is raised if no matching element is found.

### rmgpy.molecule.AtomType

**class** `rmgpy.molecule.AtomType`

A class for internal representation of atom types. Using unique objects rather than strings allows us to use fast pointer comparisons instead of slow string comparisons, as well as store extra metadata. In particular, we store metadata describing the atom type's hierarchy with regard to other atom types, and the atom types that can result when various actions involving this atom type are taken. The attributes are:

Malformed table. Text in column margin in table line 16.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label</code></td>
<td><code>str</code></td>
<td>A unique label for the atom type</td>
</tr>
<tr>
<td><code>generic</code></td>
<td><code>list</code></td>
<td>The atom types that are more generic than this one</td>
</tr>
<tr>
<td><code>specific</code></td>
<td><code>list</code></td>
<td>The atom types that are more specific than this one</td>
</tr>
<tr>
<td><code>incrementBond</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when an adjacent bond's order is increased</td>
</tr>
<tr>
<td><code>decrementBond</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when an adjacent bond's order is decreased</td>
</tr>
<tr>
<td><code>formBond</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when a new single bond is formed</td>
</tr>
<tr>
<td><code>breakBond</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when an existing single bond is broken</td>
</tr>
<tr>
<td><code>incrementRadical</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when the number of radical electrons is increased</td>
</tr>
<tr>
<td><code>decrementRadical</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when the number of radical electrons is decreased</td>
</tr>
<tr>
<td><code>incrementLonePair</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when the number of lone electron pairs is increased</td>
</tr>
<tr>
<td><code>decrementLonePair</code></td>
<td><code>list</code></td>
<td>The atom type(s) that result when the number of lone electron pairs is decreased</td>
</tr>
</tbody>
</table>

The following features are what are required in a given atom type. Any int in the list is acceptable. An empty list is a wildcard.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>single</code></td>
<td><code>list</code></td>
<td>The total number of single bonds on the atom</td>
</tr>
<tr>
<td><code>allDouble</code></td>
<td><code>list</code></td>
<td>The total number of double bonds on the atom</td>
</tr>
<tr>
<td><code>rDouble</code></td>
<td><code>list</code></td>
<td>The number of double bonds to any non-oxygen, nonsulfur element</td>
</tr>
<tr>
<td><code>oDouble</code></td>
<td><code>list</code></td>
<td>The number of double bonds to oxygen</td>
</tr>
<tr>
<td><code>sDouble</code></td>
<td><code>list</code></td>
<td>The number of double bonds to sulfur</td>
</tr>
<tr>
<td><code>triple</code></td>
<td><code>list</code></td>
<td>The total number of triple bonds on the atom</td>
</tr>
<tr>
<td><code>benzene</code></td>
<td><code>list</code></td>
<td>The total number of benzene bonds on the atom</td>
</tr>
<tr>
<td><code>lonePairs</code></td>
<td><code>list</code></td>
<td>The number of lone pairs on the atom</td>
</tr>
</tbody>
</table>

**equivalent()**

Returns `True` if two atom types `atomType1` and `atomType2` are equivalent or `False` otherwise. This function respects wildcards, e.g. `R!H` is equivalent to `C`.

**getFeatures()**

Returns a list of the features that are checked to determine atomtype

**isSpecificCaseOf()**

Returns `True` if atom type `atomType1` is a specific case of atom type `atomType2` or `False` otherwise.

```python
rmgpy.molecule.getAtomType()
```

Determine the appropriate atom type for an `Atom` object `atom` with local bond structure `bonds`, a dict containing atom-bond pairs.

---

1.6. Molecular representations (`rmgpy.molecule`) 89
The atom type of an atom describes the atom itself and (often) something about the local bond structure around that atom. This is a useful semantic tool for accelerating graph isomorphism queries, and a useful shorthand when specifying molecular substructure patterns via an RMG-style adjacency list.

We define the following basic atom types:

<table>
<thead>
<tr>
<th>Atom type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General atom types</strong></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>any atom with any local bond structure</td>
</tr>
<tr>
<td>R!H</td>
<td>any non-hydrogen atom with any local bond structure</td>
</tr>
<tr>
<td><strong>Carbon atom types</strong></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>carbon atom with any local bond structure</td>
</tr>
<tr>
<td>Cs</td>
<td>carbon atom with four single bonds</td>
</tr>
<tr>
<td>Cd</td>
<td>carbon atom with one double bond (to carbon) and two single bonds</td>
</tr>
<tr>
<td>Cdd</td>
<td>carbon atom with two double bonds</td>
</tr>
<tr>
<td>Ct</td>
<td>carbon atom with one triple bond and one single bond</td>
</tr>
<tr>
<td>Co</td>
<td>carbon atom with one double bond (to oxygen) and two single bonds</td>
</tr>
<tr>
<td>Cb</td>
<td>carbon atom with two benzene bonds and one single bond</td>
</tr>
<tr>
<td>Cbf</td>
<td>carbon atom with three benzene bonds</td>
</tr>
<tr>
<td><strong>Hydrogen atom types</strong></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>hydrogen atom with one single bond</td>
</tr>
<tr>
<td><strong>Oxygen atom types</strong></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>oxygen atom with any local bond structure</td>
</tr>
<tr>
<td>Os</td>
<td>oxygen atom with two single bonds</td>
</tr>
<tr>
<td>Od</td>
<td>oxygen atom with one double bond</td>
</tr>
<tr>
<td>0a</td>
<td>oxygen atom with no bonds</td>
</tr>
<tr>
<td><strong>Silicon atom types</strong></td>
<td></td>
</tr>
<tr>
<td>Si</td>
<td>silicon atom with any local bond structure</td>
</tr>
<tr>
<td>Sis</td>
<td>silicon atom with four single bonds</td>
</tr>
<tr>
<td>Sid</td>
<td>silicon atom with one double bond (to carbon) and two single bonds</td>
</tr>
<tr>
<td>Sidd</td>
<td>silicon atom with two double bonds</td>
</tr>
<tr>
<td>Sit</td>
<td>silicon atom with one triple bond and one single bond</td>
</tr>
<tr>
<td>Sio</td>
<td>silicon atom with one double bond (to oxygen) and two single bonds</td>
</tr>
<tr>
<td>Sib</td>
<td>silicon atom with two benzene bonds and one single bond</td>
</tr>
<tr>
<td>Sibf</td>
<td>silicon atom with three benzene bonds</td>
</tr>
<tr>
<td><strong>Sulfur atom types</strong></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>sulfur atom with any local bond structure</td>
</tr>
<tr>
<td>Ss</td>
<td>sulfur atom with two single bonds</td>
</tr>
<tr>
<td>Sd</td>
<td>sulfur atom with one double bond</td>
</tr>
<tr>
<td>Sa</td>
<td>sulfur atom with no bonds</td>
</tr>
</tbody>
</table>

**Reaction recipes**

A reaction recipe is a procedure for applying a reaction to a set of chemical species. Each reaction recipe is made up of a set of actions that, when applied sequentially, a set of chemical reactants to chemical products via that reaction’s characteristic chemical process. Each action requires a small set of parameters in order to be fully defined.

We define the following reaction recipe actions:
### Action names

<table>
<thead>
<tr>
<th>Action name</th>
<th>Arguments</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE_BOND</td>
<td><code>center1, order, center2</code></td>
<td>change the bond order of the bond between <code>center1</code> and <code>center2</code> by <code>order</code>; do not break or form bonds</td>
</tr>
<tr>
<td>FORM_BOND</td>
<td><code>center1, order, center2</code></td>
<td>form a new bond between <code>center1</code> and <code>center2</code> of type <code>order</code></td>
</tr>
<tr>
<td>BREAK_BOND</td>
<td><code>center1, order, center2</code></td>
<td>break the bond between <code>center1</code> and <code>center2</code>, which should be of type <code>order</code></td>
</tr>
<tr>
<td>GAIN_RADICAL</td>
<td><code>center, radical</code></td>
<td>increase the number of free electrons on <code>center</code> by <code>radical</code></td>
</tr>
<tr>
<td>LOSE_RADICAL</td>
<td><code>center, radical</code></td>
<td>decrease the number of free electrons on <code>center</code> by <code>radical</code></td>
</tr>
</tbody>
</table>

### `rmgpy.molecule.Atom`

**class rmgpy.molecule.Atom**

An atom. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomType</td>
<td>AtomType</td>
<td>The <code>atom type</code></td>
</tr>
<tr>
<td>element</td>
<td>Element</td>
<td>The chemical element the atom represents</td>
</tr>
<tr>
<td>radicalElectrons</td>
<td>short</td>
<td>The number of radical electrons</td>
</tr>
<tr>
<td>charge</td>
<td>short</td>
<td>The formal charge of the atom</td>
</tr>
<tr>
<td>label</td>
<td>str</td>
<td>A string label that can be used to tag individual atoms</td>
</tr>
<tr>
<td>coords</td>
<td>numpy array</td>
<td>The (x,y,z) coordinates in Angstrom</td>
</tr>
<tr>
<td>lonePairs</td>
<td>short</td>
<td>The number of lone electron pairs</td>
</tr>
</tbody>
</table>

Additionally, the `mass`, `number`, and `symbol` attributes of the atom’s element can be read (but not written) directly from the atom object, e.g. `atom.symbol` instead of `atom.element.symbol`.

**applyAction()**

Update the atom pattern as a result of applying `action`, a tuple containing the name of the reaction recipe action along with any required parameters. The available actions can be found [here](#).

**copy()**

Generate a deep copy of the current atom. Modifying the attributes of the copy will not affect the original.

**decrementLonePairs()**

Update the lone electron pairs pattern as a result of applying a LOSE_PAIR action.

**decrementRadical()**

Update the atom pattern as a result of applying a LOSE_RADICAL action, where `radical` specifies the number of radical electrons to remove.

**equivalent()**

Return `True` if `other` is indistinguishable from this atom, or `False` otherwise. If `other` is an `Atom` object, then all attributes except `label` must match exactly. If `other` is an `GroupAtom` object, then the atom must match any of the combinations in the atom pattern.

**getBondOrdersForAtom()**

This helper function is to help calculate total bond orders for an input atom.

Some special consideration for the order $B$ bond. For atoms having three $B$ bonds, the order for each is $4/3.0$, while for atoms having other than three $B$ bonds, the order for each is $3/2.0$.

**incrementLonePairs()**

Update the lone electron pairs pattern as a result of applying a GAIN_PAIR action.

**incrementRadical()**

Update the atom pattern as a result of applying a GAIN_RADICAL action, where `radical` specifies the number of radical electrons to add.

### 1.6. Molecular representations (`rmgpy.molecule`)
**isCarbon()**
Return True if the atom represents a carbon atom or False if not.

**isHydrogen()**
Return True if the atom represents a hydrogen atom or False if not.

**isNitrogen()**
Return True if the atom represents a nitrogen atom or False if not.

**isNonHydrogen()**
Return True if the atom does not represent a hydrogen atom or False if not.

**isOxygen()**
Return True if the atom represents an oxygen atom or False if not.

**isSilicon()**
Return True if the atom represents an silicon atom or False if not.

**isSpecificCaseOf()**
Return True if self is a specific case of other, or False otherwise. If other is an Atom object, then this is the same as the equivalent() method. If other is an GroupAtom object, then the atom must match or be more specific than any of the combinations in the atom pattern.

**isSulfur()**
Return True if the atom represents an sulfur atom or False if not.

**resetConnectivityValues()**
Reset the cached structure information for this vertex.

**setLonePairs()**
Set the number of lone electron pairs.

**setSpinMultiplicity()**
Set the spin multiplicity.

**updateCharge()**
Update self.charge, according to the valence, and the number and types of bonds, radicals, and lone pairs.

---

**rmgpy.molecule.Bond**

**class rmgpy.molecule.Bond**
A chemical bond. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>str</td>
<td>The bond type</td>
</tr>
</tbody>
</table>

**applyAction()**
Update the bond as a result of applying action, a tuple containing the name of the reaction recipe action along with any required parameters. The available actions can be found here.

**copy()**
Generate a deep copy of the current bond. Modifying the attributes of the copy will not affect the original.

**decrementOrder()**
Update the bond as a result of applying a CHANGE_BOND action to decrease the order by one.

**equivalent()**
Return True if other is indistinguishable from this bond, or False otherwise. other can be either a Bond or a GroupBond object.
getOtherVertex()
Given a vertex that makes up part of the edge, return the other vertex. Raise a ValueError if the given vertex is not part of the edge.

incrementOrder()
Update the bond as a result of applying a CHANGE_BOND action to increase the order by one.

isBenzene()
Return True if the bond represents a benzene bond or False if not.

isDouble()
Return True if the bond represents a double bond or False if not.

isSingle()
Return True if the bond represents a single bond or False if not.

isSpecificCaseOf()
Return True if self is a specific case of other, or False otherwise. other can be either a Bond or a GroupBond object.

isTriple()
Return True if the bond represents a triple bond or False if not.

**Bond types**

The bond type simply indicates the order of a chemical bond. We define the following bond types:

<table>
<thead>
<tr>
<th>Bond type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a single bond</td>
</tr>
<tr>
<td>D</td>
<td>a double bond</td>
</tr>
<tr>
<td>T</td>
<td>a triple bond</td>
</tr>
<tr>
<td>B</td>
<td>a benzene bond</td>
</tr>
</tbody>
</table>

**rmgpy.molecule.Molecule**

**class rmgpy.molecule.Molecule**
A representation of a molecular structure using a graph data type, extending the Graph class. The atoms and bonds attributes are aliases for the vertices and edges attributes. Other attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symmetryNumber</td>
<td>int</td>
<td>The (estimated) external + internal symmetry number of the molecule</td>
</tr>
<tr>
<td>multiplicity</td>
<td>int</td>
<td>The multiplicity of this species, multiplicity = 2*total_spin+1</td>
</tr>
</tbody>
</table>

A new molecule object can be easily instantiated by passing the SMILES or InChI string representing the molecular structure.

addAtom()
Add an atom to the graph. The atom is initialized with no bonds.

addBond()
Add a bond to the graph as an edge connecting the two atoms atom1 and atom2.

addEdge()
Add an edge to the graph. The two vertices in the edge must already exist in the graph, or a ValueError is raised.

addVertex()
Add a vertex to the graph. The vertex is initialized with no edges.
calculateCp0()  
Return the value of the heat capacity at zero temperature in J/mol*K.

calculateCpInf()  
Return the value of the heat capacity at infinite temperature in J/mol*K.

calculateSymmetryNumber()  
Return the symmetry number for the structure. The symmetry number includes both external and internal modes.

clearLabeledAtoms()  
Remove the labels from all atoms in the molecule.

connectTheDots()  
Delete all bonds, and set them again based on the Atoms’ coords. Does not detect bond type.

containsLabeledAtom()  
Return True if the molecule contains an atom with the label label and False otherwise.

copy()  
Create a copy of the current graph. If deep is True, a deep copy is made: copies of the vertices and edges are used in the new graph. If deep is False or not specified, a shallow copy is made: the original vertices and edges are used in the new graph.

countInternalRotors()  
Determine the number of internal rotors in the structure. Any single bond not in a cycle and between two atoms that also have other bonds are considered to be internal rotors.

deleteHydrogens()  
Irreversibly delete all non-labeled hydrogens without updating connectivity values. If there’s nothing but hydrogens, it does nothing. It destroys information; be careful with it.

draw()  
Generate a pictorial representation of the chemical graph using the draw module. Use path to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

findIsomorphism()  
Returns True if other is isomorphic and False otherwise, and the matching mapping. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The returned mapping also uses the atoms of self for the keys and the atoms of other for the values. The other parameter must be a Molecule object, or a TypeError is raised.

findSubgraphIsomorphisms()  
Returns True if other is subgraph isomorphic and False otherwise. Also returns the lists all of valid mappings. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The returned mappings also use the atoms of self for the keys and the atoms of other for the values. The other parameter must be a Group object, or a TypeError is raised.

fromAdjacencyList()  
Convert a string adjacency list adjlist to a molecular structure. Skips the first line (assuming it’s a label) unless withLabel is False.

fromAugmentedInChI()  
Convert an Augmented InChI string aug_inchi to a molecular structure.

fromInChI()  
Convert an InChI string inchistr to a molecular structure.
fromSMARTS()
Convert a SMARTS string `smartsstr` to a molecular structure. Uses RDKit to perform the conversion. This
Kekulizes everything, removing all aromatic atom types.

fromSMILES()
Convert a SMILES string `smilesstr` to a molecular structure.

fromXYZ()
Create an RMG molecule from a list of coordinates and a corresponding list of atomic numbers. These
are typically received from CCLib and the molecule is sent to ConnectTheDots so will only contain single
bonds.

g GetAllCycles()
Given a starting vertex, returns a list of all the cycles containing that vertex.

g GetAllCyclicVertices()
Returns all vertices belonging to one or more cycles.

g GetAllPolycyclicVertices()
Return all vertices belonging to two or more cycles, fused or spirocyclic.

g getBond()
Returns the bond connecting atoms `atom1` and `atom2`.

g getBonds()
Return a list of the bonds involving the specified `atom`.

g getDisparateRings()
Return a list of distinct polycyclic and monocyclic rings within the graph. There is some code duplication
in this function in order to maximize speed up so as to call `self.getSmallestSetOfSmallestRings()` only once.

Returns: monocyclicRingsList, polycyclicRingsList

g getEdge()
Returns the edge connecting vertices `vertex1` and `vertex2`.

g getEdges()
Return a list of the edges involving the specified `vertex`.

g getFingerprint()
Return a string containing the “fingerprint” used to accelerate graph isomorphism comparisons with other
molecules. The fingerprint is a short string containing a summary of selected information about the
molecule. Two fingerprint strings matching is a necessary (but not sufficient) condition for the associ-ated
ted molecules to be isomorphic.

g getFormula()
Return the molecular formula for the molecule.

g getLabeledAtom()
Return the atoms in the molecule that are labeled.

g getLabeledAtoms()
Return the labeled atoms as a dict with the keys being the labels and the values the atoms themselves. If
two or more atoms have the same label, the value is converted to a list of these atoms.

g getMolecularWeight()
Return the molecular weight of the molecule in kg/mol.

g getMonocyclicRings()
Return a list of cycles that are monocyclic.

g getNetCharge()
Iterate through the atoms in the structure and calculate the net charge on the overall molecule.
**getNumAtoms()**
Return the number of atoms in molecule. If element is given, ie. “H” or “C”, the number of atoms of that element is returned.

**getNumberOfRadicalElectrons()**
Return the total number of radical electrons on all atoms in the molecule. In this function, monoradical atoms count as one, biradicals count as two, etc.

**getPolycyclicRings()**
Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

**getRadicalAtoms()**
Return the atoms in the molecule that have unpaired electrons.

**getRadicalCount()**
Return the number of unpaired electrons.

**getSmallestSetOfSmallestRings()**
Return a list of the smallest set of smallest rings in the graph. The algorithm implements was adapted from a description by Fan, Panaye, Doucet, and Barbu (doi: 10.1021/ci00015a002)


**getSymmetryNumber()**
Returns the symmetry number of Molecule. First checks whether the value is stored as an attribute of Molecule. If not, it calls the calculateSymmetryNumber method.

**getURL()**
Get a URL to the molecule’s info page on the RMG website.

**hasAtom()**
Returns True if atom is an atom in the graph, or False if not.

**hasBond()**
Returns True if atoms atom1 and atom2 are connected by a bond, or False if not.

**hasEdge()**
Returns True if vertices vertex1 and vertex2 are connected by an edge, or False if not.

**hasVertex()**
Returns True if vertex is a vertex in the graph, or False if not.

**isAromatic()**
Returns True if the molecule is aromatic, or False if not. It iterates over the SSSR’s and searches for rings that consist solely of Cb atoms. Assumes that aromatic rings always consist of 6 atoms. In cases of naphthalene, where a 6 + 4 aromatic system exists, there will be at least one 6 membered aromatic ring so this algorithm will not fail for fused aromatic rings.

**isAtomInCycle()**
Return True if atom is in one or more cycles in the structure, and False if not.

**isBondInCycle()**
Return True if the bond between atoms atom1 and atom2 is in one or more cycles in the graph, or False if not.

**isCyclic()**
Return True if one or more cycles are present in the graph or False otherwise.
isEdgeInCycle()
Return True if the edge between vertices vertex1 and vertex2 is in one or more cycles in the graph, or False if not.

isIsomorphic()
Returns True if two graphs are isomorphic and False otherwise. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The other parameter must be a Molecule object, or a TypeError is raised. Also ensures multiplicities are also equal.

isLinear()
Return True if the structure is linear and False otherwise.

isMappingValid()
Check that a proposed mapping of vertices from self to other is valid by checking that the vertices and edges involved in the mapping are mutually equivalent.

isRadical()
Return True if the molecule contains at least one radical electron, or False otherwise.

isSubgraphIsomorphic()
Returns True if other is subgraph isomorphic and False otherwise. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The other parameter must be a Group object, or a TypeError is raised.

isVertexInCycle()
Return True if the given vertex is contained in one or more cycles in the graph, or False if not.

is_equal()
Method to test equality of two Molecule objects.

merge()
Merge two molecules so as to store them in a single Molecule object. The merged Molecule object is returned.

removeAtom()
Remove atom and all bonds associated with it from the graph. Does not remove atoms that no longer have any bonds as a result of this removal.

removeBond()
Remove the bond between atoms atom1 and atom2 from the graph. Does not remove atoms that no longer have any bonds as a result of this removal.

removeEdge()
Remove the specified edge from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

removeVertex()
Remove vertex and all edges associated with it from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

resetConnectivityValues()
Reset any cached connectivity information. Call this method when you have modified the graph.

saturate()
Saturate the molecule by replacing all radicals with bonds to hydrogen atoms. Changes self molecule object.

sortAtoms()
Sort the atoms in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.
sortVertices()
Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

split()
Convert a single Molecule object containing two or more unconnected molecules into separate class:Molecule objects.

toAdjacencyList()
Convert the molecular structure to a string adjacency list.

toAugmentedInChI()
Adds an extra layer to the InChI denoting the multiplicity of the molecule.
Separate layer with a forward slash character.

toAugmentedInChIKey()
Adds an extra layer to the InChIKey denoting the multiplicity of the molecule.
Simply append the multiplicity string, do not separate by a character like forward slash.

toGroup()
This method converts a list of atoms in a Molecule to a Group object.

toInChI()
Convert a molecular structure to an InChI string. Uses RDKit to perform the conversion. Perceives aromaticity.
or
Convert a molecular structure to an InChI string. Uses OpenBabel to perform the conversion.

toInChIKey()
Convert a molecular structure to an InChI Key string. Uses OpenBabel to perform the conversion.
or
Convert a molecular structure to an InChI Key string. Uses RDKit to perform the conversion.
Removes check-sum dash (-) and character so that only the 14 + 9 characters remain.

toRDKitMol()
Convert a molecular structure to a RDKit rdmol object.

toSMARTS()
Convert a molecular structure to an SMARTS string. Uses RDKit to perform the conversion. Perceives aromaticity and removes Hydrogen atoms.

toSMILES()
Convert a molecular structure to an SMILES string.
If there is a Nitrogen atom present it uses OpenBabel to perform the conversion, and the SMILES may or may not be canonical.
Otherwise, it uses RDKit to perform the conversion, so it will be canonical SMILES. While converting to an RDMolecule it will perceive aromaticity and removes Hydrogen atoms.

toSingleBonds()
Returns a copy of the current molecule, consisting of only single bonds.
This is useful for isomorphism comparison against something that was made via fromXYZ, which does not attempt to perceive bond orders.
**update()**
Update connectivity values, atom types of atoms. Update multiplicity, and sort atoms using the new connectivity values.

**updateAtomTypes()**
Iterate through the atoms in the structure, checking their atom types to ensure they are correct (i.e. accurately describe their local bond environment) and complete (i.e. are as detailed as possible).

**updateConnectivityValues()**
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

**updateLonePairs()**
Iterate through the atoms in the structure and calculate the number of lone electron pairs, assuming a neutral molecule.

**updateMultiplicity()**
Update the multiplicity of a newly formed molecule.

---

**rmgpy.molecule.GroupAtom**

class rmgpy.molecule.GroupAtom
An atom group. This class is based on the Atom class, except that it uses atom types instead of elements, and all attributes are lists rather than individual values. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomType</td>
<td>list</td>
<td>The allowed atom types (as AtomType objects)</td>
</tr>
<tr>
<td>radicalElectrons</td>
<td>list</td>
<td>The allowed numbers of radical electrons (as short integers)</td>
</tr>
<tr>
<td>charge</td>
<td>list</td>
<td>The allowed formal charges (as short integers)</td>
</tr>
<tr>
<td>label</td>
<td>str</td>
<td>A string label that can be used to tag individual atoms</td>
</tr>
<tr>
<td>lonePairs</td>
<td>list</td>
<td>The number of lone electron pairs</td>
</tr>
</tbody>
</table>

Each list represents a logical OR construct, i.e. an atom will match the group if it matches any item in the list. However, the radicalElectrons, and charge attributes are linked such that an atom must match values from the same index in each of these in order to match.

**applyAction()**
Update the atom group as a result of applying action, a tuple containing the name of the reaction recipe action along with any required parameters. The available actions can be found here.

**copy()**
Return a deep copy of the GroupAtom object. Modifying the attributes of the copy will not affect the original.

**equivalent()**
Returns True if other is equivalent to self or False if not, where other can be either an Atom or a GroupAtom object. When comparing two GroupAtom objects, this function respects wildcards, e.g. R!H is equivalent to C.

**isOxygen()**
Return True if the atom represents an oxygen atom or False if not.

**isSpecificCaseOf()**
Returns True if other is the same as self or is a more specific case of self. Returns False if some of self is not included in other or they are mutually exclusive.

**isSulfur()**
Return True if the atom represents an sulfur atom or False if not.
resetConnectivityValues()
  Reset the cached structure information for this vertex.

rmgpy.molecule.GroupBond

class rmgpy.molecule.GroupBond
  A bond group. This class is based on the Bond class, except that all attributes are lists rather than individual values. The allowed bond types are given here. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>list</td>
<td>The allowed bond orders (as character strings)</td>
</tr>
</tbody>
</table>

Each list represents a logical OR construct, i.e. a bond will match the group if it matches any item in the list.

applyAction()
  Update the bond group as a result of applying action, a tuple containing the name of the reaction recipe action along with any required parameters. The available actions can be found here.

copy()
  Return a deep copy of the GroupBond object. Modifying the attributes of the copy will not affect the original.

equivalent()
  Returns True if other is equivalent to self or False if not, where other can be either an Bond or an GroupBond object.

gotherVertex()
  Given a vertex that makes up part of the edge, return the other vertex. Raise a ValueError if the given vertex is not part of the edge.

isBenzene()
  Return True if the bond represents a benzene bond or False if not. Bonds with any wildcards will return False.

isDouble()
  Return True if the bond represents a double bond or False if not. Bonds with any wildcards will return False.

isSingle()
  Return True if the bond represents a single bond or False if not. Bonds with any wildcards will return False.

isSpecificCaseOf()
  Returns True if other is the same as self or is a more specific case of self. Returns False if some of self is not included in other or they are mutually exclusive.

isTriple()
  Return True if the bond represents a triple bond or False if not. Bonds with any wildcards will return False.

rmgpy.molecule.Group

class rmgpy.molecule.Group
  A representation of a molecular substructure group using a graph data type, extending the Graph class. The atoms and bonds attributes are aliases for the vertices and edges attributes, and store GroupAtom and GroupBond objects, respectively. Corresponding alias methods have also been provided.

addAtom()
  Add an atom to the graph. The atom is initialized with no bonds.
addBond()
Add a bond to the graph as an edge connecting the two atoms atom1 and atom2.

addEdge()
Add an edge to the graph. The two vertices in the edge must already exist in the graph, or a ValueError is raised.

addExplicitLigands()
This function Od/Sd ligand to CO or CS atomtypes if they are not already there.
Returns a ‘True’ if the group was modified otherwise returns ‘False’

addVertex()
Add a vertex to the graph. The vertex is initialized with no edges.

clearLabeledAtoms()
Remove the labels from all atoms in the molecular group.

containsLabeledAtom()
Return True if the group contains an atom with the label label and False otherwise.

copy()
Create a copy of the current graph. If deep is True, a deep copy is made: copies of the vertices and edges are used in the new graph. If deep is False or not specified, a shallow copy is made: the original vertices and edges are used in the new graph.

findIsomorphism()
Returns True if other is isomorphic and False otherwise, and the matching mapping. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The returned mapping also uses the atoms of self for the keys and the atoms of other for the values. The other parameter must be a Group object, or a TypeError is raised.

findSubgraphIsomorphisms()
Returns True if other is subgraph isomorphic and False otherwise. In other words, return True is self is more specific than other. Also returns the lists all of valid mappings. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The returned mappings also use the atoms of self for the keys and the atoms of other for the values. The other parameter must be a Group object, or a TypeError is raised.

fromAdjacencyList()
Convert a string adjacency list adjlist to a molecular structure. Skips the first line (assuming it’s a label) unless withLabel is False.

getAllCycles()
Given a starting vertex, returns a list of all the cycles containing that vertex.

getAllCyclicVertices()
Returns all vertices belonging to one or more cycles.

getAllPolycyclicVertices()
Return all vertices belonging to two or more cycles, fused or spirocyclic.

getBond()
Returns the bond connecting atoms atom1 and atom2.

getBonds()
Return a list of the bonds involving the specified atom.

getDisparateRings()
Return a list of distinct polycyclic and monocyclic rings within the graph. There is some code duplication in this function in order to maximize speed up so as to call self.getSmallestSetOfSmallestRings() only once.
Returns: monocyclicRingsList, polycyclicRingsList

**getEdge()**
Returns the edge connecting vertices vertex1 and vertex2.

**getEdges()**
Return a list of the edges involving the specified vertex.

**getLabeledAtom()**
Return the atom in the group that is labeled with the given label. Raises ValueError if no atom in the group has that label.

**getLabeledAtoms()**
Return the labeled atoms as a dict with the keys being the labels and the values the atoms themselves. If two or more atoms have the same label, the value is converted to a list of these atoms.

**getMonocyclicRings()**
Return a list of cycles that are monocyclic.

**getPolycyclicRings()**
Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

**getSmallestSetOfSmallestRings()**
Return a list of the smallest set of smallest rings in the graph. The algorithm implements was adapted from a description by Fan, Panaye, Doucet, and Barbu (doi: 10.1021/ci00015a002)


**hasAtom()**
Returns True if atom is an atom in the graph, or False if not.

**hasBond()**
Returns True if atoms atom1 and atom2 are connected by an bond, or False if not.

**hasEdge()**
Returns True if vertices vertex1 and vertex2 are connected by an edge, or False if not.

**hasVertex()**
Returns True if vertex is a vertex in the graph, or False if not.

**isCyclic()**
Return True if one or more cycles are present in the graph or False otherwise.

**isEdgeInCycle()**
Return True if the edge between vertices vertex1 and vertex2 is in one or more cycles in the graph, or False if not.

**isIdentical()**
Returns True if other is identical and False otherwise. The function isIsomorphic respects wildcards, while this function does not, make it more useful for checking groups to groups (as opposed to molecules to groups)

**isIsomorphic()**
Returns True if two graphs are isomorphic and False otherwise. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The other parameter must be a Group object, or a TypeError is raised.
isMappingValid()
Check that a proposed mapping of vertices from self to other is valid by checking that the vertices and edges involved in the mapping are mutually equivalent.

isSubgraphIsomorphic()
Returns True if other is subgraph isomorphic and False otherwise. In other words, return True if self is more specific than other. The initialMap attribute can be used to specify a required mapping from self to other (i.e. the atoms of self are the keys, while the atoms of other are the values). The other parameter must be a Group object, or a TypeError is raised.

isVertexInCycle()
Return True if the given vertex is contained in one or more cycles in the graph, or False if not.

merge()
Merge two groups so as to store them in a single Group object. The merged Group object is returned.

removeAtom()
Remove atom and all bonds associated with it from the graph. Does not remove atoms that no longer have any bonds as a result of this removal.

removeBond()
Remove the bond between atoms atom1 and atom2 from the graph. Does not remove atoms that no longer have any bonds as a result of this removal.

removeEdge()
Remove the specified edge from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

removeVertex()
Remove vertex and all edges associated with it from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

resetConnectivityValues()
Reset any cached connectivity information. Call this method when you have modified the graph.

sortAtoms()
Sort the atoms in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

sortVertices()
Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

split()
Convert a single Group object containing two or more unconnected groups into separate class:Group objects.

standardizeAtomType()
This function changes the atomTypes in a group if the atom must be a specific atomType based on its bonds and valency.
Currently only standardizes oxygen, carbon and sulfur atomTypes
We also only check when there is exactly one atomType, one bondType, one radical setting. For any group where there are wildcards or multiple attributes, we cannot apply this check.
In the case where the atomType is ambiguous based on bonds and valency, this function will not change the type.
Returns a ‘True’ if the group was modified otherwise returns ‘False’
standardizeGroup()
This function modifies groups to make them have a standard AdjList form.
Currently it makes atomtypes as specific as possible and makes CO/CS atomtypes have explicit Od/Sd ligands. Other functions can be added as necessary.
We also only check when there is exactly one atomType, one bondType, one radical setting. For any group where there are wildcards or multiple attributes, we do not apply this check.
Returns a ‘True’ if the group was modified otherwise returns ‘False’

toAdjacencyList()
Convert the molecular structure to a string adjacency list.

updateConnectivityValues()
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

updateFingerprint()
Update the molecular fingerprint used to accelerate the subgraph isomorphism checks.

Adjacency Lists

Note: 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. The new syntax, however, allows much greater flexibility, including definition of lone pairs, partial charges, wildcards, and molecule multiplicities.

Note: To quickly visualize 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

An adjacency list is the most general way of specifying a chemical molecule or molecular pattern in RMG. It is based on the adjacency list representation of the graph data type – the underlying data type for molecules and patterns in RMG – but extended to allow for specification of extra semantic information.

The first line of most adjacency lists is a unique identifier for the molecule or pattern the adjacency list represents. This is not strictly required, but is recommended in most cases. Generally the identifier should only use alphanumeric characters and the underscore, as if an identifier in many popular programming languages. However, strictly speaking any non-space ASCII character is allowed.

The subsequent lines may contain keyword-value pairs. Currently there is only one keyword, multiplicity.

For species or molecule declarations, the value after multiplicity defines the spin multiplicity of the molecule. E.g. multiplicity 1 for most ground state closed shell species, multiplicity 2 for most radical species, and multiplicity 3 for a triplet biradical. If the multiplicity line is not present then a value of (1 + number of unpaired electrons) is assumed. Thus, it can usually be omitted, but if present can be used to distinguish, for example, singlet CH2 from triplet CH2.

If defining a Functional Group, then the value must be a list, which defines the multiplicities that will be matched by the group. E.g. multiplicity [1,2,3] or, for a single value, multiplicity [1]. If a wildcard is desired, the line ’multiplicity x can be used instead to accept all multiplicities. If the multiplicity line is omitted altogether, then a wildcard is assumed.

E.g. the following two group adjlists represent identical groups.

104 Chapter 1. RMG API Reference
After the identifier line and keyword-value lines, each subsequent line describes a single atom and its local bond structure. The format of these lines is a whitespace-delimited list with tokens:

<number> [label] <element> u<unpaired> [p<pairs>] [c<charge>] <bondlist>

The first item is the number used to identify that atom. Any number may be used, though it is recommended to number the atoms sequentially starting from one. Next is an optional label used to tag that atom; this should be an asterisk followed by a unique number for the label, e.g. *1. In some cases (e.g. thermodynamics groups) there is only one labeled atom, and the label is just an asterisk with no number: *

After that is the atom’s element or atom type, indicated by its atomic symbol, followed by a sequence of tokens describing the electronic state of the atom:

- u0 number of unpaired electrons (e.g. radicals)
- p0 number of lone pairs of electrons, common on oxygen and nitrogen.
- c0 formal charge on the atom, e.g. c-1 (negatively charged), c0, c+1 (positively charged)

For Molecule definitions: The value must be a single integer (and for charge must have a + or - sign if not equal to 0) The number of unpaired electrons (i.e. radical electrons) is required, even if zero. The number of lone pairs and the formal charge are assumed to be zero if omitted.

For Group definitions: The value can be an integer or a list of integers (with signs, for charges), eg. u[0,1,2] or c[0,+1,+2,+3,+4], or may be a wildcard x which matches any valid value, eg. px is the same as p[0,1,2,3,4,...] and cx is the same as c[...,-4,-3,-2,-1,0,+1,+2,+3,+4,...]. Lists must be enclosed is square brackets, and separated by commas, without spaces. If lone pairs or formal charges are omitted from a group definition, the wildcard is assumed.

The last set of tokens is the list of bonds. To indicate a bond, place the number of the atom at the other end of the bond and the bond type within curly braces and separated by a comma, e.g. {2,D}.

Note: You must take care to make sure each bond is listed on the lines of both atoms in the bond, and that these entries have the same bond type. RMG will raise an exception if it encounters such an invalid adjacency list.

When writing a molecular substructure pattern, you may specify multiple elements, radical counts, and bond types as a comma-separated list inside square brackets. For example, to specify any carbon or oxygen atom, use the syntax [C,0]. For a single or double bond to atom 2, write {2,[S,D]}. Atom types such as R!H or Cdd may also be used as a shorthand. (Atom types like Cdd can also be used in full molecules, but this use is discouraged, as RMG can compute them automatically for full molecules.)

Below is an example adjacency list, for 1,3-hexadiene, with the weakest bond in the molecule labeled with *1 and *2. Note that hydrogen atoms can be omitted if desired, as their presence is inferred, provided that unpaired electrons, lone pairs, and charges are all correctly defined:

<table>
<thead>
<tr>
<th>multiplicity</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 C u0</td>
<td></td>
</tr>
<tr>
<td>2 C u0 {1,D}</td>
<td>{3,S}</td>
</tr>
<tr>
<td>3 C u0 {2,S}</td>
<td>{4,D}</td>
</tr>
</tbody>
</table>
The allowed element types, radicals, and bonds are listed in the following table:

<table>
<thead>
<tr>
<th>Chemical Element</th>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>Carbon atom</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>Oxygen atom</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>Hydrogen atom</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>Sulfur atom</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>Nitrogen atom</td>
</tr>
<tr>
<td></td>
<td>Si</td>
<td>Silicon atom</td>
</tr>
<tr>
<td></td>
<td>Cl</td>
<td>Chlorine atom</td>
</tr>
<tr>
<td></td>
<td>He</td>
<td>Helium atom</td>
</tr>
<tr>
<td></td>
<td>Ar</td>
<td>Argon atom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chemical Bond</th>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>Single Bond</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Double Bond</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Triple bond</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Benzene bond</td>
</tr>
</tbody>
</table>

**rmgpy.molecule.adjlist.fromAdjacencyList(adjlist, group=False, saturateH=False)**

Convert a string adjacency list `adjlist` into a set of `Atom` and `Bond` objects.

**rmgpy.molecule.adjlist.toAdjacencyList(atoms, multiplicity, label=None, group=False, removeH=False, removeLonePairs=False, oldStyle=False)**

Convert a chemical graph defined by a list of `atoms` into a string adjacency list.

**rmgpy.molecule.symmetry**

**rmgpy.molecule.symmetry.calculateAtomSymmetryNumber()**

Return the symmetry number centered at `atom` in the structure. The `atom` of interest must not be in a cycle.

**rmgpy.molecule.symmetry.calculateBondSymmetryNumber()**

Return the symmetry number centered at `bond` in the structure.

**rmgpy.molecule.symmetry.calculateAxisSymmetryNumber()**

Get the axis symmetry number correction. The “axis” refers to a series of two or more cumulated double bonds (e.g. C=C=C, etc.). Corrections for single C=C bonds are handled in getBondSymmetryNumber().

Each axis (C=C=C) has the potential to double the symmetry number. If an end has 0 or 1 groups (e.g. =C=CJJ or =C=C-R) then it cannot alter the axis symmetry and is disregarded:

<table>
<thead>
<tr>
<th>A=C=C=C..</th>
<th>A-C=C=C=C-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=1</td>
<td>s=1</td>
</tr>
</tbody>
</table>

If an end has 2 groups that are different then it breaks the symmetry and the symmetry for that axis is 1, no matter what’s at the other end:

<table>
<thead>
<tr>
<th>A\</th>
<th>A\</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=C=C=C=A</td>
<td>T=C=C=C=T</td>
</tr>
<tr>
<td>B/</td>
<td>A/</td>
</tr>
<tr>
<td>s=1</td>
<td>s=1</td>
</tr>
</tbody>
</table>

If you have one or more ends with 2 groups, and neither end breaks the symmetry, then you have an axis symmetry number of 2:
rmgpy.molecule.symmetry.calculateCyclicSymmetryNumber()

Get the symmetry number correction for cyclic regions of a molecule. For complicated fused rings the smallest set of smallest rings is used.

rmgpy.molecule.symmetry.calculateSymmetryNumber()

Return the symmetry number for the structure. The symmetry number includes both external and internal modes.

rmgpy.molecule.draw.MoleculeDrawer

class rmgpy.molecule.draw.MoleculeDrawer(options=None)

This class provides functionality for drawing the skeletal formula of molecules using the Cairo 2D graphics engine. The most common use case is simply:

```
MoleculeDrawer().draw(molecule, format='png', path='molecule.png')
```

where `molecule` is the `Molecule` object to draw. You can also pass a dict of options to the constructor to affect how the molecules are drawn.

```
draw(molecule, format, path=None)
```

Draw the given `molecule` using the given image `format` - pdf, svg, ps, or png. If `path` is given, the drawing is saved to that location on disk. The `options` dict is an optional set of key-value pairs that can be used to control the generated drawing.

This function returns the Cairo surface and context used to create the drawing, as well as a bounding box for the molecule being drawn as the tuple (`left`, `top`, `width`, `height`).

```
render(cr, offset=None)
```

Uses the Cairo graphics library to create a skeletal formula drawing of a molecule containing the list of `atoms` and dict of `bonds` to be drawn. The 2D position of each atom in `atoms` is given in the `coordinates` array. The symbols to use at each atomic position are given by the list `symbols`. You must specify the Cairo context `cr` to render to.

rmgpy.molecule.draw.ReactionDrawer

class rmgpy.molecule.draw.ReactionDrawer(options=None)

This class provides functionality for drawing chemical reactions using the skeletal formula of each reactant and product molecule via the Cairo 2D graphics engine. The most common use case is simply:

```
ReactionDrawer().draw(reaction, format='png', path='reaction.png')
```

where `reaction` is the `Reaction` object to draw. You can also pass a dict of options to the constructor to affect how the molecules are drawn.

```
draw(reaction, format, path=None)
```

Draw the given `reaction` using the given image `format` - pdf, svg, ps, or png. If `path` is given, the drawing is saved to that location on disk.

This function returns the Cairo surface and context used to create the drawing, as well as a bounding box for the molecule being drawn as the tuple (`left`, `top`, `width`, `height`).

1.6. Molecular representations (rmgpy.molecule)
1.7 Pressure dependence (rmgpy.pdep)

The `rmgpy.pdep` subpackage provides functionality for calculating the pressure-dependent rate coefficients $k(T, P)$ for unimolecular reaction networks.

A unimolecular reaction network is defined by a set of chemically reactive molecular configurations - local minima on a potential energy surface - divided into unimolecular isomers and bimolecular reactants or products. In our vernacular, reactants can associate to form an isomer, while such association is neglected for products. These configurations are connected by chemical reactions to form a network; these are referred to as path reactions. The system also consists of an excess of inert gas M, representing a thermal bath; this allows for neglecting all collisions other than those between an isomer and the bath gas.

An isomer molecule at sufficiently high internal energy can be transformed by a number of possible events:

- The isomer molecule can collide with any other molecule, resulting in an increase or decrease in energy
- The isomer molecule can isomerize to an adjacent isomer at the same energy
- The isomer molecule can dissociate into any directly connected bimolecular reactant or product channel

It is this competition between collision and reaction events that gives rise to pressure-dependent kinetics.

1.7.1 Collision events

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleExponentialDown</td>
<td>A collisional energy transfer model based on the single exponential down model</td>
</tr>
</tbody>
</table>

1.7.2 Reaction events

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculateMicrocanonicalRateCoefficient()</td>
<td>Return the microcanonical rate coefficient $k(E)$ for a reaction</td>
</tr>
<tr>
<td>applyRRKMTheory()</td>
<td>Use RRKM theory to compute $k(E)$ for a reaction</td>
</tr>
<tr>
<td>applyInverseLaplaceTransformMethod()</td>
<td>Use the inverse Laplace transform method to compute $k(E)$ for a reaction</td>
</tr>
</tbody>
</table>

1.7.3 Pressure-dependent reaction networks

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>A molecular configuration on a potential energy surface</td>
</tr>
<tr>
<td>Network</td>
<td>A collisional energy transfer model based on the single exponential down model</td>
</tr>
</tbody>
</table>

1.7.4 The master equation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateFullMEMatrix()</td>
<td>Return the full master equation matrix for a network</td>
</tr>
</tbody>
</table>
1.7.5 Master equation reduction methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msc.applyModifiedStrongCollisionMethod()</td>
<td>Reduce the master equation to phenomenological rate coefficients ( k(T, P) ) using the modified strong collision method</td>
</tr>
<tr>
<td>rs.applyReservoirStateMethod()</td>
<td>Reduce the master equation to phenomenological rate coefficients ( k(T, P) ) using the reservoir state method</td>
</tr>
<tr>
<td>cse.applyChemicallySignificantEigenvaluesMethod()</td>
<td>Reduce the master equation to phenomenological rate coefficients ( k(T, P) ) using the chemically-significant eigenvalues method</td>
</tr>
</tbody>
</table>

1.7.6 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetworkError</td>
<td>Raised when an error occurs while working with a pressure-dependent reaction network</td>
</tr>
<tr>
<td>InvalidMicrocanonicalRateError</td>
<td>Raised when the ( k(E) ) is not consistent with the high pressure-limit kinetics or thermodynamics</td>
</tr>
</tbody>
</table>

rmgpy.pdep.SingleExponentialDown

class rmgpy.pdep.SingleExponentialDown(alpha0=None, T0=None, n=0.0)

A representation of a single exponential down model of collisional energy transfer. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha0</td>
<td>The average energy transferred in a deactivating collision at the reference temperature</td>
</tr>
<tr>
<td>T0</td>
<td>The reference temperature</td>
</tr>
<tr>
<td>n</td>
<td>The temperature exponent</td>
</tr>
</tbody>
</table>

Based around the collisional energy transfer probability function

\[
P(E, E') = C(E') \exp \left( -\frac{E' - E}{\alpha} \right) \quad E < E'
\]

where the parameter \( \alpha = \langle \Delta E_d \rangle \) represents the average energy transferred in a deactivating collision. This is the most commonly-used collision model, simply because it only has one parameter to determine. The parameter \( \alpha \) is specified using the equation

\[
\alpha = \alpha_0 \left( \frac{T}{T_0} \right)^n
\]

where \( \alpha_0 \) is the value of \( \alpha \) at temperature \( T_0 \) in K. Set the exponent \( n \) to zero to obtain a temperature-independent value for \( \alpha \).

T0

The reference temperature.

alpha0

The average energy transferred in a deactivating collision at the reference temperature.

calculateCollisionEfficiency(self, double T, ndarray Elist, ndarray Jlist, ndarray densStates, double E0, double Ereac)

Calculate an efficiency factor for collisions, particularly useful for the modified strong collision method. The collisions involve the given species with density of states \( \text{densStates} \) corresponding to energies \( \text{Elist}' \) in J/mol, ground-state energy \( E0 \) in kJ/mol, and first reactive energy \( \text{Ereac} \) in kJ/mol. The collisions occur at temperature \( T \) in K and are described by the average energy transferred in a deactivating collision \( dE_{\text{down}} \) in kJ/mol. The algorithm here is implemented as described by Chang, Bozzelli, and Dean [Chang2000].

1.7. Pressure dependence (rmgpy.pdep)
**generateCollisionMatrix**

Generate and return the collision matrix \( M_{\text{coll}}/\omega = P - I \) corresponding to this collision model for a given set of energies \( E_{\text{list}} \) in J/mol, temperature \( T \) in K, and isomer density of states \( \text{densStates} \).

**getAlpha**

Return the value of the \( \alpha \) parameter - the average energy transferred in a deactivating collision - in J/mol at temperature \( T \) in K.

**Reaction events**

**Microcanonical rate coefficients**

```python
rmgpy.pdep.calculateMicrocanonicalRateCoefficient(reaction, ndarray Elist, ndarray Jlist, ndarray reacDensStates, ndarray prodDensStates=None, double T=0.0)
```

Calculate the microcanonical rate coefficient \( k(E) \) for the reaction \( \text{reaction} \) at the energies \( E_{\text{list}} \) in J/mol. \( \text{reacDensStates} \) and \( \text{prodDensStates} \) are the densities of states of the reactant and product configurations for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.
- If the above is not possible but high-pressure limit kinetics \( k_\infty(T) \) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product \( \text{prodDensStates} \) and the temperature of interest \( T \) in K can also be provided. For isomerization and association reactions \( \text{prodDensStates} \) is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.

**RRKM theory**

```python
rmgpy.pdep.applyRRKMTheory(transitionState, ndarray Elist, ndarray Jlist, ndarray densStates)
```

Calculate the microcanonical rate coefficient for a reaction using RRKM theory, where \( \text{transitionState} \) is the transition state of the reaction, \( E_{\text{list}} \) is the array of energies in J/mol at which to evaluate the microcanonical rate, and \( \text{densStates} \) is the density of states of the reactant.

RRKM (Rice-Ramsperger-Kassel-Marcus) theory is the microcanonical analogue of transition state theory. The microcanonical rate coefficient as a function of total energy \( E \) and total angular momentum quantum number \( J \) is given by

\[
k(E, J) = \frac{N^\dagger(E, J)}{\hbar \rho(E, J)}
\]

where \( N^\dagger(E, J) \) is the sum of states of the transition state and \( \rho(E, J) \) is the density of states of the reactant. If the J-rotor is treated as active, the J-dependence can be averaged in the above expression to give

\[
k(E) = \frac{N^\dagger(E)}{\hbar \rho(E)}
\]

as a function of total energy alone. This is reasonable at high temperatures, but less accurate at low temperatures.

Use of RRKM theory requires detailed information about the statistical mechanics of the reactant and transition state. However, it is generally more accurate than the inverse Laplace transform method.
Inverse Laplace transform method

**rmgpy.pdep.applyInverseLaplaceTransformMethod**

```
transitionState, Arrhenius kinetics, ndarray Elist, ndarray Jlist, ndarray densStates, double T=0.0
```

Calculate the microcanonical rate coefficient for a reaction using the inverse Laplace transform method, where *kinetics* is the high pressure limit rate coefficient, *E0* is the ground-state energy of the transition state, *Elist* is the array of energies in kJ/mol at which to evaluate the microcanonical rate, and *densStates* is the density of states of the reactant. The temperature *T* in K is not required, and is only used when the temperature exponent of the Arrhenius expression is negative (for which the inverse transform is undefined).

The inverse Laplace transform method exploits the following relationship to determine the microcanonical rate coefficient:

\[
\mathcal{L} [k(E) \rho(E)] = \int_0^{\infty} k(E) \rho(E) e^{-E/k_B T} dE = k_\infty(T) Q(T)
\]

Given a high-pressure limit rate coefficient \( k_\infty(T) \) represented as an Arrhenius expression with positive \( n \) and \( E_a \), the microcanonical rate coefficient \( k(E) \) can be determined via an inverse Laplace transform. For \( n = 0 \) the transform can be defined analytically:

\[
k(E) = A \frac{\rho(E - E_a)}{\rho(E)} \quad (n = 0)
\]

For \( n > 0 \) the transform is defined numerically. For \( n < 0 \) or \( E_a < 0 \) the transform is not defined; in this case we approximate by simply lumping the \( T^n \) or \( e^{-E_a/RT} \) terms into the preexponential factor, and use a different \( k(E) \) at each temperature.

The ILT method does not require detailed transition state information, but only the high-pressure limit kinetics. However, it assumes that (1) \( k_\infty(T) \) is valid over the temperature range from zero to infinity and (2) the activation energy \( E_a \) is physically identical to the reaction barrier \( E^\ddagger_0 - E_0 \).

**rmgpy.pdep.Configuration**

**class rmgpy.pdep.Configuration**

A representation of a molecular configuration on a potential energy surface.

**E0**

The ground-state energy of the configuration in J/mol.

**calculateCollisionFrequency()**

Return the value of the collision frequency in Hz at the given temperature \( T \) in K and pressure \( P \) in Pa. If a dictionary *bathGas* of bath gas species and corresponding mole fractions is given, the collision parameters of the bas gas species will be averaged with those of the species before computing the collision frequency.

Only the Lennard-Jones collision model is currently supported.

**calculateDensityOfStates()**

Calculate the density (and sum) of states for the configuration at the given energies above the ground state *Elist* in J/mol. The *activeJRotor* and *activeKRotor* flags control whether the J-rotor and/or K-rotor are treated as active (and therefore included in the density and sum of states). The computed density and sum of states arrays are stored on the object for future use.

**cleanup()**

Delete intermediate arrays used in computing \( k(T,P) \) values.

**generateCollisionMatrix()**

Return the collisional energy transfer probabilities matrix for the configuration at the given temperature \( T \) in K using the given energies *Elist* in kJ/mol and total angular momentum quantum numbers *Jlist*. The density of states of the configuration *densStates* in mol/kJ is also required.
getEnthalpy()  
Return the enthalpy in kJ/mol at the specified temperature $T$ in K.

getEntropy()  
Return the entropy in J/mol*K at the specified temperature $T$ in K.

getFreeEnergy()  
Return the Gibbs free energy in kJ/mol at the specified temperature $T$ in K.

getHeatCapacity()  
Return the constant-pressure heat capacity in J/mol*K at the specified temperature $T$ in K.

hasStatMech()  
Return True if all species in the configuration have statistical mechanics parameters, or False otherwise.

hasThermo()  
Return True if all species in the configuration have thermodynamics parameters, or False otherwise.

isBimolecular()  
Return True if the configuration represents a bimolecular reactant or product channel, or False otherwise.

isTransitionState()  
Return True if the configuration represents a transition state, or False otherwise.

isUnimolecular()  
Return True if the configuration represents a unimolecular isomer, or False otherwise.

mapDensityOfStates()  
Return a mapping of the density of states for the configuration to the given energies $E_{list}$ in J/mol and, if the J-rotor is not active, the total angular momentum quantum numbers $J_{list}$.

mapSumOfStates()  
Return a mapping of the density of states for the configuration to the given energies $E_{list}$ in J/mol and, if the J-rotor is not active, the total angular momentum quantum numbers $J_{list}$.

rmgpy.pdep.Network

class rmgpy.pdep.Network(label='', isomers=None, reactants=None, products=None, pathReactions=None, bathGas=None)

A representation of a unimolecular reaction network. The attributes are:
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isomers</td>
<td>A list of the unimolecular isomers in the network</td>
</tr>
<tr>
<td>reactants</td>
<td>A list of the bimolecular reactant channels in the network</td>
</tr>
<tr>
<td>products</td>
<td>A list of the bimolecular product channels in the network</td>
</tr>
<tr>
<td>pathReactions</td>
<td>A list of “path” reaction objects that connect adjacent isomers (the high-pressure-limit)</td>
</tr>
<tr>
<td>bathGas</td>
<td>A dictionary of the bath gas species (keys) and their mole fractions (values)</td>
</tr>
<tr>
<td>netReactions</td>
<td>A list of “net” reaction objects that connect any pair of isomers</td>
</tr>
<tr>
<td>T</td>
<td>The current temperature in K</td>
</tr>
<tr>
<td>P</td>
<td>The current pressure in bar</td>
</tr>
<tr>
<td>Elist</td>
<td>The current array of energy grains in kJ/mol</td>
</tr>
<tr>
<td>Jlist</td>
<td>The current array of total angular momentum quantum numbers</td>
</tr>
<tr>
<td>Nisom</td>
<td>The number of unimolecular isomers in the network</td>
</tr>
<tr>
<td>Nreac</td>
<td>The number of bimolecular reactant channels in the network</td>
</tr>
<tr>
<td>Nprod</td>
<td>The number of bimolecular product channels in the network</td>
</tr>
<tr>
<td>Ngrains</td>
<td>The number of energy grains</td>
</tr>
<tr>
<td>NJ</td>
<td>The number of angular momentum grains</td>
</tr>
<tr>
<td>activeKRotor</td>
<td>True if the K-rotor is treated as active, False if treated as adiabatic</td>
</tr>
<tr>
<td>activeJRotor</td>
<td>True if the J-rotor is treated as active, False if treated as adiabatic</td>
</tr>
<tr>
<td>rmgmode</td>
<td>True if in RMG mode, False otherwise</td>
</tr>
</tbody>
</table>

**applyChemicallySignificantEigenvaluesMethod** *(lumpingOrder=None)*

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the chemically-significant eigenvalues method. If a `lumpingOrder` is provided, the algorithm will attempt to lump the configurations (given by index) in the order provided, and return a reduced set of $k(T, P)$ values.

**applyModifiedStrongCollisionMethod** *(efficiencyModel='default')*

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the modified strong collision method.

**applyReservoirStateMethod** *

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the reservoir state method.

**calculateCollisionModel** *

Calculate the matrix of first-order rate coefficients for collisional population transfer between grains for each isomer, including the corresponding collision frequencies.

**calculateDensitiesOfStates** *

Calculate the densities of states of each configuration that has states data. The densities of states are computed such that they can be applied to each temperature in the range of interest by interpolation.

**calculateEquilibriumRatios** *

Return an array containing the fraction of each isomer and reactant channel present at equilibrium, as determined from the Gibbs free energy and using the concentration equilibrium constant $K_c$. These values are ratios, and the absolute magnitude is not guaranteed; however, the implementation scales the elements of the array so that they sum to unity.

**calculateMicrocanonicalRates** *

Calculate and return arrays containing the microcanonical rate coefficients $k(E)$ for the isomerization, dissociation, and association path reactions in the network.

**getAllSpecies** *

Return a list of all unique species in the network, including all isomers, reactant and product channels, and bath gas species.

**initialize** *(Tmin, Tmax, Pmin, Pmax, maximumGrainSize=0.0, minimumGrainCount=0, activeJRotor=True, activeKRotor=True, rmgmode=False)*

Initialize a pressure dependence calculation by computing several quantities that are independent of the
conditions. You must specify the temperature and pressure ranges of interesting using $T_{\text{min}}$ and $T_{\text{max}}$ in K and $P_{\text{min}}$ and $P_{\text{max}}$ in Pa. You must also specify the maximum energy grain size $\text{grainSize}$ in J/mol and/or the minimum number of grains $\text{grainCount}$.

**invalidate()**
Mark the network as in need of a new calculation to determine the pressure-dependent rate coefficients.

**mapDensitiesOfStates()**
Map the overall densities of states to the current energy grains. Semi-logarithmic interpolation will be used if the grain sizes of $E_{\text{list}0}$ and $E_{\text{list}}$ do not match; this should not be a significant source of error as long as the grain sizes are sufficiently small.

**printSummary(level=20)**
Print a formatted list of information about the current network. Each molecular configuration - unimolecular isomers, bimolecular reactant channels, and bimolecular product channels - is given along with its energy on the potential energy surface. The path reactions connecting adjacent molecular configurations are also given, along with their energies on the potential energy surface. The `level` parameter controls the level of logging to which the summary is written, and is DEBUG by default.

**selectEnergyGrains($T$, $\text{grainSize}=0.0$, $\text{grainCount}=0$)**
Select a suitable list of energies to use for subsequent calculations. This is done by finding the minimum and maximum energies on the potential energy surface, then adding a multiple of $k_B T$ onto the maximum energy.

You must specify either the desired grain spacing $\text{grainSize}$ in J/mol or the desired number of grains $N_{\text{grains}}$, as well as a temperature $T$ in K to use for the equilibrium calculation. You can specify both $\text{grainSize}$ and $\text{grainCount}$, in which case the one that gives the more accurate result will be used (i.e. they represent a maximum grain size and a minimum number of grains). An array containing the energy grains in J/mol is returned.

**setConditions($T$, $P$, $ymB=None$)**
Set the current network conditions to the temperature $T$ in K and pressure $P$ in Pa. All of the internal variables are updated accordingly if they are out of date. For example, those variables that depend only on temperature will not be recomputed if the temperature is the same.

**solveFullME(tlist, $x0$)**
Directly solve the full master equation using a stiff ODE solver. Pass the reaction network to solve, the temperature $T$ in K and pressure $P$ in Pa to solve at, the energies $E_{\text{list}}$ in J/mol to use, the output time points $t_{\text{list}}$ in s, the initial total populations $x0$, the full master equation matrix $M$, the accounting matrix $indices$ relating isomer and energy grain indices to indices of the master equation matrix, and the densities of states $densStates$ in mol/J of each isomer. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

**solveReducedME(tlist, $x0$)**
Directly solve the reduced master equation using a stiff ODE solver. Pass the output time points $t_{\text{list}}$ in s and the initial total populations $x0$. Be sure to run one of the methods for generating $k(T, P)$ values before calling this method. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

The master equation

rmgpy.pdep.me.generateFullMEMatrix()
Generate the full master equation matrix for the network.

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 \( y_n^A(t) \) and \( y_n^B(t) \) 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 \( N_{\text{isom}}, N_{\text{reac}}, \) and \( N_{\text{prod}} \) as the numbers of isomers, reactant channels, and product channels, respectively, in the system.

The governing equation for the population distributions \( p_i(E, J, t) \) of each isomer \( i \) and the reactant concentrations \( y_n^A(t) \) and \( y_n^B(t) \) combines the collision and reaction models to give a linear integro-differential equation:

\[
\frac{d}{dt} p_i(E, J, t) = \omega_i(T, P) \sum_{J'} \int_0^\infty P_i(E, J, E', J') p_i(E', J', t) \, dE' - \omega_i(T, P) p_i(E, J, t) + \sum_{j \neq i} k_{ij}(E, J) p_j(E, J, t) - \sum_{j \neq i} k_{ji}(E, J) p_i(E, J, t) + \sum_{n=1}^{N_{\text{isom}}} y_n^A(t) y_n^B(t) f_{in}(E, J) b_n(E, J, t) - \sum_{n=1}^{N_{\text{isom}}+N_{\text{prod}}} g_{ni}(E, J) p_i(E, J, t)
\]

\[
\frac{d}{dt} y_n^A(t) = \frac{d}{dt} y_n^B(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^\infty g_{ni}(E, J) p_i(E, J, t) \, dE - \sum_{i=1}^{N_{\text{isom}}} y_n^A(t) y_n^B(t) \int_0^\infty f_{in}(E, J) b_n(E, J, t) \, dE
\]

A summary of the variables is given below:
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:

\[
\frac{d}{dt} p_i(E, t) = \omega_i(T, P) \int_0^\infty P_i(E, E') p_i(E', t) \, dE' - \omega_i(T, P) p_i(E, t) + \sum_{j \neq i} k_{ij}(E) p_j(E, t) - \sum_{j \neq i} k_{ji}(E) p_i(E, t) + \sum_{n=1}^{N_{\text{reac}}+N_{\text{prod}}} y_{nA}(t) y_{nB}(t) f_{in}(E) b_n(E, t) - \sum_{n=1}^{N_{\text{isom}}} g_{ni}(E) p_i(E, t)
\]

\[
\frac{d}{dt} y_{nA}(t) = \frac{d}{dt} y_{nB}(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^\infty g_{ni}(E) p_i(E, t) \, dE - \sum_{i=1}^{N_{\text{isom}}} y_{nA}(t) y_{nB}(t) \int_0^\infty f_{in}(E) b_n(E, t) \, dE
\]

The equations as given are nonlinear, both due to the presence of the bimolecular reactants and because both \( \omega_i \) 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 \( y_{nA}(t) \ll y_{nB}(t) \), and all \( y(t) \) are negligible compared to the bath gas \( M \). From these assumptions the changes in \( \omega_i, P_i(E, E') \), and all \( y_{nB} \) can be neglected, which yields a linear equation system.

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 \textit{grains}. This converts the linear integro-differential equation into a system of first-order ordinary differential equations:

\[
\begin{bmatrix}
\frac{d}{dt} p_1 \\
\frac{d}{dt} p_2 \\
\frac{d}{dt} y_{1A} \\
\frac{d}{dt} y_{2A}
\end{bmatrix} =
\begin{bmatrix}
M_1 & K_{12} & \cdots & F_{11} b_{11} & F_{12} b_{21} & \cdots \\
K_{21} & M_2 & \cdots & F_{21} b_{11} & F_{22} b_{21} & \cdots \\
(g_{11})^T & (g_{12})^T & \cdots & h_1 & 0 & \cdots \\
(g_{21})^T & (g_{22})^T & \cdots & 0 & h_2 & \cdots
\end{bmatrix}
\begin{bmatrix}
p_1 \\
p_2 \\
y_{1A} \\
y_{2A}
\end{bmatrix}
\]

The diagonal matrices \( K_{ij} \) and \( F_{in} \) and the vector \( g_{ni} \) contain the microcanonical rate coefficients for isomerization,
association, and dissociation, respectively:

$$(K_{ij})_{rs} = \begin{cases} \frac{1}{2} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} k_{ij}(E) \, dE & r = s \\ 0 & r \neq s \end{cases}$$

$$(F_{in})_{rs} = \begin{cases} \frac{1}{2} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} f_{in}(E) \, dE & r = s \\ 0 & r \neq s \end{cases}$$

$$(g_{ni})_r = \frac{1}{2 \Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} g_{ni}(E) \, dE$$

The matrices $M_i$ represent the collisional transfer probabilities minus the rates of reactive loss to other isomers and to reactants and products:

$$(M_i)_{rs} = \begin{cases} \omega_i \left[ P_i(E_r, E_s) - 1 \right] - \sum_{j \neq i} N_{isom} k_{ij}(E_r) - \sum_{n=1}^{N_{react}+N_{prod}} g_{ni}(E_r) & r = s \\ \omega_i P_i(E_r, E_s) & r \neq s \end{cases}$$

The scalars $h_n$ are simply the total rate coefficient for loss of reactant channel $n$ due to chemical reactions:

$$h_n = -\sum_{i=1}^{N_{isom}} \sum_{r=1}^{N_{grains}} y_{ni} b_{in}(E_r) b_n(E_r)$$

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].

### Methods for estimating $k(T,P)$ values

The objective of each of the methods described in this section is to reduce the master equation into a small number of phenomenological rate coefficients $k(T,P)$. All of the methods share a common formalism in that they seek to express the population distribution vector $p_i$ for each unimolecular isomer $i$ as a linear combination of the total populations of all unimolecular isomers and bimolecular reactant channels.

#### The modified strong collision method

**rmgpy.pdep.msc.** applyModifiedStrongCollisionMethod()

The modified strong collision method utilizes a greatly simplified collision model that allows for a decoupling of the energy grains. In the simplified collision model, collisional stabilization of a reactive isomer is treated as a single-step process, ignoring the effects of collisional energy redistribution within the reactive energy space. An attempt to correct for the effect of collisional energy redistribution is made by modifying the collision frequency $\omega_i(T,P)$ with a collision efficiency $\beta_i(T)$ estimated from the low-pressure limit fall-off of a single isomer.

By approximating the reactive populations as existing in pseudo-steady state, the master equation is converted to a matrix equation at each energy. Solving these small matrix equations gives the pseudo-steady state populations of each isomer as a function of the total population of each isomer and reactant channel, which are then applied to determine the $k(T,P)$ values.

In practice, the modified strong collision method is the fastest and most robust of the methods, and is reasonably accurate over a wide range of temperatures and pressures.

#### The reservoir state method

**rmgpy.pdep.rs.** applyReservoirStateMethod()

In the reservoir state method, the population distribution of each isomer is partitioned into the low-energy grains...
(called the reservoir) and the high-energy grains (called the active space). The partition generally occurs at or near the lowest transition state energy for each isomer. The reservoir population is assumed to be thermalized, while the active-space population is assumed to be in pseudo-steady state. Applying these approximations converts the master equation into a single large matrix equation. Solving this matrix equation gives the pseudo-steady state populations of each isomer as a function of the total population of each isomer and reactant channel, which are then applied to determine the \( k(T, P) \) values.

The reservoir state method is only slightly more expensive than the modified strong collision method. At low temperatures the approximations used are very good, and the resulting \( k(T, P) \) values are more accurate than the modified strong collision values. However, at high temperatures the thermalized reservoir approximation breaks down, resulting in very inaccurate \( k(T, P) \) values. Thus, the reservoir state method is not robustly applicable over a wide range of temperatures and pressures.

The chemically-significant eigenvalues method

\texttt{rmgpy.pdep.cse.applyChemicallySignificantEigenvaluesMethod()}

In the chemically-significant eigenvalues method, the master equation matrix is diagonized to determine its eigenmodes. Only the slowest of these modes are relevant to the chemistry; the rest involve internal energy relaxation due to collisions. Keeping only these “chemically-significant” eigenmodes allows for reduction to \( k(T, P) \) values.

The chemically-significant eigenvalues method is the most accurate method, and is considered to be exact as long as the chemically-significant eigenmodes are separable and distinct from the internal energy relaxation eigenmodes. However, this is often only the case near the high-pressure limit, even for networks of only modest size. The chemically-significant eigenvalues method is also substantially more expensive to apply than the other methods.

1.8 QMTP (\texttt{rmgpy.qm})

The \texttt{rmgpy.qm} subpackage contains classes and functions for working with molecular geometries, and interfacing with quantum chemistry software.

1.8.1 Main

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{QMSettings}</td>
<td>A class to store settings related to quantum mechanics calculations</td>
</tr>
<tr>
<td>\texttt{QMCalculator}</td>
<td>An object to store settings and previous calculations</td>
</tr>
</tbody>
</table>

1.8.2 Molecule

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Geometry}</td>
<td>A geometry, used for quantum calculations</td>
</tr>
<tr>
<td>\texttt{QMMolecule}</td>
<td>A base class for QM Molecule calculations</td>
</tr>
</tbody>
</table>

1.8.3 QM Data

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{QMData}</td>
<td>General class for data extracted from a QM calculation</td>
</tr>
</tbody>
</table>
1.8.4 QM Verifier

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMVerifier</td>
<td>Verifies whether a QM job was successfully completed</td>
</tr>
</tbody>
</table>

1.8.5 Symmetry

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointGroup</td>
<td>A symmetry Point Group</td>
</tr>
<tr>
<td>PointGroupCalculator</td>
<td>Wrapper type to determine molecular symmetry point groups based on 3D coordinates</td>
</tr>
<tr>
<td>SymmetryJob</td>
<td>Determine the point group using the SYMMETRY program</td>
</tr>
</tbody>
</table>

1.8.6 Gaussian

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>A base class for all QM calculations that use Gaussian</td>
</tr>
<tr>
<td>GaussianMol</td>
<td>A base Class for calculations of molecules using Gaussian.</td>
</tr>
<tr>
<td>GaussianMolPM3</td>
<td>A base Class for calculations of molecules using Gaussian at PM3.</td>
</tr>
<tr>
<td>GaussianMolPM6</td>
<td>A base Class for calculations of molecules using Gaussian at PM6.</td>
</tr>
</tbody>
</table>

1.8.7 Mopac

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mopac</td>
<td>A base class for all QM calculations that use Mopac</td>
</tr>
<tr>
<td>MopacMol</td>
<td>A base Class for calculations of molecules using Mopac.</td>
</tr>
<tr>
<td>MopacMolPM3</td>
<td>A base Class for calculations of molecules using Mopac at PM3.</td>
</tr>
<tr>
<td>MopacMolPM6</td>
<td>A base Class for calculations of molecules using Mopac at PM6.</td>
</tr>
<tr>
<td>MopacMolPM7</td>
<td>A base Class for calculations of molecules using Mopac at PM7.</td>
</tr>
</tbody>
</table>

rmgpy.qm.main

class rmgpy.qm.main.QMSettings(software=None, method='pm3', fileStore=None, scratchDirectory=None, onlyCyclics=True, maxRadicalNumber=0)
A minimal class to store settings related to quantum mechanics calculations.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>software</td>
<td>str</td>
<td>Quantum chemical package name in common letters</td>
</tr>
<tr>
<td>method</td>
<td>str</td>
<td>Semi-empirical method</td>
</tr>
<tr>
<td>fileStore</td>
<td>str</td>
<td>The path to the QMfiles directory</td>
</tr>
<tr>
<td>scratchDirectory</td>
<td>str</td>
<td>The path to the scratch directory</td>
</tr>
<tr>
<td>onlyCycles</td>
<td>bool</td>
<td>True if to run QM only on ringed species</td>
</tr>
<tr>
<td>maxRadicalNumber</td>
<td>int</td>
<td>Radicals larger than this are saturated before applying HBI</td>
</tr>
</tbody>
</table>

class rmgpy.qm.main.QMCalculator(software=None, method='pm3', fileStore=None, scratchDirectory=None, onlyCyclics=True, maxRadicalNumber=0)
A Quantum Mechanics calculator object, to store settings.
The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>settings</td>
<td>QMSettings</td>
<td>Settings for QM calculations</td>
</tr>
<tr>
<td>database</td>
<td>ThermoLibrary</td>
<td>Database containing QM calculations</td>
</tr>
</tbody>
</table>

**checkPaths()**
Check the paths in the settings are OK. Make folders as necessary.

**checkReady()**
Check that it’s ready to run calculations.

**getThermoData(molecule)**
Generate thermo data for the given Molecule via a quantum mechanics calculation.

Ignores the settings onlyCyclics and maxRadicalNumber and does the calculation anyway if asked. (I.e. the code that chooses whether to call this method should consider those settings).

**initialize()**
Do any startup tasks.

**setDefaultOutputDirectory(outputDirectory)**
IF the fileStore or scratchDirectory are not already set, put them in here.

---

**rmgpy.qm.molecule**

**class rmgpy.qm.molecule.Geometry(settings, uniqueID, molecule, uniqueIDlong=None)**
A geometry, used for quantum calculations.

Created from a molecule. Geometry estimated by RDKit.

The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>settings</td>
<td>QMSettings</td>
<td>Settings for QM calculations</td>
</tr>
<tr>
<td>uniqueID</td>
<td>str</td>
<td>A short ID such as an augmented InChI Key</td>
</tr>
<tr>
<td>molecule</td>
<td>Molecule</td>
<td>RMG Molecule object</td>
</tr>
<tr>
<td>uniqueIDlong</td>
<td>str</td>
<td>A long, truly unique ID such as an augmented InChI</td>
</tr>
</tbody>
</table>

**generateRDKitGeometries()**
Use RDKit to guess geometry.

Save mol files of both crude and refined. Saves coordinates on atoms.

**getCrudeMolFilePath()**
Returns the path of the crude mol file.

**getFilePath(extension, scratch=True)**
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

**getRefinedMolFilePath()**
Returns the path the the refined mol file.

**rd_build()**
Import rmg molecule and create rdkit molecule with the same atom labeling.

**rd_embed(rdmol, numConfAttempts)**
Embed the RDKit molecule and create the crude molecule file.
saveCoordinatesFromQMData(qmdata)
    Save geometry info from QMData (eg CCLibData)

uniqueID = None
    A short unique ID such as an augmented InChI Key.

uniqueIDlong = None
    Long, truly unique, ID, such as the augmented InChI.

class rmgpy.qm.molecule.QMMolecule(molecule, settings)
    A base class for QM Molecule calculations.
    Specific programs and methods should inherit from this and define some extra attributes and methods:

    *outputFileExtension
    *inputFileExtension
    *generateQMData() ...and whatever else is needed to make this method work.

    The attributes are:

    | Attribute    | Type          | Description                                      |
    |--------------|---------------|--------------------------------------------------|
    | molecule     | Molecule      | RMG Molecule object                             |
    | settings     | QMSettings    | Settings for QM calculations                     |
    | uniqueID     | str           | A short ID such as an augmented InChI Key        |
    | uniqueIDlong | str           | A long, truly unique ID such as an augmented InChI |

calculateChiralityCorrection()
    Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
    Calculate the thermodynamic properties.
    Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()
    Check the paths in the settings are OK. Make folders as necessary.

checkReady()
    Check that it’s ready to run calculations.

createGeometry()
    Creates self.geometry with RDKit geometries

determinePointGroup()
    Determine point group using the SYMMETRY Program
    Stores the resulting PointGroup in self.pointGroup

generateQMData()
    Calculate the QM data somehow and return a CCLibData object, or None if it fails.

generateThermoData()
    Generate Thermo Data via a QM calc.
    Returns None if it fails.

getFilePath(extension, scratch=True)
    Returns the path to the file with the given extension.
    The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.
getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(\text{\texttt{\textit{attempt}}})
Get the path to the MOL file of the geometry to use for calculation \text{\texttt{\textit{attempt}}}.

If \text{\texttt{\textit{attempt}}} \leq \text{\texttt{\textit{self.scriptAttempts}}} then we use the refined coordinates, then we start to use the crude coordinates.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFilePath
Get the input file name.

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
 Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

\texttt{\texttt{\textit{rmgpy.qm.qmd}}}  

\texttt{\texttt{\texttt{\textit{class}}} \texttt{\texttt{\textit{rmgpy.qm.qmd}}}} \texttt{\texttt{\texttt{.}}} \texttt{\texttt{\texttt{QMD}}} \texttt{\texttt{\texttt{(}}} \texttt{\texttt{groundStateDegeneracy=-1, numberOfAtoms=None, stericEnergy=None,}} \texttt{\texttt{molecularMass=None, energy=0, atomicNumbers=None, rotationalConstants=None, atomCoords=None, frequencies=None, source=None)}} \texttt{\texttt{)}}

General class for data extracted from a QM calculation

\texttt{\texttt{\textit{groundStateDegeneracy}}} = \texttt{\texttt{None}}
Electronic ground state degeneracy in RMG taken as number of radicals +1

\texttt{\texttt{\textit{numberOfAtoms}}} = \texttt{\texttt{None}}
Number of atoms.

\texttt{\texttt{\textit{rmgpy.qm.qmverifier}}}  

\texttt{\texttt{\textit{class}}} \texttt{\texttt{\textit{rmgpy.qm.qmverifier}}}.\texttt{\texttt{QMVerifier}} \texttt{\texttt{(}}} \texttt{\texttt{molfile)}}
Verifies whether a QM job (externalized) was sucessfully completed by

\begin{itemize}
\item searching for specific keywords in the output files,
\item located in a specific directory (e.g. “QMFiles”)
\end{itemize}
checkForInChiKeyCollision(logFileInChI)
This method is designed in the case a MOPAC output file was found but the InChI found in the file did not
correspond to the InChI of the given molecule.

This could mean two things: 1) that the InChI Key hash does not correspond to the InChI it is hashed from.
This is the rarest case of them all 2) the complete InChI did not fit onto just one line in the MOPAC output
file. Therefore it was continued on the second line and only a part of the InChI was actually taken as the
‘whole’ InChI.

This method reads in the MOPAC input file and compares the found InChI in there to the InChI of the
given molecule.

successfulJobExists()
checks whether one of the flags is true. If so, it returns true.

rmgpy.qm.symmetry

class rmgpy.qm.symmetry.PointGroup(pointGroup, symmetryNumber, chiral)
A symmetry Point Group.
Attributes are:
• pointGroup
• symmetryNumber
• chiral
• linear

class rmgpy.qm.symmetry.PointGroupCalculator(settings, uniqueID, qmData)
Wrapper type to determine molecular symmetry point groups based on 3D coords information.
Will point to a specific algorithm, like SYMMETRY that is able to do this.

class rmgpy.qm.symmetry.SymmetryJob(settings, uniqueID, qmData)
Determine the point group using the SYMMETRY program
(http://www.cobalt.chem.ucalgary.ca/ps/symmetry/).
Required input is a line with number of atoms followed by lines for each atom including: 1) atom number 2)
x,y,z coordinates
finalTol determines how loose the point group criteria are; values are comparable to those specified in the
GaussView point group interface

calculate()
Do the entire point group calculation.
This writes the input file, then tries several times to run ‘symmetry’ with different parameters, until a point
group is found and returned.

inputFilePath
The input file’s path

parse(output)
Check the output string and extract the resulting point group, which is returned.

run(command)
Run the command, wait for it to finish, and return the stdout.

uniqueID = None
The object that holds information from a previous QM Job on 3D coords, molecule etc...
writeInputFile()
Write the input file for the SYMMETRY program.

rmgpy.qm.gaussian

class rmgpy.qm.gaussian.Gaussian
A base class for all QM calculations that use Gaussian.
Classes such as GaussianMol will inherit from this class.

failureKeys = ['ERROR TERMINATION', 'IMAGINARY FREQUENCIES']
List of phrases that indicate failure. NONE of these must be present in a successful job.

parse()
Parses the results of the Gaussian calculation, and returns a QMData object.

successKeys = ['Normal termination of Gaussian']
List of phrases to indicate success. ALL of these must be present in a successful job.

verifyOutputFile()
Check’s that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the
molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria:
1) finding an output file with
the file name equal to the InChI Key
2) NOT finding any of the keywords that denote a calculation
failure
3) finding all the keywords that denote a calculation success.
4) finding a match between the InChI
of the given molecule and the InChI found in the calculation files
5) checking that the optimized geometry,
when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

class rmgpy.qm.gaussian.GaussianMol(molecule, settings)
A base Class for calculations of molecules using Gaussian.

Inherits from both QMMolecule and Gaussian.

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.
Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be gen-
erated before this method is called.

checkPaths()
Check the paths in the settings are OK. Make folders as necessary.

checkReady()
Check that it’s ready to run calculations.

createGeometry()
Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program
Stores the resulting PointGroup in self.pointGroup

generateQMData()
Calculate the QM data and return a QMData object.
generateThermoData()
    Generate Thermo Data via a QM calc.
    Returns None if it fails.

getFilePath(extension, scratch=True)
    Returns the path to the file with the given extension.
    The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

getInChiKeyAug()
    Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(attempt)
    Get the path to the MOL file of the geometry to use for calculation attempt.
    If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(outputFile)
    Returns the appropriate cclib parser.

getThermoFilePath()
    Returns the path the thermo data file.

initialize()
    Do any startup tasks.

inputFileKeywords(attempt)
    Return the top keywords for attempt number attempt.
    NB. attempt begins at 1, not 0.

inputFilePath
    Get the input file name.

loadThermoData()
    Try loading a thermo data from a previous run.

maxAttempts
    The total number of attempts to try

outputFilePath
    Get the output file name.

parse()
    Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
    Save the generated thermo data.

scriptAttempts
    The number of attempts with different script keywords

verifyOutputFile()
    Check's that an output file exists and was successful.
    Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.
    The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI
of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

**writeInputFile**(attempt)
Using the Geometry object, write the input file for the attempt.

class rmgpy.qm.gaussian.GaussianMolPM3(molecule, settings)
Gaussian PM3 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it’s only the ‘pm3’ in the keywords that differs.

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()
Check the paths in the settings are OK. Make folders as necessary.

checkReady()
Check that it’s ready to run calculations.

createGeometry()
Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.pointGroup

generateQMData()
Calculate the QM data and return a QMData object.

generateThermoData()
Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(extension, scratch=True)
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(attempt)
Get the path to the MOL file of the geometry to use for calculation attempt.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(outputFile)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.
initialize()
Do any startup tasks.

inputFileKeywords(attempt)
Return the top keywords for attempt number attempt.

NB. attempt begins at 1, not 0.

inputFilePath
Get the input file name.

keywords = ['# pm3 opt=(verytight,gdiis) freq IOP(2/16=3)', '# pm3 opt=(verytight,gdiis) freq IOP(2/16=3) IOP(4/21=2)', '# pm3 opt=(verytight,gdiis,calcall,small) IOP(2/16=3) nosymm', '# pm3 opt=(calcall,small,maxcyc=100) IOP(2/16=3)'
Keywords that will be added at the top of the qm input file

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the Inchl found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

writeInputFile(attempt)
Using the Geometry object, write the input file for the attempt.

class rmgpy.qm.gaussian.GaussianMolPM6(molecule, settings)
Gaussian PM6 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it’s only the ‘pm6’ in the keywords that differs.

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.
checkPaths()
    Check the paths in the settings are OK. Make folders as necessary.

checkReady()
    Check that it’s ready to run calculations.

createGeometry()
    Creates self.geometry with RDKit geometries

determinePointGroup()
    Determine point group using the SYMMETRY Program
    Stores the resulting PointGroup in self.pointGroup

generateQMData()
    Calculate the QM data and return a QMData object.

generateThermoData()
    Generate Thermo Data via a QM calc.
    Returns None if it fails.

getFilePath(extension, scratch=True)
    Returns the path to the file with the given extension.
    The provided extension should include the leading dot. If called with scratch=False then it will be in the
    fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

getInChiKeyAug()
    Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(attempt)
    Get the path to the MOL file of the geometry to use for calculation attempt.
    If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coor-
    dinates.

getParser(outputFile)
    Returns the appropriate cclib parser.

getThermoFilePath()
    Returns the path the thermo data file.

initialize()
    Do any startup tasks.

inputFileKeywords(attempt)
    Return the top keywords for attempt number attempt.
    NB. attempt begins at 1, not 0.

inputFilePath
    Get the input file name.

keywords = ['# pm6 opt=(verytight,gdiis) freq IOP(2/16=3)', '# pm6 opt=(verytight,gdiis) freq IOP(2/16=3) IOP(4/21=2)',
            '# pm6 opt=(calcall,small) IOP(2/16=3) nosymm', '# pm6 opt=(calcall,small,maxcyc=100) IOP(2/16=3)'
            'Keywords that will be added at the top of the qm input file

loadThermoData()
    Try loading a thermo data from a previous run.

maxAttempts
    The total number of attempts to try

outputFilePath
    Get the output file name.
parse()
   Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
   Save the generated thermo data.

scriptAttempts
   The number of attempts with different script keywords

verifyOutputFile()
   Check’s that an output file exists and was successful.
   Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

   The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

   If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

writeInputFile(attempt)
   Using the Geometry object, write the input file for the attempt.

rmgpy.qm.mopac

class rmgpy.qm.mopac.Mopac
   A base class for all QM calculations that use MOPAC.
   Classes such as MopacMol will inherit from this class.

   failureKeys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']
   List of phrases that indicate failure NONE of these must be present in a succesful job.

   getParser(outputFile)
      Returns the appropriate cclib parser.

   successKeys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']
      List of phrases to indicate success. ALL of these must be present in a successful job.

   usePolar = False
      Keywords for the multiplicity

   verifyOutputFile()
      Check’s that an output file exists and was successful.

      Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

      The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

      If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

class rmgpy.qm.mopac.MopacMol(molecule, settings)
   A base Class for calculations of molecules using MOPAC.
Inherits from both QMolecule and Mopac.

**calculateChiralityCorrection()**
- Returns the chirality correction to entropy ($R \times \ln(2)$ if chiral) in J/mol/K.

**calculateThermoData()**
- Calculate the thermodynamic properties.
  - Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

**checkPaths()**
- Check the paths in the settings are OK. Make folders as necessary.

**checkReady()**
- Check that it’s ready to run calculations.

**createGeometry()**
- Creates self.geometry with RDKit geometries

**determinePointGroup()**
- Determine point group using the SYMMETRY Program
  - Stores the resulting PointGroup in self.pointGroup

**generateQMData()**
- Calculate the QM data and return a QMData object, or None if it fails.

**generateThermoData()**
- Generate Thermo Data via a QM calc.
  - Returns None if it fails.

**getFilePath(extension, scratch=True)**
- Returns the path to the file with the given extension.
  - The provided extension should include the leading dot. If called with `scratch=False` then it will be in the `fileStore` directory, else `scratch=True` is assumed and it will be in the `scratchDirectory` directory.

**getInChiKeyAug()**
- Returns the augmented InChi from self.molecule

**getMolFilePathForCalculation(attempt)**
- Get the path to the MOL file of the geometry to use for calculation attempt.
  - If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

**getParser(outputFile)**
- Returns the appropriate cclib parser.

**getThermoFilePath()**
- Returns the path the thermo data file.

**initialize()**
- Do any startup tasks.

**inputFileKeywords(attempt)**
- Return the top, bottom, and polar keywords.

**inputFilePath**
- Get the input file name.

**keywords = [{‘top’: ‘precise nosym THREADS=1’, ‘bottom’: ‘oldgeo thermo nosym precise THREADS=1 ’}, {‘top’: ‘precise nosym THREADS=1’, ‘bottom’: ‘oldgeo thermo nosym precise THREADS=1 ’}]**
- Keywords that will be added at the top and bottom of the qm input file
loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMDData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

writeInputFile(attempt)
Using the Geometry object, write the input file for the attempt.

class rmgpy.qm.mopac.MopacMolPM3(molecule, settings)
Mopac PM3 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it’s the same as all the MOPAC PMn calculations, only pm3

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.
Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()
Check the paths in the settings are OK. Make folders as necessary.

checkReady()
Check that it’s ready to run calculations.

createGeometry()
Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program
Stores the resulting PointGroup in self.pointGroup
generateQMData()  
Calculate the QM data and return a QMData object, or None if it fails.

generateThermoData()  
Generate Thermo Data via a QM calc.  
Returns None if it fails.

getFilePath(extension, scratch=True)  
Returns the path to the file with the given extension.  
The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

getInChIKeyAug()  
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(attempt)  
Get the path to the MOL file of the geometry to use for calculation attempt.  
If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(outputFile)  
Returns the appropriate cclib parser.

getThermoFilePath()  
Returns the path the thermo data file.

initialize()  
Do any startup tasks.

inputFileKeywords(attempt)  
Return the top, bottom, and polar keywords for attempt number attempt.  
NB. attempt begins at 1, not 0.

inputFilePath  
Get the input file name.

loadThermoData()  
Try loading a thermo data from a previous run.

maxAttempts  
The total number of attempts to try

outputFilePath  
Get the output file name.

parse()  
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()  
Save the generated thermo data.

scriptAttempts  
The number of attempts with different script keywords

verifyOutputFile()  
Check’s that an output file exists and was successful.  
Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.
The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

```python
def writeInputFile(attempt):
    # Using the Geometry object, write the input file for the attempt.

class rmgpy.qm.mopac.MopacMolPM6(molecule, settings):
    # Mopac PM6 calculations for molecules
    # This is a class of its own in case you wish to do anything differently, but for now it's the same as all the MOPAC PMn calculations, only pm6
    def calculateChiralityCorrection():
        # Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.
    def calculateThermoData():
        # Calculate the thermodynamic properties.
        # Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.
    def checkPaths():
        # Check the paths in the settings are OK. Make folders as necessary.
    def checkReady():
        # Check that it's ready to run calculations.
    def createGeometry():
        # Creates self.geometry with RDKit geometries
    def determinePointGroup():
        # Determine point group using the SYMMETRY Program
        # Stores the resulting PointGroup in self.pointGroup
    def generateQMData():
        # Calculate the QM data and return a QMData object, or None if it fails.
    def generateThermoData():
        # Generate Thermo Data via a QM calc.
        # Returns None if it fails.
    def getFilePath(extension, scratch=True):
        # Returns the path to the file with the given extension.
        # The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.
    def getInChiKeyAug():
        # Returns the augmented InChI from self.molecule
    def getMolFilePathForCalculation(attempt):
        # Get the path to the MOL file of the geometry to use for calculation attempt.
        # If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.
```

1.8. QMTP (rmgpy.qm)
**getParser** (*outputFile*)

Returns the appropriate cclib parser.

**getThermoFilePath**()

Returns the path the thermo data file.

**initialize**()

Do any startup tasks.

**inputFileKeywords** (*attempt*)

Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

**inputFilePath**

Get the input file name.

**loadThermoData**()

Try loading a thermo data from a previous run.

**maxAttempts**

The total number of attempts to try

**outputFilePath**

Get the output file name.

**parse**()

Parses the results of the Mopac calculation, and returns a QMData object.

**saveThermoData**()

Save the generated thermo data.

**scriptAttempts**

The number of attempts with different script keywords

**verifyOutputFile**()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

**writeInputFile** (*attempt*)

Using the Geometry object, write the input file for the *attempt*.

**class** **rmgpy.qm.mopac.MopacMolPM7** (*molecule, settings*)

Mopac PM7 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it’s the same as all the MOPAC PMn calculations, only pm7

**calculateChiralityCorrection**()

Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

**calculateThermoData**()

Calculate the thermodynamic properties.
Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

**checkPaths()**
- Check the paths in the settings are OK. Make folders as necessary.

**checkReady()**
- Check that it’s ready to run calculations.

**createGeometry()**
- Creates self.geometry with RDKit geometries

**determinePointGroup()**
- Determine point group using the SYMMETRY Program
  - Stores the resulting PointGroup in self.pointGroup

**generateQMData()**
- Calculate the QM data and return a QMData object, or None if it fails.

**generateThermoData()**
- Generate Thermo Data via a QM calc.
  - Returns None if it fails.

**getFilePath(extension, scratch=True)**
- Returns the path to the file with the given extension.
  - The provided extension should include the leading dot. If called with scratch=False then it will be in the fileStore directory, else scratch=True is assumed and it will be in the scratchDirectory directory.

**getInChiKeyAug()**
- Returns the augmented InChI from self.molecule

**getMolFilePathForCalculation(attempt)**
- Get the path to the MOL file of the geometry to use for calculation attempt.
  - If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

**getParser(outputFile)**
- Returns the appropriate cclib parser.

**getThermoFilePath()**
- Returns the path the thermo data file.

**initialize()**
- Do any startup tasks.

**inputFileKeywords(attempt)**
- Return the top, bottom, and polar keywords for attempt number attempt.
  - NB. attempt begins at 1, not 0.

**inputFilePath**
- Get the input file name.

**loadThermoData()**
- Try loading a thermo data from a previous run.

**maxAttempts**
- The total number of attempts to try

**outputFilePath**
- Get the output file name.
parse()

Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()

Save the generated thermo data.

scriptAttempts

The number of attempts with different script keywords

verifyOutputFile()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

writeInputFile(attempt)

Using the Geometry object, write the input file for the attempt.

1.9 Physical quantities (rmgpy.quantity)

A physical quantity is defined by a numerical value and a unit of measurement.

The `rmgpy.quantity` module contains classes and methods for working with physical quantities. Physical quantities are represented by either the `ScalarQuantity` or `ArrayQuantity` class depending on whether a scalar or vector (or tensor) value is used. The `Quantity` function automatically chooses the appropriate class based on the input value. In both cases, the value of a physical quantity is available from the `value` attribute, and the units from the `units` attribute.

For efficient computation, the value is stored internally in the SI equivalent units. The SI value can be accessed directly using the `value_si` attribute. Usually it is good practice to read the `value_si` attribute into a local variable and then use it for computations, especially if it is referred to multiple times in the calculation.

Physical quantities also allow for storing of uncertainty values for both scalars and arrays. The `uncertaintyType` attribute indicates whether the given uncertainties are additive ("+\/-") or multiplicative ("*|/"). and the `uncertainty` attribute contains the stored uncertainties. For additive uncertainties these are stored in the given units (not the SI equivalent), since they are generally not needed for efficient computations. For multiplicative uncertainties, the uncertainty values are by definition dimensionless.

1.9.1 Quantity objects

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ScalarQuantity</code></td>
<td>A scalar physical quantity, with units and uncertainty</td>
</tr>
<tr>
<td><code>ArrayQuantity</code></td>
<td>An array physical quantity, with units and uncertainty</td>
</tr>
<tr>
<td><code>Quantity()</code></td>
<td>Return a scalar or array physical quantity</td>
</tr>
</tbody>
</table>
1.9.2 Unit types

Units can be classified into categories based on the associated dimensionality. For example, miles and kilometers are both units of length; seconds and hours are both units of time, etc. Clearly, quantities of different unit types are fundamentally different.

RMG provides functions that create physical quantities (scalar or array) and validate the units for a variety of unit types. This prevents the user from inadvertently mixing up their units - e.g. by setting an enthalpy with entropy units - which should reduce errors. RMG recognizes the following unit types:

<table>
<thead>
<tr>
<th>Function</th>
<th>Unit type</th>
<th>SI unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration()</td>
<td>acceleration</td>
<td>m/s²</td>
</tr>
<tr>
<td>Area()</td>
<td>area</td>
<td>m²</td>
</tr>
<tr>
<td>Concentration()</td>
<td>concentration</td>
<td>mol/cm³</td>
</tr>
<tr>
<td>Dimensionless()</td>
<td>dimensionless</td>
<td></td>
</tr>
<tr>
<td>Energy()</td>
<td>energy</td>
<td>J/mol</td>
</tr>
<tr>
<td>Entropy()</td>
<td>entropy</td>
<td>J/mol·K</td>
</tr>
<tr>
<td>Flux()</td>
<td>flux</td>
<td>mol/cm²·s</td>
</tr>
<tr>
<td>Frequency()</td>
<td>frequency</td>
<td>cm⁻¹</td>
</tr>
<tr>
<td>Force()</td>
<td>force</td>
<td>N</td>
</tr>
<tr>
<td>Inertia()</td>
<td>inertia</td>
<td>kg·m²</td>
</tr>
<tr>
<td>Length()</td>
<td>length</td>
<td>m</td>
</tr>
<tr>
<td>Mass()</td>
<td>mass</td>
<td>kg</td>
</tr>
<tr>
<td>Momentum()</td>
<td>momentum</td>
<td>kg·m/s²</td>
</tr>
<tr>
<td>Power()</td>
<td>power</td>
<td>W</td>
</tr>
<tr>
<td>Pressure()</td>
<td>pressure</td>
<td>Pa</td>
</tr>
<tr>
<td>RateCoefficient()</td>
<td>rate coefficient</td>
<td>s⁻¹, m⁴/mol·s, m⁶/mol²·s, m⁸/mol⁴·s</td>
</tr>
<tr>
<td>Temperature()</td>
<td>temperature</td>
<td>K</td>
</tr>
<tr>
<td>Time()</td>
<td>time</td>
<td>s</td>
</tr>
<tr>
<td>Velocity()</td>
<td>velocity</td>
<td>m/s</td>
</tr>
<tr>
<td>Volume()</td>
<td>volume</td>
<td>m³</td>
</tr>
</tbody>
</table>

In RMG, all energies, heat capacities, concentrations, fluxes, and rate coefficients are treated as intensive; this means that these quantities are always expressed “per mole” or “per molecule”. All other unit types are extensive. A special exception is added for mass so as to allow for coercion of g/mol to amu.

RMG also handles rate coefficient units as a special case, as there are multiple allowed dimensionalities based on the reaction order. Note that RMG generally does not attempt to verify that the rate coefficient units match the reaction order, but only that it matches one of the possibilities.

The table above gives the SI unit that RMG uses internally to work with physical quantities. This does not necessarily correspond with the units used when outputting values. For example, pressures are often output in units of bar instead of Pa, and moments of inertia in amu * angstrom² instead of kg * m². The recommended rule of thumb is to use prefixed SI units (or aliases thereof) in the output; for example, use kJ/mol instead of kcal/mol for energy values.

rmgpy.quantity.ScalarQuantity

class rmgpy.quantity.ScalarQuantity

The ScalarQuantity class provides a representation of a scalar physical quantity, with optional units and uncertainty information. The attributes are:
### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>The numeric value of the quantity in the given units</td>
</tr>
<tr>
<td><code>units</code></td>
<td>The units the value was specified in</td>
</tr>
<tr>
<td><code>uncertainty</code></td>
<td>The numeric uncertainty in the value in the given units (unitless if multiplicative)</td>
</tr>
<tr>
<td><code>uncertaintyType</code></td>
<td>The type of uncertainty: '+</td>
</tr>
<tr>
<td><code>value_si</code></td>
<td>The numeric value of the quantity in the corresponding SI units</td>
</tr>
<tr>
<td><code>uncertainty_si</code></td>
<td>The numeric value of the uncertainty in the corresponding SI units (unitless if multiplicative)</td>
</tr>
</tbody>
</table>

It is often more convenient to perform computations using SI units instead of the given units of the quantity. For this reason, the SI equivalent of the `value` attribute can be directly accessed using the `value_si` attribute. This value is cached on the `ScalarQuantity` object for speed.

**`copy()`**

Return a copy of the quantity.

**`equals()`**

Return `True` if the everything in a quantity object matches the parameters in this object. If there are lists of values or uncertainties, each item in the list must be matching and in the same order. Otherwise, return `False` (Originally intended to return warning if units capitalization was different, however, Quantity object only parses units matching in case, so this will not be a problem.)

**`getConversionFactorFromSI()`**

Return the conversion factor for converting a quantity to a given set of `units` from the SI equivalent units.

**`getConversionFactorToSI()`**

Return the conversion factor for converting a quantity in a given set of `units` to the SI equivalent units.

**`getUncertainty()`**

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

**`getUncertaintyType()`**

The type of uncertainty: '+|-' for additive, '*|/' for multiplicative

**`getValue()`**

The numeric value of the quantity, in the given units

**`isUncertaintyAdditive()`**

Return `True` if the uncertainty is specified in additive format and `False` otherwise.

**`isUncertaintyMultiplicative()`**

Return `True` if the uncertainty is specified in multiplicative format and `False` otherwise.

**`setUncertaintyType()`**

Check the uncertainty type is valid, then set it.

**`uncertainty`**

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

**`uncertaintyType`**

The type of uncertainty: '+|-' for additive, '*|/' for multiplicative

**`value`**

The numeric value of the quantity, in the given units

---

**`rmgpy.quantity.ArrayQuantity`**

The `ArrayQuantity` class provides a representation of an array of physical quantity values, with optional units.
and uncertainty information. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The numeric value of the quantity in the given units</td>
</tr>
<tr>
<td>units</td>
<td>The units the value was specified in</td>
</tr>
<tr>
<td>uncertainty</td>
<td>The numeric uncertainty in the value (unitless if multiplicative)</td>
</tr>
<tr>
<td>uncertainty-</td>
<td>Type: '+</td>
</tr>
<tr>
<td>value_si</td>
<td>The numeric value of the quantity in the corresponding SI units</td>
</tr>
<tr>
<td>uncertainty_si</td>
<td>The numeric value of the uncertainty in the corresponding SI units (unitless if multiplicative)</td>
</tr>
</tbody>
</table>

It is often more convenient to perform computations using SI units instead of the given units of the quantity. For this reason, the SI equivalent of the value attribute can be directly accessed using the value_si attribute. This value is cached on the ArrayQuantity object for speed.

**copy()**

Return a copy of the quantity.

**equals()**

Return True if the everything in a quantity object matches the parameters in this object. If there are lists of values or uncertainties, each item in the list must be matching and in the same order. Otherwise, return False (Originally intended to return warning if units capitalization was different, however, Quantity object only parses units matching in case, so this will not be a problem.)

**getConversionFactorFromSI()**

Return the conversion factor for converting a quantity to a given set of units from the SI equivalent units.

**getConversionFactorToSI()**

Return the conversion factor for converting a quantity in a given set of units' to the SI equivalent units.

**getUncertainty()**

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

**getUncertaintyType()**

The type of uncertainty: '+|-' for additive, '*|/' for multiplicative

**getValue()**

The numeric value of the array quantity, in the given units.

**isUncertaintyAdditive()**

Return True if the uncertainty is specified in additive format and False otherwise.

**isUncertaintyMultiplicative()**

Return True if the uncertainty is specified in multiplicative format and False otherwise.

**setUncertaintyType()**

Check the uncertainty type is valid, then set it.

If you set the uncertainty then change the type, we have no idea what to do with the units. This ensures you set the type first.

**uncertainty**

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

**uncertaintyType**

The type of uncertainty: '+|-' for additive, '*|/' for multiplicative

**value**

The numeric value of the array quantity, in the given units.
rmgpy.quantity.Quantity

Create a `ScalarQuantity` or `ArrayQuantity` object for a given physical quantity. The physical quantity can be specified in several ways:

- A scalar-like or array-like value (for a dimensionless quantity)
- An array of arguments (including keyword arguments) giving some or all of the `value`, `units`, `uncertainty`, and/or `uncertaintyType`
- A tuple of the form (value,), (value,units), (value,units,uncertainty), or (value,units,uncertaintyType,uncertainty)
- An existing `ScalarQuantity` or `ArrayQuantity` object, for which a copy is made

1.10 Reactions (rmgpy.reaction)

The `rmgpy.reaction` subpackage contains classes and functions for working with chemical reaction.

1.10.1 Reaction

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaction</td>
<td>A chemical reaction</td>
</tr>
</tbody>
</table>

1.10.2 Exceptions

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactionError</td>
<td>Raised when an error occurs while working with reactions</td>
</tr>
</tbody>
</table>

```python
class rmgpy.reaction.Reaction

A chemical reaction. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>int</td>
<td>A unique nonnegative integer index</td>
</tr>
<tr>
<td>label</td>
<td>str</td>
<td>A descriptive string label</td>
</tr>
<tr>
<td>reactants</td>
<td>list</td>
<td>The reactant species (as Species objects)</td>
</tr>
<tr>
<td>products</td>
<td>list</td>
<td>The product species (as Species objects)</td>
</tr>
<tr>
<td>kinetics</td>
<td>KineticsModel</td>
<td>The kinetics model to use for the reaction</td>
</tr>
<tr>
<td>reversible</td>
<td>bool</td>
<td>True if the reaction is reversible, False if not</td>
</tr>
<tr>
<td>transition-State</td>
<td>TransitionState</td>
<td>The transition state</td>
</tr>
<tr>
<td>duplicate</td>
<td>bool</td>
<td>True if the reaction is known to be a duplicate, False if not</td>
</tr>
<tr>
<td>degeneracy</td>
<td>double</td>
<td>The reaction path degeneracy for the reaction</td>
</tr>
<tr>
<td>pairs</td>
<td>list</td>
<td>Reactant-product pairings to use in converting reaction flux to species flux</td>
</tr>
</tbody>
</table>
```

calculateMicrocanonicalRateCoefficient()  
Calculate the microcanonical rate coefficient \( k(E) \) for the reaction reaction at the energies Elist in J/mol. reacDensStates and prodDensStates are the densities of states of the reactant and product configurations
for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.
- If the above is not possible but high-pressure limit kinetics \( k_\infty(T) \) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product \( \text{prodDensStates} \) and the temperature of interest \( T \) in K can also be provided. For isomerization and association reactions \( \text{prodDensStates} \) is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.

**calculateTSTRateCoefficient()**
Evaluate the forward rate coefficient for the reaction with corresponding transition state \( TS \) at temperature \( T \) in K using (canonical) transition state theory. The TST equation is

\[
k(T) = \kappa(T) \frac{k_B T}{\hbar} \frac{Q^\dagger(T)}{Q^A(T) Q^B(T)} \exp \left( \frac{-E_0}{k_B T} \right)
\]

where \( Q^\dagger \) is the partition function of the transition state, \( Q^A \) and \( Q^B \) are the partition function of the reactants, \( E_0 \) is the ground-state energy difference from the transition state to the reactants, \( T \) is the absolute temperature, \( k_B \) is the Boltzmann constant, and \( \hbar \) is the Planck constant. \( \kappa(T) \) is an optional tunneling correction.

**canTST()**
Return True if the necessary parameters are available for using transition state theory – or the microcanonical equivalent, RRKM theory – to compute the rate coefficient for this reaction, or False otherwise.

**copy()**
Create a deep copy of the current reaction.

**draw()**
Generate a pictorial representation of the chemical reaction using the draw module. Use path to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

**fixBarrierHeight()**
Turns the kinetics into Arrhenius (if they were ArrheniusEP) and ensures the activation energy is at least the endothermicity for endothermic reactions, and is not negative only as a result of using Evans Polanyi with an exothermic reaction. If forcePositive is True, then all reactions are forced to have a non-negative barrier.

**fixDiffusionLimitedA()**
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

**generate3dTS()**
Generate the 3D structure of the transition state. Called from model.generateKinetics().

self.reactants is a list of reactants self.products is a list of products

**generatePairs()**
Generate the reactant-product pairs to use for this reaction when performing flux analysis. The exact procedure for doing so depends on the reaction type:
There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

<table>
<thead>
<tr>
<th>Reaction type</th>
<th>Template</th>
<th>Resulting pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isomerization</td>
<td>A \rightarrow C</td>
<td>(A, C)</td>
</tr>
<tr>
<td>Dissociation</td>
<td>A \rightarrow C + D</td>
<td>(A, C), (A, D)</td>
</tr>
<tr>
<td>Association</td>
<td>A + B \rightarrow C</td>
<td>(A, C), (B, C)</td>
</tr>
<tr>
<td>Bimolecular</td>
<td>A + B \rightarrow C + D</td>
<td>(A, C), (B, D) or (A, D), (B, C)</td>
</tr>
</tbody>
</table>

There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

`generateReverseRateCoefficient()`  
Generate and return a rate coefficient model for the reverse reaction. Currently this only works if the `kinetics` attribute is one of several (but not necessarily all) kinetics types.

`getEnthalpiesOfReaction()`  
Return the enthalpies of reaction in J/mol evaluated at temperatures `Tlist` in K.

`getEnthalpyOfReaction()`  
Return the enthalpy of reaction in J/mol evaluated at temperature `T` in K.

`getEntropiesOfReaction()`  
Return the entropies of reaction in J/mol*K evaluated at temperatures `Tlist` in K.

`getEntropyOfReaction()`  
Return the entropy of reaction in J/mol*K evaluated at temperature `T` in K.

`getEquilibriumConstant()`  
Return the equilibrium constant for the reaction at the specified temperature `T` in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

`getEquilibriumConstants()`  
Return the equilibrium constants for the reaction at the specified temperatures `Tlist` in K. The `type` parameter lets you specify the quantities used in the equilibrium constant: $K_a$ for activities, $K_c$ for concentrations (default), or $K_p$ for pressures. Note that this function currently assumes an ideal gas mixture.

`getFreeEnergiesOfReaction()`  
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures `Tlist` in K.

`getFreeEnergyOfReaction()`  
Return the Gibbs free energy of reaction in J/mol evaluated at temperature `T` in K.

`getRateCoefficient()`  
Return the overall rate coefficient for the forward reaction at temperature `T` in K and pressure `P` in Pa, including any reaction path degeneracies.

If `diffusionLimiter` is enabled, the reaction is in the liquid phase and we use a diffusion limitation to correct the rate. If not, then use the intrinsic rate coefficient.

`getStoichiometricCoefficient()`  
Return the stoichiometric coefficient of species `spec` in the reaction. The stoichiometric coefficient is increased by one for each time `spec` appears as a product and decreased by one for each time `spec` appears as a reactant.

`getURL()`  
Get a URL to search for this reaction in the rmg website.

`hasTemplate()`  
Return `True` if the reaction matches the template of `reactants` and `products`, which are both lists of `Species` objects, or `False` if not.
isAssociation()

Return True if the reaction represents an association reaction A + B ⇌ C or False if not.

isBalanced()

Return True if the reaction has the same number of each atom on each side of the reaction equation, or False if not.

isDissociation()

Return True if the reaction represents a dissociation reaction A ⇌ B + C or False if not.

isIsomerization()

Return True if the reaction represents an isomerization reaction A ⇌ B or False if not.

isIsomorphic()

Return True if this reaction is the same as the other reaction, or False if they are different. If eitherDirection=False then the directions must match.

isUnimolecular()

Return True if the reaction has a single molecule as either reactant or product (or both) A ⇌ B + C or A ⇌ C or A ⇌ B, or False if not.

matchesMolecules()

Return True if the given reactants represent the total set of reactants or products for the current reaction, or False if not. The reactants should be Molecule objects.

reverseThisArrheniusRate()

Reverses the given kForward, which must be an Arrhenius type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

toCantera()

Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

toChemkin()

Return the chemkin-formatted string for this reaction.

If kinetics is set to True, the chemkin format kinetics will also be returned (requires the speciesList to figure out third body colliders.) Otherwise, only the reaction string will be returned.

### 1.11 Reaction mechanism generation (rmgpy.rmg)

The rmgpy.rmg subpackage contains the main functionality for using RMG-Py to automatically generate detailed reaction mechanisms.

#### 1.11.1 Reaction models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Species</td>
<td>A chemical species, with RMG-specific functionality</td>
</tr>
<tr>
<td>CoreEdgeReactionModel</td>
<td>A reaction model comprised of core and edge species and reactions</td>
</tr>
</tbody>
</table>

#### 1.11.2 Input

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readInputFile()</td>
<td>Load an RMG job input file</td>
</tr>
<tr>
<td>saveInputFile()</td>
<td>Save an RMG job input file</td>
</tr>
</tbody>
</table>
1.11.3 Output

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saveOutputHTML()</code></td>
<td>Save the results of an RMG job to an HTML file</td>
</tr>
<tr>
<td><code>saveDiffHTML()</code></td>
<td>Save a comparison of two reaction mechanisms to an HTML file</td>
</tr>
</tbody>
</table>

1.11.4 Job classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RMG</code></td>
<td>Main class for RMG jobs</td>
</tr>
</tbody>
</table>

1.11.5 Pressure dependence

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDepReaction</code></td>
<td>A pressure-dependent “net” reaction</td>
</tr>
<tr>
<td><code>PDepNetwork</code></td>
<td>A pressure-dependent unimolecular reaction network, with RMG-specific functionality</td>
</tr>
</tbody>
</table>

1.11.6 Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputError</code></td>
<td>Raised when an error occurs while working with an RMG input file</td>
</tr>
<tr>
<td><code>OutputError</code></td>
<td>Raised when an error occurs while saving an RMG output file</td>
</tr>
<tr>
<td><code>PressureDependenceError</code></td>
<td>Raised when an error occurs while computing pressure-dependent rate coefficients</td>
</tr>
</tbody>
</table>

`rmgpy.rmg.model.CoreEdgeReactionModel`

```python
class rmgpy.rmg.model.CoreEdgeReactionModel(core=None, edge=None):
    Represent a reaction model constructed using a rate-based screening algorithm. The species and reactions in the model itself are called the `core`; the species and reactions identified as candidates for inclusion in the model are called the `edge`. The attributes are:
```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>core</code></td>
<td>The species and reactions of the current model core</td>
</tr>
<tr>
<td><code>edge</code></td>
<td>The species and reactions of the current model edge</td>
</tr>
<tr>
<td><code>networkDict</code></td>
<td>A dictionary of pressure-dependent reaction networks (Network objects) indexed by source.</td>
</tr>
<tr>
<td><code>networkList</code></td>
<td>A list of pressure-dependent reaction networks (Network objects)</td>
</tr>
<tr>
<td><code>networkCount</code></td>
<td>A counter for the number of pressure-dependent networks created</td>
</tr>
<tr>
<td><code>indexSpeciesDict</code></td>
<td>A dictionary with a unique index pointing to the species objects</td>
</tr>
</tbody>
</table>

```python
addReactionLibraryToEdge(reactionLibrary)
```
Add all species and reactions from `reactionLibrary`, a KineticsPrimaryDatabase object, to the model edge.

```python
addReactionLibraryToOutput(reactionLib)
```
Add all species and reactions from `reactionLibrary`, a KineticsPrimaryDatabase object, to the output. This does not bring any of the reactions or species into the core itself.
addReactionToCore(rxn)
Add a reaction rxn to the reaction model core (and remove from edge if necessary). This function assumes rxn has already been checked to ensure it is supposed to be a core reaction (i.e. all of its reactants AND all of its products are in the list of core species).

addReactionToEdge(rxn)
Add a reaction rxn to the reaction model edge. This function assumes rxn has already been checked to ensure it is supposed to be an edge reaction (i.e. all of its reactants OR all of its products are in the list of core species, and the others are in either the core or the edge).

addReactionToUnimolecularNetworks(newReaction, newSpecies, network=None)
Given a newly-created Reaction object newReaction, update the corresponding unimolecular reaction network. If no network exists, a new one is created. If the new reaction is an isomerization that connects two existing networks, the two networks are merged. This function is called whenever a new high-pressure limit edge reaction is created. Returns the network containing the new reaction.

addSeedMechanismToCore(seedMechanism, react=False)
Add all species and reactions from seedMechanism, a KineticsPrimaryDatabase object, to the model core. If react is True, then reactions will also be generated between the seed species. For large seed mechanisms this can be prohibitively expensive, so it is not done by default.

addSpeciesToCore(spec)
Add a species spec to the reaction model core (and remove from edge if necessary). This function also moves any reactions in the edge that gain core status as a result of this change in status to the core. If there are any such reactions, they are returned in a list.

addSpeciesToEdge(spec)
Add a species spec to the reaction model edge.

checkForExistingReaction(rxn)
Check to see if an existing reaction has the same reactants, products, and family as rxn. Returns True or False and the matched reaction (if found).

First, a shortlist of reaction is retrieved that have the same reaction keys as the parameter reaction.

Next, the reaction ID containing an identifier (e.g. label) of the reactants and products is compared between the parameter reaction and the each of the reactions in the shortlist. If a match is found, the discovered reaction is returned.

If a match is not yet found, the Library (seed mechs, reaction libs) in the reaction database are iterated over to check if a reaction was overlooked (a reaction with a different “family” key as the parameter reaction).

checkForExistingSpecies(molecule)
Check to see if an existing species contains the same molecule.Molecule as molecule. Returns True and the matched species (if found) or False and None (if not found).

enlarge(newObject=None, reactEdge=False, unimolecularReact=None, bimolecularReact=None)
Enlarge a reaction model by processing the objects in the list newObject. If newObject is a rmg.species.Species object, then the species is moved from the edge to the core and reactions generated for that species, reacting with itself and with all other species in the model core. If newObject is a rmg.unirxn.network.Network object, then reactions are generated for the species in the network with the largest leak flux.

If the reactEdge flag is True, then no newObject is needed, and instead the algorithm proceeds to react the core species together to form edge reactions.

generateKinetics(reaction)
Generate best possible kinetics for the given reaction using the kinetics database.

getLists()
Return lists of all of the species and reactions in the core and the edge.
getModelSize()
Return the numbers of species and reactions in the model core and edge. Note that this is not necessarily equal to the lengths of the corresponding species and reaction lists.

getSpecies(obj)
Retrieve species object, by polling the index species dictionary.

getStoichiometryMatrix()
Return the stoichiometry matrix for all generated species and reactions. The id of each species and reaction is the corresponding row and column, respectively, in the matrix.

inflate(rx)
Convert reactions from reactants/products that are referring to the core species index, to the respective Species objects.

initializeIndexSpeciesDict()
Populates the core species dictionary integer -> core Species with the species that are currently in the core.

makeNewPDepReaction(forward)
Make a new pressure-dependent reaction based on a list of reactants and a list of products. The reaction belongs to the specified network and has pressure-dependent kinetics given by kinetics.

No checking for existing reactions is made here. The returned PDepReaction object is not added to the global list of reactions, as that is intended to represent only the high-pressure-limit set. The reaction-Counter is incremented, however, since the returned reaction can and will exist in the model edge and/or core.

makeNewReaction(forward, checkExisting=True)
Make a new reaction given a Reaction object forward. The reaction is added to the global list of reactions. Returns the reaction in the direction that corresponds to the estimated kinetics, along with whether or not the reaction is new to the global reaction list.

The forward direction is determined using the “is_reverse” attribute of the reaction’s family. If the reaction family is its own reverse, then it is made such that the forward reaction is exothermic at 298K.

The forward reaction is appended to self.newReactionList if it is new.

makeNewSpecies(object, label='', reactive=True, checkForExisting=True)
Formally create a new species from the specified object, which can be either a Molecule object or an rmgpy.species.Species object.

markChemkinDuplicates()
Check that all reactions that will appear the chemkin output have been checked as duplicates.

Call this if you’ve done something that may have introduced undetected duplicate reactions, like add a reaction library or seed mechanism. Anything added via the expand() method should already be detected.

printEnlargeSummary(newCoreSpecies, newCoreReactions, newEdgeSpecies, newEdgeReactions, reactionsMovedFromEdge=None, reactEdge=False)
Output a summary of a model enlargement step to the log. The details of the enlargement are passed in the newCoreSpecies, newCoreReactions, newEdgeSpecies, and newEdgeReactions objects.

processNewReactions(newReactions, newSpecies, pdepNetwork=None)
Process a list of newly-generated reactions involving the new core species or explored isomer newSpecies in network pdepNetwork.

Makes a reaction and decides where to put it: core, edge, or PDepNetwork.
prune(reactionSystems, toleranceKeepInEdge, maximumEdgeSpecies, minSpeciesExistIterationsForPrune)
Remove species from the model edge based on the simulation results from the list of reactionSystems.

registerReaction(rxn)
Adds the reaction to the reaction database.

The reaction database is structured as a multi-level dictionary, for efficient search and retrieval of existing reactions.

The database has two types of dictionary keys: - reaction family - reactant(s) keys

First, the keys are generated for the parameter reaction.

Next, it is checked whether the reaction database already contains similar keys. If not, a new container is created, either a dictionary for the family key and first reactant key, or a list for the second reactant key.

Finally, the reaction is inserted as the first element in the list.

removeSpeciesFromEdge(spec)
Remove species spec from the reaction model edge.

deletedSpeciesFromEdge(spec)
Remove species spec from the reaction model edge.

retrieve(family_label, key1, key2)
Returns a list of reactions from the reaction database with the same keys as the parameters.

Returns an empty list when one of the keys could not be found.

retrieveNewSpecies(deflatedRxn)
Searches for the first reactant or product in the deflated reaction that is represented by an integer.

Such an object refers to a core species that was used to generate the reaction in the first place. Reactants or products represented by an object that is not an integer will be a newly-generated structure.

searchRetrieveReactions(rxn)
Searches through the reaction database for reactions with an identical reaction key as the key of the parameter reaction.

Both the reaction key based on the reactants as well as on the products is used to search for possible candidate reactions.

updateUnimolecularReactionNetworks()
Iterate through all of the currently-existing unimolecular reaction networks, updating those that have been marked as invalid. In each update, the phenomenological rate coefficients \( k(T, P) \) are computed for each net reaction in the network, and the resulting reactions added or updated.

class rmgpy.rmg.model.ReactionModel(species=None, reactions=None)
Represent a generic reaction model. A reaction model consists of species, a list of species, and reactions, a list of reactions.

merge(other)
Return a new ReactionModel object that is the union of this model and other.

RMG input files

rmgpy.rmg.input.readInputFile(path, rmg0)
Read an RMG input file at path on disk into the RMG object rmg.

rmgpy.rmg.input.saveInputFile(path, rmg)
Save an RMG input file at path on disk from the RMG object rmg.
**rmgpy.rmg.main.RMG**

**class** rmgpy.rmg.main.RMG(inputFile=None, outputDirectory=None)

A representation of a Reaction Mechanism Generator (RMG) job. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputFile</td>
<td>The path to the input file</td>
</tr>
<tr>
<td>databaseDirectory</td>
<td>The directory containing the RMG database</td>
</tr>
<tr>
<td>thermoLibraries</td>
<td>The thermodynamics libraries to load</td>
</tr>
<tr>
<td>reactionLibraries</td>
<td>The kinetics libraries to load</td>
</tr>
<tr>
<td>statmechLibraries</td>
<td>The statistical mechanics libraries to load</td>
</tr>
<tr>
<td>seedMechanisms</td>
<td>The seed mechanisms included in the model</td>
</tr>
<tr>
<td>kineticsFamilies</td>
<td>The kinetics families to use for reaction generation</td>
</tr>
<tr>
<td>kineticsDepositories</td>
<td>The kinetics depositories to use for looking up kinetics in each family</td>
</tr>
<tr>
<td>kineticsEstimator</td>
<td>The method to use to estimate kinetics: ‘group additivity’ or ‘rate rules’</td>
</tr>
<tr>
<td>solvent</td>
<td>If solvation estimates are required, the name of the solvent.</td>
</tr>
<tr>
<td>reactionModel</td>
<td>The core-edge reaction model generated by this job</td>
</tr>
<tr>
<td>reactionSystems</td>
<td>A list of the reaction systems used in this job</td>
</tr>
<tr>
<td>database</td>
<td>The RMG database used in this job</td>
</tr>
<tr>
<td>absoluteTolerance</td>
<td>The absolute tolerance used in the ODE/DAE solver</td>
</tr>
<tr>
<td>relativeTolerance</td>
<td>The relative tolerance used in the ODE/DAE solver</td>
</tr>
<tr>
<td>sensitivityAbsoluteTolerance</td>
<td>The absolute tolerance used in the ODE/DAE solver for the sensitivities</td>
</tr>
<tr>
<td>sensitivityRelativeTolerance</td>
<td>The relative tolerance used in the ODE/DAE solver for the sensitivities</td>
</tr>
<tr>
<td>fluxToleranceKeepInEdge</td>
<td>The relative species flux below which species are discarded from the edge</td>
</tr>
<tr>
<td>fluxToleranceMoveToCore</td>
<td>The relative species flux above which species are moved from the edge to the core</td>
</tr>
<tr>
<td>fluxToleranceInterrupt</td>
<td>The relative species flux above which the simulation will halt</td>
</tr>
<tr>
<td>maximumEdgeSpecies</td>
<td>The maximum number of edge species allowed at any time</td>
</tr>
<tr>
<td>minCoreSizeForPrune</td>
<td>Minimum number of core species before pruning is allowed</td>
</tr>
<tr>
<td>minSpeciesExistIterationsForPrune</td>
<td>Minimum number of iterations a species must exist before it can be pruned</td>
</tr>
<tr>
<td>filterReactions</td>
<td>Specify whether to filter reactions during model enlarging step</td>
</tr>
<tr>
<td>unimolecularThreshold</td>
<td>Array of flags indicating whether a species is above the unimolecular reaction threshold</td>
</tr>
<tr>
<td>bimolecularThreshold</td>
<td>Array of flags indicating whether two species are above the bimolecular reaction threshold</td>
</tr>
<tr>
<td>unimolecularReact</td>
<td>Array of flags indicating whether a species should react unimolecularly in the enlarge step</td>
</tr>
<tr>
<td>bimolecularReact</td>
<td>Array of flags indicating whether two species should react in the enlarge step</td>
</tr>
<tr>
<td>termination</td>
<td>A list of termination targets (i.e TerminationTime and TerminationConversion objects)</td>
</tr>
<tr>
<td>speciesConstraints</td>
<td>Dictates the maximum number of atoms, carbons, electrons, etc. generated by RMG</td>
</tr>
<tr>
<td>outputDirectory</td>
<td>The directory used to save output files</td>
</tr>
<tr>
<td>verbosity</td>
<td>The level of logging verbosity for console output</td>
</tr>
<tr>
<td>loadRestart</td>
<td>True if restarting a previous job, False otherwise</td>
</tr>
<tr>
<td>saveRestartPeriod</td>
<td>The time period to periodically save a restart file (Quantity), or None for never.</td>
</tr>
<tr>
<td>units</td>
<td>The unit system to use to save output files (currently must be ‘si’)</td>
</tr>
<tr>
<td>generateOutputHTML</td>
<td>True to draw pictures of the species and reactions, saving a visualized model in an output HTML file</td>
</tr>
<tr>
<td>generatePlots</td>
<td>True to generate plots of the job execution statistics after each iteration, False otherwise</td>
</tr>
<tr>
<td>verboseComments</td>
<td>True to keep the verbose comments for database estimates, False otherwise</td>
</tr>
<tr>
<td>saveEdgeSpecies</td>
<td>True to save chemkin and HTML files of the edge species, False otherwise</td>
</tr>
<tr>
<td>keepIrreversible</td>
<td>True to keep irreversibility of library reactions as is (‘&lt;=&gt;’ or ‘=&gt;’), False (default) to force all library reactions to be reversible</td>
</tr>
<tr>
<td>pressureDependence</td>
<td>Whether to process unimolecular (pressure-dependent) reaction networks</td>
</tr>
<tr>
<td>quantumMechanics</td>
<td>Whether to apply quantum mechanical calculations instead of group additivity to certain molecules</td>
</tr>
<tr>
<td>wallTime</td>
<td>The maximum amount of CPU time in the form DD:HH:MM:SS to expend on this job; used to stop gracefully so we can still get profiling information</td>
</tr>
<tr>
<td>initializationTime</td>
<td>The time at which the job was initiated, in seconds since the epoch (i.e. from time.time())</td>
</tr>
<tr>
<td>done</td>
<td>Whether the job has completed (there is nothing new to add)</td>
</tr>
</tbody>
</table>
checkInput()  
Check for a few common mistakes in the input file.

ccheckLibraries()  
Check unwanted use of libraries: Liquid phase libraries in Gas phase simulation. Loading a Liquid phase library obtained in another solvent than the one defined in the input file. Other checks can be added here.

clear()  
Clear all loaded information about the job (except the file paths).

cexecute(**kwargs)  
Execute an RMG job using the command-line arguments args as returned by the argparse package.

finish()  
Complete the model generation.

initialize(**kwargs)  
Initialize an RMG job using the command-line arguments args as returned by the argparse package.

loadInput(path=None)  
Load an RMG job from the input file located at inputFile, or from the inputFile attribute if not given as a parameter.

loadRMGJavaInput(path)  
Load an RMG-Java job from the input file located at inputFile, or from the inputFile attribute if not given as a parameter.

loadRestartFile(path)  
Load a restart file at path on disk.

loadThermoInput(path=None)  
Load an Thermo Estimation job from a thermo input file located at inputFile, or from the inputFile attribute if not given as a parameter.

logHeader(level=20)  
Output a header containing identifying information about RMG to the log.

readMeaningfulLineJava(f)  
Read a meaningful line from an RMG-Java condition file object f, returning the line with any comments removed.

register_listeners()  
Attaches listener classes depending on the options found in the RMG input file.

saveEverything()  
Saves the output HTML, the Chemkin file, and the Restart file (if appropriate).  
The restart file is only saved if self.saveRestartPeriod or self.done.

saveInput(path=None)  
Save an RMG job to the input file located at path, or from the outputFile attribute if not given as a parameter.

rmgpy.rmg.main.initializeLog(verbosity, log_file_name)  
Set up a logger for RMG to use to print output to stdout. The verbosity parameter is an integer specifying the amount of log text seen at the console; the levels correspond to those of the logging module.

rmgpy.rmg.main.makeProfileGraph(stats_file)  
Uses gprof2dot to create a graphviz dot file of the profiling information.  
This requires the gprof2dot package available via pip install gprof2dot. Render the result using the program ‘dot’ via a command like dot -Tps2 input.dot -o output.ps2.
Rendering the ps2 file to pdf requires an external pdf converter `ps2pdf output.ps2` which produces a `output.ps2.pdf` file.

```python
rmgpy.rmg.main.processProfileStats(stats_file, log_file)
```

## Saving RMG output

```python
rmgpy.rmg.output.saveOutputHTML(path, reactionModel, partCoreEdge='core')
```

Save the current set of species and reactions of `reactionModel` to an HTML file `path` on disk. As part of this process, drawings of all species are created in the species folder (if they don’t already exist) using the `rmgpy.molecule.draw` module. The `jinja` package is used to generate the HTML; if this package is not found, no HTML will be generated (but the program will carry on).

```python
rmgpy.rmg.output.saveDiffHTML(path, commonSpeciesList, speciesList1, speciesList2, commonReactions, uniqueReactions1, uniqueReactions2)
```

This function outputs the species and reactions on an HTML page for the comparison of two RMG models.

## rmgpy.rmg.pdep.PDepNetwork

```python
class rmgpy.rmg.pdep.PDepNetwork(index=-1, source=None)
```

A representation of a partial unimolecular reaction network. Each partial network has a single `source` isomer or reactant channel, and is responsible only for \( k(T, P) \) values for net reactions with source as the reactant. Multiple partial networks can have the same source, but networks with the same source and any explored isomers must be combined.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>list</td>
<td>The isomer or reactant channel that acts as the source</td>
</tr>
<tr>
<td>explored</td>
<td>list</td>
<td>A list of the unimolecular isomers whose reactions have been fully explored</td>
</tr>
</tbody>
</table>

```python
addPathReaction(newReaction)
```

Add a path reaction to the network. If the path reaction already exists, no action is taken.

```python
applyChemicallySignificantEigenvaluesMethod(lumpingOrder=None)
```

Compute the phenomenological rate coefficients \( k(T, P) \) at the current conditions using the chemically-significant eigenvalues method. If a `lumpingOrder` is provided, the algorithm will attempt to lump the configurations (given by index) in the order provided, and return a reduced set of \( k(T, P) \) values.

```python
applyModifiedStrongCollisionMethod(efficiencyModel='default')
```

Compute the phenomenological rate coefficients \( k(T, P) \) at the current conditions using the modified strong collision method.

```python
applyReservoirStateMethod()
```

Compute the phenomenological rate coefficients \( k(T, P) \) at the current conditions using the reservoir state method.

```python
calculateCollisionModel()
```

Calculate the matrix of first-order rate coefficients for collisional population transfer between grains for each isomer, including the corresponding collision frequencies.

```python
calculateDensitiesOfStates()
```

Calculate the densities of states of each configuration that has states data. The densities of states are computed such that they can be applied to each temperature in the range of interest by interpolation.

```python
calculateEquilibriumRatios()
```

Return an array containing the fraction of each isomer and reactant channel present at equilibrium, as determined from the Gibbs free energy and using the concentration equilibrium constant \( K_c \). These values are ratios, and the absolute magnitude is not guaranteed; however, the implementation scales the elements of the array so that they sum to unity.
calculateMicrocanonicalRates()
Calculate and return arrays containing the microcanonical rate coefficients $k(E)$ for the isomerization, dissociation, and association path reactions in the network.

cleanup()
Delete intermediate arrays used to compute $k(T,P)$ values.

exploreIsomer(isomer)
Explore a previously-unexplored unimolecular isomer in this partial network using the provided core-edge reaction model reactionModel, returning the new reactions and new species.

getAllSpecies()
Return a list of all unique species in the network, including all isomers, reactant and product channels, and bath gas species.

getLeakBranchingRatios($T, P$)
Return a dict with the unexplored isomers in the partial network as the keys and the fraction of the total leak coefficient as the values.

getLeakCoefficient($T, P$)
Return the pressure-dependent rate coefficient $k(T,P)$ describing the total rate of “leak” from this network. This is defined as the sum of the $k(T,P)$ values for all net reactions to unexplored unimolecular isomers.

getMaximumLeakSpecies($T, P$)
Get the unexplored (unimolecular) isomer with the maximum leak flux. Note that the leak rate coefficients vary with temperature and pressure, so you must provide these in order to get a meaningful result.

initialize($T_{min}$, $T_{max}$, $P_{min}$, $P_{max}$, maximumGrainSize=0.0, minimumGrainCount=0, activeJRotor=True, activeKRotor=True, rmgmode=False)
Initialize a pressure dependence calculation by computing several quantities that are independent of the conditions. You must specify the temperature and pressure ranges of interest using $T_{min}$ and $T_{max}$ in K and $P_{min}$ and $P_{max}$ in Pa. You must also specify the maximum energy grain size grainSize in J/mol and/or the minimum number of grains grainCount.

invalidate()
Mark the network as in need of a new calculation to determine the pressure-dependent rate coefficients

label
Get the label for this network (analogous to reaction families as a reaction’s source)

mapDensitiesOfStates()
Map the overall densities of states to the current energy grains. Semi-logarithmic interpolation will be used if the grain sizes of Elist0 and Elist do not match; this should not be a significant source of error as long as the grain sizes are sufficiently small.

merge(other)
Merge the partial network other into this network.

printSummary($level=20$)
Print a formatted list of information about the current network. Each molecular configuration - unimolecular isomers, bimolecular reactant channels, and bimolecular product channels - is given along with its energy on the potential energy surface. The path reactions connecting adjacent molecular configurations are also given, along with their energies on the potential energy surface. The level parameter controls the level of logging to which the summary is written, and is DEBUG by default.

selectEnergyGrains($T$, grainSize=0.0, grainCount=0)
Select a suitable list of energies to use for subsequent calculations. This is done by finding the minimum and maximum energies on the potential energy surface, then adding a multiple of $k_B T$ onto the maximum energy.
You must specify either the desired grain spacing \textit{grainSize} in J/mol or the desired number of grains \textit{Ngrains}, as well as a temperature \textit{T} in K to use for the equilibrium calculation. You can specify both \textit{grainSize} and \textit{grainCount}, in which case the one that gives the more accurate result will be used (i.e. they represent a maximum grain size and a minimum number of grains). An array containing the energy grains in J/mol is returned.

\textbf{setConditions} \((T, P, \text{ymB} = \text{None})\)

Set the current network conditions to the temperature \textit{T} in K and pressure \textit{P} in Pa. All of the internal variables are updated accordingly if they are out of date. For example, those variables that depend only on temperature will not be recomputed if the temperature is the same.

\textbf{solveFullME} \((tlist, x0)\)

Directly solve the full master equation using a stiff ODE solver. Pass the reaction network to solve, the temperature \textit{T} in K and pressure \textit{P} in Pa to solve at, the energies \textit{Elist} in J/mol to use, the output time points \textit{tlist} in s, the initial total populations \textit{x0}, the full master equation matrix \textit{M}, the accounting matrix \textit{indices} relating isomer and energy grain indices to indices of the master equation matrix, and the densities of states \textit{densStates} in mol/J of each isomer. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

\textbf{solveReducedME} \((tlist, x0)\)

Directly solve the reduced master equation using a stiff ODE solver. Pass the output time points \textit{tlist} in s and the initial total populations \textit{x0}. Be sure to run one of the methods for generating \(k(T, P)\) values before calling this method. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

\textbf{update} \((\text{reactionModel}, \text{pdepSettings})\)

Regenerate the \(k(T, P)\) values for this partial network if the network is marked as invalid.

\textbf{updateConfigurations} \((\text{reactionModel})\)

Sort the reactants and products of each of the network's path reactions into isomers, reactant channels, and product channels. You must pass the current \textit{reactionModel} because some decisions on sorting are made based on which species are in the model core.

\textbf{rmgpy.rmg.pdep.PDepReaction}

\textbf{class} \textbf{rmgpy.rmg.pdep.PDepReaction}(\textit{index=-1, label='', reactants=None, products=None, network=None, kinetics=None, reversible=True, transitionState=None, duplicate=False, degeneracy=1, pairs=None})

\textbf{calculateMicrocanonicalRateCoefficient}()

Calculate the microcanonical rate coefficient \(k(E)\) for the reaction \textit{reaction} at the energies \textit{Elist} in J/mol. \textit{reactDensStates} and \textit{prodDensStates} are the densities of states of the reactant and product configurations for this reaction. If the reaction is irreversible, only the reactant density of states is required; if the reaction is reversible, then both are required. This function will try to use the best method that it can based on the input data available:

- If detailed information has been provided for the transition state (i.e. the molecular degrees of freedom), then RRKM theory will be used.

- If the above is not possible but high-pressure limit kinetics \(k_\infty(T)\) have been provided, then the inverse Laplace transform method will be used.

The density of states for the product \textit{prodDensStates} and the temperature of interest \textit{T} in K can also be provided. For isomerization and association reactions \textit{prodDensStates} is required; for dissociation reactions it is optional. The temperature is used if provided in the detailed balance expression to determine the reverse kinetics, and in certain cases in the inverse Laplace transform method.
**calculateTSTRateCoefficient()**

Evaluate the forward rate coefficient for the reaction with corresponding transition state $TS$ at temperature $T$ in K using (canonical) transition state theory. The TST equation is

$$k(T) = \kappa(T) \frac{k_B T}{\hbar} \frac{Q^1(T)}{Q^A(T)Q^B(T)} \exp \left( - \frac{E_0}{k_B T} \right)$$

where $Q^1$ is the partition function of the transition state, $Q^A$ and $Q^B$ are the partition function of the reactants, $E_0$ is the ground-state energy difference from the transition state to the reactants, $T$ is the absolute temperature, $k_B$ is the Boltzmann constant, and $\hbar$ is the Planck constant. $\kappa(T)$ is an optional tunneling correction.

**canTST()**

Return True if the necessary parameters are available for using transition state theory – or the microcanonical equivalent, RRKM theory – to compute the rate coefficient for this reaction, or False otherwise.

**copy()**

Create a deep copy of the current reaction.

**draw()**

Generate a pictorial representation of the chemical reaction using the draw module. Use path to specify the file to save the generated image to; the image type is automatically determined by extension. Valid extensions are .png, .svg, .pdf, and .ps; of these, the first is a raster format and the remainder are vector formats.

**fixBarrierHeight()**

Turns the kinetics into Arrhenius (if they were ArrheniusEP) and ensures the activation energy is at least the endothermicity for endothermic reactions, and is not negative only as a result of using Evans Polanyi with an exothermic reaction. If forcePositive is True, then all reactions are forced to have a non-negative barrier.

**fixDiffusionLimitedA()**

Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

**generate3dTS()**

Generate the 3D structure of the transition state. Called from model.generateKinetics().

self.reactants is a list of reactants self.products is a list of products

**generatePairs()**

Generate the reactant-product pairs to use for this reaction when performing flux analysis. The exact procedure for doing so depends on the reaction type:

<table>
<thead>
<tr>
<th>Reaction type</th>
<th>Template</th>
<th>Resulting pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isomerization</td>
<td>A -&gt; C</td>
<td>(A,C)</td>
</tr>
<tr>
<td>Dissociation</td>
<td>A -&gt; C + D</td>
<td>(A,C), (A,D)</td>
</tr>
<tr>
<td>Association</td>
<td>A + B -&gt; C</td>
<td>(A,C), (B,C)</td>
</tr>
<tr>
<td>Bimolecular</td>
<td>A + B -&gt; C + D</td>
<td>(A,C), (B,D) or (A,D), (B,C)</td>
</tr>
</tbody>
</table>

There are a number of ways of determining the correct pairing for bimolecular reactions. Here we try a simple similarity analysis by comparing the number of heavy atoms (carbons and oxygens at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

**generateReverseRateCoefficient()**

Generate and return a rate coefficient model for the reverse reaction. Currently this only works if the kinetics attribute is one of several (but not necessarily all) kinetics types.

**getEnthalpiesOfReaction()**

Return the enthalpies of reaction in J/mol evaluated at temperatures Tlist in K.
getEnthalpyOfReaction()
Return the enthalpy of reaction in J/mol evaluated at temperature $T$ in K.

getEntropiesOfReaction()
Return the entropies of reaction in J/mol*K evaluated at temperatures $Tlist$ in K.

getEntropyOfReaction()
Return the entropy of reaction in J/mol*K evaluated at temperature $T$ in K.

getEquilibriumConstant()
Return the equilibrium constant for the reaction at the specified temperature $T$ in K. The type parameter lets you specify the quantities used in the equilibrium constant: $Ka$ for activities, $Kc$ for concentrations (default), or $Kp$ for pressures. Note that this function currently assumes an ideal gas mixture.

getEquilibriumConstants()
Return the equilibrium constants for the reaction at the specified temperatures $Tlist$ in K. The type parameter lets you specify the quantities used in the equilibrium constant: $Ka$ for activities, $Kc$ for concentrations (default), or $Kp$ for pressures. Note that this function currently assumes an ideal gas mixture.

getFreeEnergiesOfReaction()
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures $Tlist$ in K.

getFreeEnergyOfReaction()
Return the Gibbs free energy of reaction in J/mol evaluated at temperature $T$ in K.

getRateCoefficient()
Return the overall rate coefficient for the forward reaction at temperature $T$ in K and pressure $P$ in Pa, including any reaction path degeneracies.

If diffusionLimiter is enabled, the reaction is in the liquid phase and we use a diffusion limitation to correct the rate. If not, then use the intrinsic rate coefficient.

ggetSource()
Get the source of this PDepReaction

ggetStoichiometricCoefficient()
Return the stoichiometric coefficient of species $spec$ in the reaction. The stoichiometric coefficient is increased by one for each time $spec$ appears as a product and decreased by one for each time $spec$ appears as a reactant.

ggetURL()
Get a URL to search for this reaction in the rmg website.

hasTemplate()
Return True if the reaction matches the template of reactants and products, which are both lists of Species objects, or False if not.

isAssociation()
Return True if the reaction represents an association reaction $A + B \rightleftharpoons C$ or False if not.

isBalanced()
Return True if the reaction has the same number of each atom on each side of the reaction equation, or False if not.

isDissociation()
Return True if the reaction represents a dissociation reaction $A \rightleftharpoons B + C$ or False if not.

isIsomerization()
Return True if the reaction represents an isomerization reaction $A \rightleftharpoons B$ or False if not.
isIsomorphic()
Return True if this reaction is the same as the other reaction, or False if they are different. If eitherDirection=False then the directions must match.

isUnimolecular()
Return True if the reaction has a single molecule as either reactant or product (or both) A ⟷ B + C or A + B ⟷ C or A ⟷ B, or False if not.

matchesMolecules()
Return True if the given reactants represent the total set of reactants or products for the current reaction, or False if not. The reactants should be Molecule objects.

reverseThisArrheniusRate()
Reverses the given kForward, which must be an Arrhenius type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

toCantera()
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

toChemkin()
Return the chemkin-formatted string for this reaction.

If kinetics is set to True, the chemkin format kinetics will also be returned (requires the speciesList to figure out third body colliders.) Otherwise, only the reaction string will be returned.

rmgpy.rmg.model.Species

class rmgpy.rmg.model.Species(index=-1, label='', thermo=None, conformer=None, molecule=None, transportData=None, molecularWeight=None, energyTransferModel=None, reactive=True, props=None, coreSizeAtCreation=0)

calculateCp0()
Return the value of the heat capacity at zero temperature in J/mol*K.

calculateCpInf()
Return the value of the heat capacity at infinite temperature in J/mol*K.

copy()
Create a copy of the current species. If the kw argument ‘deep’ is True, then a deep copy will be made of the Molecule objects in self.molecule.

For other complex attributes, a deep copy will always be made.

fromAdjacencyList()
Load the structure of a species as a Molecule object from the given adjacency list adjlist and store it as the first entry of a list in the molecule attribute. Does not generate resonance isomers of the loaded molecule.

fromSMILES()
Load the structure of a species as a Molecule object from the given SMILES string smiles and store it as the first entry of a list in the molecule attribute. Does not generate resonance isomers of the loaded molecule.

generateEnergyTransferModel()
Generate the collisional energy transfer model parameters for the species. This “algorithm” is very much in need of improvement.

generateResonanceIsomers()
Generate all of the resonance isomers of this species. The isomers are stored as a list in the molecule attribute. If the length of molecule is already greater than one, it is assumed that all of the resonance isomers have already been generated.

1.11. Reaction mechanism generation (rmgpy.rmg)
generateStatMech()
Generate molecular degree of freedom data for the species. You must have already provided a thermodynamics model using e.g. generateThermoData().

generateTransportData()
Generate the transportData parameters for the species.

densityOfStates()
Return the density of states $\rho(E)\ dE$ at the specified energies $E_{list}$ in J/mol above the ground state.

getEnthalpy()
Return the enthalpy in J/mol for the species at the specified temperature $T$ in K.

getEntropy()
Return the entropy in J/mol*K for the species at the specified temperature $T$ in K.

getGibbsFreeEnergy()
Return the Gibbs free energy in J/mol for the species at the specified temperature $T$ in K.

getHeatCapacity()
Return the heat capacity in J/mol*K for the species at the specified temperature $T$ in K.

generatePartitionFunction()
Return the partition function for the species at the specified temperature $T$ in K.

getSumOfStates()
Return the sum of states $N(E)$ at the specified energies $E_{list}$ in J/mol.

getSymmetryNumber()
Get the symmetry number for the species, which is the highest symmetry number amongst its resonance isomers. This function is currently used for website purposes and testing only as it requires additional calculateSymmetryNumber calls.

generateThermoData()
Returns a thermoData object of the current Species object.

If the thermo object already exists, it is either of the (Wilhoit, ThermoData) type, or it is a Future.

If the type of the thermo attribute is Wilhoit, or ThermoData, then it is converted into a NASA format.

If it is a Future, then a blocking call is made to retrieve the NASA object. If the thermo object did not exist yet, the thermo object is generated.

generateTransportData()
Returns the transport data associated with this species, and calculates it if it is not yet available.

hasStatMech()
Return True if the species has statistical mechanical parameters, or False otherwise.

hasThermo()
Return True if the species has thermodynamic parameters, or False otherwise.

isIsomorphic()
Return True if the species is isomorphic to other, which can be either a Molecule object or a Species object.

toAdjacencyList()
Return a string containing each of the molecules’ adjacency lists.

toCantera()
Converts the RMG Species object to a Cantera Species object with the appropriate thermo data.

toChemkin()
Return the chemkin-formatted string for this species.
1.12 Reaction system simulation (rmgpy.solver)

The `rmgpy.solver` module contains classes used to represent and simulate reaction systems.

### 1.12.1 Reaction systems

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactionSystem</td>
<td>Base class for all reaction systems</td>
</tr>
<tr>
<td>SimpleReactor</td>
<td>A simple isothermal, isobaric, well-mixed batch reactor</td>
</tr>
</tbody>
</table>

### 1.12.2 Termination criteria

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerminationTime</td>
<td>Represent a time at which the simulation should be terminated</td>
</tr>
<tr>
<td>TerminationConversion</td>
<td>Represent a species conversion at which the simulation should be terminated</td>
</tr>
</tbody>
</table>

#### rmgpy.solver.ReactionSystem

**class rmgpy.solver.ReactionSystem**

A base class for all RMG reaction systems.

- **advance()**
  
  Simulate from the current value of the independent variable to a specified value \( t_{out} \), taking as many steps as necessary. The resulting values of \( t \), \( y \), and \( \frac{dy}{dt} \) can then be accessed via the \( t \), \( y \), and \( \frac{dy}{dt} \) attributes.

- **computeRateDerivative()**
  
  Returns derivative vector \( df/dk_j \) where \( dy/dt = f(y, t, k) \) and \( k_j \) is the rate parameter for the jth core reaction.

- **compute_network_variables()**
  
  Initialize the arrays containing network information:
  
  - **NetworkLeakCoefficients is a n x 1 array with**  
    - \( n \) the number of pressure-dependent networks.
  
  - **NetworkIndices is a n x 3 matrix with**  
    - \( n \) the number of pressure-dependent networks and 3 the maximum number of molecules allowed in either the reactant or product side of a reaction.

- **generate_reactant_product_indices()**
  
  Creates a matrix for the reactants and products.

- **generate_reaction_indices()**
  
  Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

- **generate_species_indices()**
  
  Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

- **get_species_index()**
  
  Retrieves the index that is associated with the parameter species from the species index dictionary.

- **initialize()**
  
  Initialize the DASPK solver by setting the initial values of the independent variable \( t0 \), dependent variables \( y0 \), and first derivatives \( dydt0 \). If provided, the derivatives must be consistent with the other initial conditions; if not provided, DASPK will attempt to estimate a consistent set of initial values for the derivatives. You can also set the absolute and relative tolerances \( atol \) and \( rtol \), respectively, either as single values for all dependent variables or individual values for each dependent variable.
initializeModel()
Initialize a simulation of the reaction system using the provided kinetic model. You will probably want to create your own version of this method in the derived class; don’t forget to also call the base class version, too.

initialize_tolerances()
Computes the number of differential equations and initializes the tolerance arrays.

logConversions()
Log information about the current conversion values.

logRates()
Log information about the current maximum species and network rates.

residual()
Evaluate the residual function for this model, given the current value of the independent variable \( t \), dependent variables \( y \), and first derivatives \( dy/dt \). Return a numpy array with the values of the residual function and an integer with status information (0 if okay, -2 to terminate).

set_initial_conditions()
Sets the common initial conditions of the rate equations that represent the reaction system.
- Sets the initial time of the reaction system to 0
- Initializes the species moles to a \( n \times 1 \) array with zeros

set_initial_derivative()
Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate()
Simulate the reaction system with the provided reaction model, consisting of lists of core species, core reactions, edge species, and edge reactions. As the simulation proceeds the system is monitored for validity. If the model becomes invalid (e.g. due to an excessively large edge flux), the simulation is interrupted and the object causing the model to be invalid is returned. If the simulation completes to the desired termination criteria and the model remains valid throughout, None is returned.

step()
Perform one simulation step from the current value of the independent variable toward (but not past) a specified value \( t_{out} \). The resulting values of \( t, y, \) and \( dy/dt \) can then be accessed via the \( t, y, \) and \( dy/dt \) attributes.

rmgpy.solver.SimpleReactor

class rmgpy.solver.SimpleReactor
A reaction system consisting of a homogeneous, isothermal, isobaric batch reactor. These assumptions allow for a number of optimizations that enable this solver to complete very rapidly, even for large kinetic models.

advance()
Simulate from the current value of the independent variable to a specified value \( t_{out} \), taking as many steps as necessary. The resulting values of \( t, y, \) and \( dy/dt \) can then be accessed via the \( t, y, \) and \( dy/dt \) attributes.

calculate_effective_pressure()
Computes the effective pressure for a reaction as: \( P_{eff} = P \times \sum(y_i \times eff_i / \sum(y)) \)

with:
- \( P \) the pressure of the reactor,
- \( y \) the array of initial moles of the core species
computeRateDerivative()
Returns derivative vector df/dk_j where dy/dt = f(y, t, k) and k_j is the rate parameter for the jth core reaction.

compute_network_variables()
Initialize the arrays containing network information:

- **NetworkLeakCoefficients** is a n x 1 array with n the number of pressure-dependent networks.
- **NetworkIndices** is a n x 3 matrix with n the number of pressure-dependent networks and 3 the maximum number of molecules allowed in either the reactant or product side of a reaction.

convertInitialKeysToSpeciesObjects()
Convert the initialMoleFractions dictionary from species names into species objects, using the given dictionary of species.

generate_rate_coefficients()
Populates the forward rate coefficients (kf), reverse rate coefficients (kb) and equilibrium constants (Keq) arrays with the values computed at the temperature and (effective) pressure of the reaction system.

generate_reactant_product_indices()
Creates a matrix for the reactants and products.

generate_reaction_indices()
Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices()
Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_species_index()
Retrieves the index that is associated with the parameter species from the species index dictionary.

initialize()
Initialize the DASPK solver by setting the initial values of the independent variable t0, dependent variables y0, and first derivatives dydt0. If provided, the derivatives must be consistent with the other initial conditions; if not provided, DASPK will attempt to estimate a consistent set of initial values for the derivatives. You can also set the absolute and relative tolerances atol and rtol, respectively, either as single values for all dependent variables or individual values for each dependent variable.

initializeModel()
Initialize a simulation of the simple reactor using the provided kinetic model.

initialize_tolerances()
Computes the number of differential equations and initializes the tolerance arrays.

jacobian()
Return the analytical Jacobian for the reaction system.

logConversions()
Log information about the current conversion values.

logRates()
Log information about the current maximum species and network rates.

residual()
Return the residual function for the governing DAE system for the simple reaction system.

set_colliders()
Store collider efficiencies and reaction indices for pdep reactions that have specific collider efficiencies.

set_initial_conditions()
Sets the initial conditions of the rate equations that represent the current reactor model.
The volume is set to the value derived from the ideal gas law, using the user-defined pressure, temperature, and the number of moles of initial species.

The species moles array (y0) is set to the values stored in the initial mole fractions dictionary.

The initial species concentration is computed and stored in the coreSpeciesConcentrations array.

**set_initial_derivative()**

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

**simulate()**

Simulate the reaction system with the provided reaction model, consisting of lists of core species, core reactions, edge species, and edge reactions. As the simulation proceeds the system is monitored for validity. If the model becomes invalid (e.g. due to an excessively large edge flux), the simulation is interrupted and the object causing the model to be invalid is returned. If the simulation completes to the desired termination criteria and the model remains valid throughout, `None` is returned.

**step()**

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value `tout`. The resulting values of t, y, and \( \frac{dy}{dt} \) can then be accessed via the t, y, and dydt attributes.

### Termination criteria

**class rmgpy.solver.TerminationTime**

Represent a time at which the simulation should be terminated. This class has one attribute: the termination time in seconds.

**class rmgpy.solver.TerminationConversion**

Represent a conversion at which the simulation should be terminated. This class has two attributes: the species to monitor and the fractional conversion at which to terminate.

1.13 Species (rmgpy.species)

The `rmgpy.species` subpackage contains classes and functions for working with chemical species.

1.13.1 Species

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Species</td>
<td>A chemical species</td>
</tr>
</tbody>
</table>

1.13.2 Transition state

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransitionState</td>
<td>A transition state</td>
</tr>
</tbody>
</table>

1.13.3 Exceptions

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeciesError</td>
<td>Raised when an error occurs while working with species</td>
</tr>
</tbody>
</table>
**rmgpy.species.Species**

**class rmgpy.species.Species**
A chemical species, representing a local minimum on a potential energy surface. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>A unique nonnegative integer index</td>
</tr>
<tr>
<td>label</td>
<td>A descriptive string label</td>
</tr>
<tr>
<td>thermo</td>
<td>The heat capacity model for the species</td>
</tr>
<tr>
<td>conformer</td>
<td>The molecular conformer for the species</td>
</tr>
<tr>
<td>molecule</td>
<td>A list of the Molecule objects describing the molecular structure</td>
</tr>
<tr>
<td>transportData</td>
<td>A set of transport collision parameters</td>
</tr>
<tr>
<td>molecularWeight</td>
<td>The molecular weight of the species</td>
</tr>
<tr>
<td>energyTransferModel</td>
<td>The collisional energy transfer model to use</td>
</tr>
<tr>
<td>reactive</td>
<td><code>True</code> if the species participates in reaction families, <code>False</code> if not</td>
</tr>
<tr>
<td></td>
<td>Reaction libraries and seed mechanisms that include the species are always considered regardless of this variable</td>
</tr>
<tr>
<td>props</td>
<td>A generic ‘properties’ dictionary to store user-defined flags</td>
</tr>
<tr>
<td>aug_inchi</td>
<td>Unique augmented inchi</td>
</tr>
</tbody>
</table>

*note: rmg.model.Species inherits from this class, and adds some extra methods.*

**calculateCp0()**
Return the value of the heat capacity at zero temperature in J/mol*K.

**calculateCpInf()**
Return the value of the heat capacity at infinite temperature in J/mol*K.

**copy()**
Create a copy of the current species. If the kw argument ‘deep’ is True, then a deep copy will be made of the Molecule objects in self.molecule.

For other complex attributes, a deep copy will always be made.

**fromAdjacencyList()**
Load the structure of a species as a Molecule object from the given adjacency list `adjlist` and store it as the first entry of a list in the `molecule` attribute. Does not generate resonance isomers of the loaded molecule.

**fromSMILES()**
Load the structure of a species as a Molecule object from the given SMILES string `smiles` and store it as the first entry of a list in the `molecule` attribute. Does not generate resonance isomers of the loaded molecule.

**generateEnergyTransferModel()**
Generate the collisional energy transfer model parameters for the species. This “algorithm” is very much in need of improvement.

**generateResonanceIsomers()**
Generate all of the resonance isomers of this species. The isomers are stored as a list in the `molecule` attribute. If the length of `molecule` is already greater than one, it is assumed that all of the resonance isomers have already been generated.

**generateStatMech()**
Generate molecular degree of freedom data for the species. You must have already provided a thermodynamics model using e.g. `generateThermoData()`.

`generateTransportData()`
Generate the transportData parameters for the species.

`getDensityOfStates()`
Return the density of states $\rho(E)\,dE$ at the specified energies $E_{\text{list}}$ in J/mol above the ground state.

`getEnthalpy()`
Return the enthalpy in J/mol for the species at the specified temperature $T$ in K.

`getEntropy()`
Return the entropy in J/mol*K for the species at the specified temperature $T$ in K.

`getFreeEnergy()`
Return the Gibbs free energy in J/mol for the species at the specified temperature $T$ in K.

`getHeatCapacity()`
Return the heat capacity in J/mol*K for the species at the specified temperature $T$ in K.

`getPartitionFunction()`
Return the partition function for the species at the specified temperature $T$ in K.

`getSumOfStates()`
Return the sum of states $N(E)$ at the specified energies $E_{\text{list}}$ in J/mol.

`getSymmetryNumber()`
Get the symmetry number for the species, which is the highest symmetry number amongst its resonance isomers. This function is currently used for website purposes and testing only as it requires additional `calculateSymmetryNumber` calls.

`getThermoData()`
Returns a `thermoData` object of the current Species object.

- If the thermo object already exists, it is either of the (Wilhoit, ThermoData) type, or it is a Future.
- If the type of the thermo attribute is Wilhoit, or ThermoData, then it is converted into a NASA format.
- If it is a Future, then a blocking call is made to retrieve the NASA object. If the thermo object did not exist yet, the thermo object is generated.

`getTransportData()`
Returns the transport data associated with this species, and calculates it if it is not yet available.

`hasStatMech()`
Return True if the species has statistical mechanical parameters, or False otherwise.

`hasThermo()`
Return True if the species has thermodynamic parameters, or False otherwise.

`isIsomorphic()`
Return True if the species is isomorphic to `other`, which can be either a `Molecule` object or a `Species` object.

`toAdjacencyList()`
Return a string containing each of the molecules’ adjacency lists.

`toCantera()`
Converts the RMG Species object to a Cantera Species object with the appropriate thermo data.

`toChemkin()`
Return the chemkin-formatted string for this species.
**rmgpy.species.TransitionState**

**class rmgpy.species.TransitionState**
A chemical transition state, representing a first-order saddle point on a potential energy surface. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>TDescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>A descriptive string label</td>
</tr>
<tr>
<td>conformer</td>
<td>The molecular degrees of freedom model for the species</td>
</tr>
<tr>
<td>frequency</td>
<td>The negative frequency of the first-order saddle point</td>
</tr>
<tr>
<td>tunneling</td>
<td>The type of tunneling model to use for tunneling through the reaction barrier</td>
</tr>
<tr>
<td>degeneracy</td>
<td>The reaction path degeneracy</td>
</tr>
</tbody>
</table>

**calculateTunnelingFactor()**
Calculate and return the value of the canonical tunneling correction factor for the reaction at the given temperature $T$ in K.

**calculateTunnelingFunction()**
Calculate and return the value of the microcanonical tunneling correction for the reaction at the given energies $E_{list}$ in J/mol.

**getDensityOfStates()**
Return the density of states $\rho(E)\,dE$ at the specified energies $E_{list}$ in J/mol above the ground state.

**getEnthalpy()**
Return the enthalpy in J/mol for the transition state at the specified temperature $T$ in K.

**getEntropy()**
Return the entropy in J/mol*K for the transition state at the specified temperature $T$ in K.

**getFreeEnergy()**
Return the Gibbs free energy in J/mol for the transition state at the specified temperature $T$ in K.

**getHeatCapacity()**
Return the heat capacity in J/mol*K for the transition state at the specified temperature $T$ in K.

**getPartitionFunction()**
Return the partition function for the transition state at the specified temperature $T$ in K.

**getSumOfStates()**
Return the sum of states $N(E)$ at the specified energies $E_{list}$ in J/mol.

### 1.14 Statistical mechanics (rmgpy.statmech)

The **rmgpy.statmech** subpackage contains classes that represent various statistical mechanical models of molecular degrees of freedom. These models enable the computation of macroscopic parameters (e.g. thermodynamics, kinetics, etc.) from microscopic parameters.

A molecular system consisting of $N$ atoms is described by $3N$ molecular degrees of freedom. Three of these modes involve translation of the system as a whole. Another three of these modes involve rotation of the system as a whole, unless the system is linear (e.g. diatomics), for which there are only two rotational modes. The remaining $3N - 6$ (or $3N - 5$ if linear) modes involve internal motion of the atoms within the system. Many of these modes are well-described as harmonic oscillations, while others are better modeled as torsional rotations around a bond within the system.

Molecular degrees of freedom are mathematically represented using the Schrodinger equation $\hat{H}\Psi = E\Psi$. By solving the Schrodinger equation, we can determine the available energy states of the molecular system, which enables com-
putation of macroscopic parameters. Depending on the temperature of interest, some modes (e.g. vibrations) require a quantum mechanical treatment, while others (e.g. translation, rotation) can be described using a classical solution.

### 1.14.1 Translational degrees of freedom

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdealGasTranslation</td>
<td>A model of three-dimensional translation of an ideal gas</td>
</tr>
</tbody>
</table>

### 1.14.2 Rotational degrees of freedom

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinearRotor</td>
<td>A model of two-dimensional rigid rotation of a linear molecule</td>
</tr>
<tr>
<td>NonlinearRotor</td>
<td>A model of three-dimensional rigid rotation of a nonlinear molecule</td>
</tr>
<tr>
<td>KRotor</td>
<td>A model of one-dimensional rigid rotation of a K-rotor</td>
</tr>
<tr>
<td>SphericalTopRotor</td>
<td>A model of three-dimensional rigid rotation of a spherical top molecule</td>
</tr>
</tbody>
</table>

### 1.14.3 Vibrational degrees of freedom

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HarmonicOscillator</td>
<td>A model of a set of one-dimensional harmonic oscillators</td>
</tr>
</tbody>
</table>

### 1.14.4 Torsional degrees of freedom

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HinderedRotor</td>
<td>A model of a one-dimensional hindered rotation</td>
</tr>
</tbody>
</table>

### 1.14.5 The Schrödinger equation

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPartitionFunction()</td>
<td>Calculate the partition function at a given temperature from energy levels and degeneracies</td>
</tr>
<tr>
<td>getHeatCapacity()</td>
<td>Calculate the dimensionless heat capacity at a given temperature from energy levels and degeneracies</td>
</tr>
<tr>
<td>getEnthalpy()</td>
<td>Calculate the enthalpy at a given temperature from energy levels and degeneracies</td>
</tr>
<tr>
<td>getEntropy()</td>
<td>Calculate the entropy at a given temperature from energy levels and degeneracies</td>
</tr>
<tr>
<td>getSumOfStates()</td>
<td>Calculate the sum of states for a given energy domain from energy levels and degeneracies</td>
</tr>
<tr>
<td>getDensityOfStates()</td>
<td>Calculate the density of states for a given energy domain from energy levels and degeneracies</td>
</tr>
</tbody>
</table>
1.14.6 Convolution

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convolve()</td>
<td>Return the convolution of two arrays</td>
</tr>
<tr>
<td>convolveBS()</td>
<td>Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart (BS) direct count algorithm</td>
</tr>
<tr>
<td>convolveBSSR()</td>
<td>Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart-Stein-Rabinovitch (BSSR) direct count algorithm</td>
</tr>
</tbody>
</table>

1.14.7 Molecular conformers

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformer</td>
<td>A model of a molecular conformation</td>
</tr>
</tbody>
</table>

Translational degrees of freedom

class rmgpy.statmech.IdealGasTranslation(mass=None, quantum=False)

A statistical mechanical model of translation in an 3-dimensional infinite square well by an ideal gas. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mass</td>
<td>The mass of the translating object</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

Translational energies are much smaller than \( k_B T \) except for temperatures approaching absolute zero, so a classical treatment of translation is more than adequate.

The translation of an ideal gas – a gas composed of randomly-moving, noninteracting particles of negligible size – in three dimensions can be modeled using the particle-in-a-box model. In this model, a gas particle is confined to a three-dimensional box of size \( L_x L_y L_z = V \) with the following potential:

\[
V(x,y,z) = \begin{cases} 
0 & 0 \leq x \leq L_x, 0 \leq y \leq L_y, 0 \leq z \leq L_z \\
\infty & \text{otherwise}
\end{cases}
\]

The time-independent Schrödinger equation for this system (within the box) is given by

\[
-\frac{\hbar^2}{2M} \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \Psi(x,y,z) = E \Psi(x,y,z)
\]

where \( M \) is the total mass of the particle. Because the box is finite in all dimensions, the solution of the above is quantized with the following energy levels:

\[
E_{n_x,n_y,n_z} = \frac{\hbar^2}{2M} \left[ \left( \frac{n_x \pi}{L_x} \right)^2 + \left( \frac{n_y \pi}{L_y} \right)^2 + \left( \frac{n_z \pi}{L_z} \right)^2 \right] 
\]

Above we have introduced \( n_x, n_y, \) and \( n_z \) as quantum numbers. The quantum mechanical partition function is obtained by summing over the above energy levels:

\[
Q_{\text{trans}}(T) = \sum_{n_x=1}^{\infty} \sum_{n_y=1}^{\infty} \sum_{n_z=1}^{\infty} \exp \left( -\frac{E_{n_x,n_y,n_z}}{k_B T} \right)
\]

In almost all cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the translational partition function in the classical limit:

\[
Q^{\text{cl}}_{\text{trans}}(T) = \left( \frac{2\pi M k_B T}{\hbar^2} \right)^{3/2} V
\]
For a constant-pressure problem we can use the ideal gas law to replace $V$ with $\frac{k_B T}{P}$. This gives the partition function a temperature dependence of $T^{5/2}$.

**getDensityOfStates**

```python
getDensityOfStates(self, ndarray Elist, ndarray densStates0=None) → ndarray
```

Return the density of states $\rho(E) \, dE$ at the specified energies $E$ in J/mol above the ground state. If an initial density of states $densStates0$ is given, the rotor density of states will be convoluted into these states.

**getEnthalpy**

```python
getEnthalpy(self, double T) → double
```

Return the enthalpy in J/mol for the degree of freedom at the specified temperature $T$ in K.

**getEntropy**

```python
getEntropy(self, double T) → double
```

Return the entropy in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

**getHeatCapacity**

```python
getHeatCapacity(self, double T) → double
```

Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

**getPartitionFunction**

```python
getPartitionFunction(self, double T) → double
```

Return the value of the partition function $Q(T)$ at the specified temperature $T$ in K.

**getSumOfStates**

```python
getSumOfStates(self, ndarray Elist, ndarray sumStates0=None) → ndarray
```

Return the sum of states $N(E)$ at the specified energies $E$ in J/mol above the ground state. If an initial sum of states $sumStates0$ is given, the rotor sum of states will be convoluted into these states.

**mass**

The mass of the translating object.

**quantum**

`quantum: ‘bool’`

**rmgpy.statmech.LinearRotor**

```python
class rmgpy.statmech.LinearRotor(inertia=None, symmetry=1, quantum=False, rotationalConstant=None)
```

A statistical mechanical model of a two-dimensional (linear) rigid rotor. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inertia</td>
<td>The moment of inertia of the rotor</td>
</tr>
<tr>
<td>rotationalConstant</td>
<td>The rotational constant of the rotor</td>
</tr>
<tr>
<td>symmetry</td>
<td>The symmetry number of the rotor</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default.

A linear rigid rotor is modeled as a pair of point masses $m_1$ and $m_2$ separated by a distance $R$. Since we are modeling the rotation of this system, we choose to work in spherical coordinates. Following the physics convention – where $0 \leq \theta \leq \pi$ is the zenith angle and $0 \leq \phi \leq 2\pi$ is the azimuth – the Schrodinger equation for the rotor is given by

$$
-\frac{\hbar^2}{2I} \left[ \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} \right] \Psi(\theta, \phi) = E \Psi(\theta, \phi)
$$

where $I \equiv \mu R^2$ is the moment of inertia of the rotating body, and $\mu \equiv m_1 m_2/(m_1 + m_2)$ is the reduced mass.

Note that there is no potential term in the above expression; for this reason, a rigid rotor is often referred to as a free rotor. Solving the Schrodinger equation gives the energy levels $E_J$ and corresponding degeneracies $g_J$ for
the linear rigid rotor as
\[ E_J = BJ(J + 1) \quad J = 0, 1, 2, \ldots \]
\[ g_J = 2J + 1 \]

where \( J \) is the quantum number for the rotor – sometimes called the total angular momentum quantum number – and \( B \equiv \hbar^2/2I \) is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition function for the linear rigid rotor:
\[ Q_{\text{rot}}(T) = \frac{1}{\sigma} \sum_{J=0}^{\infty} (2J + 1) e^{-BJ(J+1)/kB\ T} \]

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the linear rotor partition function in the classical limit:
\[ Q_{\text{rot}}^{\text{cl}}(T) = \frac{1}{\sigma} \frac{8\pi^2 I k_B T}{h^2} \]

Above we have also introduced \( \sigma \) as the symmetry number of the rigid rotor.

getDensityOfStates\((self, \text{ndarray Elist, ndarray densStates0=None}) \rightarrow \text{ndarray}\)
Return the density of states \( \rho(E)\ dE \) at the specified energies \( E\text{list} \) in J/mol above the ground state. If an initial density of states \( \text{densStates0} \) is given, the rotor density of states will be convoluted into these states.

getEnthalpy\((self, \text{double } T) \rightarrow \text{double}\)
Return the enthalpy in J/mol for the degree of freedom at the specified temperature \( T \) in K.

gEntrety\((self, \text{double } T) \rightarrow \text{double}\)
Return the entropy in J/mol*K for the degree of freedom at the specified temperature \( T \) in K.

gHeatCapacity\((self, \text{double } T) \rightarrow \text{double}\)
Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature \( T \) in K.

gLevelDegeneracy\((self, \text{int } J) \rightarrow \text{int}\)
Return the degeneracy of level \( J \).

gLevelEnergy\((self, \text{int } J) \rightarrow \text{double}\)
Return the energy of level \( J \) in kJ/mol.

gPartitionFunction\((self, \text{double } T) \rightarrow \text{double}\)
Return the value of the partition function \( Q(T) \) at the specified temperature \( T \) in K.

gSumOfStates\((self, \text{ndarray Elist, ndarray sumStates0=None}) \rightarrow \text{ndarray}\)
Return the sum of states \( N(E) \) at the specified energies \( E\text{list} \) in J/mol above the ground state. If an initial sum of states \( \text{sumStates0} \) is given, the rotor sum of states will be convoluted into these states.

inertia
The moment of inertia of the rotor.

quantum
quantum: ‘bool’

rotationalConstant
The rotational constant of the rotor.

symmetry
symmetry: ‘int’
rmgpy.statmech.NonlinearRotor

class rmgpy.statmech.NonlinearRotor(inertia=None, symmetry=1, quantum=False, rotationalConstant=None)
A statistical mechanical model of an N-dimensional nonlinear rigid rotor. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inertia</td>
<td>The moments of inertia of the rotor</td>
</tr>
<tr>
<td>rotationalConstant</td>
<td>The rotational constants of the rotor</td>
</tr>
<tr>
<td>symmetry</td>
<td>The symmetry number of the rotor</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

Note that the moments of inertia and the rotational constants are simply two ways of representing the same quantity; only one set of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default. In the current implementation, the quantum mechanical model has not been implemented, and a NotImplementedException will be raised if you try to use it.

A nonlinear rigid rotor is the generalization of the linear rotor to a nonlinear polyatomic system. Such a system is characterized by three moments of inertia $I_A$, $I_B$, and $I_C$ instead of just one. The solution to the Schrodinger equation for the quantum nonlinear rotor is not well defined, so we will simply show the classical result instead:

$$Q_{\text{rot}}^{cl}(T) = \frac{\pi^{1/2}}{\sigma} \left( \frac{8k_B T}{\hbar^2} \right)^{3/2} \sqrt{I_A I_B I_C}$$

getDensityOfStates(self, ndarray Elist, ndarray densStates0=None) → ndarray
Return the density of states $\rho(E)\,dE$ at the specified energies $E_{\text{list}}$ in J/mol above the ground state. If an initial density of states $\text{densStates}_0$ is given, the rotor density of states will be convoluted into these states.

genthalpy(self, double T) → double
Return the enthalpy in J/mol for the degree of freedom at the specified temperature $T$ in K.

genropy(self, double T) → double
Return the entropy in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

gheatCapacity(self, double T) → double
Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

gpartitionFunction(self, double T) → double
Return the value of the partition function $Q(T)$ at the specified temperature $T$ in K.

gsumOfStates(self, ndarray Elist, ndarray sumStates0=None) → ndarray
Return the sum of states $N(E)$ at the specified energies $E_{\text{list}}$ in J/mol above the ground state. If an initial sum of states $\text{sumStates}_0$ is given, the rotor sum of states will be convoluted into these states.

inertia
The moments of inertia of the rotor.

quantum
quantum: ‘bool’

rotationalConstant
The rotational constant of the rotor.

symmetry
symmetry: ‘int’
rmgpy.statmech.KRotor

class rmgpy.statmech.KRotor

A statistical mechanical model of an active K-rotor (a one-dimensional rigid rotor). The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inertia</td>
<td>The moment of inertia of the rotor in amu*angstrom^2</td>
</tr>
<tr>
<td>rotationalConstant</td>
<td>The rotational constant of the rotor in cm^-1</td>
</tr>
<tr>
<td>symmetry</td>
<td>The symmetry number of the rotor</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the K-rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default.

The energy levels $E_K$ of the K-rotor are given by

$$E_K = B K^2 \quad K = 0, \pm 1, \pm 2, \ldots$$

where $K$ is the quantum number for the rotor and $B \equiv \hbar^2 / 2I$ is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition function for the K-rotor:

$$Q_{rot}(T) = \frac{1}{\sigma} \left( 1 + \sum_{K=1}^{\infty} 2e^{-BK^2/k_B T} \right)$$

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the linear rotor partition function in the classical limit:

$$Q_{rot}^{cl}(T) = \frac{1}{\sigma} \left( \frac{8\pi^2 I k_B T}{\hbar^2} \right)^{1/2}$$

where $\sigma$ is the symmetry number of the K-rotor.

getDensityOfStates

Return the density of states $\rho(E) \ dE$ at the specified energies $Elist$ in J/mol above the ground state. If an initial density of states $densStates0$ is given, the rotor density of states will be convoluted into these states.

getEnthalpy

Return the enthalpy in J/mol for the degree of freedom at the specified temperature $T$ in K.

getEntropy

Return the entropy in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

getHeatCapacity

Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

getLevelDegeneracy

Return the degeneracy of level $J$.

getLevelEnergy

Return the energy of level $J$ in kJ/mol.

getPartitionFunction

Return the value of the partition function $Q(T)$ at the specified temperature $T$ in K.

getSumOfStates

Return the sum of states $N(E)$ at the specified energies $Elist$ in J/mol above the ground state. If an initial sum of states $sumStates0$ is given, the rotor sum of states will be convoluted into these states.
inertia
The moment of inertia of the rotor.

quantum
quantum: ‘bool’

rotationalConstant
The rotational constant of the rotor.

symmetry
symmetry: ‘int’

rmgpy.statmech.SphericalTopRotor
class rmgpy.statmech.SphericalTopRotor(inertia=None, symmetry=1, quantum=False, rotationalConstant=None)
A statistical mechanical model of a three-dimensional rigid rotor with a single rotational constant: a spherical
top. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inertia</td>
<td>The moment of inertia of the rotor</td>
</tr>
<tr>
<td>rotationalConstant</td>
<td>The rotational constant of the rotor</td>
</tr>
<tr>
<td>symmetry</td>
<td>The symmetry number of the rotor</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

Note that the moment of inertia and the rotational constant are simply two ways of representing the same
quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the
classical limit at all relevant temperatures; therefore, the classical model is used by default.

A spherical top rotor is simply the three-dimensional equivalent of a linear rigid rotor. Unlike the nonlinear
rotor, all three moments of inertia of a spherical top are equal, i.e. $I_A = I_B = I_C = I$. The energy levels $E_J$
and corresponding degeneracies $g_J$ of the spherical top rotor are given by

$$E_J = BJ(J + 1) \quad J = 0, 1, 2, \ldots$$

$$g_J = (2J + 1)^2$$

where $J$ is the quantum number for the rotor and $B \equiv \hbar^2 / 2I$ is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition
function for the spherical top rotor:

$$Q_{\text{rot}}(T) = \frac{1}{\sigma} \sum_{J=0}^{\infty} (2J + 1)^2 e^{-BJ(J+1)/k_BT}$$

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a
closed-form analytical expression for the linear rotor partition function in the classical limit:

$$Q_{\text{rot}}^c(T) = \frac{1}{\sigma} \left( \frac{8\pi^2 I k_B T}{\hbar^2} \right)^{3/2}$$

where $\sigma$ is the symmetry number of the spherical top. Note that the above differs from the nonlinear rotor
partition function by a factor of $\pi$.

getDensityOfStates(self, ndarray Elist, ndarray densStates0=None) → ndarray
Return the density of states $\rho(E) \, dE$ at the specified energies $Elist$ in J/mol above the ground state. If an
initial density of states $densStates0$ is given, the rotor density of states will be convoluted into these states.
getEnthalpy(self, double T) \rightarrow double
Return the enthalpy in J/mol for the degree of freedom at the specified temperature T in K.

getEntropy(self, double T) \rightarrow double
Return the entropy in J/mol*K for the degree of freedom at the specified temperature T in K.

getHeatCapacity(self, double T) \rightarrow double
Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature T in K.

getLevelDegeneracy(self, int J) \rightarrow int
Return the degeneracy of level J.

getLevelEnergy(self, int J) \rightarrow double
Return the energy of level J in kJ/mol.

getPartitionFunction(self, double T) \rightarrow double
Return the value of the partition function Q(T) at the specified temperature T in K.

getSumOfStates(self, ndarray Elist, ndarray sumStates0=None) \rightarrow ndarray
Return the sum of states N(E) at the specified energies Elist in J/mol above the ground state. If an initial sum of states sumStates0 is given, the rotor sum of states will be convoluted into these states.

inertia
The moment of inertia of the rotor.

quantum
quantum: ‘bool’

rotationalConstant
The rotational constant of the rotor.

symmetry
symmetry: ‘int’

rmgpy.statmech.HarmonicOscillator

class rmgpy.statmech.HarmonicOscillator(frequencies=None, quantum=True)
A statistical mechanical model of a set of one-dimensional independent harmonic oscillators. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>The vibrational frequencies of the oscillators</td>
</tr>
<tr>
<td>quantum</td>
<td>True to use the quantum mechanical model, False to use the classical model</td>
</tr>
</tbody>
</table>

In the majority of chemical applications, the energy levels of the harmonic oscillator are of similar magnitude to \( k_B T \), requiring a quantum mechanical treatment. Fortunately, the harmonic oscillator has an analytical quantum mechanical solution.

Many vibrational motions are well-described as one-dimensional quantum harmonic oscillators. The time-independent Schrödinger equation for such an oscillator is given by

\[
-\frac{\hbar^2}{2m} \frac{\partial^2 \Psi(x)}{\partial x^2} + \frac{1}{2} m \omega^2 x^2 \Psi(x) = E \Psi(x)
\]

where \( m \) is the total mass of the particle. The harmonic potential results in quantized solutions to the above with the following energy levels:

\[
E_n = \left( n + \frac{1}{2} \right) \hbar \omega \quad n = 0, 1, 2, \ldots
\]

Above we have introduced \( n \) as the quantum number. Note that, even in the ground state \( (n = 0) \), the harmonic oscillator has an energy that is not zero; this energy is called the zero-point energy.
The harmonic oscillator partition function is obtained by summing over the above energy levels:

\[ Q_{vib}(T) = \sum_{n=0}^{\infty} \exp \left( -\frac{(n + \frac{1}{2}) \hbar \omega}{k_B T} \right) \]

This summation can be evaluated explicitly to give a closed-form analytical expression for the vibrational partition function of a quantum harmonic oscillator:

\[ Q_{vib}(T) = \frac{e^{-\hbar \omega / 2k_B T}}{1 - e^{-\hbar \omega / k_B T}} \]

In RMG the convention is to place the zero-point energy in with the ground-state energy of the system instead of the numerator of the vibrational partition function, which gives

\[ Q_{vib}(T) = \frac{1}{1 - e^{-\hbar \omega / k_B T}} \]

The energy levels of the harmonic oscillator in chemical systems are often significant compared to the temperature of interest, so we usually use the quantum result. However, the classical limit is provided here for completeness:

\[ Q_{vib}^{cl}(T) = \frac{k_B T}{\hbar \omega} \]

**frequencies**

The vibrational frequencies of the oscillators.

**getDensityOfStates**

\[ \text{getDensityOfStates(self, ndarray Elist, ndarray densStates0=None)} \rightarrow \text{ndarray} \]

Return the density of states \( \rho(E) \, dE \) at the specified energies \( E \) in J/mol above the ground state. If an initial density of states \( \text{densStates0} \) is given, the rotor density of states will be convoluted into these states.

**getEnthalpy**

\[ \text{getEnthalpy(self, double T)} \rightarrow \text{double} \]

Return the enthalpy in J/mol for the degree of freedom at the specified temperature \( T \) in K.

**getEntropy**

\[ \text{getEntropy(self, double T)} \rightarrow \text{double} \]

Return the entropy in J/mol*K for the degree of freedom at the specified temperature \( T \) in K.

**getHeatCapacity**

\[ \text{getHeatCapacity(self, double T)} \rightarrow \text{double} \]

Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature \( T \) in K.

**getPartitionFunction**

\[ \text{getPartitionFunction(self, double T)} \rightarrow \text{double} \]

Return the value of the partition function \( Q(T) \) at the specified temperature \( T \) in K.

**getSumOfStates**

\[ \text{getSumOfStates(self, ndarray Elist, ndarray sumStates0=None)} \rightarrow \text{ndarray} \]

Return the sum of states \( N(E) \) at the specified energies \( E \) in J/mol above the ground state. If an initial sum of states \( \text{sumStates0} \) is given, the rotor sum of states will be convoluted into these states.

**quantum**

\[ \text{quantum: 'bool'} \]

**Torsional degrees of freedom**

\[ \text{class rmgpy.statmech.HinderedRotor(inertia=None, symmetry=1, barrier=None, fourier=None, rotationalConstant=None, quantum=False, semiclassical=True)} \]

A statistical mechanical model of a one-dimensional hindered rotor. The attributes are:
Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

The Schrödinger equation for a one-dimensional hindered rotor is given by

\[-\frac{\hbar^2}{2I} \frac{d^2}{d\phi^2} \Psi(\phi) + V(\phi) \Psi(\phi) = E \Psi(\phi)\]

where \(I\) is the reduced moment of inertia of the torsion and \(V(\phi)\) describes the potential of the torsion. There are two common forms for the potential: a simple cosine of the form

\[V(\phi) = \frac{1}{2} V_0 (1 - \cos \sigma \phi)\]

where \(V_0\) is the barrier height and \(\sigma\) is the symmetry number, or a more general Fourier series of the form

\[V(\phi) = A + \sum_{k=1}^{C} (a_k \cos k\phi + b_k \sin k\phi)\]

where \(A, a_k\) and \(b_k\) are fitted coefficients. Both potentials are typically defined such that the minimum of the potential is zero and is found at \(\phi = 0\).

For either the cosine or Fourier series potentials, the energy levels of the quantum hindered rotor must be determined numerically. The cosine potential does permit a closed-form representation of the classical partition function, however:

\[Q_{\text{cl}}^{\text{hind}}(T) = \left(\frac{2\pi I k_B T}{\hbar^2}\right)^{1/2} \frac{2\pi}{\sigma} \exp\left(-\frac{V_0}{2k_B T}\right) I_0\left(\frac{V_0}{2k_B T}\right)\]

A semiclassical correction to the above is usually required to provide a reasonable estimate of the partition function:

\[Q_{\text{semi}}^{\text{hind}}(T) = \frac{Q_{\text{quant}}^{\text{vib}}(T)}{Q_{\text{vib}}^{\text{cl}}(T)} \frac{Q_{\text{cl}}^{\text{vib}}(T)}{Q_{\text{vib}}^{\text{cl}}(T)} Q_{\text{cl}}^{\text{hind}}(T)\]

\[Q_{\text{semi}}^{\text{hind}}(T) = \frac{h \nu}{k_B T} \frac{1}{1 - \exp\left(-\frac{h \nu}{k_B T}\right)} \left(\frac{2\pi I k_B T}{\hbar^2}\right)^{1/2} \frac{2\pi}{\sigma} \exp\left(-\frac{V_0}{2k_B T}\right) I_0\left(\frac{V_0}{2k_B T}\right)\]

Above we have defined \(\nu\) as the vibrational frequency of the hindered rotor:

\[\nu \equiv \frac{\sigma}{2\pi} \sqrt{\frac{V_0}{2I}}\]

**barrier**

The barrier height of the cosine potential.

**energies**

energies: numpy.ndarray
fitCosinePotentialToData(self, ndarray angle, ndarray V)
Fit the given angles in radians and corresponding potential energies in J/mol to the cosine potential. For best results, the angle should begin at zero and end at $2\pi$, with the minimum energy conformation having a potential of zero be placed at zero angle. The fit is attempted at several possible values of the symmetry number in order to determine which one is correct.

fitFourierPotentialToData(self, ndarray angle, ndarray V)
Fit the given angles in radians and corresponding potential energies in J/mol to the Fourier series potential. For best results, the angle should begin at zero and end at $2\pi$, with the minimum energy conformation having a potential of zero be placed at zero angle.

fourier
The $2\times N$ array of Fourier series coefficients.

frequency
frequency: ‘double’

def getDensityOfStates(self, ndarray Elist, ndarray densStates0=None) → ndarray
Return the density of states $\rho(E)\,dE$ at the specified energies $E\text{list}$ in J/mol above the ground state. If an initial density of states $dens\text{States0}$ is given, the rotor density of states will be convoluted into these states.

def getEnthalpy(self, double T) → double
Return the enthalpy in J/mol for the degree of freedom at the specified temperature $T$ in K.

def getEntropy(self, double T) → double
Return the entropy in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

def getFrequency(self) → double
Return the frequency of vibration in cm^{-1} corresponding to the limit of harmonic oscillation.

def getHamiltonian(self, int Nbasis) → ndarray
Return the to the Hamiltonian matrix for the hindered rotor for the given number of basis functions $Nbasis$. The Hamiltonian matrix is returned in banded lower triangular form and with units of J/mol.

def getHeatCapacity(self, double T) → double
Return the heat capacity in J/mol*K for the degree of freedom at the specified temperature $T$ in K.

def getLevelDegeneracy(self, int J) → int
Return the degeneracy of level $J$.

def getLevelEnergy(self, int J) → double
Return the energy of level $J$ in J.

def getPartitionFunction(self, double T) → double
Return the value of the partition function $Q(T)$ at the specified temperature $T$ in K.

def getPotential(self, double phi) → double
Return the value of the hindered rotor potential $V(\phi)$ in J/mol at the angle $phi$ in radians.

def getSumOfStates(self, ndarray Elist, ndarray sumStates0=None) → ndarray
Return the sum of states $N(E)$ at the specified energies $E\text{list}$ in J/mol above the ground state. If an initial sum of states $sum\text{States0}$ is given, the rotor sum of states will be convoluted into these states.

inertia
The moment of inertia of the rotor.

quantum
quantum: ‘bool’

rotationalConstant
The rotational constant of the rotor.
**semiclassical**

`semiclassical: 'bool'

**solveSchrodingerEquation** *(self, int Nbasis=401) → ndarray*

Solves the one-dimensional time-independent Schrödinger equation to determine the energy levels of a one-dimensional hindered rotor with a Fourier series potential using `Nbasis` basis functions. For the purposes of this function it is usually sufficient to use 401 basis functions (the default). Returns the energy eigenvalues of the Hamiltonian matrix in J/mol.

**symmetry**

`symmetry: 'int'

**rmgpy.statmech.schrodinger**

The `rmgpy.statmech.schrodinger` module contains functionality for working with the Schrödinger equation and its solution. In particular, it contains functions for using the energy levels and corresponding degeneracies obtained from solving the Schrödinger equation to compute various thermodynamic and statistical mechanical properties, such as heat capacity, enthalpy, entropy, partition function, and the sum and density of states.

**rmgpy.statmech.schrodinger.convolve** *(ndarray rho1, ndarray rho2)*

Return the convolution of two arrays `rho1` and `rho2`.

**rmgpy.statmech.schrodinger.convolveBS** *(ndarray Elist, ndarray rho0, double energy, int degeneracy=1)*

Convolve a molecular degree of freedom into a density or sum of states using the Beyer-Swinehart (BS) direct count algorithm. This algorithm is suitable for unevenly-spaced energy levels in the array of energy grains `Elist` (in J/mol), but assumes the solution of the Schrödinger equation gives evenly-spaced energy levels with spacing `energy` in kJ/mol and degeneracy `degeneracy`.

**rmgpy.statmech.schrodinger.convolveBSSR** *(ndarray Elist, ndarray rho0, energy, degeneracy=unitDegeneracy, int n0=0)*

Convolve a molecular degree of freedom into a density or sum of states using the Beyer-Swinehart-Stein-Rabinovitch (BSSR) direct count algorithm. This algorithm is suitable for unevenly-spaced energy levels in both the array of energy grains `Elist` (in J/mol) and the energy levels corresponding to the solution of the Schrödinger equation.

**rmgpy.statmech.schrodinger.getDensityOfStates** *(ndarray Elist, energy, degeneracy=unitDegeneracy, int n0=0, ndarray densStates0=None) → ndarray*

Return the values of the dimensionless density of states $\rho(E)\,dE$ for a given set of energies `Elist` in J/mol above the ground state using an initial density of states `densStates0`. The solution to the Schrödinger equation is given using functions `energy` and `degeneracy` that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at `n0` and increases by ones.

**rmgpy.statmech.schrodinger.getEnthalpy** *(double T, energy, degeneracy=unitDegeneracy, int n0=0, int nmax=10000, double tol=1e-12) → double*

Return the value of the dimensionless enthalpy $H(T)/RT$ at a given temperature T in K. The solution to the Schrödinger equation is given using functions `energy` and `degeneracy` that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at `n0` and increases by ones. You can also change the relative tolerance `tol` and the maximum allowed value of the quantum number `nmax`.

**rmgpy.statmech.schrodinger.getEntropy** *(double T, energy, degeneracy=unitDegeneracy, int n0=0, int nmax=10000, double tol=1e-12) → double*

Return the value of the dimensionless entropy $S(T)/R$ at a given temperature T in K. The solution to the Schrödinger equation is given using functions `energy` and `degeneracy` that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at `n0` and increases by ones. You can also change the relative tolerance `tol` and the maximum allowed value of the quantum number `nmax`.

1.14. Statistical mechanics *(rmgpy.statmech)* 175
at \( n_0 \) and increases by ones. You can also change the relative tolerance \( tol \) and the maximum allowed value of the quantum number \( n_{\text{max}} \).

\[
\text{rmgpy.statmech.schrodinger.getHeatCapacity(double } T, \text{ energy, degeneracy=unitDegeneracy, int } n_0=0, \text{ int } n_{\text{max}}=10000, \text{ double } tol=1e-12) \rightarrow \text{double}
\]

Return the value of the dimensionless heat capacity \( C_v(T)/R \) at a given temperature \( T \) in K. The solution to the Schrodinger equation is given using functions \( \text{energy} \) and \( \text{degeneracy} \) that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at \( n_0 \) and increases by ones. You can also change the relative tolerance \( tol \) and the maximum allowed value of the quantum number \( n_{\text{max}} \).

\[
\text{rmgpy.statmech.schrodinger.getPartitionFunction(double } T, \text{ energy, degeneracy=unitDegeneracy, int } n_0=0, \text{ int } n_{\text{max}}=10000, \text{ double } tol=1e-12) \rightarrow \text{double}
\]

Return the value of the partition function \( Q(T) \) at a given temperature \( T \) in K. The solution to the Schrodinger equation is given using functions \( \text{energy} \) and \( \text{degeneracy} \) that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at \( n_0 \) and increases by ones. You can also change the relative tolerance \( tol \) and the maximum allowed value of the quantum number \( n_{\text{max}} \).

\[
\text{rmgpy.statmech.schrodinger.getSumOfStates(ndarray } E\text{list, energy, degeneracy=unitDegeneracy, int } n_0=0, \text{ ndarray } \text{sumStates0=0=None) } \rightarrow \text{ndarray}
\]

Return the values of the sum of states \( N(E) \) for a given set of energies \( E\text{list} \) in J/mol above the ground state using an initial sum of states \( \text{sumStates0} \). The solution to the Schrodinger equation is given using functions \( \text{energy} \) and \( \text{degeneracy} \) that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at \( n_0 \) and increases by ones.

\[
\text{rmgpy.statmech.schrodinger.unitDegeneracy}(n)
\]

\text{rmgpy.statmech.Conformer}

\text{class} \text{rmgpy.statmech.Conformer}(E0=None, modes=None, spinMultiplicity=1, opticalIsomers=1, number=None, mass=None, coordinates=None)

A representation of an individual molecular conformation. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_0 )</td>
<td>The ground-state energy (including zero-point energy) of the conformation</td>
</tr>
<tr>
<td>( \text{modes} )</td>
<td>A list of the molecular degrees of freedom</td>
</tr>
<tr>
<td>( \text{spinMultiplicity} )</td>
<td>The degeneracy of the electronic ground state</td>
</tr>
<tr>
<td>( \text{opticalIsomers} )</td>
<td>The number of optical isomers</td>
</tr>
<tr>
<td>( \text{number} )</td>
<td>An array of atomic numbers of each atom in the conformer</td>
</tr>
<tr>
<td>( \text{mass} )</td>
<td>An array of masses of each atom in the conformer</td>
</tr>
<tr>
<td>( \text{coordinates} )</td>
<td>An array of 3D coordinates of each atom in the conformer</td>
</tr>
</tbody>
</table>

Note that the \( \text{spinMultiplicity} \) reflects the electronic mode of the molecular system.

\( E_0 \)

The ground-state energy (including zero-point energy) of the conformation.

\text{coordinates}

An array of 3D coordinates of each atom in the conformation.

\text{getActiveModes(self, bool activeJRotor=False, bool activeKRotor=True) } \rightarrow \text{list}

Return a list of the active molecular degrees of freedom of the molecular system.

\text{getCenterOfMass(self, atoms=None) } \rightarrow \text{ndarray}

Calculate and return the [three-dimensional] position of the center of mass of the conformation in m. If a list
atoms of atoms is specified, only those atoms will be used to calculate the center of mass. Otherwise, all atoms will be used.

**getDensityOfStates**(*self*, *ndarray Elist*) → *ndarray*
Return the density of states \( \rho(E) \, dE \) at the specified energies *Elist* above the ground state.

**getEnthalpy**(*self*, *double T*) → *double*
Return the enthalpy in J/mol for the system at the specified temperature *T* in K.

**getEntropy**(*self*, *double T*) → *double*
Return the entropy in J/mol*K for the system at the specified temperature *T* in K.

**getFreeEnergy**(*self*, *double T*) → *double*
Return the Gibbs free energy in J/mol for the system at the specified temperature *T* in K.

**getHeatCapacity**(*self*, *double T*) → *double*
Return the heat capacity in J/mol*K for the system at the specified temperature *T* in K.

**getInternalReducedMomentOfInertia**(*self*, *pivots*, *top1*) → *double*
Calculate and return the reduced moment of inertia for an internal torsional rotation around the axis defined by the two atoms in *pivots*. The list *top1* contains the atoms that should be considered as part of the rotating top; this list should contain the pivot atom connecting the top to the rest of the molecule. The procedure used is that of Pitzer \(^1\), which is described as \( I^{(2,3)} \) by East and Radom \(^2\). In this procedure, the molecule is divided into two tops: those at either end of the hindered rotor bond. The moment of inertia of each top is evaluated using an axis passing through the center of mass of both tops. Finally, the reduced moment of inertia is evaluated from the moment of inertia of each top via the formula

\[
\frac{1}{I^{(2,3)}} = \frac{1}{I_1} + \frac{1}{I_2}
\]

**getMomentOfInertiaTensor**(*self*) → *ndarray*
Calculate and return the moment of inertia tensor for the conformer in kg*m\(^2\). If the coordinates are not at the center of mass, they are temporarily shifted there for the purposes of this calculation.

**getNumberDegreesOfFreedom**(*self*)
Return the number of degrees of freedom in a species object, which should be 3N, and raises an exception if it is not.

**getPartitionFunction**(*self*, *double T*) → *double*
Return the partition function \( Q(T) \) for the system at the specified temperature *T* in K.

**getPrincipalMomentsOfInertia**(*self*)
Calculate and return the principal moments of inertia and corresponding principal axes for the conformer. The moments of inertia are in kg*m\(^2\), while the principal axes have unit length.

**getSumOfStates**(*self*, *ndarray Elist*) → *ndarray*
Return the sum of states \( N(E) \) at the specified energies *Elist* in kJ/mol above the ground state.

**getSymmetricTopRotors**(*self*)
Return objects representing the external J-rotor and K-rotor under the symmetric top approximation. For nonlinear molecules, the J-rotor is a 2D rigid rotor with a rotational constant \( B \) determined as the geometric mean of the two most similar rotational constants. The K-rotor is a 1D rigid rotor with a rotational constant \( A - B \) determined by the difference between the remaining molecular rotational constant and the J-rotor rotational constant.

**getTotalMass**(*self*, *atoms=None*) → *double*
Calculate and return the total mass of the atoms in the conformer in kg. If a list *atoms* of atoms is specified, only those atoms will be used to calculate the center of mass. Otherwise, all atoms will be used.

---


mass
An array of masses of each atom in the conformer.

modes
modes: list

number
An array of atomic numbers of each atom in the conformer.

opticalIsomers
opticalIsomers: ‘int’

spinMultiplicity
spinMultiplicity: ‘int’

1.15 Thermodynamics (rmgpy.thermo)

The rmgpy.thermo subpackage contains classes that represent various thermodynamic models of heat capacity.

1.15.1 Heat capacity models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThermoData</td>
<td>A heat capacity model based on a set of discrete heat capacity points</td>
</tr>
<tr>
<td>Wilhoit</td>
<td>A heat capacity model based on the Wilhoit polynomial</td>
</tr>
<tr>
<td>NASA</td>
<td>A heat capacity model based on a set of NASA polynomials</td>
</tr>
<tr>
<td>NASAPolynomial</td>
<td>A heat capacity model based on a single NASA polynomial</td>
</tr>
</tbody>
</table>

rmgpy.thermo.ThermoData

class rmgpy.thermo.ThermoData(Tdata=None, Cpdata=None, H298=None, S298=None, Cp0=None, CpInf=None, Tmin=None, Tmax=None, E0=None, comment='')

A heat capacity model based on a set of discrete heat capacity data points. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tdata</td>
<td>An array of temperatures at which the heat capacity is known</td>
</tr>
<tr>
<td>Cpdata</td>
<td>An array of heat capacities at the given temperatures</td>
</tr>
<tr>
<td>H298</td>
<td>The standard enthalpy of formation at 298 K</td>
</tr>
<tr>
<td>S298</td>
<td>The standard entropy at 298 K</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>E0</td>
<td>The energy at zero Kelvin (including zero point energy)</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

Cp0
The heat capacity at zero temperature.

CpInf
The heat capacity at infinite temperature.

Cpdata
An array of heat capacities at the given temperatures.

E0
The ground state energy (J/mol) at zero Kelvin, including zero point energy, or None if not yet specified.
H298
The standard enthalpy of formation at 298 K.

S298
The standard entropy of formation at 298 K.

Tdata
An array of temperatures at which the heat capacity is known.

Tmax
The maximum temperature at which the model is valid, or None if not defined.

Tmin
The minimum temperature at which the model is valid, or None if not defined.

comment

comment: str

discrepancy(self, HeatCapacityModel other) → double

Return some measure of how dissimilar self is from other.

The measure is arbitrary, but hopefully useful for sorting purposes. Discrepancy of 0 means they are identical.

denthalpy(self, double T) → double

Return the enthalpy in J/mol at the specified temperature T in K.

getEntropy(self, double T) → double

Return the entropy in J/mol*K at the specified temperature T in K.

getFreeEnergy(self, double T) → double

Return the Gibbs free energy in J/mol at the specified temperature T in K.

getHeatCapacity(self, double T) → double

Return the constant-pressure heat capacity in J/mol*K at the specified temperature T in K.

isIdenticalTo(self, HeatCapacityModel other) → bool

Returns True if self and other report very similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isSimilarTo(self, HeatCapacityModel other) → bool

Returns True if self and other report similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isTemperatureValid(self, double T) → bool

Return True if the temperature T in K is within the valid temperature range of the thermodynamic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

toNASA(self, double Tmin, double Tmax, double Tint, bool fixedTint=False, bool weighting=True, int continuity=3) → NASA

Convert the object to a NASA object. You must specify the minimum and maximum temperatures of the fit Tmin and Tmax in K, as well as the intermediate temperature Tint in K to use as the bridge between the two fitted polynomials. The remaining parameters can be used to modify the fitting algorithm used:

• fixedTint - False to allow Tint to vary in order to improve the fit, or True to keep it fixed

• weighting - True to weight the fit by \(T^{-1}\) to emphasize good fit at lower temperatures, or False to not use weighting

• continuity - The number of continuity constraints to enforce at Tint:

  - 0: no constraints on continuity of \(C_p(T)\) at Tint
  - 1: constrain \(C_p(T)\) to be continuous at Tint
-2: constrain $C_p(T)$ and $\frac{dC_p}{dT}$ to be continuous at $T_{int}$

-3: constrain $C_p(T)$, $\frac{dC_p}{dT}$, and $\frac{d^2C_p}{dT^2}$ to be continuous at $T_{int}$

-4: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, and $\frac{d^3C_p}{dT^3}$ to be continuous at $T_{int}$

-5: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, $\frac{d^3C_p}{dT^3}$, and $\frac{d^4C_p}{dT^4}$ to be continuous at $T_{int}$

Note that values of continuity of 5 or higher effectively constrain all the coefficients to be equal and should be equivalent to fitting only one polynomial (rather than two).

Returns the fitted NASA object containing the two fitted NASAPolynomial objects.

```python
toWilhoit(self, B=None) \rightarrow Wilhoit
```

Convert the Benson model to a Wilhoit model. For the conversion to succeed, you must have set the $Cp0$ and $CpInf$ attributes of the Benson model.

- B: the characteristic temperature in Kelvin.

### rmgpy.thermo.Wilhoit

```python
class rmgpy.thermo.Wilhoit(Cp0=None, CpInf=None, a0=0.0, a1=0.0, a2=0.0, a3=0.0, H0=None, S0=None, B=None, Tmin=None, Tmax=None, comment='')
```

A heat capacity model based on the Wilhoit equation. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>The zeroth-order Wilhoit polynomial coefficient</td>
</tr>
<tr>
<td>a1</td>
<td>The first-order Wilhoit polynomial coefficient</td>
</tr>
<tr>
<td>a2</td>
<td>The second-order Wilhoit polynomial coefficient</td>
</tr>
<tr>
<td>a3</td>
<td>The third-order Wilhoit polynomial coefficient</td>
</tr>
<tr>
<td>H0</td>
<td>The integration constant for enthalpy (not $H$ at $T$=0)</td>
</tr>
<tr>
<td>S0</td>
<td>The integration constant for entropy (not $S$ at $T$=0)</td>
</tr>
<tr>
<td>E0</td>
<td>The energy at zero Kelvin (including zero point energy)</td>
</tr>
<tr>
<td>B</td>
<td>The Wilhoit scaled temperature coefficient in K</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The Wilhoit polynomial is an expression for heat capacity that is guaranteed to give the correct limits at zero and infinite temperature, and gives a very reasonable shape to the heat capacity profile in between:

$$C_p(T) = C_p(0) + [C_p(\infty) - C_p(0)] y^2 \left[ 1 + (y - 1) \sum_{i=0}^{3} a_i y^i \right]$$

Above, $y \equiv T/(T + B)$ is a scaled temperature that ranges from zero to one based on the value of the coefficient $B$, and $a_0$, $a_1$, $a_2$, and $a_3$ are the Wilhoit polynomial coefficients.

The enthalpy is given by

$$H(T) = H_0 + C_p(0) T + [C_p(\infty) - C_p(0)] T$$

\[
\begin{aligned}
&\left\{ 2 + \sum_{i=0}^{3} a_i \right\} \left[ \frac{1}{2} y - 1 + \left( \frac{1}{y} - 1 \right) \ln \frac{T}{y} \right] + y^2 \sum_{i=0}^{3} \frac{y^i}{(i+2)(i+3)} \sum_{j=0}^{3} f_{ij} a_j \\
&= f_{ij} = 3 + j \text{ if } i = j, \quad f_{ij} = 1 \text{ if } i > j, \quad \text{ and } \quad f_{ij} = 0 \text{ if } i < j.
\end{aligned}
\]

where $f_{ij} = 3 + j$ if $i = j$, $f_{ij} = 1$ if $i > j$, and $f_{ij} = 0$ if $i < j.$
The entropy is given by

\[ S(T) = S_0 + C_p(\infty) \ln T - \left[ C_p(\infty) - C_p(0) \right] \left[ \ln y + \left( 1 + y \sum_{i=0}^{3} \frac{a_i y^i}{2 + i} \right) y \right] \]

The low-temperature limit \( C_p(0) \) is \( 3.5R \) for linear molecules and \( 4R \) for nonlinear molecules. The high-temperature limit \( C_p(\infty) \) is taken to be \( [3N_{\text{atoms}} - 1.5] R \) for linear molecules and \( [3N_{\text{atoms}} - (2 + 0.5N_{\text{rotors}})] R \) for nonlinear molecules, for a molecule composed of \( N_{\text{atoms}} \) atoms and \( N_{\text{rotors}} \) internal rotors.

**B**

The Wilhoit scaled temperature coefficient.

**Cp0**

The heat capacity at zero temperature.

**CpInf**

The heat capacity at infinite temperature.

**E0**

The ground state energy (\( J/\text{mol} \)) at zero Kelvin, including zero point energy.

For the Wilhoit class, this is calculated as the Enthalpy at 0.001 Kelvin.

**H0**

The integration constant for enthalpy.

NB. this is not equal to the enthalpy at 0 Kelvin, which you can access via E0.

**S0**

The integration constant for entropy.

**Tmax**

The maximum temperature at which the model is valid, or \textit{None} if not defined.

**Tmin**

The minimum temperature at which the model is valid, or \textit{None} if not defined.

**a0**

a0: ‘double’

**a1**

a1: ‘double’

**a2**

a2: ‘double’

**a3**

a3: ‘double’

**comment**

comment: str

**copy** (\textit{self}) \rightarrow \textit{Wilhoit}

Return a copy of the Wilhoit object.

**discrepancy** (\textit{self}, \textit{HeatCapacityModel other}) \rightarrow double

Return some measure of how dissimilar \textit{self} is from \textit{other}.

The measure is arbitrary, but hopefully useful for sorting purposes. Discrepancy of 0 means they are identical.
fitToData(self, ndarray Tdata, ndarray Cpdata, double Cp0, double CpInf, double H298, double S298, double B0=500.0)
Fit a Wilhoit model to the data points provided, allowing the characteristic temperature \( B \) to vary so as to improve the fit. This procedure requires an optimization, using the `fminbound` function in the `scipy.optimize` module. The data consists of a set of heat capacity points \( \text{Cpdata} \) in J/mol*K at a given set of temperatures \( Tdata \) in K, along with the enthalpy \( H298 \) in kJ/mol and entropy \( S298 \) in J/mol*K at 298 K. The linearity of the molecule, number of vibrational frequencies, and number of internal rotors (linear, Nfreq, and Nrotors, respectively) is used to set the limits at zero and infinite temperature.

fitToDataForConstantB(self, ndarray Tdata, ndarray Cpdata, double Cp0, double CpInf, double H298, double S298, double B)
Fit a Wilhoit model to the data points provided using a specified value of the characteristic temperature \( B \). The data consists of a set of dimensionless heat capacity points \( \text{Cpdata} \) at a given set of temperatures \( Tdata \) in K, along with the enthalpy \( H298 \) at 298 K, and the dimensionless entropy \( S298 \) at 298 K.

getEnthalpy(self, double T) → double
Return the enthalpy in J/mol at the specified temperature \( T \) in K.

getEntropy(self, double T) → double
Return the entropy in J/mol*K at the specified temperature \( T \) in K.

getFreeEnergy(self, double T) → double
Return the Gibbs free energy in J/mol at the specified temperature \( T \) in K.

getHeatCapacity(self, double T) → double
Return the constant-pressure heat capacity in J/mol*K at the specified temperature \( T \) in K.

isIdenticalTo(self, HeatCapacityModel other) → bool
Returns True if \( \text{self} \) and \( \text{other} \) report very similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isSimilarTo(self, HeatCapacityModel other) → bool
Returns True if \( \text{self} \) and \( \text{other} \) report similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isTemperatureValid(self, double T) → bool
Return True if the temperature \( T \) in K is within the valid temperature range of the thermodynamic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

toNASA(self, double Tmin, double Tmax, double Tint, bool fixedTint=False, bool weighting=True, int continuity=3) → NASA
Convert the Wilhoit object to a NASA object. You must specify the minimum and maximum temperatures of the fit \( T\text{min} \) and \( T\text{max} \) in K, as well as the intermediate temperature \( T\text{int} \) in K to use as the bridge between the two fitted polynomials. The remaining parameters can be used to modify the fitting algorithm used:

- **fixedTint** - False to allow \( T\text{int} \) to vary in order to improve the fit, or True to keep it fixed
- **weighting** - True to weight the fit by \( T^{-1} \) to emphasize good fit at lower temperatures, or False to not use weighting
- **continuity** - The number of continuity constraints to enforce at \( T\text{int} \):
  - 0: no constraints on continuity of \( C_p(T) \) at \( T\text{int} \)
  - 1: constrain \( C_p(T) \) to be continuous at \( T\text{int} \)
  - 2: constrain \( C_p(T) \) and \( \frac{dC_p}{dT} \) to be continuous at \( T\text{int} \)
  - 3: constrain \( C_p(T) \), \( \frac{dC_p}{dT} \), and \( \frac{d^2C_p}{dT^2} \) to be continuous at \( T\text{int} \)
-4: constrain \( C_p(T), \frac{dC_p}{dT}, \frac{d^2C_p}{dT^2}, \) and \( \frac{d^3C_p}{dT^3} \) to be continuous at \( T_{int} \)

-5: constrain \( C_p(T), \frac{dC_p}{dT}, \frac{d^2C_p}{dT^2}, \frac{d^3C_p}{dT^3}, \) and \( \frac{d^4C_p}{dT^4} \) to be continuous at \( T_{int} \)

Note that values of continuity of 5 or higher effectively constrain all the coefficients to be equal and should be equivalent to fitting only one polynomial (rather than two).

Returns the fitted NASA object containing the two fitted NASAPolynomial objects.

toThermoData(self) → ThermoData
Convert the Wilhoit model to a ThermoData object.

rmgpy.thermo.NASA

class rmgpy.thermo.NASA(polynomials=None, Tmin=None, Tmax=None, E0=None, Cp0=None, CpInf=None, comment='')
A heat capacity model based on a set of one, two, or three NASAPolynomial objects. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>polynomials</td>
<td>The list of NASA polynomials to use in this model</td>
</tr>
<tr>
<td>Tmin</td>
<td>The minimum temperature in K at which the model is valid, or zero if unknown</td>
</tr>
<tr>
<td>Tmax</td>
<td>The maximum temperature in K at which the model is valid, or zero if unknown</td>
</tr>
<tr>
<td>E0</td>
<td>The energy at zero Kelvin (including zero point energy)</td>
</tr>
<tr>
<td>comment</td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The NASA polynomial is another representation of the heat capacity, enthalpy, and entropy using seven or nine coefficients \( a = [a_{-2} \ a_{-1} \ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6] \). The relevant thermodynamic parameters are evaluated via the expressions

\[
\frac{C_p(T)}{R} = a_{-2}T^{-2} + a_{-1}T^{-1} + a_0 + a_1T + a_2T^2 + a_3T^3 + a_4T^4
\]

\[
\frac{H(T)}{RT} = -a_{-2}T^{-2} + a_{-1}T^{-1}\ln T + a_0 + \frac{1}{2}a_1T + \frac{1}{3}a_2T^2 + \frac{1}{4}a_3T^3 + \frac{1}{5}a_4T^4 + \frac{a_5}{T}
\]

\[
\frac{S(T)}{R} = -\frac{1}{2}a_{-2}T^{-2} - a_{-1}T^{-1} + a_0\ln T + a_1T + \frac{1}{2}a_2T^2 + \frac{1}{3}a_3T^3 + \frac{1}{4}a_4T^4 + a_6
\]

In the seven-coefficient version, \( a_{-2} = a_{-1} = 0 \).

As simple polynomial expressions, the NASA polynomial is faster to evaluate when compared to the Wilhoit model; however, it does not have the nice physical behavior of the Wilhoit representation. Often multiple NASA polynomials are used to accurately represent the thermodynamics of a system over a wide temperature range.

\( \textbf{Cp0} \)

The heat capacity at zero temperature.

\( \textbf{CpInf} \)

The heat capacity at infinite temperature.

\( \textbf{E0} \)

The ground state energy (J/mol) at zero Kelvin, including zero point energy, or None if not yet specified.

\( \textbf{Tmax} \)

The maximum temperature at which the model is valid, or None if not defined.

\( \textbf{Tmin} \)

The minimum temperature at which the model is valid, or None if not defined.
**changeBaseEnthalpy**(*self*, *double deltaH*) → NASA

Add deltaH in J/mol to the base enthalpy of formation H298 and return the modified NASA object.

**comment**

*comment: str*

**discrepancy**(*self*, *HeatCapacityModel other*) → double

Return some measure of how dissimilar *self* is from *other*.

The measure is arbitrary, but hopefully useful for sorting purposes. Discrepancy of 0 means they are identical.

**getEnthalpy**(*self*, *double T*) → double

Return the dimensionless enthalpy \( H(T) / RT \) at the specified temperature \( T \) in K.

**getEntropy**(*self*, *double T*) → double

Return the dimensionless entropy \( S(T) / R \) at the specified temperature \( T \) in K.

**getFreeEnergy**(*self*, *double T*) → double

Return the dimensionless Gibbs free energy \( G(T) / RT \) at the specified temperature \( T \) in K.

**getHeatCapacity**(*self*, *double T*) → double

Return the dimensionless constant-pressure heat capacity \( C_p(T) / R \) at the specified temperature \( T \) in K.

**isIdenticalTo**(*self*, *HeatCapacityModel other*) → bool

Returns True if *self* and *other* report very similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

**isSimilarTo**(*self*, *HeatCapacityModel other*) → bool

Returns True if *self* and *other* report similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

**isTemperatureValid**(*self*, *double T*) → bool

Return True if the temperature \( T \) in K is within the valid temperature range of the thermodynamic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.

**poly1**

*poly1: rmgpy.thermo.nasa.NASAPolynomial*

**poly2**

*poly2: rmgpy.thermo.nasa.NASAPolynomial*

**poly3**

*poly3: rmgpy.thermo.nasa.NASAPolynomial*

**polynomials**

The set of one, two, or three NASA polynomials.

**selectPolynomial**(*self*, *double T*) → NASAPolynomial

**toCantera**(*self*)

Return the cantera equivalent NasaPoly2 object from this NASA object.

**toThermoData**(*self*) → ThermoData

Convert the Wilhoit model to a ThermoData object.

**toWilhoit**(*self*) → Wilhoit

Convert a MultiNASA object multiNASA to a Wilhoit object. You must specify the linearity of the molecule linear, the number of vibrational modes Nfreq, and the number of hindered rotor modes Nrots so the algorithm can determine the appropriate heat capacity limits at zero and infinite temperature.

Here is an example of a NASA entry:
A heat capacity model based on the NASA polynomial. Both the seven-coefficient and nine-coefficient variations are supported. The attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>coeffs</code></td>
<td>The seven or nine NASA polynomial coefficients</td>
</tr>
<tr>
<td><code>Tmin</code></td>
<td>The minimum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>Tmax</code></td>
<td>The maximum temperature in K at which the model is valid, or zero if unknown or undefined</td>
</tr>
<tr>
<td><code>E0</code></td>
<td>The energy at zero Kelvin (including zero point energy)</td>
</tr>
<tr>
<td><code>comment</code></td>
<td>Information about the model (e.g. its source)</td>
</tr>
</tbody>
</table>

The NASA polynomial is another representation of the heat capacity, enthalpy, and entropy using seven or nine coefficients $a = \{a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$. The relevant thermodynamic parameters are evaluated via the expressions

\[
\frac{C_p(T)}{R} = a_{-2}T^{-2} + a_{-1}T^{-1} + a_0 + a_1T + a_2T^2 + a_3T^3 + a_4T^4
\]

\[
\frac{H(T)}{RT} = -a_{-2}T^{-2} + a_{-1}T^{-1} \ln T + a_0 + \frac{1}{2}a_1T + \frac{1}{3}a_2T^2 + \frac{1}{4}a_3T^3 + \frac{1}{5}a_4T^4 + \frac{a_5}{T}
\]

\[
\frac{S(T)}{R} = -\frac{1}{2}a_{-2}T^{-2} - a_{-1}T^{-1} \ln T + a_0 + a_1T + \frac{1}{2}a_2T^2 + \frac{1}{3}a_3T^3 + \frac{1}{4}a_4T^4 + a_6
\]
In the seven-coefficient version, $a_{-2} = a_{-1} = 0$.

As simple polynomial expressions, the NASA polynomial is faster to evaluate when compared to the Wilhoit model; however, it does not have the nice physical behavior of the Wilhoit representation. Often multiple NASA polynomials are used to accurately represent the thermodynamics of a system over a wide temperature range; the NASA class is available for this purpose.

- $C_p^0$  
  The heat capacity at zero temperature.

- $C_p^\infty$  
  The heat capacity at infinite temperature.

- $E_0$  
  The ground state energy (J/mol) at zero Kelvin, including zero point energy, or None if not yet specified.

- $T_{\text{max}}$  
  The maximum temperature at which the model is valid, or None if not defined.

- $T_{\text{min}}$  
  The minimum temperature at which the model is valid, or None if not defined.

- $c_0$ 
  $c_0$: ‘double’

- $c_1$ 
  $c_1$: ‘double’

- $c_2$ 
  $c_2$: ‘double’

- $c_3$ 
  $c_3$: ‘double’

- $c_4$ 
  $c_4$: ‘double’

- $c_5$ 
  $c_5$: ‘double’

- $c_6$ 
  $c_6$: ‘double’

- $\text{changeBaseEnthalpy}(self, \text{double} \ deltaH)$  
  Add $\delta H$ in J/mol to the base enthalpy of formation $H_{298}$.

- $c_{m1}$ 
  $c_{m1}$: ‘double’

- $c_{m2}$ 
  $c_{m2}$: ‘double’

- $\text{coefs}$  
  The set of seven or nine NASA polynomial coefficients.

- $\text{comment}$ 
  $\text{comment}$: str

- $\text{discrepancy}(self, \text{HeatCapacityModel} \ other)$ → double  
  Return some measure of how dissimilar $self$ is from $other$.
  
  The measure is arbitrary, but hopefully useful for sorting purposes. Discrepancy of 0 means they are identical.
getEnthalpy(self, double T) → double
    Return the enthalpy in J/mol at the specified temperature T in K.

getEntropy(self, double T) → double
    Return the entropy in J/mol*K at the specified temperature T in K.

getFreeEnergy(self, double T) → double
    Return the Gibbs free energy in J/mol at the specified temperature T in K.

getHeatCapacity(self, double T) → double
    Return the constant-pressure heat capacity in J/mol*K at the specified temperature T in K.

isIdenticalTo(self, HeatCapacityModel other) → bool
    Returns True if self and other report very similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isSimilarTo(self, HeatCapacityModel other) → bool
    Returns True if self and other report similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.

isTemperatureValid(self, double T) → bool
    Return True if the temperature T in K is within the valid temperature range of the thermodynamic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.


rmgpy.cantherm, 3
rmgpy.chemkin, 10
rmgpy.constants, 13
rmgpy.data, 13
rmgpy.kinetics, 64
rmgpy.molecule, 84
rmgpy.molecule.adjlist, 104
rmgpy.pdep, 108
rmgpy.qm, 118
rmgpy.quantity, 136
rmgpy.reaction, 140
rmgpy.rmg, 143
rmgpy.solver, 157
rmgpy.species, 160
rmgpy.statmech, 163
rmgpy.statmech.schrodinger, 175
rmgpy.thermo, 178
INDEX

A (rmgpy.kinetics.Arrhenius attribute), 66
a0 (rmgpy.thermo.Wilhoit attribute), 181
a1 (rmgpy.thermo.Wilhoit attribute), 181
a2 (rmgpy.thermo.Wilhoit attribute), 181
a3 (rmgpy.thermo.Wilhoit attribute), 181
addAction() (rmgpy.data.kinetics.ReactionRecipe method), 40
addAtom() (rmgpy.molecule.Group method), 100
addAtom() (rmgpy.molecule.Molecule method), 93
addAtomLabelsForReaction() (rmgpy.data.kinetics.KineticsFamily method), 25
addBond() (rmgpy.molecule.Group method), 101
addBond() (rmgpy.molecule.Molecule method), 93
addEdge() (rmgpy.molecule.graph.Graph method), 86
addEdge() (rmgpy.molecule.Group method), 101
addEdge() (rmgpy.molecule.Molecule method), 93
addExplicitLigands() (rmgpy.molecule.Group method), 101
addKineticsRulesFromTrainingSet() (rmgpy.data.kinetics.KineticsFamily method), 25
addPathReaction() (rmgpy.rmg.pdep.PDepNetwork method), 150
addReactionLibraryToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 144
addReactionLibraryToOutput() (rmgpy.rmg.model.CoreEdgeReactionModel method), 144
addReactionToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 144
addReactionToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
addReactionToUnimolecularNetworks() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
addSeedMechanismToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
addSpeciesToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
addSpeciesToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
addVertex() (rmgpy.molecule.graph.Graph method), 86
addVertex() (rmgpy.molecule.Group method), 101
addVertex() (rmgpy.molecule.Molecule method), 93
advance() (rmgpy.solver.ReactionSystem method), 157
advance() (rmgpy.solver.SimpleReactor method), 158
alpha (rmgpy.kinetics.Troe attribute), 80
alpha0 (rmgpy.pdep.SingleExponentialDown attribute), 109
ancestors() (rmgpy.data.base.Database method), 15
ancestors() (rmgpy.data.kinetics.KineticsDepository method), 22
ancestors() (rmgpy.data.kinetics.KineticsFamily method), 25
ancestors() (rmgpy.data.kinetics.KineticsGroups method), 30
ancestors() (rmgpy.data.kinetics.KineticsLibrary method), 32
ancestors() (rmgpy.data.kinetics.KineticsRules method), 34
ancestors() (rmgpy.data.statmech.StatmechDepository method), 42
ancestors() (rmgpy.data.statmech.StatmechGroups method), 48
ancestors() (rmgpy.data.statmech.StatmechLibrary method), 50
ancestors() (rmgpy.data.thermo.ThermoDepository method), 58
ancestors() (rmgpy.data.thermo.ThermoGroups method), 60
ancestors() (rmgpy.data.thermo.ThermoLibrary method), 62
applyAction() (rmgpy.molecule.Atom method), 91
applyAction() (rmgpy.molecule.Bond method), 92
applyAction() (rmgpy.molecule.GroupAtom method), 99
applyAction() (rmgpy.molecule.GroupBond method), 100
applyChemicallySignificantEigenvaluesMethod() (in module rmgpy.pdep.cse), 118
applyChemicallySignificantEigenvaluesMethod() (in module rmgpy.pdep.cse), 118
calculateChiralityCorrection()
(rmgpy.qm.mopac.MopacMolPM7 method), 134

calculateCollisionEfficiency()
(rmgpy.pdep.SingleExponentialDown method), 109

calculateCollisionFrequency()
(rmgpy.pdep.Configuration method), 111

calculateCollisionModel()
(rmgpy.pdep.Network method), 113

calculateCollisionModel()
(rmgpy.rmg.pdep.PDepNetwork method), 150

calculateCp0()
(rmgpy.molecule.Molecule method), 93
calculateCp0()
(rmgpy.rmg.model.Species method), 155
calculateCp0()
(rmgpy.species.Species method), 161

calculateCpInf()
(rmgpy.molecule.Molecule method), 94
calculateCpInf()
(rmgpy.rmg.model.Species method), 155

calculateCpInf()
(rmgpy.species.Species method), 161

calculateCyclicSymmetryNumber()
(in module rmgpy.molecule.symmetry), 107

calculateDegeneracy()
(rmgpy.data.kinetics.KineticsFamily method), 25

calculateDensitiesOfStates()
(rmgpy.pdep.Network method), 113
calculateDensitiesOfStates()
(rmgpy.rmg.pdep.PDepNetwork method), 150

calculateDensityOfStates()
(rmgpy.pdep.Configuration method), 111

calculateEquilibriumRatios()
(rmgpy.pdep.Network method), 113
calculateEquilibriumRatios()
(rmgpy.rmg.pdep.PDepNetwork method), 150

calculateMicrocanonicalRateCoefficient() (in module rmgpy.rmg), 110
calculateMicrocanonicalRateCoefficient() (in module rmgpy.pdep), 110
calculateMicrocanonicalRateCoefficient()
(rmgpy.data.kinetics.DepositoryReaction method), 17

calculateMicrocanonicalRateCoefficient()
(rmgpy.data.kinetics.LibraryReaction method), 37

calculateMicrocanonicalRateCoefficient() (in module rmgpy.data.kinetics.TemplateReaction method), 53
calculateMicrocanonicalRateCoefficient() (rmgpy.reaction.Reaction method), 141
calculateMicrocanonicalRateCoefficient() (rmgpy.rmg.pdep.PDepReaction method), 152
calculateTunnelingFactor() (rmgpy.kinetics.Eckart method), 83
calculateTunnelingFactor() (rmgpy.kinetics.Wigner method), 82
calculateTunnelingFactor()
(rmgpy.species.TransitionState method), 163

calculateTunnelingFunction() (rmgpy.kinetics.Eckart method), 83
calculateTunnelingFunction() (rmgpy.kinetics.Wigner method), 82
calculateTunnelingFunction()
(rmgpy.species.TransitionState method), 163

calculateNormDistance()
(rmgpy.data.kinetics.KineticsRules method), 34

calculateSymmetryNumber()
(in module rmgpy.molecule.symmetry), 107

calculateSymmetryNumber()
(rmgpy.molecule.Molecule method), 94

calculateThermoData()
(rmgpy.qm.gaussian.GaussianMol method), 124
calculateThermoData()
(rmgpy.qm.gaussian.GaussianMolPM3 method), 126
calculateThermoData()
(rmgpy.qm.gaussian.GaussianMolPM6 method), 127
calculateThermoData()
(rmgpy.qm.molecule.QMMolecule method), 121
calculateThermoData()
(rmgpy.qm.mopac.MopacMol method), 130
calculateThermoData()
(rmgpy.qm.mopac.MopacMolPM3 method), 131
calculateThermoData()
(rmgpy.qm.mopac.MopacMolPM6 method), 133
calculateThermoData()
(rmgpy.qm.mopac.MopacMolPM7 method), 134

calculateTSTRateCoefficient()
(rmgpy.data.kinetics.DepositoryReaction method), 17

calculateTSTRateCoefficient()
(rmgpy.data.kinetics.LibraryReaction method), 37

calculateTSTRateCoefficient() (rmgpy.data.kinetics.TemplateReaction method), 53

calculateTSTRateCoefficient() (rmgpy.reaction.Reaction method), 141

calculateTSTRateCoefficient() (rmgpy.rmg.pdep.PDepReaction method), 152
canTST() (rmgpy.data.kinetics.LibraryReaction method), 37
changeBaseEnthalpy() (rmgpy.thermo.NASA method), 183
changeBaseEnthalpy() (rmgpy.thermo.NASAPolynomial method), 186
changeRate() (rmgpy.kinetics.Arrhenius method), 66
changeRate() (rmgpy.kinetics.Chebyshev method), 74
changeRate() (rmgpy.kinetics.Lindemann method), 78
changeRate() (rmgpy.kinetics.MultiArrhenius method), 68
changeRate() (rmgpy.kinetics.MultiPDepArrhenius method), 72
changeRate() (rmgpy.kinetics.PDepArrhenius method), 71
changeRate() (rmgpy.kinetics.ThirdBody method), 76
changeRate() (rmgpy.kinetics.Troe method), 80
checkAllSet() (rmgpy.qm.main.QMSettings method), 119
checkForDuplicates() (rmgpy.data.kinetics.KineticsLibrary method), 32
checkForExistingReaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
checkForExistingSpecies() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
checkForInChiKeyCollision() (rmgpy.qm.qmverifier.QMVerifier method), 122
checkInput() (rmgpy.rmg.main.RMG method), 148
checkLibraries() (rmgpy.rmg.main.RMG method), 149
checkPaths() (rmgpy.qm.gaussian.GaussianMol method), 124
checkPaths() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
checkPaths() (rmgpy.qm.gaussian.GaussianMolPM6 method), 127
checkPaths() (rmgpy.qm.main.QMCalculator method), 120
checkPaths() (rmgpy.qm.molecule.QMMolecule method), 121
checkPaths() (rmgpy.qm.mopac.MopacMol method), 130
checkPaths() (rmgpy.qm.mopac.MopacMolPM3 method), 131
checkPaths() (rmgpy.qm.mopac.MopacMolPM6 method), 133
checkPaths() (rmgpy.qm.mopac.MopacMolPM7 method), 135
checkReady() (rmgpy.qm.gaussian.GaussianMol method), 124
checkReady() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
checkReady() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
checkReady() (rmgpy.qm.main.QMCalculator method), 120
checkReady() (rmgpy.qm.molecule.QMMolecule method), 121
checkReady() (rmgpy.qm.mopac.MopacMol method), 130
checkReady() (rmgpy.qm.mopac.MopacMolPM3 method), 131
checkReady() (rmgpy.qm.mopac.MopacMolPM6 method), 133
checkReady() (rmgpy.qm.mopac.MopacMolPM7 method), 135
cleanup() (rmgpy.pdep.Configuration method), 111
cleanup() (rmgpy.rmg.pdep.PDepNetwork method), 151
clear() (rmgpy.rmg.main.RMG method), 149
clearLabeledAtoms() (rmgpy.molecule.Group method), 101
clearLabeledAtoms() (rmgpy.molecule.Molecule method), 94
cm1 (rmgpy.thermo.NASAPolynomial attribute), 186
cm2 (rmgpy.thermo.NASAPolynomial attribute), 186
coeffs (rmgpy.kinetics.Chebyshev attribute), 74
coeffs (rmgpy.thermo.NASAPolynomial attribute), 186
cm1 (rmgpy.thermo.NASAPolynomial attribute), 186
cm2 (rmgpy.thermo.NASAPolynomial attribute), 186
coef (rmgpy.kinetics.Chebyshev attribute), 74
comment (rmgpy.kinetics.Arrhenius attribute), 67
comment (rmgpy.kinetics.Chebyshev attribute), 74
comment (rmgpy.kinetics.KineticsData attribute), 65
comment (rmgpy.kinetics.Lindemann attribute), 78
comment (rmgpy.kinetics.MultiArrhenius attribute), 68
comment (rmgpy.kinetics.MultiPDepArrhenius attribute), 72
comment (rmgpy.kinetics.PDepArrhenius attribute), 71
comment (rmgpy.kinetics.PDepKineticsData attribute), 69
comment (rmgpy.kinetics.ThirdBody attribute), 76
comment (rmgpy.kinetics.Troe attribute), 80
comment (rmgpy.thermo.NASA attribute), 184
comment (rmgpy.thermo.NASAPolynomial attribute), 186
comment (rmgpy.thermo.ThermoData attribute), 179
comment (rmgpy.thermo.Wilhoit attribute), 181
compute_network_variables() (rmgpy.solver.ReactionSystem method), 157
compute_network_variables() (rmgpy.solver.SimpleReactor method), 159
Index
descendants() (rmgpy.data.statmech.StatmechLibrary method), 50
descendants() (rmgpy.data.thermo.ThermoDepository method), 58
descendants() (rmgpy.data.thermo.ThermoGroups method), 60
descendants() (rmgpy.data.thermo.ThermoLibrary method), 62
descendTree() (rmgpy.data.base.Database method), 15
descendTree() (rmgpy.data.kinetics.KineticsDepository method), 22
descendTree() (rmgpy.data.kinetics.KineticsFamily method), 25
descendTree() (rmgpy.data.kinetics.KineticsGroups method), 30
descendTree() (rmgpy.data.kinetics.KineticsLibrary method), 32
descendTree() (rmgpy.data.kinetics.KineticsRules method), 34
descendTree() (rmgpy.data.statmech.StatmechDepository method), 42
descendTree() (rmgpy.data.statmech.StatmechGroups method), 48
descendTree() (rmgpy.data.statmech.StatmechLibrary method), 50
descendTree() (rmgpy.data.thermo.ThermoDepository method), 58
descendTree() (rmgpy.data.thermo.ThermoGroups method), 60
descendTree() (rmgpy.data.thermo.ThermoLibrary method), 62
determinePointGroup() (rmgpy.qm.gaussian.GaussianMol method), 124
determinePointGroup() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
determinePointGroup() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
determinePointGroup() (rmgpy.qm.molecule.QMMolecule method), 121
determinePointGroup() (rmgpy.qm.mopac.MopacMol method), 130
determinePointGroup() (rmgpy.qm.mopac.MopacMolPM3 method), 131
determinePointGroup() (rmgpy.qm.mopac.MopacMolPM6 method), 133
determinePointGroup() (rmgpy.qm.mopac.MopacMolPM7 method), 135
DirectFit (class in rmgpy.data.statmechfit), 45
discrepancy() (rmgpy.kinetics.Arrhenius method), 67
discrepancy() (rmgpy.kinetics.Chebyshev method), 74
discrepancy() (rmgpy.kinetics.KineticsData method), 65
discrepancy() (rmgpy.kinetics.Lindemann method), 78
discrepancy() (rmgpy.kinetics.MultiArrhenius method), 68
discrepancy() (rmgpy.kinetics.MultiPDepArrhenius method), 72
discrepancy() (rmgpy.kinetics.PDepArrhenius method), 71
discrepancy() (rmgpy.kinetics.PDepKineticsData method), 69
discrepancy() (rmgpy.kinetics.ThirdBody method), 76
discrepancy() (rmgpy.thermo.NASA method), 184
discrepancy() (rmgpy.thermo.NASAPolynomial method), 186
discrepancy() (rmgpy.thermo.ThermoData method), 179
discrepancy() (rmgpy.thermo.Wilhoit method), 181
draw() (rmgpy.cantherm.PressureDependenceJob method), 8
draw() (rmgpy.data.kinetics.DepositoryReaction method), 18
draw() (rmgpy.data.kinetics.LibraryReaction method), 37
draw() (rmgpy.data.kinetics.TemplateReaction method), 53
draw() (rmgpy.molecule.draw.MoleculeDrawer method), 107
draw() (rmgpy.molecule.Molecule method), 94
draw() (rmgpy.reaction.Reaction method), 141
draw() (rmgpy.rmg.pdep.PDepReaction method), 153

e
E0 (rmgpy.pdep.Configuration attribute), 111
E0 (rmgpy.statmech.Conformer attribute), 176
E0 (rmgpy.thermo.NASA attribute), 183
E0 (rmgpy.thermo.NASAPolynomial attribute), 186
E0 (rmgpy.thermo.ThermoData attribute), 178
E0 (rmgpy.thermo.Wilhoit attribute), 181
E0_prod (rmgpy.kinetics.Eckart attribute), 83
E0_reac (rmgpy.kinetics.Eckart attribute), 83
E0_TS (rmgpy.kinetics.Eckart attribute), 83
Ea (rmgpy.kinetics.Arrhenius attribute), 66
Eckart (class in rmgpy.kinetics), 82
Edge (class in rmgpy.molecule.graph), 86
efficiencies (rmgpy.kinetics.Chebyshev attribute), 74
efficiencies (rmgpy.kinetics.Lindemann attribute), 78
efficiencies (rmgpy.kinetics.MultiPDepArrhenius attribute), 72
efficiencies (rmgpy.kinetics.PDepArrhenius attribute), 71
efficiencies (rmgpy.kinetics.PDepKineticsData attribute), 69
efficiencies (rmgpy.kinetics.ThirdBody attribute), 76
Element (class in rmgpy.molecule), 88
energies (rmgpy.statmech.HinderedRotor attribute), 173
enlarge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
Entry (class in rmgpy.data.base), 20
equals() (rmgpy.quantity.ArrayQuantity method), 139
equals() (rmgpy.quantity.ScalarQuantity method), 138
equivalent() (rmgpy.molecule.Atom method), 91
equivalent() (rmgpy.molecule.AtomType method), 89
equivalent() (rmgpy.molecule.Bond method), 92
equivalent() (rmgpy.molecule.graph.Edge method), 86
equivalent() (rmgpy.molecule.graph.Vertex method), 85
equivalent() (rmgpy.molecule.GroupAtom method), 99
equivalent() (rmgpy.molecule.GroupBond method), 100
estimateKinetics() (rmgpy.data.kinetics.KineticsRules method), 34
estimateKineticsUsingRateRules() (rmgpy.data.kinetics.KineticsFamily method), 26
estimateRadicalThermoViaHBI() (rmgpy.data.thermo.ThermoDatabase method), 55
estimateThermoViaGroupAdditivity() (rmgpy.data.thermo.ThermoDatabase method), 55
execute() (rmgpy.cantherm.CanTherm method), 6
execute() (rmgpy.cantherm.KineticsJob method), 6
execute() (rmgpy.cantherm.StatMechJob method), 9
execute() (rmgpy.cantherm.ThermoJob method), 9
exploreIsomer() (rmgpy.rmg.main.RMG method), 149
findIsomorphism() (rmgpy.molecule.graph.Graph method), 86
findIsomorphism() (rmgpy.molecule.Group method), 101
findIsomorphism() (rmgpy.molecule.Molecule method), 94
findIsomorphism() (rmgpy.molecule.vf2.VF2 method), 88
findSubgraphIsomorphisms() (rmgpy.molecule.graph.Graph method), 86
findSubgraphIsomorphisms() (rmgpy.molecule.Group method), 101
findSubgraphIsomorphisms() (rmgpy.molecule.Molecule method), 94
findSubgraphIsomorphisms() (rmgpy.molecule.vf2.VF2 method), 88
finish() (rmgpy.rmg.main.RMG method), 149
fitCosinePotentialToData() (rmgpy.statmech.HinderedRotor method), 173
fitFourierPotentialToData() (rmgpy.statmech.HinderedRotor method), 174
fitStatmechDirect() (in module rmgpy.data.statmechfit), 44
fitStatmechPseudo() (in module rmgpy.data.statmechfit), 44
fitStatmechPseudoRotors() (in module rmgpy.data.statmechfit), 44
fitToData() (rmgpy.thermo.Wilhoit method), 181
fitToData() (rmgpy.thermo.Wilhoit method), 181
fitToDataForConstantB() (rmgpy.thermo.Wilhoit method), 182
fixBarrierHeight() (rmgpy.data.kinetics.DepositoryReaction method), 18
fixBarrierHeight() (rmgpy.data.kinetics.LibraryReaction method), 37
fixBarrierHeight() (rmgpy.data.kinetics.TemplateReaction method), 53
fixBarrierHeight() (rmgpy.reaction.Reaction method), 141
fixBarrierHeight() (rmgpy.rmg.pdep.PDepReaction method), 153
fixDiffusionLimitedA() (rmgpy.data.kinetics.DepositoryReaction method), 18
fixDiffusionLimitedA() (rmgpy.data.kinetics.LibraryReaction method), 37
fixDiffusionLimitedA() (rmgpy.data.kinetics.TemplateReaction method), 53
fixDiffusionLimitedA() (rmgpy.reaction.Reaction method), 141

F
failureKeys (rmgpy.qm.gaussian.Gaussian attribute), 124
failureKeys (rmgpy.qm.mopac.Mopac attribute), 129
fillRulesByAveragingUp() (rmgpy.data.kinetics.KineticsRules method), 34

Index 199
fixDiffusionLimitedA() (rmgpy.rmg.pdep.PDepReaction method), 153
fouriér (rmgpy.statmech.HinderedRotor attribute), 174
frequencies (rmgpy.statmech.HarmonicOscillator attribute), 172
frequency (rmgpy.kinetics.Eckart attribute), 83
frequency (rmgpy.kinetics.Wigner attribute), 82
frequency (rmgpy.statmech.HinderedRotor attribute), 174
fromAdjacencyList() (in module rmgpy.molecule.adjlist), 106
fromAdjacencyList() (rmgpy.molecule.Group method), 101
fromAdjacencyList() (rmgpy.molecule.Molecule method), 94
fromAdjacencyList() (rmgpy.rmg.model.Species method), 155
fromAdjacencyList() (rmgpy.species.Species method), 161
fromAugmentedInChI() (rmgpy.molecule.Molecule method), 94
fromInChI() (rmgpy.molecule.Molecule method), 94
fromSMARTS() (rmgpy.molecule.Molecule method), 94
fromSMILES() (rmgpy.molecule.Molecule method), 95
fromSMILES() (rmgpy.rmg.model.Species method), 155
fromSMILES() (rmgpy.species.Species method), 161
fromXYZ() (rmgpy.molecule.Molecule method), 95

G
Gaussian (class in rmgpy.qm.gaussian), 124
GaussianLog (class in rmgpy.cantherm.gaussian), 4
GaussianMol (class in rmgpy.qm.gaussian), 124
GaussianMolPM3 (class in rmgpy.qm.gaussian), 126
GaussianMolPM6 (class in rmgpy.qm.gaussian), 127
generate3dTS() (rmgpy.data.kinetics.DepositoryReaction method), 18
generate3dTS() (rmgpy.data.kinetics.LibraryReaction method), 37
generate3dTS() (rmgpy.data.kinetics.TemplateReaction method), 53
generate3dTS() (rmgpy.reaction.Reaction method), 141
generate3dTS() (rmgpy.rmg.pdep.PDepReaction method), 153
generate_rate_coefficients() (rmgpy.solver.SimpleReactor method), 159
generate_reactant_product_indices() (rmgpy.solver.ReactionSystem method), 157
generate_reactant_product_indices() (rmgpy.solver.SimpleReactor method), 159
generate_reaction_indices() (rmgpy.solver.ReactionSystem method), 157
generate_reaction_indices() (rmgpy.solver.SimpleReactor method), 159
generate_species_indices() (rmgpy.solver.ReactionSystem method), 157
generate_species_indices() (rmgpy.solver.SimpleReactor method), 159
generateCollisionMatrix() (rmgpy.pdep.Configuration method), 111
generateCollisionMatrix() (rmgpy.pdep.SingleExponentialDown method), 110
generateEnergyTransferModel() (rmgpy.rmg.model.Species method), 155
generateEnergyTransferModel() (rmgpy.species.Species method), 161
generateFrequencies() (rmgpy.data.statmech.GroupFrequencies method), 20
generateFullMEMatrix() (in module rmgpy.pdep.me), 114
generateGroupAdditivityValues() (rmgpy.data.kinetics.KineticsGroups method), 30
generateKinetics() (rmgpy.cantherm.KineticsJob method), 6
generateKinetics() (rmgpy.rmg.model.CoreEdgeReactionModel method), 145
generateOldLibraryEntry() (rmgpy.data.statmech.StatmechGroups method), 48
generateOldLibraryEntry() (rmgpy.data.statmech.StatmechLibrary method), 51
generateOldLibraryEntry() (rmgpy.data.thermo.ThermoGroups method), 60
generateOldLibraryEntry() (rmgpy.data.thermo.ThermoLibrary method), 62
generateOldTree() (rmgpy.data.base.Database method), 15
generateOldTree() (rmgpy.data.kinetics.KineticsDepository method), 23
generateOldTree() (rmgpy.data.kinetics.KineticsFamily method), 26
generateOldTree() (rmgpy.data.kinetics.KineticsGroups method), 30
generateOldTree() (rmgpy.data.kinetics.KineticsLibrary method), 32
generateOldTree() (rmgpy.data.kinetics.KineticsRules method), 34
generateOldTree() (rmgpy.data.statmech.StatmechDepository method), 42
generateOldTree() (rmgpy.data.statmech.StatmechGroups method), 48
generateOldTree() (rmgpy.data.statmech.StatmechLibrary method), 50

200

Index

200
generateOldTree() (rmgpy.data.thermo.ThermoDepository method), 51
generateOldTree() (rmgpy.data.thermo.ThermoGroups method), 60
generateOldTree() (rmgpy.data.thermo.ThermoLibrary method), 62
generatePairs() (rmgpy.data.kinetics.DepositoryReaction method), 18
generatePairs() (rmgpy.data.kinetics.LibraryReaction method), 37
generatePairs() (rmgpy.data.kinetics.TemplateReaction method), 53
generatePairs() (rmgpy.reaction.Reaction method), 141
generatePairs() (rmgpy.rmg.pdep.PDepReaction method), 153
generatePressureList() (rmgpy.cantherm.PressureDependenceJob method), 9
generateProductTemplate() (rmgpy.data.kinetics.KineticsFamily method), 26
generateQMDdata() (rmgpy.qm.gaussian.GaussianMol method), 124
generateQMDdata() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
generateQMDdata() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
generateQMDdata() (rmgpy.qm.molecule.QMMolecule method), 121
generateQMDdata() (rmgpy.qm.mopac.MopacMol method), 130
generateQMDdata() (rmgpy.qm.mopac.MopacMolPM3 method), 131
generateQMDdata() (rmgpy.qm.mopac.MopacMolPM6 method), 133
generateQMDdata() (rmgpy.qm.mopac.MopacMolPM7 method), 135
generateRDKitGeometries() (rmgpy.qm.molecule.Geometry method), 120
generateReactions() (rmgpy.data.kinetics.KineticsDatabase method), 21
generateReactions() (rmgpy.data.kinetics.KineticsFamily method), 26
generateReactionsFromFamilies() (rmgpy.data.kinetics.KineticsDatabase method), 21
generateReactionsFromLibraries() (rmgpy.data.kinetics.KineticsDatabase method), 21
generateResonanceIsomers() (rmgpy.rmg.model.Species method), 155
generateReverseRateCoefficient() (rmgpy.data.kinetics.DepositoryReaction method), 18
generateReverseRateCoefficient() (rmgpy.data.kinetics.LibraryReaction method), 38
generateReverseRateCoefficient() (rmgpy.data.kinetics.TemplateReaction method), 54
generateReverseRateCoefficient() (rmgpy.reaction.Reaction method), 142
generateReverseRateCoefficient() (rmgpy.rmg.pdep.PDepReaction method), 153
generateStatMech() (rmgpy.rmg.model.Species method), 155
generateStatMech() (rmgpy.species.Species method), 161
generateTemperatureList() (rmgpy.cantherm.PressureDependenceJob method), 9
generateThermo() (rmgpy.cantherm.ThermoJob method), 9
generateThermoData() (rmgpy.qm.gaussian.GaussianMol method), 125
generateThermoData() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
generateThermoData() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
generateThermoData() (rmgpy.qm.molecule.QMMolecule method), 121
generateThermoData() (rmgpy.qm.mopac.MopacMol method), 130
generateThermoData() (rmgpy.qm.mopac.MopacMolPM3 method), 131
generateThermoData() (rmgpy.qm.mopac.MopacMolPM6 method), 133
generateThermoData() (rmgpy.qm.mopac.MopacMolPM7 method), 135
generateThermoData() (rmgpy.qm.mopac.MopacMolPM7 method), 135
generateTransportData() (rmgpy.rmg.model.Species method), 156
generateTransportData() (rmgpy.species.Species method), 162
generic_visit() (rmgpy.cantherm.output.PrettifyVisitor method), 7
getActiveModes() (rmgpy.statmech.Conformer method), 120
get_species_index() (rmgpy.solver.ReactionSystem method), 157
get_species_index() (rmgpy.solver.SimpleReactor method), 159
Geometry (class in rmgpy.cantherm.geometry), 5
Geometry (class in rmgpy.qm.molecule), 120
getActiveModes() (rmgpy.statmech.Conformer method), 120
getSpeciesIndex() (rmgpy.solver.ReactionSystem method), 157
getSpeciesIndex() (rmgpy.solver.SimpleReactor method), 159

getEffectiveColliderEfficiencies()
    (rmgpy.kinetics.PDepArrhenius method), 71
getEffectiveColliderEfficiencies()
    (rmgpy.kinetics.PDepKineticsData method), 69
getEffectiveColliderEfficiencies()
    (rmgpy.kinetics.ThirdBody method), 76
getEffectiveColliderEfficiencies() (rmgpy.kinetics.Troe method), 81
getEffectivePressure() (rmgpy.kinetics.Chebyshev method), 75
getEffectivePressure() (rmgpy.kinetics.Lindemann method), 78
getEffectivePressure() (rmgpy.kinetics.MultiPDepArrhenius method), 73
getEffectivePressure() (rmgpy.kinetics.PDepArrhenius method), 71
getEffectivePressure() (rmgpy.kinetics.PDepKineticsData method), 69
getEffectivePressure() (rmgpy.kinetics.ThirdBody method), 76
getEffectivePressure() (rmgpy.kinetics.Troe method), 81
gElement() (in module rmgpy.molecule), 89
getEnthalpiesOfReaction()
    (rmgpy.data.kinetics.DepositoryReaction method), 18
getEnthalpiesOfReaction()
    (rmgpy.data.kinetics.LibraryReaction method), 38
getEnthalpiesOfReaction()
    (rmgpy.data.kinetics.TemplateReaction method), 54
getEnthalpiesOfReaction() (rmgpy.reaction.Reaction method), 142
getEnthalpiesOfReaction()
    (rmgpy.rmg.pdep.PDepReaction method), 153
getEntriesToSave() (rmgpy.data.base.Database method), 15
getEntriesToSave() (rmgpy.data.kinetics.KineticsDepository method), 23
getEntriesToSave() (rmgpy.data.kinetics.KineticsFamily method), 26
getEntriesToSave() (rmgpy.data.kinetics.KineticsGroups method), 30
getEntriesToSave() (rmgpy.data.kinetics.KineticsLibrary method), 32
getEntriesToSave() (rmgpy.data.kinetics.KineticsRules method), 35
getEntriesToSave() (rmgpy.data.statmech.StatmechDepository method), 42
getEntriesToSave() (rmgpy.data.statmech.StatmechGroups method), 48
getEntriesToSave() (rmgpy.data.statmech.StatmechLibrary method), 51
getEntriesToSave() (rmgpy.data.thermo.ThermoDepository method), 58
getEntriesToSave() (rmgpy.data.thermo.ThermoGroups method), 60
getEntriesToSave() (rmgpy.data.thermo.ThermoLibrary method), 62
gGetEntropiesOfReaction()
    (rmgpy.data.kinetics.DepositoryReaction method), 18
getEntropiesOfReaction()
    (rmgpy.data.kinetics.LibraryReaction method), 38
getEntropiesOfReaction()
(rmgpy.data.kinetics.TemplateReaction method), 54
getEntropiesOfReaction()  (rmgpy.reaction.Reaction method), 142
getEntropiesOfReaction()
(rmgpy.rmg.pdep.PDepReaction method), 154
getEntropy() (in module rmgpy.statmech.schrodinger), 175
getEntropy() (rmgpy.pdep.Configuration method), 112
getEntropy() (rmgpy.rmg.model.Species method), 156
getEntropy() (rmgpy.species.Species method), 162
getEntropy() (rmgpy.species.TransitionState method), 163
getEntropy() (rmgpy.statmech.Conformer method), 177
getEntropy() (rmgpy.statmech.HarmonicOscillator method), 172
getEntropy() (rmgpy.statmech.HinderedRotor method), 174
getEntropy() (rmgpy.statmech.IdealGasTranslation method), 166
getEntropy() (rmgpy.statmech.KRotor method), 169
getEntropy() (rmgpy.statmech.LinearRotor method), 167
getEntropy() (rmgpy.statmech.NonlinearRotor method), 168
getEntropy() (rmgpy.statmech.SphericalTopRotor method), 171
getEntropy() (rmgpy.thermo.NASA method), 184
getEntropy() (rmgpy.thermo.NASA Polynomial method), 187
getEntropy() (rmgpy.thermo.ThermoData method), 179
getEntropy() (rmgpy.thermo.Wilhoit method), 182
getEntropyOfReaction() (rmgpy.data.kinetics.DepositoryReaction method), 18
getEntropyOfReaction() (rmgpy.data.kinetics.LibraryReaction method), 38
getEntropyOfReaction() (rmgpy.data.kinetics.TemplateReaction method), 54
getEntropyOfReaction() (rmgpy.reaction.Reaction method), 142
getEntropyOfReaction() (rmgpy.rmg.pdep.PDepReaction method), 154
getEquilibriumConstant() (rmgpy.rmg.pdep.PDepReaction method), 154
getEquilibriumConstants() (rmgpy.data.kinetics.DepositoryReaction method), 19
getEquilibriumConstants() (rmgpy.data.kinetics.LibraryReaction method), 38
getEquilibriumConstants() (rmgpy.data.kinetics.TemplateReaction method), 54
getEquilibriumConstants() (rmgpy.reaction.Reaction method), 142
getEquilibriumConstants() (rmgpy.rmg.pdep.PDepReaction method), 154
getFeatures() (rmgpy.molecule.AtomType method), 89
getFilePath() (rmgpy.qm.gaussian.GaussianMol method), 125
getFilePath() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
getFilePath() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
getFilePath() (rmgpy.qm.molecule.Geometry method), 120
getFilePath() (rmgpy.qm.molecule.QMMolecule method), 121
getFilePath() (rmgpy.qm.mopac.MopacMol method), 130
getFilePath() (rmgpy.qm.mopac.MopacMolPM3 method), 132
getFilePath() (rmgpy.qm.mopac.MopacMolPM6 method), 133
getFilePath() (rmgpy.qm.mopac.MopacMolPM7 method), 135
generateFingerprint() (rmgpy.molecule.Molecule method), 95
getFormula() (rmgpy.molecule.Molecule method), 95
getForwardReactionForFamilyEntry() (rmgpy.data.kinetics.KineticsDatabase method), 21
generateFreeEnergiesOfReaction() (rmgpy.data.kinetics.DepositoryReaction method), 19
generateFreeEnergiesOfReaction() (rmgpy.data.kinetics.LibraryReaction method), 38
generateFreeEnergiesOfReaction() (rmgpy.data.kinetics.TemplateReaction method), 54
generateFreeEnergiesOfReaction() (rmgpy.reaction.Reaction method), 142
generateFreeEnergiesOfReaction() (rmgpy.rmg.pdep.PDepReaction method), 154
getFreeEnergy() (rmgpy.pdep.Configuration method), 112
getFreeEnergy() (rmgpy.rmg.model.Species method), 156
getFreeEnergy() (rmgpy.species.Species method), 162
getFreeEnergy() (rmgpy.species.TransitionState method), 163
getFreeEnergy() (rmgpy.statmech.Conformer method), 177
getFreeEnergy() (rmgpy.thermo.NASA method), 184
getFreeEnergy() (rmgpy.thermo.NASAPolynomial method), 187
getFreeEnergy() (rmgpy.thermo.ThermoData method), 179
getFreeEnergy() (rmgpy.thermo.Wilhoit method), 182
getFreeEnergyOfReaction() (rmgpy.data.kinetics.DepositoryReaction method), 19
getFreeEnergyOfReaction() (rmgpy.data.kinetics.LibraryReaction method), 38
getFreeEnergyOfReaction() (rmgpy.data.kinetics.TemplateReaction method), 54
getFreeEnergyOfReaction() (rmgpy.reaction.Reaction method), 142
getFreeEnergyOfReaction() (rmgpy.rmg.pdep.PDepReaction method), 154
getFrequency() (rmgpy.statmech.HinderedRotor method), 174
getFrequencyGroups() (rmgpy.data.statmech.StatmechGroups method), 48
getHamiltonian() (rmgpy.statmech.HinderedRotor method), 174
getHeatCapacity() (in module rmgpy.statmech.schrodinger), 176
getHeatCapacity() (rmgpy.pdep.Configuration method), 112
getHeatCapacity() (rmgpy.rmg.model.Species method), 156
getHeatCapacity() (rmgpy.species.Species method), 162
getHeatCapacity() (rmgpy.species.TransitionState method), 163
getHeatCapacity() (rmgpy.statmech.Conformer method), 177
getHeatCapacity() (rmgpy.statmech.HarmonicOscillator method), 172
getHeatCapacity() (rmgpy.statmech.HinderedRotor method), 174
getHeatCapacity() (rmgpy.statmech.IdealGasTranslation method), 166
getHeatCapacity() (rmgpy.statmech.KRotor method), 169
getHeatCapacity() (rmgpy.statmech.LinearRotor method), 167
getHeatCapacity() (rmgpy.statmech.NonlinearRotor method), 168
getHeatCapacity() (rmgpy.statmech.SphericalTopRotor method), 171
getHeatCapacity() (rmgpy.thermo.NASA method), 184
getHeatCapacity() (rmgpy.thermo.NASAPolynomial method), 187
getHeatCapacity() (rmgpy.thermo.ThermoData method), 179
getHeatCapacity() (rmgpy.thermo.Wilhoit method), 182
getInChiKeyAug() (rmgpy.qm.gaussian.GaussianMol method), 125
getInChiKeyAug() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
getInChiKeyAug() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
getInChiKeyAug() (rmgpy.qm.molecule.QMMolecule method), 121
getInChiKeyAug() (rmgpy.qm.mopac.MopacMol method), 130
getInChiKeyAug() (rmgpy.qm.mopac.MopacMolPM3 method), 132
getInChiKeyAug() (rmgpy.qm.mopac.MopacMolPM6 method), 133
getInChiKeyAug() (rmgpy.qm.mopac.MopacMolPM7 method), 135
getInternalReducedMomentOfInertia() (rmgpy.cantherm.geometry.Geometry method), 5
getInternalReducedMomentOfInertia() (rmgpy.statmech.Conformer method), 177
getKinetics() (rmgpy.data.kinetics.KineticsFamily method), 26
getKineticsForTemplate() (rmgpy.data.kinetics.KineticsFamily method), 26
getKineticsFromDepository() (rmgpy.data.kinetics.KineticsFamily method), 26
getLabeledAtom() (rmgpy.molecule.Group method), 102
getLabeledAtom() (rmgpy.molecule.Molecule method), 95
getLabeledAtoms() (rmgpy.molecule.Group method), 102
getLabeledAtoms() (rmgpy.molecule.Molecule method), 95
getLabeledReactantsAndProducts() (rmgpy.data.kinetics.KineticsFamily method), 27
getLeakBranchingRatios() (rmgpy.rmg.pdep.PDepNetwork method), 151
getLeakCoefficient() (rmgpy.rmg.pdep.PDepNetwork method), 151
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLevelDegeneracy()</td>
<td>rmgpy.statmech.HinderedRotor method</td>
<td>174</td>
</tr>
<tr>
<td>getLevelDegeneracy()</td>
<td>rmgpy.statmech.KRotor method</td>
<td>169</td>
</tr>
<tr>
<td>getLevelDegeneracy()</td>
<td>rmgpy.statmech.LinearRotor method</td>
<td>167</td>
</tr>
<tr>
<td>getLevelDegeneracy()</td>
<td>rmgpy.statmech.SphericalTopRotor method</td>
<td>171</td>
</tr>
<tr>
<td>getLevelEnergy()</td>
<td>rmgpy.statmech.HinderedRotor method</td>
<td>174</td>
</tr>
<tr>
<td>getLevelEnergy()</td>
<td>rmgpy.statmech.KRotor method</td>
<td>169</td>
</tr>
<tr>
<td>getLevelEnergy()</td>
<td>rmgpy.statmech.LinearRotor method</td>
<td>167</td>
</tr>
<tr>
<td>getLevelEnergy()</td>
<td>rmgpy.statmech.SphericalTopRotor method</td>
<td>171</td>
</tr>
<tr>
<td>getLists()</td>
<td>rmgpy.rmg.model.CoreEdgeReactionModel method</td>
<td>145</td>
</tr>
<tr>
<td>getMaximumLeakSpecies()</td>
<td>rmgpy.rmg.pdep.PDepNetwork method</td>
<td>151</td>
</tr>
<tr>
<td>getModelSize()</td>
<td>rmgpy.rmg.model.CoreEdgeReactionModel method</td>
<td>145</td>
</tr>
<tr>
<td>getMolecularWeight()</td>
<td>rmgpy.molecule.Molecule method</td>
<td>95</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.gaussian.GaussianMol method</td>
<td>125</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.gaussian.GaussianMolPM3 method</td>
<td>126</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.gaussian.GaussianMolPM6 method</td>
<td>128</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.molecule.QMMolecule method</td>
<td>122</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.mopac.MopacMol method</td>
<td>130</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.mopac.MopacMolPM3 method</td>
<td>132</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.mopac.MopacMolPM6 method</td>
<td>133</td>
</tr>
<tr>
<td>getMolFilePathForCalculation()</td>
<td>rmgpy.qm.mopac.MopacMolPM7 method</td>
<td>135</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.schrodinger</td>
<td>176</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.rmg.model.Species method</td>
<td>156</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.species.Species method</td>
<td>162</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.species.TransitionState method</td>
<td>163</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.Conformer method</td>
<td>177</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.HarmonicOscillator method</td>
<td>172</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.HinderedRotor method</td>
<td>174</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.IdealGasTranslation method</td>
<td>166</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.KRotor method</td>
<td>169</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.LinearRotor method</td>
<td>167</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.NonlinearRotor method</td>
<td>168</td>
</tr>
<tr>
<td>getPartitionFunction()</td>
<td>rmgpy.statmech.SphericalTopRotor method</td>
<td>177</td>
</tr>
<tr>
<td>getMonocyclicRings()</td>
<td>rmgpy.molecule.Group method</td>
<td>102</td>
</tr>
<tr>
<td>getMonocyclicRings()</td>
<td>rmgpy.molecule.Molecule method</td>
<td>95</td>
</tr>
<tr>
<td>getNetCharge()</td>
<td>rmgpy.molecule.Molecule method</td>
<td>95</td>
</tr>
<tr>
<td>getNumAtoms()</td>
<td>rmgpy.molecule.Molecule method</td>
<td>95</td>
</tr>
<tr>
<td>getNumberDegreesOfFreedom()</td>
<td>rmgpy.statmech.Conformer method</td>
<td>177</td>
</tr>
<tr>
<td>getNumberOfRadicalElectrons()</td>
<td>rmgpy.molecule.Molecule method</td>
<td>96</td>
</tr>
<tr>
<td>getOtherVertex()</td>
<td>rmgpy.molecule.Bond method</td>
<td>92</td>
</tr>
<tr>
<td>getOtherVertex()</td>
<td>rmgpy.molecule.graph.Edge method</td>
<td>86</td>
</tr>
<tr>
<td>getOtherVertex()</td>
<td>rmgpy.molecule.GroupBond method</td>
<td>100</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.gaussian.GaussianMol method</td>
<td>125</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.gaussian.GaussianMolPM3 method</td>
<td>126</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.gaussian.GaussianMolPM6 method</td>
<td>128</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.mopac.Mopac method</td>
<td>129</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.mopac.MopacMol method</td>
<td>130</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.mopac.MopacMolPM3 method</td>
<td>132</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.mopac.MopacMolPM6 method</td>
<td>133</td>
</tr>
<tr>
<td>getParser()</td>
<td>rmgpy.qm.mopac.MopacMolPM7 method</td>
<td>135</td>
</tr>
<tr>
<td>getMomentOfInertiaTensor()</td>
<td>rmgpy.cantherm.geometry.Geometry method</td>
<td>5</td>
</tr>
<tr>
<td>getMomentOfInertiaTensor()</td>
<td>rmgpy.statmech.Conformer method</td>
<td>177</td>
</tr>
<tr>
<td>getMonocyclicRings()</td>
<td>rmgpy.molecule.graph.Graph method</td>
<td>87</td>
</tr>
</tbody>
</table>
getPolycyclicRings() (rmgpy.molecule.graph.Graph method), 171
getPolycyclicRings() (rmgpy.molecule.Group method), 102
getPolycyclicRings() (rmgpy.molecule.Molecule method), 96
getPossibleStructures() (rmgpy.data.base.LogicOr method), 39
getPotential() (rmgpy.statmech.HinderedRotor method), 174
getPrincipalMomentsOfInertia() (rmgpy.cantherm.geometry.Geometry method), 5
getPrincipalMomentsOfInertia() (rmgpy.statmech.Conformer method), 177
getRadicalAtoms() (rmgpy.molecule.Molecule method), 96
getRadicalCount() (rmgpy.molecule.Molecule method), 96
getRateCoefficient() (rmgpy.data.kinetics.DepositoryReaction method), 19
getRateCoefficient() (rmgpy.data.kinetics.LibraryReaction method), 38
getRateCoefficient() (rmgpy.data.kinetics.TemplateReaction method), 54
getRateCoefficient() (rmgpy.kinetics.Arrhenius method), 67
getRateCoefficient() (rmgpy.kinetics.Chebyshev method), 75
getRateCoefficient() (rmgpy.kinetics.KineticsData method), 65
getRateCoefficient() (rmgpy.kinetics.Lindemann method), 78
getRateCoefficient() (rmgpy.kinetics.MultiArrhenius method), 68
getRateCoefficient() (rmgpy.kinetics.MultiPDepArrhenius method), 73
getRateCoefficient() (rmgpy.kinetics.PDepArrhenius method), 71
getRateCoefficient() (rmgpy.kinetics.PDepKineticsData method), 69
getRateCoefficient() (rmgpy.kinetics.ThirdBody method), 76
getRateCoefficient() (rmgpy.kinetics.Troe method), 81
getRateCoefficient() (rmgpy.reaction.Reaction method), 142
getRateCoefficient() (rmgpy.rmg.pdep.PDepReaction method), 154
getRateRule() (rmgpy.data.kinetics.KineticsFamily method), 27
getReactionPairs() (rmgpy.data.kinetics.KineticsFamily method), 27
getReactionTemplate() (rmgpy.data.kinetics.KineticsGroups method), 27
getReactionTemplateLabels() (rmgpy.data.kinetics.KineticsFamily method), 30
getRefinedMolFilePath() (rmgpy.qm.molecule.Geometry method), 120
getReverse() (rmgpy.data.kinetics.ReactionRecipe method), 40
getRingGroupsFromComments() (rmgpy.data.thermo.ThermoDatabase method), 56
getRootTemplate() (rmgpy.data.kinetics.KineticsFamily method), 27
getSmallestSetOfSmallestRings() (rmgpy.molecule.graph.Graph method), 87
getSmallestSetOfSmallestRings() (rmgpy.molecule.Group method), 102
getSource() (rmgpy.data.kinetics.DepositoryReaction method), 19
getSource() (rmgpy.data.kinetics.LibraryReaction method), 38
getSource() (rmgpy.data.kinetics.TemplateReaction method), 54
getSource() (rmgpy.data.kinetics.KineticsFamily method), 27
getSource() (rmgpy.data.kinetics.KineticsGroups method), 30
getSource() (rmgpy.data.kinetics.KineticsLibrary method), 32
getSource() (rmgpy.data.kinetics.KineticsRules method), 35
getSource() (rmgpy.data.statmech.StatmechDepository method), 42
getSource() (rmgpy.data.statmech.StatmechGroups method), 48
getSource() (rmgpy.data.statmech.StatmechLibrary method), 51
getSource() (rmgpy.data.thermo.ThermoDepository method), 58
getSource() (rmgpy.data.thermo.ThermoGroups method), 60

Index

207
getSpecies() (rmgpy.data.thermo.ThermoLibrary method), 62
getSpecies() (rmgpy.rmg.model.CoreEdgeReactionModel method), 146
getSpeciesIdentifier() (in module rmgpy.chemkin), 12
getStatmechData() (rmgpy.data.statmech.StatmechDatabase method), 41
getStatmechData() (rmgpy.data.statmech.StatmechGroups method), 48
getStatmechDataFromDepository() (rmgpy.data.statmech.StatmechDatabase method), 41
getStatmechDataFromGroups() (rmgpy.data.statmech.StatmechDatabase method), 41
getStatmechDataFromLibrary() (rmgpy.data.statmech.StatmechDatabase method), 41
getStoichiometricCoefficient() (rmgpy.data.kinetics.DepositoryReaction method), 19
getStoichiometricCoefficient() (rmgpy.data.kinetics.LibraryReaction method), 38
getStoichiometricCoefficient() (rmgpy.data.kinetics.TemplateReaction method), 54
getStoichiometricCoefficient() (rmgpy.data.kinetics.Reaction method), 142
getStoichiometricCoefficient() (rmgpy.rmg.pdep.PDepReaction method), 154
getStoichiometryMatrix() (rmgpy.rmg.model.CoreEdgeReactionModel method), 146
getSumOfStates() (in module rmgpy.statmech.schrodinger), 176
getSumOfStates() (rmgpy.rmg.model.Species method), 156
getSumOfStates() (rmgpy.species.Species method), 162
getSumOfStates() (rmgpy.species.TransitionState method), 163
getSumOfStates() (rmgpy.statmech.Conformer method), 177
getSumOfStates() (rmgpy.statmech.HarmonicOscillator method), 172
getSumOfStates() (rmgpy.statmech.HinderedRotor method), 174
getSumOfStates() (rmgpy.statmech.IdealGasTranslation method), 166
getSumOfStates() (rmgpy.statmech.KRotor method), 169
getSumOfStates() (rmgpy.statmech.LinearRotor method), 167
getSumOfStates() (rmgpy.statmech.NonlinearRotor method), 168
getSumOfStates() (rmgpy.statmech.SphericalTopRotor method), 171
getSymmetricTopRotors() (rmgpy.statmech.Conformer method), 177
getSymmetryNumber() (rmgpy.molecule.Molecule method), 96
getSymmetryNumber() (rmgpy.rmg.model.Species method), 156
getSymmetryNumber() (rmgpy.species.Species method), 162
getThermoData() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoData() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoData() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoData() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoData() (rmgpy.qm.main.QMCalculator method), 120
getThermoData() (rmgpy.rmg.model.Species method), 156
getThermoData() (rmgpy.species.Species method), 162
getThermoDataFromDepository() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoDataFromGroups() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoDataFromLibraries() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoDataFromLibrary() (rmgpy.data.thermo.ThermoDatabase method), 56
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMol method), 125
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
getThermoFilePath() (rmgpy.qm.molecule.QMMolecule method), 122
getThermoFilePath() (rmgpy.qm.mopac.MopacMol method), 130
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM3 method), 132
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM6 method), 134
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM7 method), 135
getTotalMass() (rmgpy.cantherm.geometry.Geometry method), 5
getTotalMass() (rmgpy.statmech.Conformer method), 177
getTrainingDepository() (rmgpy.data.kinetics.KineticsFamily method), 27
getTransportData() (rmgpy.rmg.model.Species method), 156
getTransportData() (rmgpy.species.Species method), 162
getUncertainty() (rmgpy.quantity.ArrayQuantity method), 139
getUncertainty() (rmgpy.quantity.ScalarQuantity method), 138
getUncertaintyType() (rmgpy.quantity.ArrayQuantity method), 139
getUncertaintyType() (rmgpy.quantity.ScalarQuantity method), 138
getURL() (rmgpy.data.kinetics.DepositoryReaction method), 19
getURL() (rmgpy.data.kinetics.LibraryReaction method), 38
getURL() (rmgpy.data.kinetics.TemplateReaction method), 54
getURL() (rmgpy.molecule.Molecule method), 96
getURL() (rmgpy.reaction.Reaction method), 142
getURL() (rmgpy.rmg.pdep.PDepReaction method), 154
getValue() (rmgpy.quantity.ArrayQuantity method), 139
getValue() (rmgpy.quantity.ScalarQuantity method), 138
Graph (class in rmgpy.molecule.graph), 86
groundStateDegeneracy (rmgpy.qm.qmdata.QMData attribute), 122
Group (class in rmgpy.molecule), 100
GroupAtom (class in rmgpy.molecule), 99
GroupBond (class in rmgpy.molecule), 100
GroupFrequencies (class in rmgpy.data.statmech), 20
hasTemplate() (rmgpy.data.kinetics.TemplateReaction method), 54
hasTemplate() (rmgpy.reaction.Reaction method), 142
hasTemplate() (rmgpy.rmg.pdep.PDepReaction method), 154
hasThermo() (rmgpy.pdep.Configuration method), 112
hasThermo() (rmgpy.rmg.model.Species method), 156
hasThermo() (rmgpy.species.Species method), 162
hasVertex() (rmgpy.molecule.graph.Graph method), 87
hasVertex() (rmgpy.molecule.Group method), 102
hasVertex() (rmgpy.molecule.Molecule method), 96
highPlimit (rmgpy.kinetics.Chebyshev attribute), 75
highPlimit (rmgpy.kinetics.Lindemann attribute), 78
highPlimit (rmgpy.kinetics.MultiPDepArrhenius attribute), 73
highPlimit (rmgpy.kinetics.PDepArrhenius attribute), 71
highPlimit (rmgpy.kinetics.PDepKineticsData attribute), 69
highPlimit (rmgpy.kinetics.ThirdBody attribute), 77
highPlimit (rmgpy.kinetics.Troe attribute), 81
HinderedRotor (class in rmgpy.statmech), 172
hinderedRotor_d_heatCapacity_d_barr() (in module rmgpy.data.statmechfit), 44
hinderedRotor_d_heatCapacity_d_freq() (in module rmgpy.data.statmechfit), 44
hinderedRotor_heatCapacity() (in module rmgpy.data.statmechfit), 44
HinderedRotor_HeatCapacity() (in module rmgpy.data.statmechfit), 44
IdealGasTranslation (class in rmgpy.statmech), 165
incrementLonePairs() (rmgpy.molecule.Atom method), 91
incrementOrder() (rmgpy.molecule.Bond method), 93
increment Radical() (rmgpy.molecule.Atom method), 91
inertia (rmgpy.statmech.HinderedRotor attribute), 174
inertia (rmgpy.statmech.KRotor attribute), 170
inertia (rmgpy.statmech.LinearRotor attribute), 167
inertia (rmgpy.statmech.NonlinearRotor attribute), 168
inertia (rmgpy.statmech.SphericalTopRotor attribute), 171
inflate() (rmgpy.rmg.model.CoreEdgeReactionModel method), 146
initialize() (rmgpy.data.statmechfit.DirectFit method), 45
initialize() (rmgpy.data.statmechfit.PseudoFit method), 47
initialize() (rmgpy.data.statmechfit.PseudoRotorFit method), 46
initialize() (rmgpy.pdep.Network method), 113
initialize() (rmgpy.qm.gaussian.GaussianMol method), 125
initialize() (rmgpy.qm.gaussian.GaussianMolPM3 method), 126
initialize() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
isIdenticalTo() (rmgpy.kinetics.Arrhenius method), 67
isIdenticalTo() (rmgpy.kinetics.Chebyshev method), 75
isIdenticalTo() (rmgpy.kinetics.KineticsData method), 65
isIdenticalTo() (rmgpy.kinetics.Lindemann method), 68
isIdenticalTo() (rmgpy.kinetics.MultiArrhenius method), 68
isIdenticalTo() (rmgpy.kinetics.PDepArrhenius method), 71
isIdenticalTo() (rmgpy.kinetics.PDepKineticsData method), 70
isIdenticalTo() (rmgpy.kinetics.ThirdBody method), 77
isIdenticalTo() (rmgpy.kinetics.Troe method), 79

isIdenticalTo() (rmgpy.thermo.NASA method), 184
isIdenticalTo() (rmgpy.thermo.NASAPolynomial method), 187

isIsomerization() (rmgpy.data.kinetics.DepositoryReaction method), 19
isIsomerization() (rmgpy.data.kinetics.LibraryReaction method), 39
isIsomerization() (rmgpy.data.kinetics.TemplateReaction method), 55
isIsomerization() (rmgpy.reaction.Reaction method), 143
isIsomerization() (rmgpy.rmg.pdep.PDepReaction method), 154
isIsomorphic() (rmgpy.data.kinetics.DepositoryReaction method), 19
isIsomorphic() (rmgpy.data.kinetics.LibraryReaction method), 39
isIsomorphic() (rmgpy.data.kinetics.TemplateReaction method), 55
isIsomorphic() (rmgpy.molecule.graph.Graph method), 87
isIsomorphic() (rmgpy.molecule.Group method), 102
isIsomorphic() (rmgpy.molecule.Molecule method), 97
isIsomorphic() (rmgpy.molecule.vf2.VF2 method), 88
isIsomorphic() (rmgpy.reaction.Reaction method), 143
isIsomorphic() (rmgpy.rmg.model.Species method), 156
isIsomorphic() (rmgpy.rmg.pdep.PDepReaction method), 154
isIsomorphic() (rmgpy.species.Species method), 162
isLinear() (rmgpy.molecule.Molecule method), 97
isMappingValid() (rmgpy.molecule.graph.Graph method), 87
isMappingValid() (rmgpy.molecule.Group method), 102
isMappingValid() (rmgpy.molecule.Molecule method), 97
isMoleculeForbidden() (rmgpy.data.kinetics.KineticsFamily method), 27
isNitrogen() (rmgpy.molecule.Atom method), 92
isNonHydrogen() (rmgpy.molecule.Atom method), 92
isOxygen() (rmgpy.molecule.Atom method), 92
isOxygen() (rmgpy.molecule.GroupAtom method), 99
isPressureDependent() (rmgpy.kinetics.Arrhenius method), 67
isPressureDependent() (rmgpy.kinetics.Chebyshev method), 75
isPressureDependent() (rmgpy.kinetics.KineticsData method), 65
isPressureDependent() (rmgpy.kinetics.Lindemann method), 79
isPressureDependent() (rmgpy.kinetics.MultiArrhenius method), 68
isPressureDependent() (rmgpy.kinetics.MultiPDepArrhenius method), 73
isPressureDependent() (rmgpy.kinetics.PDepArrhenius method), 71
isPressureDependent() (rmgpy.kinetics.PDepKineticsData method), 70
isPressureDependent() (rmgpy.kinetics.ThirdBody method), 77
isPressureDependent() (rmgpy.kinetics.Troe method), 79
isPressureValid() (rmgpy.kinetics.Chebyshev method), 81
isPressureValid() (rmgpy.kinetics.Lindemann method), 79
isPressureValid() (rmgpy.kinetics.MultiPDepArrhenius method), 73
isPressureValid() (rmgpy.kinetics.PDepArrhenius method), 71
isPressureValid() (rmgpy.kinetics.PDepKineticsData method), 70
isPressureValid() (rmgpy.kinetics.ThirdBody method), 77
isPressureValid() (rmgpy.kinetics.Troe method), 79
isRadical() (rmgpy.molecule.Molecule method), 97
isSilicon() (rmgpy.molecule.Atom method), 92
isSimilarTo() (rmgpy.kinetics.Arrhenius method), 67
isSimilarTo() (rmgpy.kinetics.Chebyshev method), 75
isSimilarTo() (rmgpy.kinetics.KineticsData method), 65
isSimilarTo() (rmgpy.kinetics.Lindemann method), 79
isSimilarTo() (rmgpy.kinetics.MultiArrhenius method), 68
isSimilarTo() (rmgpy.kinetics.MultiPDepArrhenius method), 73
isSimilarTo() (rmgpy.kinetics.PDepArrhenius method), 71
isSimilarTo() (rmgpy.kinetics.PDepKineticsData method), 70
isSimilarTo() (rmgpy.kinetics.ThirdBody method), 77
isSimilarTo() (rmgpy.kinetics.Troe method), 79
isSingle() (rmgpy.molecule.Bond method), 93

Index 211
Index
method), 23
loadOldLibrary() (rmgpy.data.kinetics.KineticsFamily
method), 28
loadOldLibrary() (rmgpy.data.kinetics.KineticsGroups
method), 31
loadOldLibrary() (rmgpy.data.kinetics.KineticsLibrary
method), 33
loadOldLibrary() (rmgpy.data.kinetics.KineticsRules
method), 35
loadOldLibrary() (rmgpy.data.statmech.StatmechDepository
method), 42
loadOldLibrary() (rmgpy.data.statmech.StatmechGroups
method), 49
loadOldLibrary() (rmgpy.data.statmech.StatmechLibrary
method), 51
loadOldLibrary() (rmgpy.data.thermo.ThermoDepository
method), 58
loadOldLibrary() (rmgpy.data.thermo.ThermoGroups
method), 61
loadOldLibrary() (rmgpy.data.thermo.ThermoLibrary
method), 63
loadOldTemplate() (rmgpy.data.kinetics.KineticsFamily
method), 28
loadOldTree() (rmgpy.data.base.Database method), 16
loadOldTree() (rmgpy.data.kinetics.KineticsDepository
method), 23
loadOldTree() (rmgpy.data.kinetics.KineticsFamily
method), 28
loadOldTree() (rmgpy.data.kinetics.KineticsGroups
method), 31
loadOldTree() (rmgpy.data.kinetics.KineticsLibrary
method), 33
loadOldTree() (rmgpy.data.kinetics.KineticsRules
method), 35
loadOldTree() (rmgpy.data.statmech.StatmechDepository
method), 43
loadOldTree() (rmgpy.data.statmech.StatmechGroups
method), 49
loadOldTree() (rmgpy.data.statmech.StatmechLibrary
method), 51
loadOldTree() (rmgpy.data.thermo.ThermoDepository
method), 58
loadOldTree() (rmgpy.data.thermo.ThermoGroups
method), 61
loadOldTree() (rmgpy.data.thermo.ThermoLibrary
method), 63
loadRecipe() (rmgpy.data.kinetics.KineticsFamily
method), 28
loadRecommendedFamiliesList()
(rmgpy.data.kinetics.KineticsDatabase
method), 22
loadRestartFile() (rmgpy.rmg.main.RMG method), 149
loadRMGJavaInput() (rmgpy.rmg.main.RMG method), 149
loadScanEnergies() (rmgpy.cantherm.gaussian.GaussianLog
method), 5
loadSpeciesDictionary() (in module rmgpy.chemkin), 11
loadTemplate() (rmgpy.data.kinetics.KineticsFamily
method), 28
loadThermoData() (rmgpy.qm.gaussian.GaussianMol
method), 125
loadThermoData() (rmgpy.qm.gaussian.GaussianMolPM3
method), 127
loadThermoData() (rmgpy.qm.gaussian.GaussianMolPM6
method), 128
loadThermoData() (rmgpy.qm.molecule.QMMolecule
method), 122
loadThermoData() (rmgpy.qm.mopac.MopacMol
method), 130
loadThermoData() (rmgpy.qm.mopac.MopacMolPM3
method), 132
loadThermoData() (rmgpy.qm.mopac.MopacMolPM6
method), 134
loadThermoData() (rmgpy.qm.mopac.MopacMolPM7
method), 135
loadThermoInput() (rmgpy.rmg.main.RMG method), 149
loadTransportFile() (in module rmgpy.chemkin), 11
loadZeroPointEnergy() (rmgpy.cantherm.gaussian.GaussianLog
method), 5
logConversions() (rmgpy.solver.ReactionSystem
method), 158
logConversions() (rmgpy.solver.SimpleReactor method), 159
logFooter() (rmgpy.cantherm.CanTherm method), 6
logHeader() (rmgpy.cantherm.CanTherm method), 6
logHeader() (rmgpy.rmg.main.RMG method), 149
LogicAnd (class in rmgpy.data.base), 39
LogicNode (class in rmgpy.data.base), 39
LogicOr (class in rmgpy.data.base), 39
logRates() (rmgpy.solver.ReactionSystem method), 158
logRates() (rmgpy.solver.SimpleReactor method), 159
makeLogicNode() (in module rmgpy.data.base), 40
makeNewPDepReaction()
(rmgpy.rmg.model.CoreEdgeReactionModel
method), 146
makeNewReaction() (rmgpy.rmg.model.CoreEdgeReactionModel
method), 146
makeNewSpecies() (rmgpy.rmg.model.CoreEdgeReactionModel
method), 146
makeProfileGraph() (in module rmgpy.rmg.main), 149
mapDensitiesOfStates() (rmgpy.pdep.Network method), 114
mapDensitiesOfStates() (rmgpy.rmg.pdep.PDepNetwork
method), 151
mapDensityOfStates() (rmgpy.pdep.Configuration
method), 112
mapSumOfStates() (rmgpy.pdep.Configuration method), 112
markChemkinDuplicates() (rmgpy.rmg.model.CoreEdgeReactionModel method), 146
markDuplicateReactions() (in module rmgpy.chemkin), 12
markValidDuplicates() (rmgpy.data.kinetics.KineticsLibrary method), 33
mass (rmgpy.statmech.Conformer attribute), 177
mass (rmgpy.statmech.IdealGasTranslation attribute), 166
matchesMolecules() (rmgpy.data.kinetics.DepositoryReaction method), 19
matchesMolecules() (rmgpy.data.kinetics.LibraryReaction method), 39
matchesMolecules() (rmgpy.data.kinetics.TemplateReaction method), 55
matchesMolecules() (rmgpy.reaction.Reaction method), 143
matchesMolecules() (rmgpy.rmg.pdep.PDepReaction method), 155
matchLogicOr() (rmgpy.data.base.LogicOr method), 40
matchNodeToChild() (rmgpy.data.base.Database method), 16
matchNodeToChild() (rmgpy.data.kinetics.KineticsDepository method), 23
matchNodeToChild() (rmgpy.data.kinetics.KineticsFamily method), 28
matchNodeToChild() (rmgpy.data.kinetics.KineticsGroups method), 31
matchNodeToChild() (rmgpy.data.kinetics.KineticsLibrary method), 33
matchNodeToChild() (rmgpy.data.kinetics.KineticsRules method), 35
matchNodeToChild() (rmgpy.data.statmech.StatmechDepository method), 43
matchNodeToChild() (rmgpy.data.statmech.StatmechGroups method), 49
matchNodeToChild() (rmgpy.data.statmech.StatmechLibrary method), 51
matchNodeToChild() (rmgpy.data.thermo.ThermoDepository method), 58
matchNodeToChild() (rmgpy.data.thermo.ThermoGroups method), 61
matchNodeToChild() (rmgpy.data.thermo.ThermoLibrary method), 63
matchNodeToNode() (rmgpy.data.kinetics.KineticsDepository method), 33
matchNodeToNode() (rmgpy.data.kinetics.KineticsRules method), 35
matchNodeToNode() (rmgpy.data.statmech.StatmechDepository method), 43
matchNodeToNode() (rmgpy.data.statmech.StatmechGroups method), 49
matchNodeToNode() (rmgpy.data.statmech.StatmechLibrary method), 51
matchNodeToNode() (rmgpy.data.thermo.ThermoDepository method), 58
matchNodeToNode() (rmgpy.data.thermo.ThermoGroups method), 61
matchNodeToNode() (rmgpy.data.thermo.ThermoLibrary method), 63
matchNodeToStructure() (rmgpy.data.base.Database method), 16
matchNodeToStructure() (rmgpy.data.kinetics.KineticsDepository method), 23
matchNodeToStructure() (rmgpy.data.kinetics.KineticsFamily method), 28
matchNodeToStructure() (rmgpy.data.kinetics.KineticsGroups method), 31
matchNodeToStructure() (rmgpy.data.kinetics.KineticsLibrary method), 33
matchNodeToStructure() (rmgpy.data.statmech.StatmechDepository method), 43
matchNodeToStructure() (rmgpy.data.statmech.StatmechGroups method), 49
matchNodeToStructure() (rmgpy.data.statmech.StatmechLibrary method), 51
matchNodeToStructure() (rmgpy.data.thermo.ThermoDepository method), 58
matchNodeToStructure() (rmgpy.data.thermo.ThermoGroups method), 61
matchNodeToStructure() (rmgpy.data.thermo.ThermoLibrary method), 63
matchToStructure() (rmgpy.data.base.LogicAnd method), 39
matchToStructure() (rmgpy.data.base.LogicOr method), 40

Index
maxAttempts (rmgpy.qm.gaussian.GaussianMol attribute), 125
maxAttempts (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 127
maxAttempts (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 128
maxAttempts (rmgpy.qm.molecule.QMMolecule attribute), 122
maxAttempts (rmgpy.qm.mopac.MopacMol attribute), 131
maxAttempts (rmgpy.qm.mopac.MopacMolPM3 attribute), 132
maxAttempts (rmgpy.qm.mopac.MopacMolPM6 attribute), 134
maxAttempts (rmgpy.qm.mopac.MopacMolPM7 attribute), 135
maximumGrainSize (rmgpy.cantherm.PressureDependenceJob attribute), 9
merge() (rmgpy.molecule.graph.Graph method), 87
merge() (rmgpy.molecule.Group method), 103
merge() (rmgpy.molecule.Molecule method), 97
merge() (rmgpy.rmg.model.ReactionModel method), 147
modes (rmgpy.statmech.Conformer attribute), 178
Molecule (class in rmgpy.molecule), 93
MoleculeDrawer (class in rmgpy.molecule.draw), 107
Mopac (class in rmgpy.qm.mopac), 129
MopacMol (class in rmgpy.qm.mopac), 129
MopacMolPM3 (class in rmgpy.qm.mopac), 131
MopacMolPM6 (class in rmgpy.qm.mopac), 133
MopacMolPM7 (class in rmgpy.qm.mopac), 134
MultiArrhenius (class in rmgpy.kinetics), 67
MultiPDepArrhenius (class in rmgpy.kinetics), 72

n (rmgpy.kinetics.Arrhenius attribute), 67
n (rmgpy.pdep.SingleExponentialDown attribute), 110
NASA (class in rmgpy.thermo), 183
NASAPolynomial (class in rmgpy.thermo), 185
Network (class in rmgpy.pdep), 112
NonlinearRotor (class in rmgpy.statmech), 168
number (rmgpy.statmech.Conformer attribute), 178
numberOfAtoms (rmgpy.qm.qmdata.QMData attribute), 122

opticalIsomers (rmgpy.statmech.Conformer attribute), 178
outputFilePath (rmgpy.qm.gaussian.GaussianMol attribute), 125
outputFilePath (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 127
outputFilePath (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 128
outputFilePath (rmgpy.qm.molecule.QMMolecule attribute), 122
outputFilePath (rmgpy.qm.mopac.MopacMol attribute), 131
outputFilePath (rmgpy.qm.mopac.MopacMolPM3 attribute), 132
outputFilePath (rmgpy.qm.mopac.MopacMolPM6 attribute), 134
outputFilePath (rmgpy.qm.mopac.MopacMolPM7 attribute), 135

parse() (rmgpy.qm.gaussian.Gaussian method), 124
parse() (rmgpy.qm.gaussian.GaussianMol method), 125
parse() (rmgpy.qm.gaussian.GaussianMolPM3 method), 127
parse() (rmgpy.qm.gaussian.GaussianMolPM6 method), 128
parse() (rmgpy.qm.molecule.QMMolecule method), 122
parse() (rmgpy.qm.mopac.MopacMol method), 131
parse() (rmgpy.qm.mopac.MopacMolPM3 method), 132
parse() (rmgpy.qm.mopac.MopacMolPM6 method), 134
parse() (rmgpy.qm.mopac.MopacMolPM7 method), 135
parse() (rmgpy.qm.symmetry.SymmetryJob method), 123
parseCommandLineArguments() (rmgpy.cantherm.CanTherm method), 7
parseOldLibrary() (rmgpy.data.base.Database method), 16
parseOldLibrary() (rmgpy.data.kinetics.KineticsDepository method), 24
parseOldLibrary() (rmgpy.data.kinetics.KineticsFamily method), 28
parseOldLibrary() (rmgpy.data.kinetics.KineticsGroups method), 31
parseOldLibrary() (rmgpy.data.kinetics.KineticsLibrary method), 33
parseOldLibrary() (rmgpy.data.kinetics.KineticsRules method), 36
parseOldLibrary() (rmgpy.data.statmech.StatmechDepository method), 43
parseOldLibrary() (rmgpy.data.statmech.StatmechGroups method), 49
parseOldLibrary() (rmgpy.data.statmech.StatmechLibrary method), 52
parseOldLibrary() (rmgpy.data.thermo.ThermoDepository method), 59
parseOldLibrary() (rmgpy.data.thermo.ThermoGroups method), 61
parseOldLibrary() (rmgpy.data.thermo.ThermoLibrary method), 63
Pdata (rmgpy.kinetics.PDepKineticsData attribute), 69
PDepArrhenius (class in rmgpy.kinetics), 70
readThermoEntry() (in module rmgpy.chemkin), 11
reconstructKineticsFromSource()
    (rmgpy.data.kinetics.KineticsDatabase method), 22
register_listeners() (rmgpy.rmg.main.RMG method), 149
registerReaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 147
removeAtom() (rmgpy.molecule.Group method), 103
removeAtom() (rmgpy.molecule.Molecule method), 97
removeBond() (rmgpy.molecule.Group method), 103
removeBond() (rmgpy.molecule.Molecule method), 97
removeCommentFromLine() (in module rmgpy.chemkin), 11
removeEdge() (rmgpy.molecule.graph.Graph method), 87
removeEdge() (rmgpy.molecule.Molecule method), 97
removeGroup() (rmgpy.data.base.Database method), 16
removeGroup() (rmgpy.data.kinetics.KineticsDepository method), 24
removeGroup() (rmgpy.data.kinetics.KineticsFamily method), 28
removeGroup() (rmgpy.data.kinetics.KineticsGroups method), 31
removeGroup() (rmgpy.data.kinetics.KineticsLibrary method), 33
removeGroup() (rmgpy.data.kinetics.KineticsRules method), 36
removeGroup() (rmgpy.data.statmech.StatmechDepository method), 43
removeGroup() (rmgpy.data.statmech.StatmechGroups method), 50
removeGroup() (rmgpy.data.statmech.StatmechLibrary method), 52
removeGroup() (rmgpy.data.thermo.ThermoDepository method), 59
removeGroup() (rmgpy.data.thermo.ThermoGroups method), 61
removeGroup() (rmgpy.data.thermo.ThermoLibrary method), 63
removeSpeciesFromEdge()
    (rmgpy.rmg.model.CoreEdgeReactionModel method), 147
removeVertex() (rmgpy.molecule.graph.Graph method), 88
removeVertex() (rmgpy.molecule.Group method), 103
removeVertex() (rmgpy.molecule.Molecule method), 97
render() (rmgpy.molecule.draw.MoleculeDrawer method), 107
resetConnectivityValues() (rmgpy.molecule.Atom method), 92
resetConnectivityValues() (rmgpy.molecule.graph.Graph method), 88
resetConnectivityValues() (rmgpy.molecule.graph.Vertex method), 85
resetConnectivityValues() (rmgpy.molecule.Group method), 103
resetConnectivityValues() (rmgpy.molecule.GroupAtom method), 99
resetConnectivityValues() (rmgpy.molecule.Molecule method), 97
residual() (rmgpy.solver.ReactionSystem method), 158
residual() (rmgpy.solver.SimpleReactor method), 159
retrieve() (rmgpy.rmg.model.CoreEdgeReactionModel method), 147
retrieveNewSpecies() (rmgpy.rmg.model.CoreEdgeReactionModel method), 147
retrieveOriginalEntry() (rmgpy.data.kinetics.KineticsFamily method), 29
retrieveTemplate() (rmgpy.data.kinetics.KineticsFamily method), 29
reverseThisArrheniusRate()
    (rmgpy.data.kinetics.DepositoryReaction method), 20
reverseThisArrheniusRate()
    (rmgpy.data.kinetics.LibraryReaction method), 39
reverseThisArrheniusRate()
    (rmgpy.data.kinetics.TemplateReaction method), 55
reverseThisArrheniusRate() (rmgpy.reaction.Reaction method), 143
reverseThisArrheniusRate() (rmgpy.rmg.pdep.PDepReaction method), 155
RMG (class in rmgpy.rmg.main), 148
rmgpy.cantherm (module), 3
rmgpy.chemkin (module), 10
rmgpy.constants (module), 13
rmgpy.data (module), 13
rmgpy.chemkin (module), 10
rmgpy.chemkin (module), 10
rmgpy.kinetics (module), 64
rmgpy.molecule (module), 84
rmgpy.molecule.adjlist (module), 104
rmgpy.pdep (module), 108
rmgpy.qm (module), 118
rmgpy.quantity (module), 136
rmgpy.reaction (module), 140
rmgpy.rmg (module), 143
rmgpy.solver (module), 157
rmgpy.species (module), 160
rmgpy.statmech (module), 163
rmgpy.statmech.schrodinger (module), 175
rmgpy.thermo (module), 178
rotationalConstant (rmgpy.statmech.HinderedRotor attribute), 174
rotationalConstant (rmgpy.statmech.KRotor attribute), 170
rotationalConstant (rmgpy.statmech.LinearRotor attribute), 167
rotationalConstant (rmgpy.statmech.NonlinearRotor attribute), 168
rotationalConstant (rmgpy.statmech.SphericalTopRotor attribute), 171
run() (rmgpy.qm.symmetry.SymmetryJob method), 123
S
S0 (rmgpy.thermo.Wilhoit attribute), 181
S298 (rmgpy.thermo.ThermoData attribute), 179
saturate() (rmgpy.molecule.Molecule method), 97
save() (rmgpy.cantherm.KineticsJob method), 6
save() (rmgpy.cantherm.StatMechJob method), 9
save() (rmgpy.data.base.Database method), 17
save() (rmgpy.data.kinetics.KineticsDatabase method), 22
save() (rmgpy.data.kinetics.KineticsDepository method), 24
save() (rmgpy.data.kinetics.KineticsFamily method), 29
save() (rmgpy.data.kinetics.KineticsGroups method), 31
save() (rmgpy.data.kinetics.KineticsLibrary method), 33
save() (rmgpy.data.kinetics.KineticsRules method), 36
save() (rmgpy.data.statmech.StatmechDatabase method), 41
save() (rmgpy.data.statmech.StatmechDepository method), 43
save() (rmgpy.data.statmech.StatmechGroups method), 50
save() (rmgpy.data.statmech.StatmechLibrary method), 52
save() (rmgpy.data.thermo.ThermoDepository method), 59
save() (rmgpy.data.thermo.ThermoGroups method), 61
save() (rmgpy.data.thermo.ThermoLibrary method), 64
saveDiffHTML() (in module rmgpy.rmg.output), 150
saveEntry() (rmgpy.data.kinetics.KineticsDepository method), 24
saveEntry() (rmgpy.data.kinetics.KineticsFamily method), 29
saveEntry() (rmgpy.data.kinetics.KineticsLibrary method), 34
saveEntry() (rmgpy.data.kinetics.KineticsRules method), 36
saveEntry() (rmgpy.data.statmech.StatmechDepository method), 43
saveEntry() (rmgpy.data.statmech.StatmechFamily method), 50
saveEntry() (rmgpy.data.statmech.StatmechGroups method), 52
saveEntry() (rmgpy.data.statmech.StatmechLibrary method), 59
saveEntry() (rmgpy.data.thermo.ThermoDepository method), 61
saveEntry() (rmgpy.data.thermo.ThermoGroups method), 64
saveEntry() (rmgpy.data.thermo.ThermoLibrary method), 64
saveEverything() (rmgpy.rmg.main.RMG method), 149
saveFamilies() (rmgpy.data.kinetics.KineticsDatabase method), 22
saveGroups() (rmgpy.data.kinetics.KineticsFamily method), 29
saveGroups() (rmgpy.data.statmech.StatmechDatabase method), 41
saveGroups() (rmgpy.data.thermo.ThermoDatabase method), 57
saveHTMLFile() (in module rmgpy.chemkin), 12
saveInput() (rmgpy.rmg.main.RMG method), 149
saveInputFile() (in module rmgpy.rmg.input), 147
saveInputFile() (rmgpy.cantherm.PressureDependenceJob method), 9
saveJavaKineticsLibrary() (in module rmgpy.chemkin), 12
saveLibraries() (rmgpy.data.statmech.StatmechDatabase method), 22
saveSpeciesDictionary() (in module rmgpy.chemkin), 11
saveThermoData() (rmgpy.qm.gaussian.GaussianMol method), 125
saveThermoData() (rmgpy.qm.gaussian.GaussianMolPM3 method), 127
saveThermoData() (rmgpy.qm.gaussian.GaussianMolPM6 method), 129
saveThermoData() (rmgpy.qm.molecule.QMMolecule method), 122
saveThermoData() (rmgpy.qm.mopac.MopacMol method), 131
saveThermoData() (rmgpy.qm.mopac.MopacMolPM3 method), 132
saveThermoData() (rmgpy.qm.mopac.MopacMolPM6 method), 134
saveThermoData() (rmgpy.qm.mopac.MopacMolPM6 method), 136
saveTrainingReactions() (rmgpy.data.kinetics.KineticsFamily method), 29
saveTransportFile() (in module rmgpy.chemkin), 12
ScalarQuantity (class in rmgpy.quantity), 137
scriptAttempts (rmgpy.qm.gaussian.GaussianMol attribute), 125
scriptAttempts (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 127
scriptAttempts (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 129
scriptAttempts (rmgpy.qm.molecule.QMMolecule attribute), 122
scriptAttempts (rmgpy.qm.mopac.MopacMol attribute), 131
scriptAttempts (rmgpy.qm.mopac.MopacMolPM3 attribute), 132
scriptAttempts (rmgpy.qm.mopac.MopacMolPM6 attribute), 134
scriptAttempts (rmgpy.qm.mopac.MopacMolPM7 attribute), 136
searchRetrieveReactions() (rmgpy.rmg.model.CoreEdgeReactionModel method), 147
selectEnergyGrains() (rmgpy.pdep.Network method), 114
selectEnergyGrains() (rmgpy.rmg.pdep.PDepNetwork method), 151
selectPolynomial() (rmgpy.thermo.NASA method), 184
semiclassical (rmgpy.statmech.HinderedRotor attribute), 174
set_colliders() (rmgpy.solver.SimpleReactor method), 159
set_initial_derivative() (rmgpy.solver.SimpleReactor method), 160
setCanteraKinetics() (rmgpy.kinetics.Arrhenius method), 67
setCanteraKinetics() (rmgpy.kinetics.Chebyshev method), 75
setCanteraKinetics() (rmgpy.kinetics.KineticsData method), 66
setCanteraKinetics() (rmgpy.kinetics.Lindemann method), 79
setCanteraKinetics() (rmgpy.kinetics.MultiArrhenius method), 68
setCanteraKinetics() (rmgpy.kinetics.MultiPDepArrhenius method), 73
setCanteraKinetics() (rmgpy.kinetics.PDepArrhenius method), 72
setCanteraKinetics() (rmgpy.kinetics.PDepKineticsData method), 70
setCanteraKinetics() (rmgpy.kinetics.ThirdBody method), 77
setCanteraKinetics() (rmgpy.kinetics.Troe method), 81
setConditions() (rmgpy.pdep.Network method), 114
setConditions() (rmgpy.rmg.pdep.PDepNetwork method), 152
setDefaultOutputDirectory() (rmgpy.qm.main.QMCalculator method), 120
setLonePairs() (rmgpy.molecule.Atom method), 92
setSpinMultiplicity() (rmgpy.molecule.Atom method), 92
setUncertaintyType() (rmgpy.quantity.ArrayQuantity method), 139
setUncertaintyType() (rmgpy.quantity.ScalarQuantity method), 138
SimpleReactor (class in rmgpy.solver), 158
simulate() (rmgpy.solver.ReactionSystem method), 158
simulate() (rmgpy.solver.SimpleReactor method), 160
SingleExponentialDown (class in rmgpy.pdep), 109
solve() (rmgpy.data.statmechfit.DirectFit method), 45
solve() (rmgpy.data.statmechfit.PseudoFit method), 47
solve() (rmgpy.data.statmechfit.PseudoRotorFit method), 46
solveFullME() (rmgpy.pdep.Network method), 114
solveFullME() (rmgpy.rmg.pdep.PDepNetwork method), 152
solveReducedME() (rmgpy.pdep.Network method), 114
solveReducedME() (rmgpy.rmg.pdep.PDepNetwork method), 152
solveSchrodingerEquation() (rmgpy.statmech.HinderedRotor method), 175
sortAtoms() (rmgpy.molecule.Group method), 103
sortAtoms() (rmgpy.molecule.Molecule method), 97
sortVertices() (rmgpy.molecule.graph.Graph method), 88
sortVertices() (rmgpy.molecule.Group method), 103
toCantera() (rmgpy.data.kinetics.TemplateReaction method), 55

toCantera() (rmgpy.reaction.Reaction method), 143

toCantera() (rmgpy.rmg.model.Species method), 156

toCantera() (rmgpy.rmg.pdep.PDepReaction method), 155

toCantera() (rmgpy.species.Species method), 162

toCantera() (rmgpy.thermo.nasa method), 184

toCanteraKinetics() (rmgpy.kinetics.Arrhenius method), 67

toChemkin() (rmgpy.data.kinetics.DepositoryReaction method), 20

toChemkin() (rmgpy.data.kinetics.LibraryReaction method), 39

toChemkin() (rmgpy.data.kinetics.TemplateReaction method), 55

toChemkin() (rmgpy.reaction.Reaction method), 143

toChemkin() (rmgpy.rmg.model.Species method), 156

toChemkin() (rmgpy.rmg.pdep.PDepReaction method), 155

toChemkin() (rmgpy.species.Species method), 162

toGroup() (rmgpy.molecule.Molecule method), 98

toHTML() (rmgpy.kinetics.Arrhenius method), 67

toHTML() (rmgpy.kinetics.Chebyshev method), 75

toHTML() (rmgpy.kinetics.KineticsData method), 66

toHTML() (rmgpy.kinetics.Lindemann method), 79

toHTML() (rmgpy.kinetics.MultiArrhenius method), 88

toHTML() (rmgpy.kinetics.MultiPDepArrhenius method), 73

toHTML() (rmgpy.kinetics.PDepArrhenius method), 72

toHTML() (rmgpy.kinetics.PDepKineticsData method), 70

toHTML() (rmgpy.kinetics.ThirdBody method), 77

toHTML() (rmgpy.kinetics.Troe method), 81

toInChI() (rmgpy.molecule.Molecule method), 98

toInChIKey() (rmgpy.molecule.Molecule method), 98

toNASA() (rmgpy.thermo.ThermoData method), 179

toNASA() (rmgpy.thermo.Wilhoit method), 182

toRDKitMol() (rmgpy.molecule.Molecule method), 98

toSingleBonds() (rmgpy.molecule.Molecule method), 98

toSMARTS() (rmgpy.molecule.Molecule method), 98

toSMILES() (rmgpy.molecule.Molecule method), 98

toThermoData() (rmgpy.thermo.nasa method), 184

toThermoData() (rmgpy.thermo.Wilhoit method), 183

TransitionState (class in rmgpy.species), 163

Troe (class in rmgpy.kinetics), 79

uncertainty (rmgpy.quantity.ArrayQuantity attribute), 139

uncertainty (rmgpy.quantity.ScalarQuantity attribute), 138

uncertaintyType (rmgpy.quantity.ArrayQuantity attribute), 139

uncertaintyType (rmgpy.quantity.ScalarQuantity attribute), 138

uniqueID (rmgpy.qm.molecule.Geometry attribute), 121

uniqueID (rmgpy.qm.symmetry.SymmetryJob attribute), 123

uniqueIDlong (rmgpy.qm.molecule.Geometry attribute), 121

unitDegeneracy() (in module rmgpy.statmech.schrodinger), 176

update() (rmgpy.molecule.Molecule method), 98

update() (rmgpy.rmg.pdep.PDepNetwork method), 152

updateAtomTypes() (rmgpy.molecule.Molecule method), 99

updateCharge() (rmgpy.molecule.Atom method), 92

updateConfigurations() (rmgpy.rmg.pdep.PDepNetwork method), 152

updateConnectivityValues() (rmgpy.molecule.graph.Graph method), 88

updateConnectivityValues() (rmgpy.molecule.Group method), 104

updateConnectivityValues() (rmgpy.molecule.Molecule method), 99

updateFingerprint() (rmgpy.molecule.Group method), 104

updateLonePairs() (rmgpy.molecule.Molecule method), 99

updateMultiplicity() (rmgpy.molecule.Molecule method), 99

updateUnimolecularReactionNetworks() (rmgpy.rmg.model.CoreEdgeReactionModel method), 147

usePolar (rmgpy.qm.mopac.Mopac attribute), 129

V

value (rmgpy.quantity.ArrayQuantity attribute), 139

value (rmgpy.quantity.ScalarQuantity attribute), 138

verifyOutputFile() (rmgpy.qm.gaussian.Gaussian method), 124

verifyOutputFile() (rmgpy.qm.gaussian.GaussianMol method), 125

verifyOutputFile() (rmgpy.qm.gaussian.GaussianMolPM3 method), 127

verifyOutputFile() (rmgpy.qm.gaussian.GaussianMolPM6 method), 129

verifyOutputFile() (rmgpy.qm.mopac.Mopac method), 129

verifyOutputFile() (rmgpy.qm.mopac.MopacMol method), 131

verifyOutputFile() (rmgpy.qm.mopac.MopacMolPM3 method), 132

verifyOutputFile() (rmgpy.qm.mopac.MopacMolPM6 method), 134
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>verifyOutputFile() (rmgpy.qm.mopac.MopacMolPM7 method)</td>
<td>136</td>
</tr>
<tr>
<td>Vertex (class in rmgpy.molecule.graph)</td>
<td>85</td>
</tr>
<tr>
<td>VF2 (class in rmgpy.molecule.vf2)</td>
<td>88</td>
</tr>
<tr>
<td>visit() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_Call() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_Dict() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_List() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_Num() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_Str() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>visit_Tuple() (rmgpy.cantherm.output.PrettifyVisitor method)</td>
<td>7</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.gaussian.GaussianMol method)</td>
<td>126</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.gaussian.GaussianMolPM3 method)</td>
<td>127</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.gaussian.GaussianMolPM6 method)</td>
<td>129</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.mopac.MopacMol method)</td>
<td>131</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.mopac.MopacMolPM3 method)</td>
<td>133</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.mopac.MopacMolPM6 method)</td>
<td>134</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.mopac.MopacMolPM7 method)</td>
<td>136</td>
</tr>
<tr>
<td>writeInputFile() (rmgpy.qm.symmetry.SymmetryJob method)</td>
<td>123</td>
</tr>
<tr>
<td>writeKineticsEntry() (in module rmgpy.chemkin)</td>
<td>12</td>
</tr>
<tr>
<td>writeThermoEntry() (in module rmgpy.chemkin)</td>
<td>12</td>
</tr>
</tbody>
</table>