



RMG-Py API Reference

Release 1.0.0

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

July 17, 2015

CONTENTS

1	RMG API Reference	3
1.1	CanTherm (rmgpy.cantherm)	3
1.2	Chemkin files (rmgpy.chemkin)	10
1.3	Physical constants (rmgpy.constants)	13
1.4	Database (rmgpy.data)	13
1.5	Kinetics (rmgpy.kinetics)	59
1.6	Molecular representations (rmgpy.molecule)	77
1.7	Pressure dependence (rmgpy.pdep)	99
1.8	QMTP (rmgpy.qm)	109
1.9	Physical quantities (rmgpy.quantity)	127
1.10	Reactions (rmgpy.reaction)	130
1.11	Reaction mechanism generation (rmgpy.rmg)	134
1.12	Reaction system simulation (rmgpy.solver)	147
1.13	Species (rmgpy.species)	149
1.14	Statistical mechanics (rmgpy.statmech)	152
1.15	Thermodynamics (rmgpy.thermo)	166
	Bibliography	177
	Python Module Index	179
	Index	181

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/>

RMG API REFERENCE

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.

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

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

Class	Description
<i>GaussianLog</i>	Extract chemical parameters from Gaussian log files

1.1.2 Geometry

Class	Description
<i>Geometry</i>	The three-dimensional geometry of a molecular conformation

1.1.3 Input

Function	Description
<code>loadInputFile()</code>	Load a CanTherm job input file

1.1.4 Job classes

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

1.1.5 Exceptions

Exception	Description
<code>GaussianError</code>	Raised when an error occurs while working with a Gaussian log file

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 `io(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/m². 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:

Attribute	Type	Description
<i>coordinates</i>	<code>numpy.ndarray</code>	An $N \times 3$ array containing the 3D coordinates of each atom
<i>number</i>	<code>numpy.ndarray</code>	An array containing the integer atomic number of each atom
<i>mass</i>	<code>numpy.ndarray</code>	An array containing the atomic mass in kg/mol of each atom

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 Pitzer¹, which is described as $I^{(2,3)}$ by East and Radom². 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 $\text{kg}\cdot\text{m}^2$. 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 $\text{kg}\cdot\text{m}^2$, while the principal axes have unit length.

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.

¹ Pitzer, K. S. *J. Chem. Phys.* **14**, p. 239-243 (1946).

² East, A. L. L. and Radom, L. *J. Chem. Phys.* **106**, p. 6655-6674 (1997).

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:

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

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.prettyfy`(*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 `prettyfy()` 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:

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

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 $A \rightarrow$ 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*)

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.

generatePressureList()

Returns an array of pressures based on the interpolation *model*, minimum and maximum pressures *Pmin* and *Pmax* in Pa, and the number of pressures *Pcount*. 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 *Pmin* and *Pmax* 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 *Tmin* and *Tmax* in K, and the number of temperatures *Tcount*. 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 *Tmin* and *Tmax* 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

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

1.2.2 Writing Chemkin files

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

1.2.3 Exceptions

Exception	Description
<code>ChemkinError</code>	Raised when an error occurs while working with a Chemkin file

Reading Chemkin files

Main functions

`rmgpy.chemkin.loadChemkinFile(path, dictionaryPath=None, transportPath=None, readComments=True, thermoPath=None)`

Load a Chemkin input file located at *path* on disk to *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

`rmgpy.chemkin.loadSpeciesDictionary(path)`

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

`rmgpy.chemkin.loadTransportFile(path, speciesDict)`

Load a Chemkin transport properties file located at *path* and store the properties on the species in *speciesDict*.

Helper functions

`rmgpy.chemkin.readKineticsEntry(entry, speciesDict, Aunits, Eunits)`

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

`rmgpy.chemkin.readReactionComments(reaction, comments, read=True)`

Parse the *comments* associated with a given *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 *read* if False, the reaction is returned as an “Unclassified” LibraryReaction.

`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.

`rmgpy.chemkin.readThermoEntry(entry, Tmin=0, Tint=0, Tmax=0)`

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

Format specification at <http://www2.galcit.caltech.edu/EDL/public/formats/chemkin.html>

`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

`rmgpy.chemkin.saveChemkinFile(path, species, reactions, verbose=True, checkForDuplicates=True)`

Save a Chemkin input file to *path* on disk containing the provided lists of *species* and *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)`

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

`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 ϵ/k_B in Kelvins.
3. The Lennard-Jones collision diameter σ in Angstroms.
4. The dipole moment μ in Debye. Note: a Debye is $10^{-18} \text{ cm}^3/2 \text{ erg}^{1/2}$.
5. The polarizability α 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.

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

```
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 1.1: Physical constants defined in the `rmgpy.constants` module

Symbol	Constant	Value	Description
E_h	E_h	$4.35974434 \times 10^{-18}$ J	Hartree energy
F	F	96485.3365 C/mol	Faraday constant
G	G	6.67384×10^{-11} m ³ /kg · s ²	Newtonian gravitational constant
N_A	Na	$6.02214179 \times 10^{23}$ mol ⁻¹	Avogadro constant
R	R	8.314472 J/mol · K	gas law constant
a_0	a0	$5.2917721092 \times 10^{-11}$ m	Bohr radius
c	c	299792458 m/s	speed of light in a vacuum
e	e	$1.602176565 \times 10^{-19}$ C	elementary charge
g	g	9.80665 m/s ²	standard acceleration due to gravity
h	h	$6.62606896 \times 10^{-34}$ J · s	Planck constant
\hbar	hbar	$1.054571726 \times 10^{-34}$ J · s	reduced Planck constant
k_B	kB	$1.3806504 \times 10^{-23}$ J/K	Boltzmann constant
m_e	m_e	$9.10938291 \times 10^{-31}$ kg	electron rest mass
m_n	m_n	$1.674927351 \times 10^{-27}$ kg	neutron rest mass
m_p	m_p	$1.672621777 \times 10^{-27}$ kg	proton rest mass
m_u	amu	$1.660538921 \times 10^{-27}$ kg	atomic mass unit
π	pi	3.14159...	

1.4 Database (rmgpy.data)

1.4.1 General classes

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

1.4.2 Thermodynamics database

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

1.4.3 Kinetics database

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

1.4.4 Statistical mechanics database

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

1.4.5 Statistical mechanics fitting

Class/Function	Description
<i>DirectFit</i>	DQED class for fitting a small number of vibrational frequencies and hindered rotors
<i>PseudoFit</i>	DQED class for fitting a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both
<i>PseudoRotorFit</i>	DQED class for fitting a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only
<i>fitStatmechDirect()</i>	Directly fit a small number of vibrational frequencies and hindered rotors
<i>fitStatmechPseudo()</i>	Fit a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both
<i>fitStatmechPseudoRotorFit()</i>	Fit a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only
<i>fitStatmechToHeatCapacityVib()</i>	Fit vibrational and torsional degrees of freedom to heat capacity data

1.4.6 Exceptions

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

`rmgpy.data.base.Database`

class `rmgpy.data.base.Database`(*entries=None, top=None, label='', name='', solvent=None, short-Desc='', longDesc=''*)

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.

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*.

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

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.

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

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

where Q^\ddagger 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 a factor of `getDiffusionFactor` to account for the diffusion limit.

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 an Arrhenius or KineticsData object.

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.

getRate()

Return the net rate of reaction at temperature *T* and pressure *P*. The parameter *conc* is a map with species as keys and concentrations as values. A reactant not found in the *conc* map is treated as having zero concentration.

If passed a *totalConc*, it won't bother recalculating it.

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 *DepositoryReaction* this should be a *KineticsDepository* 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 $A + B \rightleftharpoons C$ 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) $A \rightleftharpoons B + C$ or $A + B \rightleftharpoons C$ or $A \rightleftharpoons B$, 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.

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:

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

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.

```
generateReactions(reactants, products=None, failsSpeciesConstraints=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, failsSpeciesConstraints=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*, *failsSpeciesConstraints=None*)

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*, *failsSpeciesConstraints=None*)

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).

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.

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*.

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

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.

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(*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.KineticsFamily

```
class rmgpy.data.kinetics.KineticsFamily(entries=None, top=None, label='', name='', reverse='',
                                         shortDesc='', longDesc='', forwardTemplate=None,
                                         forwardRecipe=None, reverseTemplate=None, re-
                                         verseRecipe=None, forbidden=None)
```

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

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

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.

addKineticsRulesFromTrainingSet (*thermoDatabase=None*)

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

ancestors (*node*)

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

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.

calculateDegeneracy (*reaction*)

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

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*, *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.

fillKineticsRulesByAveragingUp (*rootTemplate=None*, *alreadyDone=None*)

Fill in gaps in the kinetics rate rules by averaging child nodes.

generateOldTree (*entries*, *level*)

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

generateProductTemplate(*reactants*)

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*, *failsSpeciesConstraints=None*)

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 `Species` 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.

getRateCoefficientUnits()

Return the units of the forward kinetics generated for the family. Raises a `ValueError` if this could not be determined (e.g. if the family has not yet been added to the appropriate list).

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.

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`).

getSpecies(*path*)

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

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(*label*, *group*, *shortDesc*='', *longDesc*='')

Load information about a forbidden structure.

loadOld(*path*)

Load an old-style RMG kinetics group additivity database 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*.

loadOldTemplate(*path*)

Load an old-style RMG reaction family template from the location *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.

loadRecipe(*actions*)

Load information about the reaction recipe.

loadTemplate(*reactants*, *products*, *ownReverse*=False)

Load information about the reaction template.

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*.

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

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.

save(*path*)

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

saveDepository(*depository*, *path*)

Save the given kinetics family *depository* to the 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*.

saveGroups(*path*)

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

saveOld(*path*)

Save the old RMG kinetics groups to the given *path* on disk.

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.

saveOldTemplate(*path*)

Save an old-style RMG reaction family template from the location *path*.

saveOldTree(*path*)

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

rmgpy.data.kinetics.KineticsGroups

```
class rmgpy.data.kinetics.KineticsGroups(entries=None, top=None, label='', name='', short-
    Desc='', longDesc='', forwardTemplate=None,
    forwardRecipe=None, reverseTemplate=None, re-
    verseRecipe=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.

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(*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.

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*.

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

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.

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.

checkForDuplicates()

Check that all duplicate reactions in the kinetics library are properly marked (i.e. with their duplicate attribute set to *True*).

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.

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

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.

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 library to the file object *f*.

saveOld(*path*)

Save an old-style reaction library to *path*. This creates files named `species.txt`, `reactions.txt`, and `pdepreactions.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.

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*)

Fill in gaps in the kinetics rate rules by averaging child nodes.

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.

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

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.

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 *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

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

where Q^\ddagger 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 a factor of `getDiffusionFactor` to account for the diffusion limit.

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	$A \rightarrow C$	(A,C)
Dissociation	$A \rightarrow C + D$	(A,C), (A,D)
Association	$A + B \rightarrow C$	(A,C), (B,C)
Bimolecular	$A + B \rightarrow C + D$	(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 an Arrhenius or KineticsData object.

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.

getRate()

Return the net rate of reaction at temperature *T* and pressure *P*. The parameter *conc* is a map with species as keys and concentrations as values. A reactant not found in the *conc* map is treated as having zero concentration.

If passed a *totalConc*, it won't bother recalculating it.

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 $A + B \rightleftharpoons C$ 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) $A \rightleftharpoons B + C$ or $A + B \rightleftharpoons C$ or $A \rightleftharpoons B$, 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.

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:

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

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.

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 pre-labeled.

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*.

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

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.

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.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. $\|\Delta\vec{x}\| \leq \epsilon_x \cdot \|\vec{x}\|$.
- *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 *x0*, 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

class `rmgpy.data.statmechfit.PseudoRotorFit(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 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. $\|\vec{f}\| \leq \epsilon_f$.
- *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 *x0*, 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

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 *x0*, 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.StatmechGroups

class `rmgpy.data.statmech.StatmechGroups` (*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.

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 pre-labeled.

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.

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

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 corresponding thermodynamics object.

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.

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.

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.

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

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.

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)
```

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

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

where Q^\ddagger 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 a factor of `getDiffusionFactor` to account for the diffusion limit.

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 an Arrhenius or KineticsData object.

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.

getRate()

Return the net rate of reaction at temperature *T* and pressure *P*. The parameter *conc* is a map with species as keys and concentrations as values. A reactant not found in the *conc* map is treated as having zero concentration.

If passed a *totalConc*, it won't bother recalculating it.

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 $A + B \rightleftharpoons C$ 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) $A \rightleftharpoons B + C$ or $A + B \rightleftharpoons C$ or $A \rightleftharpoons B$, 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.

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.

findCp0andCpInf(*species*, *thermoData*)

Calculate the Cp0 and CpInf values, and add them to the *thermoData* object.

Modifies *thermoData* in place and doesn't return anything

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)

getThermoData(*species*, *trainingSet=None*, *quantumMechanics=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.

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.

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*.

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

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.

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.

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.

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*.

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

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.

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.

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.

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*.

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

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.

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

Class	Description
<i>KineticsData</i>	A kinetics model based on a set of discrete rate coefficient points in temperature
<i>Arrhenius</i>	A kinetics model based on the (modified) Arrhenius expression
<i>MultiArrhenius</i>	A kinetics model based on a sum of <i>Arrhenius</i> expressions

1.5.2 Pressure-dependent kinetics models

Class	Description
<i>PDepKineticsData</i>	A kinetics model based on a set of discrete rate coefficient points in temperature and pressure
<i>PDepArrhenius</i>	A kinetics model based on a set of Arrhenius expressions for a range of pressures
<i>MultiPDepArrhenius</i>	A kinetics model based on a sum of <i>PDepArrhenius</i> expressions
<i>Chebyshev</i>	A kinetics model based on a Chebyshev polynomial representation
<i>ThirdBody</i>	A low pressure-limit kinetics model based on the (modified) Arrhenius expression, with a third body
<i>Lindemann</i>	A kinetics model of pressure-dependent falloff based on the Lindemann model
<i>Troe</i>	A kinetics model of pressure-dependent falloff based on the Lindemann model with the Troe falloff factor

1.5.3 Tunneling models

Class	Description
<i>Wigner</i>	A one-dimensional tunneling model based on the Wigner expression
<i>Eckart</i>	A one-dimensional tunneling model based on the (asymmetric) Eckart expression

rmgpy.kinetics.KineticsData

class rmgpy.kinetics.**KineticsData**(*Tdata=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. The attributes are:

Attribute	Description
<i>Tdata</i>	An array of temperatures at which rate coefficient values are known
<i>kdata</i>	An array of rate coefficient values
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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 m³, 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 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.

toHTML(*self*)

Return an HTML rendering.

rmgpy.kinetics.Arrhenius

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:

Attribute	Description
<i>A</i>	The preexponential factor
<i>T0</i>	The reference temperature
<i>n</i>	The temperature exponent
<i>Ea</i>	The activation energy
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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.

comment

comment: str

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 *kuits* 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 m³, 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.

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:

Attribute	Description
<i>arrhenius</i>	A list of the Arrhenius kinetics
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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.

getRateCoefficient(*self*, *double T*, *double P=0.0*) → double

Return the rate coefficient in the appropriate combination of m³, 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.

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:

Attribute	Description
<i>Tdata</i>	An array of temperatures at which rate coefficient values are known
<i>Pdata</i>	An array of pressures at which rate coefficient values are known
<i>kdata</i>	An array of rate coefficient values at each temperature and pressure
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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

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* with the given *fractions*.

getRateCoefficient(*self, double T, double P=0.0*) → double

Return the rate coefficient in the appropriate combination of m³, 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.

toHTML(*self*)

Return an HTML rendering.

rmgpy.kinetics.PDepArrhenius

class rmgpy.kinetics.PDepArrhenius(*pressures=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:

Attribute	Description
<i>pressures</i>	The list of pressures
<i>arrhenius</i>	The list of Arrhenius objects at each pressure
<i>Tmin</i>	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure in bar at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure in bar at which the model is valid, or zero if unknown or undefined
<i>efficiencies</i>	A dict associating chemical species with associated efficiencies
<i>order</i>	The reaction order (1 = first, 2 = second, etc.)
<i>comment</i>	Information about the model (e.g. its source)

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 E_a 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.**getEffectivePressure**(*self*, *double P*, *list species*, *ndarray fractions*) → doubleReturn the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* with the given *fractions*.**getRateCoefficient**(*self*, *double T*, *double P=0*) → doubleReturn the rate coefficient in the appropriate combination of m³, 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*) → boolReturn 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*) → boolReturn 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.

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:

Attribute	Description
<i>arrhenius</i>	A list of the PDepArrhenius kinetics at each temperature
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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

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* with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0.0*) → double

Return the rate coefficient in the appropriate combination of m³, 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 +/- .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.

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:

Attribute	Description
<i>coeffs</i>	Matrix of Chebyshev coefficients, such that the resulting $k(T, P)$ has units of cm ³ , mol, s
<i>kunits</i>	The units of the rate coefficient
<i>degreeT</i>	The number of terms in the inverse temperature direction
<i>degreeP</i>	The number of terms in the log pressure direction
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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, α_{tp} is a constant, $\phi_n(x)$ is the Chebyshev polynomial of degree n evaluated at x , and

$$\tilde{T} \equiv \frac{2T^{-1} - T_{\min}^{-1} - T_{\max}^{-1}}{T_{\max}^{-1} - T_{\min}^{-1}}$$

$$\tilde{P} \equiv \frac{2 \log P - \log P_{\min} - \log P_{\max}}{\log P_{\max} - \log P_{\min}}$$

are reduced temperature and reduced pressure designed to map the ranges (T_{\min}, T_{\max}) and (P_{\min}, P_{\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*) → 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.

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* with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0*) → double

Return the rate coefficient in the appropriate combination of m³, 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

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:

Attribute	Description
<i>arrheniusLow</i>	The Arrhenius kinetics at the low-pressure limit
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>efficiencies</i>	A dict associating chemical species with associated efficiencies
<i>comment</i>	Information about the model (e.g. its source)

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

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* 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 $\text{m}^3, \text{mol},$ 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,

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.

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:

Attribute	Description
<i>arrheniusHigh</i>	The Arrhenius kinetics at the high-pressure limit
<i>arrheniusLow</i>	The Arrhenius kinetics at the low-pressure limit
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>efficiencies</i>	A dict associating chemical species with associated efficiencies
<i>comment</i>	Information about the model (e.g. its source)

The Lindemann model qualitatively predicts the falloff of some simple pressure-dependent reaction kinetics. The formulation is as follows:

$$k(T, P) = k_{\infty}(T) \left[\frac{P_r}{1 + P_r} \right]$$

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`

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* 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.

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:

Attribute	Description
<i>arrheniusHigh</i>	The Arrhenius kinetics at the high-pressure limit
<i>arrheniusLow</i>	The Arrhenius kinetics at the low-pressure limit
<i>alpha</i>	The α parameter
<i>T1</i>	The T_1 parameter
<i>T2</i>	The T_2 parameter
<i>T3</i>	The T_3 parameter
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>efficiencies</i>	A dict associating chemical species with associated efficiencies
<i>comment</i>	Information about the model (e.g. its source)

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

$$\log F = \left\{ 1 + \left[\frac{\log P_r + c}{n - d(\log P_r + c)} \right]^2 \right\}^{-1} \log F_{\text{cent}}$$

$$c = -0.4 - 0.67 \log F_{\text{cent}}$$

$$n = 0.75 - 1.27 \log F_{\text{cent}}$$

$$d = 0.14$$

$$F_{\text{cent}} = (1 - \alpha) \exp(-T/T_3) + \alpha \exp(-T/T_1) + \exp(-T_2/T)$$

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.

T1

The Troe T_1 parameter.

T2

The Troe T_2 parameter.

T3

The Troe T_3 parameter.

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.

alpha

alpha: 'double'

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

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* 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.

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:

Attribute	Description
<i>frequency</i>	The imaginary frequency of the transition state

An early formulation for incorporating the effect of tunneling is that of Wigner [1932Wigner]:

$$\kappa(T) = 1 + \frac{1}{24} \left(\frac{h |\nu_{\text{TS}}|}{k_{\text{B}} T} \right)^2$$

where h is the Planck constant, ν_{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.

calculateTunnelingFactor(*self*, *double T*) → double

Calculate and return the value of the Wigner tunneling correction for the reaction at the temperature T in K.

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

class rmgpy.kinetics.**Eckart**(*frequency*, *E0_reac*, *E0_TS*, *E0_prod=None*)

A tunneling model based on the Eckart model. The attributes are:

Attribute	Description
<i>frequency</i>	The imaginary frequency of the transition state
<i>E0_reac</i>	The ground-state energy of the reactants
<i>E0_TS</i>	The ground-state energy of the transition state
<i>E0_prod</i>	The ground-state energy of the products

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

$$V(x) = \frac{\hbar^2}{2m} \left[\frac{Ae^x}{1 + e^x} + \frac{Be^x}{(1 + e^x)^2} \right]$$

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

$$x_{\text{max}} = \ln \left(\frac{B + A}{B - A} \right)$$

$$V(x_{\max}) = \frac{\hbar^2}{2m} \frac{(A+B)^2}{4B}$$

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

$$2\pi a = \frac{2\sqrt{\alpha_1 \xi}}{\alpha_1^{-1/2} + \alpha_2^{-1/2}}$$

$$2\pi b = \frac{2\sqrt{|(\xi - 1)\alpha_1 + \alpha_2|}}{\alpha_1^{-1/2} + \alpha_2^{-1/2}}$$

$$2\pi d = 2\sqrt{|\alpha_1 \alpha_2 - 4\pi^2/16|}$$

$$\alpha_1 = 2\pi \frac{\Delta V_1}{h |\nu_{\text{TS}}|}$$

$$\alpha_2 = 2\pi \frac{\Delta V_2}{h |\nu_{\text{TS}}|}$$

$$\xi = \frac{E}{\Delta V_1}$$

ΔV_1 and ΔV_2 are the thermal energy difference between the transition state and the reactants and products, respectively; ν_{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 $\beta \equiv 1/k_{\text{B}}T$):

$$\kappa(T) = e^{\beta \Delta V_1} \int_0^\infty \kappa(E) e^{-\beta E} dE$$

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*) → *double*

Calculate and return the value of the Eckart tunneling correction for the reaction at the temperature T in K.

calculateTunnelingFunction(*self*, *ndarray Elist*) → *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

Class	Description
<i>Vertex</i>	A generic vertex (node) in a graph
<i>Edge</i>	A generic edge (arc) in a graph
<i>Graph</i>	A generic graph data type

1.6.2 Graph isomorphism

Class	Description
<i>VF2</i>	Graph isomorphism using the VF2 algorithm

1.6.3 Elements and atom types

Class/Function	Description
<i>Element</i>	A model of a chemical element
<i>getElement()</i>	Return the <i>Element</i> object for a given atomic number or symbol
<i>AtomType</i>	A model of an atom type: an element and local bond structure
<i>getAtomType()</i>	Return the <i>AtomType</i> object for a given atom in a molecule

1.6.4 Molecules

Class	Description
<i>Atom</i>	An atom in a molecule
<i>Bond</i>	A bond in a molecule
<i>Molecule</i>	A molecular structure represented using a chemical graph

1.6.5 Functional groups

Class	Description
<i>GroupAtom</i>	An atom in a functional group
<i>GroupBond</i>	A bond in a functional group
<i>Group</i>	A functional group structure represented using a chemical graph

1.6.6 Adjacency lists

Function	Description
<i>fromAdjacencyList()</i>	Convert an adjacency list to a set of atoms and bonds
<i>toAdjacencyList()</i>	Convert a set of atoms and bonds to an adjacency list

1.6.7 Symmetry numbers

Class	Description
<code>calculateAtomSymmetryNumber()</code>	Calculate the atom-centered symmetry number for an atom in a molecule
<code>calculateBondSymmetryNumber()</code>	Calculate the bond-centered symmetry number for a bond in a molecule
<code>calculateAxisSymmetryNumber()</code>	Calculate the axis-centered symmetry number for a double bond axis in a molecule
<code>calculateCyclicSymmetryNumber()</code>	Calculate the ring-centered symmetry number for a ring in a molecule
<code>calculateSymmetryNumber()</code>	Calculate the total internal + external symmetry number for a molecule

1.6.8 Molecule and reaction drawing

Class	Description
<code>MoleculeDrawer</code>	Draw the skeletal formula of a molecule
<code>ReactionDrawer</code>	Draw a chemical reaction

1.6.9 Exceptions

Exception	Description
<code>ElementError</code>	Raised when an error occurs while working with chemical elements
<code>AtomTypeError</code>	Raised when an error occurs while working with atom types
<code>InvalidAdjacencyListError</code>	Raised when an invalid adjacency list is encountered
<code>ActionError</code>	Raised when an error occurs while working with a reaction recipe action

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 \(1965\)](#).

Attribute	Type	Description
<code>connectivity1</code>	int	The number of nearest neighbors
<code>connectivity2</code>	int	The sum of the neighbors' <code>connectivity1</code> values
<code>connectivity3</code>	int	The sum of the neighbors' <code>connectivity2</code> values
<code>sortingLabel</code>	int	An integer label used to sort the vertices

`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.

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.

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.

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.

getEdge()

Returns the edge connecting vertices *vertex1* and *vertex2*.

getEdges()

Return a list of the edges involving the specified *vertex*.

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)

B. T. Fan, A. Panaye, J. P. Doucet, and A. Barbu. "Ring Perception: A New Algorithm for Directly Finding the Smallest Set of Smallest Rings from a Connection Table." *J. Chem. Inf. Comput. Sci.* **33**, p. 657-662 (1993).

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.

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:

Attribute	Type	Description
<i>number</i>	int	The atomic number of the element
<i>symbol</i>	str	The symbol used for the element
<i>name</i>	str	The IUPAC name of the element
<i>mass</i>	float	The mass of the element in kg/mol
<i>covRadius</i>	float	Covalent bond radius in Angstrom

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.

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:

Attribute	Type	Description
<i>label</i>	str	A unique label for the atom type
<i>generic</i>	list	The atom types that are more generic than this one
<i>specific</i>	list	The atom types that are more specific than this one
<i>incrementBond</i>	list	The atom type(s) that result when an adjacent bond's order is incremented
<i>decrementBond</i>	list	The atom type(s) that result when an adjacent bond's order is decremented
<i>formBond</i>	list	The atom type(s) that result when a new single bond is formed to this atom type
<i>breakBond</i>	list	The atom type(s) that result when an existing single bond to this atom type is broken
<i>incrementRadical</i>	list	The atom type(s) that result when the number of radical electrons is incremented
<i>decrementRadical</i>	list	The atom type(s) that result when the number of radical electrons is decremented
<i>incrementLone-Pair</i>	list	The atom type(s) that result when the number of lone electron pairs is incremented
<i>decrementLone-Pair</i>	list	The atom type(s) that result when the number of lone electron pairs is decremented

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.

isSpecificCaseOf()

Returns True if atom type *atomType1* is a specific case of atom type *atomType2* or False otherwise.

rmgpy.molecule.getAtomType()

Determine the appropriate atom type for an Atom object *atom* with local bond structure *bonds*, a dict containing atom-bond pairs.

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:

Atom type	Description
<i>General atom types</i>	
R	any atom with any local bond structure
R!H	any non-hydrogen atom with any local bond structure
<i>Carbon atom types</i>	
C	carbon atom with any local bond structure
Cs	carbon atom with four single bonds
Cd	carbon atom with one double bond (to carbon) and two single bonds
Cdd	carbon atom with two double bonds
Ct	carbon atom with one triple bond and one single bond
C0	carbon atom with one double bond (to oxygen) and two single bonds
Cb	carbon atom with two benzene bonds and one single bond
Cbf	carbon atom with three benzene bonds
<i>Hydrogen atom types</i>	
H	hydrogen atom with one single bond
<i>Oxygen atom types</i>	
Continued on next page	

Table 1.2 – continued from previous page

Atom type	Description
O	oxygen atom with any local bond structure
Os	oxygen atom with two single bonds
Od	oxygen atom with one double bond
Oa	oxygen atom with no bonds
<i>Silicon atom types</i>	
Si	silicon atom with any local bond structure
Sis	silicon atom with four single bonds
Sid	silicon atom with one double bond (to carbon) and two single bonds
Sidd	silicon atom with two double bonds
Sit	silicon atom with one triple bond and one single bond
SiO	silicon atom with one double bond (to oxygen) and two single bonds
Sib	silicon atom with two benzene bonds and one single bond
Sibf	silicon atom with three benzene bonds
<i>Sulfur atom types</i>	
S	sulfur atom with any local bond structure
Ss	sulfur atom with two single bonds
Sd	sulfur atom with one double bond
Sa	sulfur atom with no bonds

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 name	Arguments	Action
CHANGE_BOND	<i>center1</i> , <i>order</i> , <i>center2</i>	change the bond order of the bond between <i>center1</i> and <i>center2</i> by <i>order</i> ; do not break or form bonds
FORM_BOND	<i>center1</i> , <i>order</i> , <i>center2</i>	form a new bond between <i>center1</i> and <i>center2</i> of type <i>order</i>
BREAK_BOND	<i>center1</i> , <i>order</i> , <i>center2</i>	break the bond between <i>center1</i> and <i>center2</i> , which should be of type <i>order</i>
GAIN_RADICAL	<i>center</i> , <i>radical</i>	increase the number of free electrons on <i>center</i> by <i>radical</i>
LOSE_RADICAL	<i>center</i> , <i>radical</i>	decrease the number of free electrons on <i>center</i> by <i>radical</i>

rmgpy.molecule.Atom

class rmgpy.molecule.Atom

An atom. The attributes are:

Attribute	Type	Description
<i>atomType</i>	AtomType	The <i>atom type</i>
<i>element</i>	Element	The chemical element the atom represents
<i>radicalElectrons</i>	short	The number of radical electrons
<i>charge</i>	short	The formal charge of the atom
<i>label</i>	str	A string label that can be used to tag individual atoms
<i>coords</i>	numpy array	The (x,y,z) coordinates in Angstrom
<i>lonePairs</i>	short	The number of lone electron pairs

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.

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.

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.

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.

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:

Attribute	Type	Description
<i>order</i>	str	The <i>bond type</i>

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:

Bond type	Description
S	a single bond
D	a double bond
T	a triple bond
B	a benzene bond

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:

Attribute	Type	Description
<i>symmetryNumber</i>	int	The (estimated) external + internal symmetry number of the molecule
<i>multiplicity</i>	int	The multiplicity of this species, $\text{multiplicity} = 2 * \text{total_spin} + 1$

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.

createMultiplicityLayer()

Creates the string with the multiplicity information that is appended to the *InChI* to create an augmented *InChI*.

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.

findAllDelocalizationPaths()

Find all the delocalization paths allyl to the radical center indicated by *atomI*. Used to generate resonance isomers.

findAllDelocalizationPathsLonePairRadical()

Find all the delocalization paths of lone electron pairs next to the radical center indicated by *atomI*. Used to generate resonance isomers.

findAllDelocalizationPathsN5dd_N5ts()

Find all the resonance structures of nitrogen atoms with two double bonds (N5dd) and nitrogen atoms with one triple and one single bond (N5ts)

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 *adlist* to a molecular structure. Skips the first line (assuming it's a label) unless *withLabel* is False.

fromInChI()

Convert an InChI string *inchistr* to a molecular structure. Uses RDKit to perform the conversion. This Kekulizes everything, removing all aromatic atom types.

fromRDKitMol()

Convert a RDKit Mol object *rdkitmol* to a molecular structure. Uses RDKit to perform the conversion. This Kekulizes everything, removing all aromatic atom types.

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. Uses RDKit to perform the conversion. This Kekulizes everything, removing all aromatic atom types.

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.

generateResonanceIsomers()

Generate and return all of the resonance isomers of this molecule.

getAdjacentResonanceIsomers()

Generate all of the resonance isomers formed by one allyl radical shift.

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.

getAromaticResonanceIsomers()

Generate the aromatic form of the molecule.

getBond()

Returns the bond connecting atoms *atom1* and *atom2*.

getBonds()

Return a list of the bonds involving the specified *atom*.

getEdge()

Returns the edge connecting vertices *vertex1* and *vertex2*.

getEdges()

Return a list of the edges involving the specified *vertex*.

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 associated molecules to be isomorphic.

getFormula()

Return the molecular formula for the molecule.

getKekulizedResonanceIsomers()

Generate the kekulized (single-double bond) form of the molecule.

getLabeledAtom()

Return the atoms in the molecule that are labeled.

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.

getLonePairRadicalResonanceIsomers()

Generate all of the resonance isomers formed by lone electron pair - radical shifts.

getMolecularWeight()

Return the molecular weight of the molecule in kg/mol.

getN5dd_N5tsResonanceIsomers()

Generate all of the resonance isomers formed by shifts between N5dd and N5ts.

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.

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 implemented was adapted from a description by Fan, Panaye, Doucet, and Barbu (doi: 10.1021/ci00015a002)

B. T. Fan, A. Panaye, J. P. Doucet, and A. Barbu. "Ring Perception: A New Algorithm for Directly Finding the Smallest Set of Smallest Rings from a Connection Table." *J. Chem. Inf. Comput. Sci.* **33**, p. 657-662 (1993).

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 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.

isAromatic()

Returns True if the molecule is aromatic, or False if not. 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.

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.

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.

toOBMol()

Convert a molecular structure to an OpenBabel OBMol object. Uses [OpenBabel](#) to perform the conversion.

toRDKitMol()

Convert a molecular structure to a RDKit rdmol object. Uses [RDKit](#) to perform the conversion. Perceives aromaticity and, unless `removeHs==False`, removes Hydrogen atoms.

If `returnMapping==True` then it also returns a dictionary mapping the atoms to RDKit's atom indices.

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

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:

Attribute	Type	Description
<i>atomType</i>	list	The allowed atom types (as <code>AtomType</code> objects)
<i>radicalElectrons</i>	list	The allowed numbers of radical electrons (as short integers)
<i>charge</i>	list	The allowed formal charges (as short integers)
<i>label</i>	str	A string label that can be used to tag individual atoms
<i>lonePairs</i>	list	The number of lone electron pairs

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 an `GroupAtom` object. When comparing two `GroupAtom` objects, this function respects wildcards, e.g. R!H is equivalent to C.

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.

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:

Attribute	Type	Description
<i>order</i>	list	The allowed bond orders (as character strings)

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.

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.

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.

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.

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` if *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*.

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.

getSmallestSetOfSmallestRings()

Return a list of the smallest set of smallest rings in the graph. The algorithm implemented was adapted from a description by Fan, Panaye, Doucet, and Barbu (doi: 10.1021/ci00015a002)

B. T. Fan, A. Panaye, J. P. Doucet, and A. Barbu. "Ring Perception: A New Algorithm for Directly Finding the Smallest Set of Smallest Rings from a Connection Table." *J. Chem. Inf. Comput. Sci.* **33**, p. 657-662 (1993).

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.

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, eg. `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.

```
group1
multiplicity x
1    R!H u0
```

```
group2
1    R!H u0
```

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 (eg. 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 in 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,S}`. Multiple bonds from the same atom should be separated by whitespace.

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,O]`. 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:

```
HXD13
multiplicity 1
1   C u0      {2,D}
2   C u0 {1,D} {3,S}
3   C u0 {2,S} {4,D}
4   C u0 {3,D} {5,S}
5 *1 C u0 {4,S} {6,S}
6 *2 C u0 {5,S}
```

The allowed element types, radicals, and bonds are listed in the following table:

	Notation	Explanation
Chemical Element	C	Carbon atom
	O	Oxygen atom
	H	Hydrogen atom
	S	Sulfur atom
	N	Nitrogen atom
Nonreactive Elements	Si	Silicon atom
	Cl	Chlorine atom
	He	Helium atom
	Ar	Argon atom
Chemical Bond	S	Single Bond
	D	Double Bond
	T	Triple bond
	B	Benzene bond

`rmgpy.molecule.adjlist.fromAdjacencyList(adjlist, group=False, saturateH=False)`

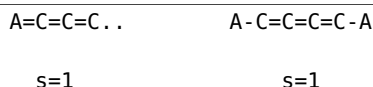
Convert a string adjacency list *adjlist* into a set of Atom and Bond objects.

`rmgpy.molecule.adjlist.toAdjacencyList(atoms, multiplicity, label=None, group=False, removeH=False, removeLonePairs=False, oldStyle=False)`

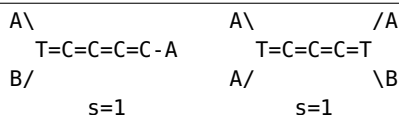
Convert a chemical graph defined by a list of *atoms* into a string adjacency list.

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()`.

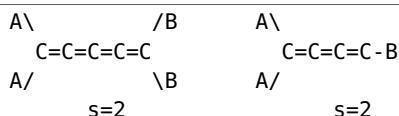
Each axis (C=C=C) has the potential to double the symmetry number. If an end has 0 or 1 groups (eg. =C=CJJ or =C=C-R) then it cannot alter the axis symmetry and is disregarded:



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:



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*).

1.7 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

Class	Description
<i>SingleExponentialDown</i>	A collisional energy transfer model based on the single exponential down model

1.7.2 Reaction events

Function	Description
<code>calculateMicrocanonicalRateCoefficient()</code>	Return the microcanonical rate coefficient $k(E)$ for a reaction
<code>applyRRKMTheory()</code>	Use RRKM theory to compute $k(E)$ for a reaction
<code>applyInverseLaplaceTransformMethod()</code>	Use the inverse Laplace transform method to compute $k(E)$ for a reaction

1.7.3 Pressure-dependent reaction networks

Class	Description
<code>Configuration</code>	A molecular configuration on a potential energy surface
<code>Network</code>	A collisional energy transfer model based on the single exponential down model

1.7.4 The master equation

Function	Description
<code>generateFullMEMatrix()</code>	Return the full master equation matrix for a network

1.7.5 Master equation reduction methods

Function	Description
<code>msc.applyModifiedStrongCollisionMethod()</code>	Reduce the master equation to phenomenological rate coefficients $k(T, P)$ using the modified strong collision method
<code>rs.applyReservoirStateMethod()</code>	Reduce the master equation to phenomenological rate coefficients $k(T, P)$ using the reservoir state method
<code>cse.applyChemicallySignificantEigenvaluesMethod()</code>	Reduce the master equation to phenomenological rate coefficients $k(T, P)$ using the chemically-significant eigenvalues method

1.7.6 Exceptions

Exception	Description
<code>NetworkError</code>	Raised when an error occurs while working with a pressure-dependent reaction network
<code>InvalidMicrocanonicalRateError</code>	Raised when the $k(E)$ is not consistent with the high pressure-limit kinetics or thermodynamics

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:

Attribute	Description
<i>alpha0</i>	The average energy transferred in a deactivating collision at the reference temperature
<i>T0</i>	The reference temperature
<i>n</i>	The temperature exponent

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 α is specified using the equation

$$\alpha = \alpha_0 \left(\frac{T}{T_0}\right)^n$$

where α_0 is the value of α at temperature T_0 in K. Set the exponent n to zero to obtain a temperature-independent value for α .

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 *densStates* corresponding to energies *Elist* in J/mol, ground-state energy *E0* in kJ/mol, and first reactive energy *Ereac* in kJ/mol. The collisions occur at temperature *T* in K and are described by the average energy transferred in a deactivating collision *dEdown* in kJ/mol. The algorithm here is implemented as described by Chang, Bozzelli, and Dean [Chang2000].

generateCollisionMatrix(*self*, double *T*, ndarray *densStates*, ndarray *Elist*, ndarray *Jlist=None*)

Generate and return the collision matrix $\mathbf{M}_{\text{coll}}/\omega = \mathbf{P} - \mathbf{I}$ corresponding to this collision model for a given set of energies *Elist* in J/mol, temperature *T* in K, and isomer density of states *densStates*.

getAlpha(*self*, double *T*) → double

Return the value of the α parameter - the average energy transferred in a deactivating collision - in J/mol at temperature *T* in K.

n

n: 'double'

Reaction events

Microcanonical rate coefficients

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 *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.

RRKM theory

`rmgpy.pdep.applyRRKMTheory(transitionState, ndarray Elist, ndarray Jlist, ndarray densStates)`

Calculate the microcanonical rate coefficient for a reaction using RRKM theory, where *transitionState* is the transition state of the reaction, *Elist* is the array of energies in J/mol at which to evaluate the microcanonical rate, and *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

$$k(E, J) = \frac{N^\ddagger(E, J)}{h\rho(E, J)}$$

where $N^\ddagger(E, J)$ 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

$$k(E) = \frac{N^\ddagger(E)}{h\rho(E)}$$

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 $E_0^{\ddagger} - E_0$.

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 bath 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 *Elist* in J/mol and, if the J-rotor is not active, the total angular momentum quantum numbers *Jlist*.

mapSumOfStates()

Return a mapping of the density of states for the configuration to the given energies *Elist* in J/mol and, if the J-rotor is not active, the total angular momentum quantum numbers *Jlist*.

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:

Attribute	Description
<i>isomers</i>	A list of the unimolecular isomers in the network
<i>reactants</i>	A list of the bimolecular reactant channels in the network
<i>products</i>	A list of the bimolecular product channels in the network
<i>pathReactions</i>	A list of “path” reaction objects that connect adjacent isomers (the high-pressure-limit)
<i>bathGas</i>	A dictionary of the bath gas species (keys) and their mole fractions (values)
<i>netReactions</i>	A list of “net” reaction objects that connect any pair of isomers
<i>T</i>	The current temperature in K
<i>P</i>	The current pressure in bar
<i>Elist</i>	The current array of energy grains in kJ/mol
<i>Jlist</i>	The current array of total angular momentum quantum numbers
<i>Nisom</i>	The number of unimolecular isomers in the network
<i>Nreac</i>	The number of bimolecular reactant channels in the network
<i>Nprod</i>	The number of bimolecular product channels in the network
<i>Ngrains</i>	The number of energy grains
<i>NJ</i>	The number of angular momentum grains
<i>activeKRotor</i>	True if the K-rotor is treated as active, False if treated as adiabatic
<i>activeJRotor</i>	True if the J-rotor is treated as active, False if treated as adiabatic
<i>rmgmode</i>	True if in RMG mode, False otherwise

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 interest using *Tmin* and *Tmax* in K and *Pmin* and *Pmax* 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

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.

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 *grainSize* in J/mol or the desired number of grains *Ngrains*, as well as a temperature *T* in K to use for the equilibrium calculation. You can specify both *grainSize* and *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 *Elist* in J/mol to use, the output time points *tlist* 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 *tlist* in s

and the initial total populations x_0 . 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 $b_i(E, J)$:

$$\lim_{t \rightarrow \infty} p_i(E, J, t) \propto b_i(E, J)$$

We can simplify by eliminating the angular momentum quantum number to get

$$p_i(E, t) = \sum_J p_i(E, J, t)$$

Let us also denote the (time-dependent) total population of isomer i by $x_i(t)$:

$$x_i(t) \equiv \sum_J \int_0^\infty p_i(E, J, t) dE$$

The two molecules of a reactant or product channel are free to move apart from one another and interact independently with other molecules in the system. Accordingly, we treat these channels as fully thermalized, leaving as the only variable the total concentrations $y_{nA}(t)$ and $y_{nB}(t)$ of the molecules A_n and B_n of reactant channel n . (Since the product channels act as infinite sinks, their populations do not need to be considered explicitly.)

Finally, we will use N_{isom} , N_{reac} , and N_{prod} as the numbers of isomers, reactant channels, and product channels, respectively, in the system.

The governing equation for the population distributions $p_i(E, J, t)$ of each isomer i and the reactant concentrations $y_{nA}(t)$ and $y_{nB}(t)$ combines the collision and reaction models to give a linear integro-differential equation:

$$\begin{aligned} \frac{d}{dt} p_i(E, J, t) = & \omega_i(T, P) \sum_{J'} \int_0^\infty P_i(E, J, E', J') p_i(E', J', t) dE' - \omega_i(T, P) p_i(E, J, t) \\ & + \sum_{j \neq i}^{N_{\text{isom}}} k_{ij}(E, J) p_j(E, J, t) - \sum_{j \neq i}^{N_{\text{isom}}} k_{ji}(E, J) p_i(E, J, t) \\ & + \sum_{n=1}^{N_{\text{reac}}} y_{nA}(t) y_{nB}(t) f_{in}(E, J) b_n(E, J, t) - \sum_{n=1}^{N_{\text{reac}} + N_{\text{prod}}} g_{ni}(E, J) p_i(E, J, t) \\ \frac{d}{dt} y_{nA}(t) = & \frac{d}{dt} y_{nB}(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^\infty g_{ni}(E, J) p_i(E, J, t) dE \\ & - \sum_{i=1}^{N_{\text{isom}}} y_{nA}(t) y_{nB}(t) \int_0^\infty f_{in}(E, J) b_n(E, J, t) dE \end{aligned}$$

A summary of the variables is given below:

Variable	Meaning
$p_i(E, J, t)$	Population distribution of isomer i
$y_{nA}(t)$	Total population of species A_n in reactant channel n
$\omega_i(T, P)$	Collision frequency of isomer i
$P_i(E, J, E', J')$	Collisional transfer probability from (E', J') to (E, J) for isomer i
$k_{ij}(E, J)$	Microcanonical rate coefficient for isomerization from isomer j to isomer i
$f_{im}(E, J)$	Microcanonical rate coefficient for association from reactant channel m to isomer i
$g_{nj}(E, J)$	Microcanonical rate coefficient for dissociation from isomer j to reactant or product channel n
$b_n(E, J, t)$	Boltzmann distribution for reactant channel n
N_{isom}	Total number of isomers
N_{reac}	Total number of reactant channels
N_{prod}	Total number of product channels

The above is called the two-dimensional master equation because it contains two dimensions: total energy E and total angular momentum quantum number J . In the first equation (for isomers), the first pair of terms correspond to collision, the second pair to isomerization, and the final pair to association/dissociation. Similarly, in the second equation above (for reactant channels), the pair of terms refer to dissociation/association.

We can also simplify the above to the one-dimensional form, which only has E as a dimension:

$$\begin{aligned}
 \frac{d}{dt} p_i(E, t) &= \omega_i(T, P) \int_0^\infty P_i(E, E') p_i(E', t) dE' - \omega_i(T, P) p_i(E, t) \\
 &+ \sum_{j \neq i}^{N_{\text{isom}}} k_{ij}(E) p_j(E, t) - \sum_{j \neq i}^{N_{\text{isom}}} k_{ji}(E) p_i(E, t) \\
 &+ \sum_{n=1}^{N_{\text{reac}}} y_{nA}(t) y_{nB}(t) f_{in}(E) b_n(E, t) - \sum_{n=1}^{N_{\text{reac}}+N_{\text{prod}}} g_{ni}(E) p_i(E, t) \\
 \frac{d}{dt} y_{nA}(t) &= \frac{d}{dt} y_{nB}(t) = \sum_{i=1}^{N_{\text{isom}}} \int_0^\infty g_{ni}(E) p_i(E, t) dE \\
 &- \sum_{i=1}^{N_{\text{isom}}} y_{nA}(t) y_{nB}(t) \int_0^\infty f_{in}(E) b_n(E, t) dE
 \end{aligned}$$

The equations as given are nonlinear, both due to the presence of the bimolecular reactants and because both ω_i and $P_i(E, E')$ depend on the composition, which is changing with time. The rate coefficients can be derived from considering the pseudo-first-order situation where $y_{nA}(t) \ll y_{nB}(t)$, and all $y(t)$ are negligible compared to the bath gas M. From these assumptions the changes in ω_i , $P_i(E, E')$, and all y_{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 *grains*. This converts the linear integro-differential equation into a system of first-order ordinary differential equations:

$$\frac{d}{dt} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ y_{1A} \\ y_{2A} \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{M}_1 & \mathbf{K}_{12} & \dots & \mathbf{F}_{11} \mathbf{b}_1 y_{1B} & \mathbf{F}_{12} \mathbf{b}_2 y_{2B} & \dots \\ \mathbf{K}_{21} & \mathbf{M}_2 & \dots & \mathbf{F}_{21} \mathbf{b}_1 y_{1B} & \mathbf{F}_{22} \mathbf{b}_2 y_{2B} & \dots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ (\mathbf{g}_{11})^T & (\mathbf{g}_{12})^T & \dots & h_1 & 0 & \dots \\ (\mathbf{g}_{21})^T & (\mathbf{g}_{22})^T & \dots & 0 & h_2 & \dots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ y_{1A} \\ y_{2A} \\ \vdots \end{bmatrix}$$

The diagonal matrices \mathbf{K}_{ij} and \mathbf{F}_{in} and the vector \mathbf{g}_{ni} contain the microcanonical rate coefficients for isomerization, association, and dissociation, respectively:

$$\begin{aligned}
 (\mathbf{K}_{ij})_{rs} &= \begin{cases} \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} k_{ij}(E) dE & r = s \\ 0 & r \neq s \end{cases} \\
 (\mathbf{F}_{in})_{rs} &= \begin{cases} \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} f_{in}(E) dE & r = s \\ 0 & r \neq s \end{cases} \\
 (\mathbf{g}_{ni})_r &= \frac{1}{\Delta E_r} \int_{E_r - \Delta E_r/2}^{E_r + \Delta E_r/2} g_{ni}(E) dE
 \end{aligned}$$

The matrices \mathbf{M}_i represent the collisional transfer probabilities minus the rates of reactive loss to other isomers and to reactants and products:

$$(\mathbf{M}_i)_{rs} = \begin{cases} \omega_i [P_i(E_r, E_r) - 1] - \sum_{j \neq i}^{N_{\text{isom}}} k_{ij}(E_r) - \sum_{n=1}^{N_{\text{reac}} + N_{\text{prod}}} g_{ni}(E_r) & r = s \\ \omega_i P_i(E_r, E_s) & r \neq s \end{cases}$$

The scalars h_n are simply the total rate coefficient for loss of reactant channel n due to chemical reactions:

$$h_n = - \sum_{i=1}^{N_{\text{isom}}} \sum_{r=1}^{N_{\text{grains}}} y_{nB} f_{in}(E_r) b_n(E_r)$$

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 \mathbf{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

`rmgpy.pdep.cse.applyChemicallySignificantEigenvaluesMethod()`

In the chemically-significant eigenvalues method, the master equation matrix is diagonalized 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 (`rmgpy.qm`)

The `rmgpy.qm` subpackage contains classes and functions for working with molecular geometries, and interfacing with quantum chemistry software.

1.8.1 Main

Class	Description
<code>QMSettings</code>	A class to store settings related to quantum mechanics calculations
<code>QMCalculator</code>	An object to store settings and previous calculations

1.8.2 Molecule

Class	Description
<code>Geometry</code>	A geometry, used for quantum calculations
<code>QMolecule</code>	A base class for QM Molecule calculations

1.8.3 QM Data

Class/Function	Description
<code>QMData</code>	General class for data extracted from a QM calculation
<code>CCLibData</code>	QM Data extracted from a cclib data object

1.8.4 QM Verifier

Class/Function	Description
<i>QMVerifier</i>	Verifies whether a QM job was successfully completed

1.8.5 Symmetry

Class/Function	Description
<i>PointGroup</i>	A symmetry Point Group
<i>PointGroupCalculator</i>	Wrapper type to determine molecular symmetry point groups based on 3D coordinates
<i>SymmetryJob</i>	Determine the point group using the SYMMETRY program

1.8.6 Gaussian

Class/Function	Description
<i>Gaussian</i>	A base class for all QM calculations that use Gaussian
<i>GaussianMol</i>	A base Class for calculations of molecules using Gaussian.
<i>GaussianMolPM3</i>	A base Class for calculations of molecules using Gaussian at PM3.
<i>GaussianMolPM6</i>	A base Class for calculations of molecules using Gaussian at PM6.

1.8.7 Mopac

Class/Function	Description
<i>Mopac</i>	A base class for all QM calculations that use Mopac
<i>MopacMol</i>	A base Class for calculations of molecules using Mopac.
<i>MopacMolPM3</i>	A base Class for calculations of molecules using Mopac at PM3.
<i>MopacMolPM6</i>	A base Class for calculations of molecules using Mopac at PM6.
<i>MopacMolPM7</i>	A base Class for calculations of molecules using Mopac at PM7.

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.

Attribute	Type	Description
<i>software</i>	str	Quantum chemical package name in common letters
<i>method</i>	str	Semi-empirical method
<i>fileStore</i>	str	The path to the QMfiles directory
<i>scratchDirectory</i>	str	The path to the scratch directory
<i>onlyCyclics</i>	bool	True if to run QM only on ringed species
<i>maxRadicalNumber</i>	int	Radicals larger than this are saturated before applying HBI

checkAllSet()

Check that all the required settings are set.

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:

Attribute	Type	Description
<i>settings</i>	QMSettings	Settings for QM calculations
<i>database</i>	ThermoLibrary	Database containing QM calculations

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:

Attribute	Type	Description
<i>settings</i>	QMSettings	Settings for QM calculations
<i>uniqueID</i>	str	A short ID such as an augmented InChI Key
<i>molecule</i>	Molecule	RMG Molecule object
<i>uniqueIDlong</i>	str	A long, truly unique ID such as an augmented InChI

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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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.QMolecule(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
<i>molecule</i>	Molecule	RMG Molecule object
<i>settings</i>	QMSettings	Settings for QM calculations
<i>uniqueID</i>	str	A short ID such as an augmented InChI Key
<i>uniqueIDlong</i>	str	A long, truly unique ID such as an augmented InChI

calculateChiralityCorrection()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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.

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 CCLibData object.

saveThermoData()

Save the generated thermo data.

scriptAttempts

The number of attempts with different script keywords

rmgpy.qm.qmdata

```
class rmgpy.qm.qmdata.QMData(groundStateDegeneracy=None,          numberOfAtoms=None,
                             stericEnergy=None, molecularMass=None, energy=0, atomicNum-
                             bers=[], rotationalConstants=[], atomCoords=[], frequencies=[],
                             source=None)
```

General class for data extracted from a QM calculation

groundStateDegeneracy = None

Electronic ground state degeneracy in RMG taken as number of radicals +1

numberOfAtoms = None

Number of atoms.

rotationalConstants = None

Rotational constants, in Hz.

stericEnergy = None

Steric energy.

```
class rmgpy.qm.qmdata.CCLibData(cclib_data, groundStateDegeneracy)
```

QM Data extracted from a cclib data object

This data objects collects information that CCLib was able to retrieve from a quantum chemistry output file.

cclib_data = None

data object returned by a parsing tool like CCLib.parse()

rmgpy.qm.qmverifier

class rmgpy.qm.qmverifier.QMVerifier(*molfile*)

Verifies whether a QM job (externalized) was successfully completed by

- searching for specific keywords in the output files,
- located in a specific directory (e.g. “QMFiles”)

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 succesful job.

parse()

Parses the results of the Gaussian calculation, and returns a CCLibData 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 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.

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 ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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 CCLibData 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 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 ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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)']

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 CCLibData 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.**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 ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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 = ['*# pm6 opt=(verytight,gdiis) freq IOP(2/16=3)*', '*# pm6 opt=(verytight,gdiis) freq IOP(2/16=3) IOP(4/21=2)*']

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 CCLibData 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']

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 `QMMolecule` and `Mopac`.

calculateChiralityCorrection()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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’, ‘bottom’: ‘oldgeo thermo nosym precise ‘}, {‘top’: ‘precise nosym gnorm=0.0 nonr’,

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 CCLibData 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.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 ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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 CCLibData 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.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

calculateChiralityCorrection()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ 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)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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 CCLibData 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 ($R \cdot \ln(2)$ 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*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot.

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 CCLibData 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

Class	Description
<i>ScalarQuantity</i>	A scalar physical quantity, with units and uncertainty
<i>ArrayQuantity</i>	An array physical quantity, with units and uncertainty
<i>Quantity()</i>	Return a scalar or array physical quantity

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:

Function	Unit type	SI unit
Acceleration()	acceleration	m/s^2
Area()	area	m^2
Concentration()	concentration	mol/cm^3
Dimensionless()	dimensionless	
Energy()	energy	J/mol
Entropy()	entropy	$\text{J/mol} \cdot \text{K}$
Flux()	flux	$\text{mol/cm}^2 \cdot \text{s}$
Frequency()	frequency	cm^{-1}
Force()	force	N
Inertia()	inertia	$\text{kg} \cdot \text{m}^2$
Length()	length	m
Mass()	mass	kg
Momentum()	momentum	$\text{kg} \cdot \text{m/s}$
Power()	power	W
Pressure()	pressure	Pa
RateCoefficient()	rate coefficient	s^{-1} , $\text{m}^3/\text{mol} \cdot \text{s}$, $\text{m}^6/\text{mol}^2 \cdot \text{s}$, $\text{m}^9/\text{mol}^3 \cdot \text{s}$
Temperature()	temperature	K
Time()	time	s
Velocity()	velocity	m/s
Volume()	volume	m^3

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 $\text{amu} \cdot \text{angstrom}^2$ instead of $\text{kg} \cdot \text{m}^2$. 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	Description
<i>value</i>	The numeric value of the quantity in the given units
<i>units</i>	The units the value was specified in
<i>uncertainty</i>	The numeric uncertainty in the value
<i>uncertaintyType</i>	The type of uncertainty: '+' '-' for additive, '*' '/' for multiplicative
<i>value_si</i>	The numeric value of the quantity in the corresponding SI units

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, and set the uncertainty to -1.

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 quantity, in the given units

rmgpy.quantity.ArrayQuantity

class rmgpy.quantity.ArrayQuantity

The *ScalarQuantity* class provides a representation of an array of physical quantity values, with optional units and uncertainty information. The attributes are:

Attribute	Description
<i>value</i>	The numeric value of the quantity in the given units
<i>units</i>	The units the value was specified in
<i>uncertainty</i>	The numeric uncertainty in the value
<i>uncertaintyType</i>	The type of uncertainty: '+' '-' for additive, '*' '/' for multiplicative
<i>value_si</i>	The numeric value of the quantity in the corresponding SI units

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.

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.

rmgpy.quantity.Quantity

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

Class	Description
<i>Reaction</i>	A chemical reaction

1.10.2 Reaction model

Class	Description
<code>ReactionModel</code>	A reaction model, comprising a list of species and a list of reactions

1.10.3 Exceptions

Class	Description
<code>ReactionError</code>	Raised when an error occurs while working with reactions

`rmgpy.reaction.Reaction`

class `rmgpy.reaction.Reaction`

A chemical reaction. The attributes are:

Attribute	Type	Description
<i>index</i>	<code>int</code>	A unique nonnegative integer index
<i>label</i>	<code>str</code>	A descriptive string label
<i>reactants</i>	<code>list</code>	The reactant species (as <code>Species</code> objects)
<i>products</i>	<code>list</code>	The product species (as <code>Species</code> objects)
<i>kinetics</i>	<code>KineticsModel</code>	The kinetics model to use for the reaction
<i>reversible</i>	<code>bool</code>	True if the reaction is reversible, False if not
<i>transition-State</i>	<code>TransitionState</code>	The transition state
<i>duplicate</i>	<code>bool</code>	True if the reaction is known to be a duplicate, False if not
<i>degeneracy</i>	<code>double</code>	The reaction path degeneracy for the reaction
<i>pairs</i>	<code>list</code>	Reactant-product pairings to use in converting reaction flux to species flux

`calculateMicrocanonicalRateCoefficient()`

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *Elist* in J/mol. *reactDensStates* 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

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

where Q^\ddagger 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 a factor of getDiffusionFactor to account for the diffusion limit.

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 an Arrhenius or KineticsData object.

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.

getRate()

Return the net rate of reaction at temperature T and pressure P . The parameter *conc* is a map with species as keys and concentrations as values. A reactant not found in the *conc* map is treated as having zero concentration.

If passed a *totalConc*, it won't bother recalculating it.

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 $A + B \rightleftharpoons C$ 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) $A \rightleftharpoons B + C$ or $A + B \rightleftharpoons C$ or $A \rightleftharpoons B$, 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.

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.reaction.ReactionModel**class rmgpy.reaction.ReactionModel**

A chemical reaction model, composed of a list of species and a list of reactions involving those species. The attributes are:

Attribute	Type	Description
<i>species</i>	list	The species involved in the reaction model
<i>reactions</i>	list	The reactions comprising the reaction model

generateStoichiometryMatrix()

Generate the stoichiometry matrix corresponding to the current reaction system. The stoichiometry matrix is defined such that the rows correspond to the *index* attribute of each species object, while the columns correspond to the *index* attribute of each reaction object.

getReactionRates()

Return an array of reaction rates for each reaction in the model core and edge. The id of the reaction is the index into the vector.

merge()

Return a new ReactionModel object that is the union of this model and *other*.

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

Class	Description
<i>Species</i>	A chemical species, with RMG-specific functionality
<i>CoreEdgeReactionModel</i>	A reaction model comprised of core and edge species and reactions

1.11.2 Input

Function	Description
<i>readInputFile()</i>	Load an RMG job input file
<i>saveInputFile()</i>	Save an RMG job input file

1.11.3 Output

Function	Description
<code>saveOutputHTML()</code>	Save the results of an RMG job to an HTML file
<code>saveDiffHTML()</code>	Save a comparison of two reaction mechanisms to an HTML file

1.11.4 Job classes

Class	Description
<code>RMG</code>	Main class for RMG jobs

1.11.5 Pressure dependence

Class	Description
<code>PDepReaction</code>	A pressure-dependent “net” reaction
<code>PDepNetwork</code>	A pressure-dependent unimolecular reaction network, with RMG-specific functionality

1.11.6 Exceptions

Exception	Description
<code>InputError</code>	Raised when an error occurs while working with an RMG input file
<code>OutputError</code>	Raised when an error occurs while saving an RMG output file
<code>PressureDependenceError</code>	Raised when an error occurs while computing pressure-dependent rate coefficients

rmgpy.rmg.model.CoreEdgeReactionModel

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:

Attribute	Description
<i>core</i>	The species and reactions of the current model core
<i>edge</i>	The species and reactions of the current model edge
<i>networkDict</i>	A dictionary of pressure-dependent reaction networks (<code>Network</code> objects) indexed by source.
<i>networkList</i>	A list of pressure-dependent reaction networks (<code>Network</code> objects)
<i>network-Count</i>	A counter for the number of pressure-dependent networks created

addReactionLibraryToEdge (*reactionLibrary*)

Add all species and reactions from *reactionLibrary*, a `KineticsPrimaryDatabase` object, to the model edge.

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 this 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).

checkForExistingSpecies(*molecule*)

Check to see if an existing species contains the same structure. Structure as *structure*. Returns True or False and the matched species (if found, or None if not).

enlarge(*newObject*)

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.

failsSpeciesConstraints(*species*)

Pass in either a *Species* or *Molecule* object and checks whether it passes the speciesConstraints set by the user. If not, returns *True* for failing speciesConstraints.

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.

getReactionRates(*T*, *P*, *Ci*)

Return an array of reaction rates for each reaction in the model core and edge. The id of the reaction is the

index into the vector.

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.

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*)

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*)

Remove species from the model edge based on the simulation results from the list of *reactionSystems*.

react(*database*, *speciesA*, *speciesB=None*)

Generates reactions involving *rmgpy.species.Species* speciesA and speciesB.

removeSpeciesFromEdge(*spec*)

Remove species *spec* from the reaction model edge.

saveChemkinFile(*path*, *verbose_path*, *dictionaryPath=None*, *transportPath=None*, *saveEdgeSpecies=False*)

Save a Chemkin file for the current model as well as any desired output species and reactions to *path*. If *saveEdgeSpecies* is True, then a chemkin file and dictionary file for the core and edge species and reactions will be saved.

updateUnimolecularReactionNetworks(*database*)

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.

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, logFile=None, outputDirectory=None, scratchDirectory=None*)

A representation of a Reaction Mechanism Generator (RMG) job. The attributes are:

Attribute	Description
<i>inputFile</i>	The path to the input file
<i>logFile</i>	The path to the log file
<i>databaseDirectory</i>	The directory containing the RMG database
<i>thermoLibraries</i>	The thermodynamics libraries to load
<i>reactionLibraries</i>	The kinetics libraries to load
<i>statmechLibraries</i>	The statistical mechanics libraries to load
<i>seedMechanisms</i>	The seed mechanisms included in the model
<i>kineticsFamilies</i>	The kinetics families to use for reaction generation
<i>kineticsDepositories</i>	The kinetics depositories to use for looking up kinetics in each family
<i>kineticsEstimator</i>	The method to use to estimate kinetics: 'group additivity' or 'rate rules'
<i>solvent</i>	If solvation estimates are required, the name of the solvent.
<i>reactionModel</i>	The core-edge reaction model generated by this job
<i>reactionSystems</i>	A list of the reaction systems used in this job
<i>database</i>	The RMG database used in this job
<i>absoluteTolerance</i>	The absolute tolerance used in the ODE/DAE solver
<i>relativeTolerance</i>	The relative tolerance used in the ODE/DAE solver
<i>sensitivityAbsoluteTolerance</i>	The absolute tolerance used in the ODE/DAE solver for the sensitivities
<i>sensitivityRelativeTolerance</i>	The relative tolerance used in the ODE/DAE solver for the sensitivities
<i>fluxToleranceKeepInEdge</i>	The relative species flux below which species are discarded from the edge
<i>fluxToleranceMoveToCore</i>	The relative species flux above which species are moved from the edge to the core
<i>fluxToleranceInterrupt</i>	The relative species flux above which the simulation will halt
<i>maximumEdgeSpecies</i>	The maximum number of edge species allowed at any time
<i>termination</i>	A list of termination targets (i.e <code>TerminationTime</code> and <code>TerminationConversion</code> objects)
<i>speciesConstraints</i>	Dictates the maximum number of atoms, carbons, electrons, etc. generated by RMG
<i>outputDirectory</i>	The directory used to save output files
<i>scratchDirectory</i>	The directory used to save temporary files
<i>verbosity</i>	The level of logging verbosity for console output
<i>loadRestart</i>	True if restarting a previous job, False otherwise
<i>saveRestartPeriod</i>	The time period to periodically save a restart file (<code>Quantity</code>), or None for never.
<i>units</i>	The unit system to use to save output files (currently must be 'si')

Table 1.3 – continued from previous page

Attribute	Description
<i>drawMolecules</i>	True to draw pictures of the species in the core, False otherwise
<i>generatePlots</i>	True to generate plots of the job execution statistics after each iteration, False otherwise
<i>verboseComments</i>	True to keep the verbose comments for database estimates, False otherwise
<i>saveEdgeSpecies</i>	True to save chemkin and HTML files of the edge species, False otherwise
<i>pressureDependence</i>	Whether to process unimolecular (pressure-dependent) reaction networks
<i>quantumMechanics</i>	Whether to apply quantum mechanical calculations instead of group additivity to certain molecular types
<i>wallTime</i>	The maximum amount of CPU time in seconds to expend on this job; used to stop gracefully so we can
<i>initializationTime</i>	The time at which the job was initiated, in seconds since the epoch (i.e. from time.time())
<i>done</i>	Whether the job has completed (there is nothing new to add)

checkInput()

Check for a few common mistakes in the input file.

checkLibraries()

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).

execute(args)

Execute an RMG job using the command-line arguments *args* as returned by the *argparse* package.

finish()

Complete the model generation.

generateExecutionPlots(execTime, coreSpeciesCount, coreReactionCount, edgeSpeciesCount, edgeReactionCount, memoryUse, restartSize)

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

initialize(args)

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.

makeOutputSubdirectory(folder)

Create a subdirectory *folder* in the output directory. If the folder already exists (e.g. from a previous job) its contents are deleted.

readMeaningfulLineJava(*f*)

Read a meaningful line from an RMG-Java condition file object *f*, returning the line with any comments removed.

saveChemkinFiles()

Save the current reaction model to a set of Chemkin files.

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`.

saveExecutionStatistics(*execTime*, *coreSpeciesCount*, *coreReactionCount*, *edgeSpeciesCount*, *edgeReactionCount*, *memoryUse*, *restartSize*)

Save the statistics of the RMG job to an Excel spreadsheet for easy viewing after the run is complete. The statistics are saved to the file *statistics.xls* in the output directory. The `xlwt` package is used to create the spreadsheet file; if this package is not installed, no file is saved.

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.

saveOutputHTML()

Save the current reaction model to a pretty HTML file.

saveRestartFile(*path*, *reactionModel*, *delay=0*)

Save a restart file to *path* on disk containing the contents of the provided *reactionModel*. The *delay* parameter is a time in seconds; if the restart file is not at least that old, the save is aborted. (Use the default value of 0 to force the restart file to be saved.)

rmgpy.rmg.main.initializeLog(*verbose*, *log_file_name*)

Set up a logger for RMG 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.

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.

rmgpy.rmg.main.processProfileStats(*stats_file*, *log_file*)

Saving RMG output

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).

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

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.

Attribute	Type	Description
<i>source</i>	list	The isomer or reactant channel that acts as the source
<i>explored</i>	list	A list of the unimolecular isomers whose reactions have been fully explored

addPathReaction(*newReaction, newSpecies*)

Add a path reaction to the network. If the path reaction already exists, no action is taken.

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.

cleanup()

Delete intermediate arrays used to compute $k(T, P)$ values.

exploreIsomer(*isomer, reactionModel, database*)

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 nonexplored 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(*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 interest using *Tmin* and *Tmax* in K and *Pmin* and *Pmax* 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 *grainSize* in J/mol or the desired number of grains *Ngrains*, as well as a temperature *T* in K to use for the equilibrium calculation. You can specify both *grainSize* and *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 *Elist* in J/mol to use, the output time points *tlist* 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 *tlist* 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.

update(*reactionModel*, *database*, *pdepSettings*)

Regenerate the $k(T, P)$ values for this partial network if the network is marked as invalid.

updateConfigurations(*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 *reactionModel* because some decisions on sorting are made based on which species are in the model core.

rmgpy.rmg.pdep.PDepReaction

```
class rmgpy.rmg.pdep.PDepReaction(index=-1, label='', reactants=None, products=None, net-
                                work=None, kinetics=None, reversible=True, transition-
                                State=None, duplicate=False, degeneracy=1, pairs=None)
```

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

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

where Q^\ddagger 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 a factor of `getDiffusionFactor` to account for the diffusion limit.

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	$A \rightarrow C$	(A,C)
Dissociation	$A \rightarrow C + D$	(A,C), (A,D)
Association	$A + B \rightarrow C$	(A,C), (B,C)
Bimolecular	$A + B \rightarrow C + D$	(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 an Arrhenius or `KineticsData` object.

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.

getRate()

Return the net rate of reaction at temperature *T* and pressure *P*. The parameter *conc* is a map with species as keys and concentrations as values. A reactant not found in the *conc* map is treated as having zero concentration.

If passed a *totalConc*, it won't bother recalculating it.

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()

Get the source of this PDepReaction

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 $A + B \rightleftharpoons C$ 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) $A \rightleftharpoons B + C$ or $A + B \rightleftharpoons C$ or $A \rightleftharpoons B$, 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.

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, dipoleMoment=None,
                              polarizability=None, Zrot=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.

fromAdjacencyList()

Load the structure of a species as a `Molecule` object from the given adjacency list *adlist* 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(database)

Generate molecular degree of freedom data for the species. You must have already provided a thermodynamics model using e.g. `generateThermoData()`.

generateThermoData(database, thermoClass=<type 'rmgpy.thermo.nasa.NASA'>, quantumMechanics=None)

Generates thermo data, first checking Libraries, then using either QM or Database.

If quantumMechanics is not None, it is asked to calculate the thermo. Failing that, the database is used.

The database generates the thermo data for each structure (resonance isomer), picks that with lowest H298 value.

It then calls `processThermoData()`, to convert (via Wilhoit) to NASA and set the E0.

Result stored in *self.thermo* and returned.

generateTransportData(database)

Generate the transportData parameters for the species.

getDensityOfStates()

Return the density of states $\rho(E) dE$ at the specified energies *Elist* 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 $Elist$ 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.

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.

processThermoData(database, thermo0, thermoClass=<type 'rmgpy.thermo.nasa.NASA'>)

Converts via Wilhoit into required *thermoClass* and sets *E0*.

Resulting thermo is stored (*self.thermo*) and returned.

toAdjacencyList()

Return a string containing each of the molecules' adjacency lists.

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

Class	Description
<code>ReactionSystem</code>	Base class for all reaction systems
<code>SimpleReactor</code>	A simple isothermal, isobaric, well-mixed batch reactor

1.12.2 Termination criteria

Class	Description
<code>TerminationTime</code>	Represent a time at which the simulation should be terminated
<code>TerminationConversion</code>	Represent a species conversion at which the simulation should be terminated

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 *tout*, 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 *dydt* attributes.

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.

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 *dydt*. Return a numpy array with the values of the residual function and an integer with status information (0 if okay, -2 to terminate).

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.

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 *tout*, 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 *dydt* attributes.

computeRateDerivative()

Returns derivative vector df/dk_j where $dy/dt = f(y, t, k)$ and k_j is the rate parameter for the *j*th core reaction.

convertInitialKeysToSpeciesObjects()

Convert the initialMoleFractions dictionary from species names into species objects, using the given dictionary of species.

initialize()

Initialize the DASPK solver by setting the initial values of the independent variable t_0 , dependent variables y_0 , and first derivatives $dydt_0$. 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.

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.

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

Class	Description
<i>Species</i>	A chemical species

1.13.2 Transition state

Class	Description
<i>TransitionState</i>	A transition state

1.13.3 Exceptions

Class	Description
<i>SpeciesError</i>	Raised when an error occurs while working with species

rmgpy.species.Species

class rmgpy.species.Species

A chemical species, representing a local minimum on a potential energy surface. The attributes are:

Attribute	Description
<i>index</i>	A unique nonnegative integer index
<i>label</i>	A descriptive string label
<i>thermo</i>	The heat capacity model for the species
<i>conformer</i>	The molecular conformer for the species
<i>molecule</i>	A list of the <i>Molecule</i> objects describing the molecular structure
<i>transportData</i>	A set of transport collision parameters
<i>molecularWeight</i>	The molecular weight of the species
<i>dipoleMoment</i>	The molecular dipole moment
<i>polarizability</i>	The polarizability alpha
<i>Zrot</i>	The rotational relaxation collision number
<i>energyTransferModel</i>	The collisional energy transfer model to use
<i>reactive</i>	True if the species participates in reactions, False if not
<i>'props'</i>	A generic 'properties' dictionary to store user-defined flags

note: `rmgpy.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.

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.

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.

getDensityOfStates()

Return the density of states $\rho(E) dE$ at the specified energies *Elist* 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 *Elist* 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.

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.

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:

Attribute	Description
<i>label</i>	A descriptive string label
<i>conformer</i>	The molecular degrees of freedom model for the species
<i>frequency</i>	The negative frequency of the first-order saddle point
<i>tunneling</i>	The type of tunneling model to use for tunneling through the reaction barrier
<i>degeneracy</i>	The reaction path degeneracy

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 *Elist* in J/mol.

getDensityOfStates()

Return the density of states $\rho(E) dE$ at the specified energies *Elist* 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 *Elist* 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 computation 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

Class	Description
<i>IdealGasTranslation</i>	A model of three-dimensional translation of an ideal gas

1.14.2 Rotational degrees of freedom

Class	Description
<i>LinearRotor</i>	A model of two-dimensional rigid rotation of a linear molecule
<i>NonlinearRotor</i>	A model of three-dimensional rigid rotation of a nonlinear molecule
<i>KRotor</i>	A model of one-dimensional rigid rotation of a K-rotor
<i>SphericalTopRotor</i>	A model of three-dimensional rigid rotation of a spherical top molecule

1.14.3 Vibrational degrees of freedom

Class	Description
<i>HarmonicOscillator</i>	A model of a set of one-dimensional harmonic oscillators

1.14.4 Torsional degrees of freedom

Class	Description
<i>HinderedRotor</i>	A model of a one-dimensional hindered rotation

1.14.5 The Schrodinger equation

Class	Description
<i>getPartitionFunction()</i>	Calculate the partition function at a given temperature from energy levels and degeneracies
<i>getHeatCapacity()</i>	Calculate the dimensionless heat capacity at a given temperature from energy levels and degeneracies
<i>getEnthalpy()</i>	Calculate the enthalpy at a given temperature from energy levels and degeneracies
<i>getEntropy()</i>	Calculate the entropy at a given temperature from energy levels and degeneracies
<i>getSumOfStates()</i>	Calculate the sum of states for a given energy domain from energy levels and degeneracies
<i>getDensityOfStates()</i>	Calculate the density of states for a given energy domain from energy levels and degeneracies

1.14.6 Convolution

Class	Description
<i>convolve()</i>	Return the convolution of two arrays
<i>convolveBS()</i>	Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart (BS) direct count algorithm
<i>convolveBSSR()</i>	Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart-Stein-Rabinovitch (BSSR) direct count algorithm

1.14.7 Molecular conformers

Class	Description
<i>Conformer</i>	A model of a molecular conformation

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:

Attribute	Description
<i>mass</i>	The mass of the translating object
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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 Schrodinger 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:

$$E_{n_x, n_y, n_z} = \frac{\hbar^2}{2M} \left[\left(\frac{n_x \pi}{L_x} \right)^2 + \left(\frac{n_y \pi}{L_y} \right)^2 + \left(\frac{n_z \pi}{L_z} \right)^2 \right] \quad n_x, n_y, n_z = 1, 2, \dots$$

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:

$$Q_{\text{trans}}^{\text{cl}}(T) = \left(\frac{2\pi M k_B T}{h^2} \right)^{3/2} V$$

For a constant-pressure problem we can use the ideal gas law to replace V with $k_B T/P$. This gives the partition function a temperature dependence of $T^{5/2}$.

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*) → double

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*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(*self*, double *T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, ndarray *Elist*, ndarray *sumStates0=None*) → 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.

mass

The mass of the translating object.

quantum

quantum: 'bool'

rmgpy.statmech.LinearRotor

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:

Attribute	Description
<i>inertia</i>	The moment of inertia of the rotor
<i>rotationalConstant</i>	The rotational constant of the rotor
<i>symmetry</i>	The symmetry number of the rotor
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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

$$E_J = BJ(J+1) \quad J = 0, 1, 2, \dots$$

$$g_J = 2J + 1$$

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:

$$Q_{\text{rot}}(T) = \frac{1}{\sigma} \sum_{J=0}^{\infty} (2J+1) e^{-BJ(J+1)/k_B T}$$

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 σ as the symmetry number of the rigid rotor.

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*) → *double*

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*self*, *double T*) → 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*) → int

Return the degeneracy of level *J*.

getLevelEnergy(*self*, *int J*) → double

Return the energy of level *J* in kJ/mol.

getPartitionFunction(*self*, *double T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, *ndarray Elist*, *ndarray sumStates0=None*) → 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.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:

Attribute	Description
<i>inertia</i>	The moments of inertia of the rotor
<i>rotationalConstant</i>	The rotational constants of the rotor
<i>symmetry</i>	The symmetry number of the rotor
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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 `NotImplementedError` 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:

$$Q_{\text{rot}}^{\text{cl}}(T) = \frac{\pi^{1/2}}{\sigma} \left(\frac{8k_B T}{h^2} \right)^{3/2} \sqrt{I_A I_B I_C}$$

getDensityOfStates(*self*, *ndarray Elist*, *ndarray densStates0=None*) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *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*) → double

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*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(*self*, *double T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, *ndarray Elist*, *ndarray sumStates0=None*) → 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 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**(*inertia=None*, *symmetry=1*, *quantum=False*, *rotationalConstant=None*)

A statistical mechanical model of an active K-rotor (a one-dimensional rigid rotor). The attributes are:

Attribute	Description
<i>inertia</i>	The moment of inertia of the rotor in amu*angstrom^2
<i>rotationalConstant</i>	The rotational constant of the rotor in cm^-1
<i>symmetry</i>	The symmetry number of the rotor
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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

$$E_K = BK^2 \quad K = 0, \pm 1, \pm 2, \dots$$

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:

$$Q_{\text{rot}}(T) = \frac{1}{\sigma} \left(1 + \sum_{K=1}^{\infty} 2e^{-BK^2/k_{\text{B}}T} \right)$$

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:

$$Q_{\text{rot}}^{\text{cl}}(T) = \frac{1}{\sigma} \left(\frac{8\pi^2 I k_{\text{B}} T}{h^2} \right)^{1/2}$$

where σ is the symmetry number of the K-rotor.

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*) → double

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*self*, double *T*) → 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*) → int

Return the degeneracy of level *J*.

getLevelEnergy(*self*, int *J*) → double

Return the energy of level *J* in kJ/mol.

getPartitionFunction(*self*, double *T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, ndarray *Elist*, ndarray *sumStates0=None*) → 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.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:

Attribute	Description
<i>inertia</i>	The moment of inertia of the rotor
<i>rotationalConstant</i>	The rotational constant of the rotor
<i>symmetry</i>	The symmetry number of the rotor
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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

$$E_J = BJ(J+1) \quad J = 0, 1, 2, \dots$$

$$g_J = (2J+1)^2$$

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:

$$Q_{\text{rot}}(T) = \frac{1}{\sigma} \sum_{J=0}^{\infty} (2J+1)^2 e^{-BJ(J+1)/k_B T}$$

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:

$$Q_{\text{rot}}^{\text{cl}}(T) = \frac{1}{\sigma} \left(\frac{8\pi^2 I k_B T}{h^2} \right)^{3/2}$$

where σ is the symmetry number of the spherical top. Note that the above differs from the nonlinear rotor partition function by a factor of π .

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*) → double

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*self*, double *T*) → 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*) → int

Return the degeneracy of level *J*.

getLevelEnergy(*self*, int *J*) → double

Return the energy of level *J* in kJ/mol.

getPartitionFunction(*self*, double *T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, ndarray *Elist*, ndarray *sumStates0=None*) → 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:

Attribute	Description
<i>frequencies</i>	The vibrational frequencies of the oscillators
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model

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 Schrodinger equation for such an oscillator is given by

$$-\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} \Psi(x) + \frac{1}{2} m \omega^2 x^2 \Psi(x) = E \Psi(x)$$

where m is the total mass of the particle. The harmonic potential results in quantized solutions to the above with the following energy levels:

$$E_n = \left(n + \frac{1}{2} \right) \hbar \omega \quad n = 0, 1, 2, \dots$$

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:

$$Q_{\text{vib}}(T) = \sum_{n=0}^{\infty} \exp \left(-\frac{\left(n + \frac{1}{2} \right) \hbar \omega}{k_B T} \right)$$

This summation can be evaluated explicitly to give a closed-form analytical expression for the vibrational partition function of a quantum harmonic oscillator:

$$Q_{\text{vib}}(T) = \frac{e^{-\hbar \omega / 2 k_B T}}{1 - e^{-\hbar \omega / k_B T}}$$

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

$$Q_{\text{vib}}(T) = \frac{1}{1 - e^{-\hbar \omega / k_B T}}$$

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:

$$Q_{\text{vib}}^{\text{cl}}(T) = \frac{k_B T}{\hbar \omega}$$

frequencies

The vibrational frequencies of the oscillators.

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*) → *double*

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

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(*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(*self*, *double T*) → *double*

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getSumOfStates(*self*, *ndarray Elist*, *ndarray sumStates0=None*) → *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.

quantum

quantum: 'bool'

Torsional degrees of freedom

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:

Attribute	Description
<i>inertia</i>	The moment of inertia of the rotor
<i>rotationalConstant</i>	The rotational constant of the rotor
<i>symmetry</i>	The symmetry number of the rotor
<i>fourier</i>	The $2 \times N$ array of Fourier series coefficients
<i>barrier</i>	The barrier height of the cosine potential
<i>quantum</i>	True to use the quantum mechanical model, False to use the classical model
<i>semiclassical</i>	True to use the semiclassical correction, False otherwise

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 Schrodinger 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 σ 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:

$$Q_{\text{hind}}^{\text{cl}}(T) = \left(\frac{2\pi I k_{\text{B}} T}{h^2} \right)^{1/2} \frac{2\pi}{\sigma} \exp \left(-\frac{V_0}{2k_{\text{B}} T} \right) I_0 \left(\frac{V_0}{2k_{\text{B}} T} \right)$$

A semiclassical correction to the above is usually required to provide a reasonable estimate of the partition function:

$$\begin{aligned} Q_{\text{hind}}^{\text{semi}}(T) &= \frac{Q_{\text{vib}}^{\text{quant}}(T)}{Q_{\text{vib}}^{\text{cl}}(T)} Q_{\text{hind}}^{\text{cl}}(T) \\ &= \frac{h\nu}{k_{\text{B}} T} \frac{1}{1 - \exp(-h\nu/k_{\text{B}} T)} \left(\frac{2\pi I k_{\text{B}} T}{h^2} \right)^{1/2} \frac{2\pi}{\sigma} \exp \left(-\frac{V_0}{2k_{\text{B}} T} \right) I_0 \left(\frac{V_0}{2k_{\text{B}} T} \right) \end{aligned}$$

Above we have defined ν 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 2π , 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 2π , with the minimum energy conformation having a potential of zero be placed at zero angle.

fourier

The $2xN$ array of Fourier series coefficients.

frequency

frequency: 'double'

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) → *double*

Return the enthalpy in J/mol for the degree of freedom at the specified temperature *T* in K.

getEntropy(*self*, *double* T) → *double*

Return the entropy in J/mol*K for the degree of freedom at the specified temperature *T* in K.

getFrequency(*self*) → *double*

Return the frequency of vibration in cm^{-1} corresponding to the limit of harmonic oscillation.

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.

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.

getLevelDegeneracy(*self*, *int J*) → int

Return the degeneracy of level *J*.

getLevelEnergy(*self*, *int J*) → double

Return the energy of level *J* in J.

getPartitionFunction(*self*, *double T*) → double

Return the value of the partition function $Q(T)$ at the specified temperature *T* in K.

getPotential(*self*, *double phi*) → double

Return the value of the hindered rotor potential $V(\phi)$ in J/mol at the angle *phi* in radians.

getSumOfStates(*self*, *ndarray Elist*, *ndarray sumStates0=None*) → 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.

semiclassical

semiclassical: 'bool'

solveSchrodingerEquation(*self*, *int Nbasis=401*) → ndarray

Solves the one-dimensional time-independent Schrodinger 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 Schrodinger equation and its solution. In particular, it contains functions for using the energy levels and corresponding degeneracies obtained from solving the Schrodinger 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 Schrodinger 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 Schrodinger 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 Schrodinger 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 Schrodinger 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 Schrodinger 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.getHeatCapacity`(*double T*, *energy*, *degeneracy=unitDegeneracy*, *int n0=0*, *int nmax=10000*, *double tol=1e-12*) → *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 *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.getPartitionFunction`(*double T*, *energy*, *degeneracy=unitDegeneracy*, *int n0=0*, *int nmax=10000*, *double tol=1e-12*) → *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 *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.getSumOfStates`(*ndarray Elist*, *energy*, *degeneracy=unitDegeneracy*, *int n0=0*, *ndarray sumStates0=None*) → *ndarray*

Return the values of the sum of states $N(E)$ for a given set of energies *Elist* in J/mol above the ground state using an initial sum of states *sumStates0*. The solution to the Schrodinger 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.unitDegeneracy`(*n*)

rmgpy.statmech.Conformer

`class 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:

Attribute	Description
<i>E0</i>	The ground-state energy (including zero-point energy) of the conformer
<i>modes</i>	A list of the molecular degrees of freedom
<i>spinMultiplicity</i>	The degeneracy of the electronic ground state
<i>opticalIsomers</i>	The number of optical isomers
<i>number</i>	An array of atomic numbers of each atom in the conformer
<i>mass</i>	An array of masses of each atom in the conformer
<i>coordinates</i>	An array of 3D coordinates of each atom in the conformer

Note that the *spinMultiplicity* reflects the electronic mode of the molecular system.

E0

The ground-state energy (including zero-point energy) of the conformer.

coordinates

An array of 3D coordinates of each atom in the conformer.

getActiveModes(*self*, *bool activeJRotor=False*, *bool activeKRotor=True*) → list

Return a list of the active molecular degrees of freedom of the molecular system.

getCenterOfMass(*self*, *atoms=None*) → ndarray

Calculate and return the [three-dimensional] position of the center of mass of the conformer 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 Pitzer³, which is described as $I^{(2,3)}$ by East and Radom⁴. 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*m². If the coordinates are not at the center of mass, they are temporarily shifted there for the purposes of this calculation.

³ Pitzer, K. S. *J. Chem. Phys.* **14**, p. 239-243 (1946).

⁴ East, A. L. L. and Radom, L. *J. Chem. Phys.* **106**, p. 6655-6674 (1997).

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 $\text{kg}\cdot\text{m}^2$, 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

Class	Description
<i>ThermoData</i>	A heat capacity model based on a set of discrete heat capacity points
<i>Wilhoit</i>	A heat capacity model based on the Wilhoit polynomial
<i>NASA</i>	A heat capacity model based on a set of NASA polynomials
<i>NASAPolynomial</i>	A heat capacity model based on a single NASA polynomial

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:

Attribute	Description
<i>Tdata</i>	An array of temperatures at which the heat capacity is known
<i>Cpdata</i>	An array of heat capacities at the given temperatures
<i>H298</i>	The standard enthalpy of formation at 298 K
<i>S298</i>	The standard entropy at 298 K
<i>Cp0</i>	The heat capacity at zero temperature
<i>CpInf</i>	The heat capacity at infinite temperature
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>E0</i>	The energy at zero Kelvin (including zero point energy)
<i>comment</i>	Information about the model (e.g. its source)

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

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.

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 *Tint*
 - 3: constrain $C_p(T)$, $\frac{dC_p}{dT}$, and $\frac{d^2C_p}{dT^2}$ to be continuous at *Tint*
 - 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 *Tint*
 - 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 *Tint*

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.

toWilhoit(*self*) → *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.

rmgpy.thermo.Wilhoit

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:

Attribute	Description
<i>Cp0</i>	The heat capacity at zero temperature
<i>CpInf</i>	The heat capacity at infinite temperature
<i>a0</i>	The zeroth-order Wilhoit polynomial coefficient
<i>a1</i>	The first-order Wilhoit polynomial coefficient
<i>a2</i>	The second-order Wilhoit polynomial coefficient
<i>a3</i>	The third-order Wilhoit polynomial coefficient
<i>H0</i>	The integration constant for enthalpy (not H at T=0)
<i>S0</i>	The integration constant for entropy (not S at T=0)
<i>E0</i>	The energy at zero Kelvin (including zero point energy)
<i>B</i>	The Wilhoit scaled temperature coefficient in K
<i>Tmin</i>	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
<i>comment</i>	Information about the model (e.g. its source)

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

$$H(T) = H_0 + C_p(0)T + [C_p(\infty) - C_p(0)] T \left\{ \left[2 + \sum_{i=0}^3 a_i \right] \left[\frac{1}{2}y - 1 + \left(\frac{1}{y} - 1 \right) \ln \frac{T}{y} \right] + y^2 \sum_{i=0}^3 \frac{y^i}{(i+2)(i+3)} \sum_{j=0}^3 f_{ij} a_j \right\}$$

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

$$S(T) = S_0 + C_p(\infty) \ln T - [C_p(\infty) - C_p(0)] \left[\ln y + \left(1 + y \sum_{i=0}^3 \frac{a_i y^i}{2+i} \right) y \right]$$

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_{atoms} atoms and N_{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/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 `None` if not defined.

Tmin

The minimum temperature at which the model is valid, or `None` if not defined.

a0

a0: 'double'

a1

a1: 'double'

a2

a2: 'double'

a3

a3: 'double'

comment

comment: str

copy(*self*) → Wilhoit

Return a copy of the Wilhoit object.

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

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 *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 *Cpdata* at a given set of temperatures *Tdata* in K, along with the dimensionless heat capacity at zero and infinite temperature, the dimensionless 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 *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 Wilhoit 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 *Tint*
 - 3: constrain $C_p(T)$, $\frac{dC_p}{dT}$, and $\frac{d^2C_p}{dT^2}$ to be continuous at *Tint*
 - 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 *Tint*
 - 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 *Tint*

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*, *comment=''*)

A heat capacity model based on a set of one, two, or three NASAPolynomial objects. The attributes are:

Attribute	Description
<i>polynomials</i>	The list of NASA polynomials to use in this model
<i>Tmin</i>	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
<i>E0</i>	The energy at zero Kelvin (including zero point energy)
<i>comment</i>	Information about the model (e.g. its source)

The NASA polynomial is another representation of the heat capacity, enthalpy, and entropy using seven or nine coefficients $\mathbf{a} = [a_{-2} \ a_{-1} \ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6]$. 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.

E0

The ground state energy (J/mol) at zero Kelvin, including zero point energy, or `None` if not yet specified.

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.

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`

toThermoData(*self*, *double Cp0=0.0*, *double CpInf=0.0*) → `ThermoData`

Convert the Wilhoit model to a `ThermoData` object.

toWilhoit(*self*, *double Cp0*, *double CpInf*) → `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 *Nrotors* so the algorithm can determine the appropriate heat capacity limits at zero and infinite temperature.

Here is an example of a NASA entry:

```
entry(
  index = 2,
  label = "octane",
  molecule =
    """
      1 C 0 {2,S}
      2 C 0 {1,S} {3,S}
      3 C 0 {2,S} {4,S}
      4 C 0 {3,S} {5,S}
      5 C 0 {4,S} {6,S}
      6 C 0 {5,S} {7,S}
      7 C 0 {6,S} {8,S}
      8 C 0 {7,S}
    """,
  thermo = NASA(
    polynomials = [
      NASAPolynomial(coeffs=[1.25245480E+01, -1.01018826E-02, 2.21992610E-04, -2.84863722E-07, 1.12410138E-10, -2.
      NASAPolynomial(coeffs=[2.09430708E+01, 4.41691018E-02, -1.53261633E-05, 2.30544803E-09, -1.29765727E-13, -3.
    ],
    Tmin = (200, 'K'),
    Tmax = (6000, 'K'),
  ),
  reference = Reference(authors=["check on burcat"], title='burcat', year="1999", url="http://www.me.berkeley.edu
  referenceType = "review",
  shortDesc = u"",
  longDesc =
    u""
    """
)
```

rmgpy.thermo.NASAPolynomial

class rmgpy.thermo.**NASAPolynomial**(*coeffs=None, Tmin=None, Tmax=None, E0=None, comment=''*)

A heat capacity model based on the NASA polynomial. Both the seven-coefficient and nine-coefficient variations are supported. The attributes are:

Attribute	Description
<i>coeffs</i>	The seven or nine NASA polynomial coefficients
<i>Tmin</i>	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
<i>E0</i>	The energy at zero Kelvin (including zero point energy)
<i>comment</i>	Information about the model (e.g. its source)

The NASA polynomial is another representation of the heat capacity, enthalpy, and entropy using seven or nine coefficients $\mathbf{a} = [a_{-2} \ a_{-1} \ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6]$. 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; the [NASA](#) class is available for this purpose.

E0

The ground state energy (J/mol) at zero Kelvin, including zero point energy, or **None** if not yet specified.

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.

c0

c0: 'double'

c1

c1: 'double'

c2

c2: 'double'

c3

c3: 'double'

c4

c4: 'double'

c5

c5: 'double'

c6

c6: 'double'

changeBaseEnthalpy(*self*, double *deltaH*)Add *deltaH* in J/mol to the base enthalpy of formation H298.**cm1**

cm1: 'double'

cm2

cm2: 'double'

coeffs

The set of seven or nine NASA polynomial coefficients.

comment

comment: str

discrepancy(*self*, *HeatCapacityModel* *other*) → doubleReturn 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*) → doubleReturn the enthalpy in J/mol at the specified temperature *T* in K.**getEntropy**(*self*, double *T*) → doubleReturn the entropy in J/mol*K at the specified temperature *T* in K.**getFreeEnergy**(*self*, double *T*) → doubleReturn the Gibbs free energy in J/mol at the specified temperature *T* in K.**getHeatCapacity**(*self*, double *T*) → doubleReturn the constant-pressure heat capacity in J/mol*K at the specified temperature *T* in K.**isIdenticalTo**(*self*, *HeatCapacityModel* *other*) → boolReturns 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*) → boolReturns 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*) → boolReturn 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.

BIBLIOGRAPHY

- [1932Wigner] E. Wigner. *Phys. Rev.* **40**, p. 749-759 (1932). doi:[10.1103/PhysRev.40.749](https://doi.org/10.1103/PhysRev.40.749)
- [1959Bell] R. P. Bell. *Trans. Faraday Soc.* **55**, p. 1-4 (1959). doi:[10.1039/TF9595500001](https://doi.org/10.1039/TF9595500001)
- [Chang2000] A. Y. Chang, J. W. Bozzelli, and A. M. Dean. *Z. Phys. Chem.* **214**, p. 1533-1568 (2000). doi:[10.1524/zpch.2000.214.11.1533](https://doi.org/10.1524/zpch.2000.214.11.1533)
- [Gilbert1990] R. G. Gilbert and S. C. Smith. *Theory of Unimolecular and Recombination Reactions*. Blackwell Sci. (1990).
- [Baer1996] T. Baer and W. L. Hase. *Unimolecular Reaction Dynamics*. Oxford University Press (1996).
- [Holbrook1996] K. A. Holbrook, M. J. Pilling, and S. H. Robertson. *Unimolecular Reactions*. Second Edition. John Wiley and Sons (1996).
- [Forst2003] W. Forst. *Unimolecular Reactions: A Concise Introduction*. Cambridge University Press (2003).
- [Pilling2003] M. J. Pilling and S. H. Robertson. *Annu. Rev. Phys. Chem.* **54**, p. 245-275 (2003). doi:[10.1146/annurev.physchem.54.011002.103822](https://doi.org/10.1146/annurev.physchem.54.011002.103822)

r

- `rmgpy.cantherm`, 3
- `rmgpy.chemkin`, 10
- `rmgpy.constants`, 13
- `rmgpy.data`, 13
- `rmgpy.kinetics`, 59
- `rmgpy.molecule`, 77
- `rmgpy.molecule.adjlist`, 95
- `rmgpy.pdep`, 99
- `rmgpy.qm`, 109
- `rmgpy.quantity`, 127
- `rmgpy.reaction`, 130
- `rmgpy.rmg`, 134
- `rmgpy.solver`, 147
- `rmgpy.species`, 149
- `rmgpy.statmech`, 152
- `rmgpy.statmech.schrodinger`, 163
- `rmgpy.thermo`, 166

A

- A (rmgpy.kinetics.Arrhenius attribute), 61
- a0 (rmgpy.thermo.Wilhoit attribute), 170
- a1 (rmgpy.thermo.Wilhoit attribute), 170
- a2 (rmgpy.thermo.Wilhoit attribute), 170
- a3 (rmgpy.thermo.Wilhoit attribute), 170
- addAction() (rmgpy.data.kinetics.ReactionRecipe method), 37
- addAtom() (rmgpy.molecule.Group method), 93
- addAtom() (rmgpy.molecule.Molecule method), 86
- addBond() (rmgpy.molecule.Group method), 93
- addBond() (rmgpy.molecule.Molecule method), 86
- addEdge() (rmgpy.molecule.graph.Graph method), 79
- addEdge() (rmgpy.molecule.Group method), 93
- addEdge() (rmgpy.molecule.Molecule method), 86
- addKineticsRulesFromTrainingSet() (rmgpy.data.kinetics.KineticsFamily method), 24
- addPathReaction() (rmgpy.rmg.pdep.PDepNetwork method), 141
- addReactionLibraryToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 135
- addReactionLibraryToOutput() (rmgpy.rmg.model.CoreEdgeReactionModel method), 135
- addReactionToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 135
- addReactionToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- addReactionToUnimolecularNetworks() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- addSeedMechanismToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- addSpeciesToCore() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- addSpeciesToEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- addVertex() (rmgpy.molecule.graph.Graph method), 79
- addVertex() (rmgpy.molecule.Group method), 93
- addVertex() (rmgpy.molecule.Molecule method), 86
- advance() (rmgpy.solver.ReactionSystem method), 148
- advance() (rmgpy.solver.SimpleReactor method), 148
- alpha (rmgpy.kinetics.Troe attribute), 74
- alpha0 (rmgpy.pdep.SingleExponentialDown attribute), 101
- ancestors() (rmgpy.data.base.Database method), 15
- ancestors() (rmgpy.data.kinetics.KineticsDepository method), 22
- ancestors() (rmgpy.data.kinetics.KineticsFamily method), 24
- ancestors() (rmgpy.data.kinetics.KineticsGroups method), 27
- ancestors() (rmgpy.data.kinetics.KineticsLibrary method), 29
- ancestors() (rmgpy.data.kinetics.KineticsRules method), 31
- ancestors() (rmgpy.data.statmech.StatmechDepository method), 39
- ancestors() (rmgpy.data.statmech.StatmechGroups method), 44
- ancestors() (rmgpy.data.statmech.StatmechLibrary method), 46
- ancestors() (rmgpy.data.thermo.ThermoDepository method), 53
- ancestors() (rmgpy.data.thermo.ThermoGroups method), 55
- ancestors() (rmgpy.data.thermo.ThermoLibrary method), 57
- applyAction() (rmgpy.molecule.Atom method), 84
- applyAction() (rmgpy.molecule.Bond method), 85
- applyAction() (rmgpy.molecule.GroupAtom method), 92
- applyAction() (rmgpy.molecule.GroupBond method), 92
- applyChemicallySignificantEigenvaluesMethod() (in module rmgpy.pdep.cse), 109
- applyChemicallySignificantEigenvaluesMethod() (rmgpy.pdep.Network method), 104
- applyChemicallySignificantEigenvaluesMethod() (rmgpy.rmg.pdep.PDepNetwork method), 141
- applyForward() (rmgpy.data.kinetics.ReactionRecipe method), 37

- [applyInverseLaplaceTransformMethod\(\)](#) (in module [rmgpy.pdep](#)), [102](#)
[applyModifiedStrongCollisionMethod\(\)](#) (in module [rmgpy.pdep.msc](#)), [108](#)
[applyModifiedStrongCollisionMethod\(\)](#) ([rmgpy.pdep.Network](#) method), [104](#)
[applyModifiedStrongCollisionMethod\(\)](#) ([rmgpy.rmg.pdep.PDepNetwork](#) method), [141](#)
[applyRecipe\(\)](#) ([rmgpy.data.kinetics.KineticsFamily](#) method), [24](#)
[applyReservoirStateMethod\(\)](#) (in module [rmgpy.pdep.rs](#)), [108](#)
[applyReservoirStateMethod\(\)](#) ([rmgpy.pdep.Network](#) method), [104](#)
[applyReservoirStateMethod\(\)](#) ([rmgpy.rmg.pdep.PDepNetwork](#) method), [141](#)
[applyReverse\(\)](#) ([rmgpy.data.kinetics.ReactionRecipe](#) method), [37](#)
[applyRRKMTheory\(\)](#) (in module [rmgpy.pdep](#)), [102](#)
[ArrayQuantity](#) (class in [rmgpy.quantity](#)), [129](#)
[Arrhenius](#) (class in [rmgpy.kinetics](#)), [61](#)
[arrhenius](#) ([rmgpy.kinetics.MultiArrhenius](#) attribute), [63](#)
[arrhenius](#) ([rmgpy.kinetics.MultiPDepArrhenius](#) attribute), [67](#)
[arrhenius](#) ([rmgpy.kinetics.PDepArrhenius](#) attribute), [65](#)
[arrheniusHigh](#) ([rmgpy.kinetics.Lindemann](#) attribute), [72](#)
[arrheniusHigh](#) ([rmgpy.kinetics.Troe](#) attribute), [74](#)
[arrheniusLow](#) ([rmgpy.kinetics.Lindemann](#) attribute), [72](#)
[arrheniusLow](#) ([rmgpy.kinetics.ThirdBody](#) attribute), [70](#)
[arrheniusLow](#) ([rmgpy.kinetics.Troe](#) attribute), [74](#)
[Atom](#) (class in [rmgpy.molecule](#)), [83](#)
[AtomType](#) (class in [rmgpy.molecule](#)), [81](#)
- ## B
- [B](#) ([rmgpy.thermo.Wilhoit](#) attribute), [169](#)
[barrier](#) ([rmgpy.statmech.HinderedRotor](#) attribute), [162](#)
[Bond](#) (class in [rmgpy.molecule](#)), [85](#)
- ## C
- [c0](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c1](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c2](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c3](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c4](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c5](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[c6](#) ([rmgpy.thermo.NASAPolynomial](#) attribute), [174](#)
[calculate\(\)](#) ([rmgpy.qm.symmetry.SymmetryJob](#) method), [114](#)
[calculateAtomSymmetryNumber\(\)](#) (in module [rmgpy.molecule.symmetry](#)), [98](#)
[calculateAxisSymmetryNumber\(\)](#) (in module [rmgpy.molecule.symmetry](#)), [98](#)
[calculateBondSymmetryNumber\(\)](#) (in module [rmgpy.molecule.symmetry](#)), [98](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.gaussian.GaussianMol](#) method), [115](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.gaussian.GaussianMolPM3](#) method), [117](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.gaussian.GaussianMolPM6](#) method), [118](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.molecule.QMMolecule](#) method), [112](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.mopac.MopacMol](#) method), [121](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.mopac.MopacMolPM3](#) method), [122](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.mopac.MopacMolPM6](#) method), [124](#)
[calculateChiralityCorrection\(\)](#) ([rmgpy.qm.mopac.MopacMolPM7](#) method), [125](#)
[calculateCollisionEfficiency\(\)](#) ([rmgpy.pdep.SingleExponentialDown](#) method), [101](#)
[calculateCollisionFrequency\(\)](#) ([rmgpy.pdep.Configuration](#) method), [103](#)
[calculateCollisionModel\(\)](#) ([rmgpy.pdep.Network](#) method), [104](#)
[calculateCollisionModel\(\)](#) ([rmgpy.rmg.pdep.PDepNetwork](#) method), [141](#)
[calculateCp0\(\)](#) ([rmgpy.molecule.Molecule](#) method), [86](#)
[calculateCp0\(\)](#) ([rmgpy.rmg.model.Species](#) method), [146](#)
[calculateCp0\(\)](#) ([rmgpy.species.Species](#) method), [150](#)
[calculateCpInf\(\)](#) ([rmgpy.molecule.Molecule](#) method), [86](#)
[calculateCpInf\(\)](#) ([rmgpy.rmg.model.Species](#) method), [146](#)
[calculateCpInf\(\)](#) ([rmgpy.species.Species](#) method), [150](#)
[calculateCyclicSymmetryNumber\(\)](#) (in module [rmgpy.molecule.symmetry](#)), [98](#)
[calculateDegeneracy\(\)](#) ([rmgpy.data.kinetics.KineticsFamily](#) method), [24](#)
[calculateDensitiesOfStates\(\)](#) ([rmgpy.pdep.Network](#) method), [104](#)
[calculateDensitiesOfStates\(\)](#) ([rmgpy.rmg.pdep.PDepNetwork](#) method), [141](#)
[calculateDensityOfStates\(\)](#) ([rmgpy.pdep.Configuration](#) method), [103](#)

- `calculateEquilibriumRatios()` (rmgpy.pdep.Network method), 104
`calculateEquilibriumRatios()` (rmgpy.rmg.pdep.PDepNetwork method), 141
`calculateMicrocanonicalRateCoefficient()` (in module rmgpy.pdep), 101
`calculateMicrocanonicalRateCoefficient()` (rmgpy.data.kinetics.DepositoryReaction method), 17
`calculateMicrocanonicalRateCoefficient()` (rmgpy.data.kinetics.LibraryReaction method), 33
`calculateMicrocanonicalRateCoefficient()` (rmgpy.data.kinetics.TemplateReaction method), 48
`calculateMicrocanonicalRateCoefficient()` (rmgpy.reaction.Reaction method), 131
`calculateMicrocanonicalRateCoefficient()` (rmgpy.rmg.pdep.PDepReaction method), 143
`calculateMicrocanonicalRates()` (rmgpy.pdep.Network method), 105
`calculateMicrocanonicalRates()` (rmgpy.rmg.pdep.PDepNetwork method), 141
`calculateSymmetryNumber()` (in module rmgpy.molecule.symmetry), 98
`calculateSymmetryNumber()` (rmgpy.molecule.Molecule method), 86
`calculateThermoData()` (rmgpy.qm.gaussian.GaussianMol method), 115
`calculateThermoData()` (rmgpy.qm.gaussian.GaussianMolPM3 method), 117
`calculateThermoData()` (rmgpy.qm.gaussian.GaussianMolPM6 method), 119
`calculateThermoData()` (rmgpy.qm.molecule.QMMolecule method), 112
`calculateThermoData()` (rmgpy.qm.mopac.MopacMol method), 121
`calculateThermoData()` (rmgpy.qm.mopac.MopacMolPM3 method), 122
`calculateThermoData()` (rmgpy.qm.mopac.MopacMolPM6 method), 124
`calculateThermoData()` (rmgpy.qm.mopac.MopacMolPM7 method), 125
`calculateTSTRateCoefficient()` (rmgpy.data.kinetics.DepositoryReaction method), 17
`calculateTSTRateCoefficient()` (rmgpy.data.kinetics.LibraryReaction method), 34
`calculateTSTRateCoefficient()` (rmgpy.data.kinetics.TemplateReaction method), 49
`calculateTSTRateCoefficient()` (rmgpy.reaction.Reaction method), 131
`calculateTSTRateCoefficient()` (rmgpy.rmg.pdep.PDepReaction method), 143
`calculateTunnelingFactor()` (rmgpy.kinetics.Eckart method), 76
`calculateTunnelingFactor()` (rmgpy.kinetics.Wigner method), 75
`calculateTunnelingFactor()` (rmgpy.species.TransitionState method), 151
`calculateTunnelingFunction()` (rmgpy.kinetics.Eckart method), 76
`calculateTunnelingFunction()` (rmgpy.kinetics.Wigner method), 75
`calculateTunnelingFunction()` (rmgpy.species.TransitionState method), 151
`CanTherm` (class in rmgpy.cantherm), 6
`canTST()` (rmgpy.data.kinetics.DepositoryReaction method), 17
`canTST()` (rmgpy.data.kinetics.LibraryReaction method), 34
`canTST()` (rmgpy.data.kinetics.TemplateReaction method), 49
`canTST()` (rmgpy.reaction.Reaction method), 132
`canTST()` (rmgpy.rmg.pdep.PDepReaction method), 143
`cclib_data` (rmgpy.qm.qmdata.CCLibData attribute), 113
`CCLibData` (class in rmgpy.qm.qmdata), 113
`changeBaseEnthalpy()` (rmgpy.thermo.NASA method), 172
`changeBaseEnthalpy()` (rmgpy.thermo.NASAPolynomial method), 175
`changeRate()` (rmgpy.kinetics.Arrhenius method), 62
`changeRate()` (rmgpy.kinetics.Chebyshev method), 68
`changeRate()` (rmgpy.kinetics.Lindemann method), 72
`changeRate()` (rmgpy.kinetics.MultiArrhenius method), 63
`changeRate()` (rmgpy.kinetics.MultiPDepArrhenius method), 67
`changeRate()` (rmgpy.kinetics.PDepArrhenius method), 66
`changeRate()` (rmgpy.kinetics.ThirdBody method), 70
`changeRate()` (rmgpy.kinetics.Troe method), 74
`changeT0()` (rmgpy.kinetics.Arrhenius method), 62
`Chebyshev` (class in rmgpy.kinetics), 68
`checkAllSet()` (rmgpy.qm.main.QMSettings method), 110
`checkForDuplicates()` (rmgpy.data.kinetics.KineticsLibrary method), 29
`checkForExistingReaction()` (rmgpy.rmg.model.CoreEdgeReactionModel method), 136

- [checkForExistingSpecies\(\)](#)
 (rmgpy.rmg.model.CoreEdgeReactionModel method), [136](#)
- [checkForInChiKeyCollision\(\)](#)
 (rmgpy.qm.qmverifier.QMVerifier method), [114](#)
- [checkInput\(\)](#) (rmgpy.rmg.main.RMG method), [139](#)
- [checkLibraries\(\)](#) (rmgpy.rmg.main.RMG method), [139](#)
- [checkPaths\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), [115](#)
- [checkPaths\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), [117](#)
- [checkPaths\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), [119](#)
- [checkPaths\(\)](#) (rmgpy.qm.main.QMCalculator method), [111](#)
- [checkPaths\(\)](#) (rmgpy.qm.molecule.QMMolecule method), [112](#)
- [checkPaths\(\)](#) (rmgpy.qm.mopac.MopacMol method), [121](#)
- [checkPaths\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), [122](#)
- [checkPaths\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), [124](#)
- [checkPaths\(\)](#) (rmgpy.qm.mopac.MopacMolPM7 method), [126](#)
- [checkReady\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), [115](#)
- [checkReady\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), [117](#)
- [checkReady\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), [119](#)
- [checkReady\(\)](#) (rmgpy.qm.main.QMCalculator method), [111](#)
- [checkReady\(\)](#) (rmgpy.qm.molecule.QMMolecule method), [112](#)
- [checkReady\(\)](#) (rmgpy.qm.mopac.MopacMol method), [121](#)
- [checkReady\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), [122](#)
- [checkReady\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), [124](#)
- [checkReady\(\)](#) (rmgpy.qm.mopac.MopacMolPM7 method), [126](#)
- [cleanup\(\)](#) (rmgpy.pdep.Configuration method), [103](#)
- [cleanup\(\)](#) (rmgpy.rmg.pdep.PDepNetwork method), [141](#)
- [clear\(\)](#) (rmgpy.rmg.main.RMG method), [139](#)
- [clearLabeledAtoms\(\)](#) (rmgpy.molecule.Group method), [93](#)
- [clearLabeledAtoms\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
- [cm1](#) (rmgpy.thermo.NASAPolynomial attribute), [175](#)
- [cm2](#) (rmgpy.thermo.NASAPolynomial attribute), [175](#)
- [coeffs](#) (rmgpy.kinetics.Chebyshev attribute), [69](#)
- [coeffs](#) (rmgpy.thermo.NASAPolynomial attribute), [175](#)
- [comment](#) (rmgpy.kinetics.Arrhenius attribute), [62](#)
- [comment](#) (rmgpy.kinetics.Chebyshev attribute), [69](#)
- [comment](#) (rmgpy.kinetics.KineticsData attribute), [60](#)
- [comment](#) (rmgpy.kinetics.Lindemann attribute), [72](#)
- [comment](#) (rmgpy.kinetics.MultiArrhenius attribute), [63](#)
- [comment](#) (rmgpy.kinetics.MultiPDepArrhenius attribute), [67](#)
- [comment](#) (rmgpy.kinetics.PDepArrhenius attribute), [66](#)
- [comment](#) (rmgpy.kinetics.PDepKineticsData attribute), [64](#)
- [comment](#) (rmgpy.kinetics.ThirdBody attribute), [70](#)
- [comment](#) (rmgpy.kinetics.Troe attribute), [74](#)
- [comment](#) (rmgpy.thermo.NASA attribute), [172](#)
- [comment](#) (rmgpy.thermo.NASAPolynomial attribute), [175](#)
- [comment](#) (rmgpy.thermo.ThermoData attribute), [167](#)
- [comment](#) (rmgpy.thermo.Wilhoit attribute), [170](#)
- [computeGroupAdditivityThermo\(\)](#)
 (rmgpy.data.thermo.ThermoDatabase method), [51](#)
- [computeRateDerivative\(\)](#) (rmgpy.solver.SimpleReactor method), [148](#)
- [Configuration](#) (class in rmgpy.pdep), [103](#)
- [Conformer](#) (class in rmgpy.statmech), [164](#)
- [connectTheDots\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
- [containsLabeledAtom\(\)](#) (rmgpy.molecule.Group method), [93](#)
- [containsLabeledAtom\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
- [convertDuplicatesToMulti\(\)](#)
 (rmgpy.data.kinetics.KineticsLibrary method), [29](#)
- [convertInitialKeysToSpeciesObjects\(\)](#)
 (rmgpy.solver.SimpleReactor method), [148](#)
- [convolve\(\)](#) (in module rmgpy.statmech.schrodinger), [163](#)
- [convolveBS\(\)](#) (in module rmgpy.statmech.schrodinger), [163](#)
- [convolveBSSR\(\)](#) (in module rmgpy.statmech.schrodinger), [163](#)
- [coordinates](#) (rmgpy.statmech.Conformer attribute), [165](#)
- [copy\(\)](#) (rmgpy.cantherm.PressureDependenceJob method), [8](#)
- [copy\(\)](#) (rmgpy.data.kinetics.DepositoryReaction method), [17](#)
- [copy\(\)](#) (rmgpy.data.kinetics.LibraryReaction method), [34](#)
- [copy\(\)](#) (rmgpy.data.kinetics.TemplateReaction method), [49](#)
- [copy\(\)](#) (rmgpy.molecule.Atom method), [84](#)
- [copy\(\)](#) (rmgpy.molecule.Bond method), [85](#)
- [copy\(\)](#) (rmgpy.molecule.graph.Edge method), [79](#)
- [copy\(\)](#) (rmgpy.molecule.graph.Graph method), [79](#)
- [copy\(\)](#) (rmgpy.molecule.graph.Vertex method), [78](#)
- [copy\(\)](#) (rmgpy.molecule.Group method), [93](#)

[copy\(\)](#) (rmgpy.molecule.GroupAtom method), [92](#)
[copy\(\)](#) (rmgpy.molecule.GroupBond method), [92](#)
[copy\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
[copy\(\)](#) (rmgpy.quantity.ArrayQuantity method), [130](#)
[copy\(\)](#) (rmgpy.quantity.ScalarQuantity method), [129](#)
[copy\(\)](#) (rmgpy.reaction.Reaction method), [132](#)
[copy\(\)](#) (rmgpy.rmg.pdep.PDepReaction method), [143](#)
[copy\(\)](#) (rmgpy.thermo.Wilhoit method), [170](#)
[CoreEdgeReactionModel](#) (class in rmgpy.rmg.model), [135](#)
[countInternalRotors\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
[Cp0](#) (rmgpy.thermo.ThermoData attribute), [167](#)
[Cp0](#) (rmgpy.thermo.Wilhoit attribute), [169](#)
[Cpdata](#) (rmgpy.thermo.ThermoData attribute), [167](#)
[CpInf](#) (rmgpy.thermo.ThermoData attribute), [167](#)
[CpInf](#) (rmgpy.thermo.Wilhoit attribute), [169](#)
[createGeometry\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), [115](#)
[createGeometry\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), [117](#)
[createGeometry\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), [119](#)
[createGeometry\(\)](#) (rmgpy.qm.molecule.QMMolecule method), [112](#)
[createGeometry\(\)](#) (rmgpy.qm.mopac.MopacMol method), [121](#)
[createGeometry\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), [122](#)
[createGeometry\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), [124](#)
[createGeometry\(\)](#) (rmgpy.qm.mopac.MopacMolPM7 method), [126](#)
[createMultiplicityLayer\(\)](#) (rmgpy.molecule.Molecule method), [86](#)

D

[Database](#) (class in rmgpy.data.base), [15](#)
[decrementLonePairs\(\)](#) (rmgpy.molecule.Atom method), [84](#)
[decrementOrder\(\)](#) (rmgpy.molecule.Bond method), [85](#)
[decrementRadical\(\)](#) (rmgpy.molecule.Atom method), [84](#)
[degreeP](#) (rmgpy.kinetics.Chebyshev attribute), [69](#)
[degreeT](#) (rmgpy.kinetics.Chebyshev attribute), [69](#)
[deleteHydrogens\(\)](#) (rmgpy.molecule.Molecule method), [86](#)
[DepositoryReaction](#) (class in rmgpy.data.kinetics), [17](#)
[descendants\(\)](#) (rmgpy.data.base.Database method), [15](#)
[descendants\(\)](#) (rmgpy.data.kinetics.KineticsDepository method), [22](#)
[descendants\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), [24](#)
[descendants\(\)](#) (rmgpy.data.kinetics.KineticsGroups method), [28](#)

[descendants\(\)](#) (rmgpy.data.kinetics.KineticsLibrary method), [30](#)
[descendants\(\)](#) (rmgpy.data.kinetics.KineticsRules method), [31](#)
[descendants\(\)](#) (rmgpy.data.statmech.StatmechDepository method), [39](#)
[descendants\(\)](#) (rmgpy.data.statmech.StatmechGroups method), [45](#)
[descendants\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), [47](#)
[descendants\(\)](#) (rmgpy.data.thermo.ThermoDepository method), [53](#)
[descendants\(\)](#) (rmgpy.data.thermo.ThermoGroups method), [55](#)
[descendants\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), [57](#)
[descendTree\(\)](#) (rmgpy.data.base.Database method), [15](#)
[descendTree\(\)](#) (rmgpy.data.kinetics.KineticsDepository method), [22](#)
[descendTree\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), [24](#)
[descendTree\(\)](#) (rmgpy.data.kinetics.KineticsGroups method), [27](#)
[descendTree\(\)](#) (rmgpy.data.kinetics.KineticsLibrary method), [30](#)
[descendTree\(\)](#) (rmgpy.data.kinetics.KineticsRules method), [31](#)
[descendTree\(\)](#) (rmgpy.data.statmech.StatmechDepository method), [39](#)
[descendTree\(\)](#) (rmgpy.data.statmech.StatmechGroups method), [44](#)
[descendTree\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), [46](#)
[descendTree\(\)](#) (rmgpy.data.thermo.ThermoDepository method), [53](#)
[descendTree\(\)](#) (rmgpy.data.thermo.ThermoGroups method), [55](#)
[descendTree\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), [57](#)
[determinePointGroup\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), [116](#)
[determinePointGroup\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), [117](#)
[determinePointGroup\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), [119](#)
[determinePointGroup\(\)](#) (rmgpy.qm.molecule.QMMolecule method), [112](#)
[determinePointGroup\(\)](#) (rmgpy.qm.mopac.MopacMol method), [121](#)
[determinePointGroup\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), [123](#)
[determinePointGroup\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), [124](#)

- determinePointGroup() (rmgpy.qm.mopac.MopacMolPM7 method), 126
- DirectFit (class in rmgpy.data.statmechfit), 42
- discrepancy() (rmgpy.kinetics.Arrhenius method), 62
- discrepancy() (rmgpy.kinetics.Chebyshev method), 69
- discrepancy() (rmgpy.kinetics.KineticsData method), 60
- discrepancy() (rmgpy.kinetics.Lindemann method), 72
- discrepancy() (rmgpy.kinetics.MultiArrhenius method), 63
- discrepancy() (rmgpy.kinetics.MultiPDepArrhenius method), 67
- discrepancy() (rmgpy.kinetics.PDepArrhenius method), 66
- discrepancy() (rmgpy.kinetics.PDepKineticsData method), 64
- discrepancy() (rmgpy.kinetics.ThirdBody method), 70
- discrepancy() (rmgpy.kinetics.Troe method), 74
- discrepancy() (rmgpy.thermo.NASA method), 172
- discrepancy() (rmgpy.thermo.NASAPolynomial method), 175
- discrepancy() (rmgpy.thermo.ThermoData method), 167
- discrepancy() (rmgpy.thermo.Wilhoit method), 170
- draw() (rmgpy.cantherm.PressureDependenceJob method), 8
- draw() (rmgpy.data.kinetics.DepositoryReaction method), 17
- draw() (rmgpy.data.kinetics.LibraryReaction method), 34
- draw() (rmgpy.data.kinetics.TemplateReaction method), 49
- draw() (rmgpy.molecule.draw.MoleculeDrawer method), 98
- draw() (rmgpy.molecule.draw.ReactionDrawer method), 99
- draw() (rmgpy.molecule.Molecule method), 86
- draw() (rmgpy.reaction.Reaction method), 132
- draw() (rmgpy.rmg.pdep.PDepReaction method), 143
- ## E
- E0 (rmgpy.pdep.Configuration attribute), 103
- E0 (rmgpy.statmech.Conformer attribute), 165
- E0 (rmgpy.thermo.NASA attribute), 172
- E0 (rmgpy.thermo.NASAPolynomial attribute), 174
- E0 (rmgpy.thermo.ThermoData attribute), 167
- E0 (rmgpy.thermo.Wilhoit attribute), 169
- E0_prod (rmgpy.kinetics.Eckart attribute), 76
- E0_reac (rmgpy.kinetics.Eckart attribute), 76
- E0_TS (rmgpy.kinetics.Eckart attribute), 76
- Ea (rmgpy.kinetics.Arrhenius attribute), 61
- Eckart (class in rmgpy.kinetics), 75
- Edge (class in rmgpy.molecule.graph), 79
- efficiencies (rmgpy.kinetics.Chebyshev attribute), 69
- efficiencies (rmgpy.kinetics.Lindemann attribute), 72
- efficiencies (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
- efficiencies (rmgpy.kinetics.PDepArrhenius attribute), 66
- efficiencies (rmgpy.kinetics.PDepKineticsData attribute), 64
- efficiencies (rmgpy.kinetics.ThirdBody attribute), 70
- efficiencies (rmgpy.kinetics.Troe attribute), 74
- Element (class in rmgpy.molecule), 81
- energies (rmgpy.statmech.HinderedRotor attribute), 162
- enlarge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- Entry (class in rmgpy.data.base), 20
- equals() (rmgpy.quantity.ArrayQuantity method), 130
- equals() (rmgpy.quantity.ScalarQuantity method), 129
- equivalent() (rmgpy.molecule.Atom method), 84
- equivalent() (rmgpy.molecule.AtomType method), 82
- equivalent() (rmgpy.molecule.Bond method), 85
- equivalent() (rmgpy.molecule.graph.Edge method), 79
- equivalent() (rmgpy.molecule.graph.Vertex method), 78
- equivalent() (rmgpy.molecule.GroupAtom method), 92
- equivalent() (rmgpy.molecule.GroupBond method), 92
- estimateKinetics() (rmgpy.data.kinetics.KineticsRules method), 32
- estimateKineticsUsingGroupAdditivity() (rmgpy.data.kinetics.KineticsFamily method), 24
- estimateKineticsUsingGroupAdditivity() (rmgpy.data.kinetics.KineticsGroups method), 28
- estimateKineticsUsingRateRules() (rmgpy.data.kinetics.KineticsFamily method), 24
- estimateRadicalThermoViaHBI() (rmgpy.data.thermo.ThermoDatabase method), 51
- estimateThermoViaGroupAdditivity() (rmgpy.data.thermo.ThermoDatabase method), 51
- execute() (rmgpy.cantherm.CanTherm method), 6
- execute() (rmgpy.cantherm.KineticsJob method), 6
- execute() (rmgpy.cantherm.StatMechJob method), 9
- execute() (rmgpy.cantherm.ThermoJob method), 9
- execute() (rmgpy.rmg.main.RMG method), 139
- exploreIsomer() (rmgpy.rmg.pdep.PDepNetwork method), 141
- ## F
- failsSpeciesConstraints() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- failureKeys (rmgpy.qm.gaussian.Gaussian attribute), 115
- failureKeys (rmgpy.qm.mopac.Mopac attribute), 120
- fillKineticsRulesByAveragingUp() (rmgpy.data.kinetics.KineticsFamily method), 24

- fillRulesByAveragingUp() (rmgpy.data.kinetics.KineticsRules method), 32
 findAllDelocalizationPaths() (rmgpy.molecule.Molecule method), 87
 findAllDelocalizationPathsLonePairRadical() (rmgpy.molecule.Molecule method), 87
 findAllDelocalizationPathsN5dd_N5ts() (rmgpy.molecule.Molecule method), 87
 findCp0andCpInf() (rmgpy.data.thermo.ThermoDatabase method), 52
 findIsomorphism() (rmgpy.molecule.graph.Graph method), 79
 findIsomorphism() (rmgpy.molecule.Group method), 93
 findIsomorphism() (rmgpy.molecule.Molecule method), 87
 findIsomorphism() (rmgpy.molecule.vf2.VF2 method), 81
 findSubgraphIsomorphisms() (rmgpy.molecule.graph.Graph method), 79
 findSubgraphIsomorphisms() (rmgpy.molecule.Group method), 93
 findSubgraphIsomorphisms() (rmgpy.molecule.Molecule method), 87
 findSubgraphIsomorphisms() (rmgpy.molecule.vf2.VF2 method), 81
 finish() (rmgpy.rmg.main.RMG method), 139
 fitCosinePotentialToData() (rmgpy.statmech.HinderedRotor method), 162
 fitFourierPotentialToData() (rmgpy.statmech.HinderedRotor method), 162
 fitStatmechDirect() (in module rmgpy.data.statmechfit), 41
 fitStatmechPseudo() (in module rmgpy.data.statmechfit), 41
 fitStatmechPseudoRotors() (in module rmgpy.data.statmechfit), 41
 fitStatmechToHeatCapacity() (in module rmgpy.data.statmechfit), 41
 fitToData() (rmgpy.kinetics.Arrhenius method), 62
 fitToData() (rmgpy.kinetics.Chebyshev method), 69
 fitToData() (rmgpy.kinetics.PDepArrhenius method), 66
 fitToData() (rmgpy.thermo.Wilhoit method), 170
 fitToDataForConstantB() (rmgpy.thermo.Wilhoit method), 170
 fixBarrierHeight() (rmgpy.data.kinetics.DepositoryReaction method), 18
 fixBarrierHeight() (rmgpy.data.kinetics.LibraryReaction method), 34
 fixBarrierHeight() (rmgpy.data.kinetics.TemplateReaction method), 49
 fixBarrierHeight() (rmgpy.reaction.Reaction method), 132
 fixBarrierHeight() (rmgpy.rmg.pdep.PDepReaction method), 144
 fixDiffusionLimitedA() (rmgpy.data.kinetics.DepositoryReaction method), 18
 fixDiffusionLimitedA() (rmgpy.data.kinetics.LibraryReaction method), 34
 fixDiffusionLimitedA() (rmgpy.data.kinetics.TemplateReaction method), 49
 fixDiffusionLimitedA() (rmgpy.reaction.Reaction method), 132
 fixDiffusionLimitedA() (rmgpy.rmg.pdep.PDepReaction method), 144
 fourier (rmgpy.statmech.HinderedRotor attribute), 162
 frequencies (rmgpy.statmech.HarmonicOscillator attribute), 160
 frequency (rmgpy.kinetics.Eckart attribute), 76
 frequency (rmgpy.kinetics.Wigner attribute), 75
 frequency (rmgpy.statmech.HinderedRotor attribute), 162
 fromAdjacencyList() (in module rmgpy.molecule.adjlist), 97
 fromAdjacencyList() (rmgpy.molecule.Group method), 93
 fromAdjacencyList() (rmgpy.molecule.Molecule method), 87
 fromAdjacencyList() (rmgpy.rmg.model.Species method), 146
 fromAdjacencyList() (rmgpy.species.Species method), 150
 fromInChI() (rmgpy.molecule.Molecule method), 87
 fromRDKitMol() (rmgpy.molecule.Molecule method), 87
 fromSMARTS() (rmgpy.molecule.Molecule method), 87
 fromSMILES() (rmgpy.molecule.Molecule method), 87
 fromSMILES() (rmgpy.rmg.model.Species method), 146
 fromSMILES() (rmgpy.species.Species method), 150
 fromXYZ() (rmgpy.molecule.Molecule method), 87
- ## G
- Gaussian (class in rmgpy.qm.gaussian), 115
 GaussianLog (class in rmgpy.cantherm.gaussian), 4
 GaussianMol (class in rmgpy.qm.gaussian), 115
 GaussianMolPM3 (class in rmgpy.qm.gaussian), 117
 GaussianMolPM6 (class in rmgpy.qm.gaussian), 118
 generate3dTS() (rmgpy.data.kinetics.DepositoryReaction method), 18
 generate3dTS() (rmgpy.data.kinetics.LibraryReaction method), 34
 generate3dTS() (rmgpy.data.kinetics.TemplateReaction method), 49
 generate3dTS() (rmgpy.reaction.Reaction method), 132
 generate3dTS() (rmgpy.rmg.pdep.PDepReaction method), 144

`generateCollisionMatrix()` (`rmgpy.pdep.Configuration` method), 103

`generateCollisionMatrix()` (`rmgpy.pdep.SingleExponentialDown` method), 101

`generateEnergyTransferModel()` (`rmgpy.rmg.model.Species` method), 146

`generateExecutionPlots()` (`rmgpy.rmg.main.RMG` method), 139

`generateFrequencies()` (`rmgpy.data.statmech.GroupFrequency` method), 20

`generateFullMEMatrix()` (in module `rmgpy.pdep.me`), 106

`generateGroupAdditivityValues()` (`rmgpy.data.kinetics.KineticsGroups` method), 28

`generateKinetics()` (`rmgpy.cantherm.KineticsJob` method), 6

`generateKinetics()` (`rmgpy.rmg.model.CoreEdgeReactionModel` method), 136

`generateOldLibraryEntry()` (`rmgpy.data.statmech.StatmechGroups` method), 45

`generateOldLibraryEntry()` (`rmgpy.data.statmech.StatmechLibrary` method), 47

`generateOldLibraryEntry()` (`rmgpy.data.thermo.ThermoGroups` method), 55

`generateOldLibraryEntry()` (`rmgpy.data.thermo.ThermoLibrary` method), 58

`generateOldTree()` (`rmgpy.data.base.Database` method), 15

`generateOldTree()` (`rmgpy.data.kinetics.KineticsDepository` method), 22

`generateOldTree()` (`rmgpy.data.kinetics.KineticsFamily` method), 24

`generateOldTree()` (`rmgpy.data.kinetics.KineticsGroups` method), 28

`generateOldTree()` (`rmgpy.data.kinetics.KineticsLibrary` method), 30

`generateOldTree()` (`rmgpy.data.kinetics.KineticsRules` method), 32

`generateOldTree()` (`rmgpy.data.statmech.StatmechDepository` method), 39

`generateOldTree()` (`rmgpy.data.statmech.StatmechGroups` method), 45

`generateOldTree()` (`rmgpy.data.statmech.StatmechLibrary` method), 47

`generateOldTree()` (`rmgpy.data.thermo.ThermoDepository` method), 54

`generateOldTree()` (`rmgpy.data.thermo.ThermoGroups` method), 56

`generateOldTree()` (`rmgpy.data.thermo.ThermoLibrary` method), 58

`generatePairs()` (`rmgpy.data.kinetics.DepositoryReaction` method), 18

`generatePairs()` (`rmgpy.data.kinetics.LibraryReaction` method), 34

`generatePairs()` (`rmgpy.data.kinetics.TemplateReaction` method), 49

`generatePairs()` (`rmgpy.reaction.Reaction` method), 132

`generatePairs()` (`rmgpy.rmg.pdep.PDepReaction` method), 144

`generatePressureList()` (`rmgpy.cantherm.PressureDependenceJob` method), 9

`generateProductTemplate()` (`rmgpy.data.kinetics.KineticsFamily` method), 25

`generateQMDData()` (`rmgpy.qm.gaussian.GaussianMol` method), 116

`generateQMDData()` (`rmgpy.qm.gaussian.GaussianMolPM3` method), 117

`generateQMDData()` (`rmgpy.qm.gaussian.GaussianMolPM6` method), 119

`generateQMDData()` (`rmgpy.qm.molecule.QMMolecule` method), 112

`generateQMDData()` (`rmgpy.qm.mopac.MopacMol` method), 121

`generateQMDData()` (`rmgpy.qm.mopac.MopacMolPM3` method), 123

`generateQMDData()` (`rmgpy.qm.mopac.MopacMolPM6` method), 124

`generateQMDData()` (`rmgpy.qm.mopac.MopacMolPM7` method), 126

`generateRDKitGeometries()` (`rmgpy.qm.molecule.Geometry` method), 111

`generateReactions()` (`rmgpy.data.kinetics.KineticsDatabase` method), 20

`generateReactions()` (`rmgpy.data.kinetics.KineticsFamily` method), 25

`generateReactionsFromFamilies()` (`rmgpy.data.kinetics.KineticsDatabase` method), 20

`generateReactionsFromLibraries()` (`rmgpy.data.kinetics.KineticsDatabase` method), 21

`generateReactionsFromLibrary()` (`rmgpy.data.kinetics.KineticsDatabase` method), 21

`generateResonanceIsomers()` (`rmgpy.molecule.Molecule` method), 87

`generateResonanceIsomers()` (`rmgpy.rmg.model.Species` method), 146

`generateResonanceIsomers()` (`rmgpy.species.Species` method), 150

- generateReverseRateCoefficient()
(rmgpy.data.kinetics.DepositoryReaction method), 18
- generateReverseRateCoefficient()
(rmgpy.data.kinetics.LibraryReaction method), 35
- generateReverseRateCoefficient()
(rmgpy.data.kinetics.TemplateReaction method), 50
- generateReverseRateCoefficient()
(rmgpy.reaction.Reaction method), 132
- generateReverseRateCoefficient()
(rmgpy.rmg.pdep.PDepReaction method), 144
- generateStatMech() (rmgpy.rmg.model.Species method), 146
- generateStoichiometryMatrix()
(rmgpy.reaction.ReactionModel method), 134
- generateTemperatureList()
(rmgpy.cantherm.PressureDependenceJob method), 9
- generateThermo() (rmgpy.cantherm.ThermoJob method), 9
- generateThermoData() (rmgpy.qm.gaussian.GaussianMol method), 116
- generateThermoData() (rmgpy.qm.gaussian.GaussianMolPM3 method), 117
- generateThermoData() (rmgpy.qm.gaussian.GaussianMolPM6 method), 119
- generateThermoData() (rmgpy.qm.molecule.QMMolecule method), 112
- generateThermoData() (rmgpy.qm.mopac.MopacMol method), 121
- generateThermoData() (rmgpy.qm.mopac.MopacMolPM3 method), 123
- generateThermoData() (rmgpy.qm.mopac.MopacMolPM6 method), 124
- generateThermoData() (rmgpy.qm.mopac.MopacMolPM7 method), 126
- generateThermoData() (rmgpy.rmg.model.Species method), 146
- generateTransportData() (rmgpy.rmg.model.Species method), 146
- generic_visit() (rmgpy.cantherm.output.PrettifyVisitor method), 7
- Geometry (class in rmgpy.cantherm.geometry), 5
- Geometry (class in rmgpy.qm.molecule), 111
- getActiveModes() (rmgpy.statmech.Conformer method), 165
- getAdjacentResonanceIsomers()
(rmgpy.molecule.Molecule method), 87
- getAllCycles() (rmgpy.molecule.graph.Graph method), 79
- getAllCycles() (rmgpy.molecule.Group method), 93
- getAllCycles() (rmgpy.molecule.Molecule method), 87
- getAllCyclicVertices() (rmgpy.molecule.graph.Graph method), 79
- getAllCyclicVertices() (rmgpy.molecule.Group method), 93
- getAllCyclicVertices() (rmgpy.molecule.Molecule method), 88
- getAllPolycyclicVertices() (rmgpy.molecule.graph.Graph method), 79
- getAllPolycyclicVertices() (rmgpy.molecule.Group method), 94
- getAllPolycyclicVertices() (rmgpy.molecule.Molecule method), 88
- getAllRules() (rmgpy.data.kinetics.KineticsRules method), 32
- getAllSpecies() (rmgpy.pdep.Network method), 105
- getAllSpecies() (rmgpy.rmg.pdep.PDepNetwork method), 141
- getAllThermoData() (rmgpy.data.thermo.ThermoDatabase method), 52
- getAlpha() (rmgpy.pdep.SingleExponentialDown method), 101
- getAromaticResonanceIsomers()
(rmgpy.molecule.Molecule method), 88
- getAtomType() (in module rmgpy.molecule), 82
- getBond() (rmgpy.molecule.Group method), 94
- getBond() (rmgpy.molecule.Molecule method), 88
- getBonds() (rmgpy.molecule.Group method), 94
- getBonds() (rmgpy.molecule.Molecule method), 88
- getCenterOfMass() (rmgpy.cantherm.geometry.Geometry method), 5
- getCenterOfMass() (rmgpy.statmech.Conformer method), 165
- getConversionFactorFromSI()
(rmgpy.quantity.ArrayQuantity method), 130
- getConversionFactorFromSI()
(rmgpy.quantity.ScalarQuantity method), 129
- getConversionFactorToSI()
(rmgpy.quantity.ArrayQuantity method), 130
- getConversionFactorToSI()
(rmgpy.quantity.ScalarQuantity method), 129
- getCrudeMolFilePath() (rmgpy.qm.molecule.Geometry method), 111
- getDensityOfStates() (in module rmgpy.statmech.schrodinger), 163
- getDensityOfStates() (rmgpy.rmg.model.Species method), 146
- getDensityOfStates() (rmgpy.species.Species method), 151

- [getDensityOfStates\(\) \(rmgpy.species.TransitionState method\), 152](#)
[getDensityOfStates\(\) \(rmgpy.statmech.Conformer method\), 165](#)
[getDensityOfStates\(\) \(rmgpy.statmech.HarmonicOscillator method\), 160](#)
[getDensityOfStates\(\) \(rmgpy.statmech.HinderedRotor method\), 162](#)
[getDensityOfStates\(\) \(rmgpy.statmech.IdealGasTranslation method\), 154](#)
[getDensityOfStates\(\) \(rmgpy.statmech.KRotor method\), 158](#)
[getDensityOfStates\(\) \(rmgpy.statmech.LinearRotor method\), 155](#)
[getDensityOfStates\(\) \(rmgpy.statmech.NonlinearRotor method\), 156](#)
[getDensityOfStates\(\) \(rmgpy.statmech.SphericalTopRotor method\), 159](#)
[getEdge\(\) \(rmgpy.molecule.graph.Graph method\), 80](#)
[getEdge\(\) \(rmgpy.molecule.Group method\), 94](#)
[getEdge\(\) \(rmgpy.molecule.Molecule method\), 88](#)
[getEdges\(\) \(rmgpy.molecule.graph.Graph method\), 80](#)
[getEdges\(\) \(rmgpy.molecule.Group method\), 94](#)
[getEdges\(\) \(rmgpy.molecule.Molecule method\), 88](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.Chebyshev method\), 69](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.Lindemann method\), 72](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.MultiPDepArrhenius method\), 67](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.PDepsArrhenius method\), 66](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.PDepsKineticsData method\), 64](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.ThirdBody method\), 70](#)
[getEffectivePressure\(\) \(rmgpy.kinetics.Troe method\), 74](#)
[getElement\(\) \(in module rmgpy.molecule\), 81](#)
[getEnthalpiesOfReaction\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEnthalpiesOfReaction\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEnthalpiesOfReaction\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEnthalpiesOfReaction\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEnthalpiesOfReaction\(\) \(rmgpy.rmg.pdep.PDepsReaction method\), 144](#)
[getEnthalpy\(\) \(in module rmgpy.statmech.schrodinger\), 164](#)
[getEnthalpy\(\) \(rmgpy.pdep.Configuration method\), 103](#)
[getEnthalpy\(\) \(rmgpy.rmg.model.Species method\), 146](#)
[getEnthalpy\(\) \(rmgpy.species.Species method\), 151](#)
[getEnthalpy\(\) \(rmgpy.species.TransitionState method\), 152](#)
[getEnthalpy\(\) \(rmgpy.statmech.Conformer method\), 165](#)
[getEnthalpy\(\) \(rmgpy.statmech.HarmonicOscillator method\), 160](#)
[getEnthalpy\(\) \(rmgpy.statmech.HinderedRotor method\), 162](#)
[getEnthalpy\(\) \(rmgpy.statmech.IdealGasTranslation method\), 154](#)
[getEnthalpy\(\) \(rmgpy.statmech.KRotor method\), 158](#)
[getEnthalpy\(\) \(rmgpy.statmech.LinearRotor method\), 155](#)
[getEnthalpy\(\) \(rmgpy.statmech.NonlinearRotor method\), 156](#)
[getEnthalpy\(\) \(rmgpy.statmech.SphericalTopRotor method\), 159](#)
[getEnthalpy\(\) \(rmgpy.thermo.NASA method\), 172](#)
[getEnthalpy\(\) \(rmgpy.thermo.NASAPolynomial method\), 175](#)
[getEnthalpy\(\) \(rmgpy.thermo.ThermoData method\), 167](#)
[getEnthalpy\(\) \(rmgpy.thermo.Wilhoit method\), 170](#)
[getEnthalpyOfReaction\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEnthalpyOfReaction\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEnthalpyOfReaction\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEnthalpyOfReaction\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEnthalpyOfReaction\(\) \(rmgpy.rmg.pdep.PDepsReaction method\), 144](#)
[getEntries\(\) \(rmgpy.data.kinetics.KineticsRules method\), 32](#)
[getEntriesToSave\(\) \(rmgpy.data.base.Database method\), 15](#)
[getEntriesToSave\(\) \(rmgpy.data.kinetics.KineticsDepository method\), 22](#)
[getEntriesToSave\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 25](#)
[getEntriesToSave\(\) \(rmgpy.data.kinetics.KineticsGroups method\), 28](#)
[getEntriesToSave\(\) \(rmgpy.data.kinetics.KineticsLibrary method\), 30](#)
[getEntriesToSave\(\) \(rmgpy.data.kinetics.KineticsRules method\), 32](#)
[getEntriesToSave\(\) \(rmgpy.data.statmech.StatmechDepository method\), 39](#)

[getEntriesToSave\(\) \(rmgpy.data.statmech.StatmechGroups method\), 45](#)
[getEntriesToSave\(\) \(rmgpy.data.statmech.StatmechLibrary method\), 47](#)
[getEntriesToSave\(\) \(rmgpy.data.thermo.ThermoDepository method\), 54](#)
[getEntriesToSave\(\) \(rmgpy.data.thermo.ThermoGroups method\), 56](#)
[getEntriesToSave\(\) \(rmgpy.data.thermo.ThermoLibrary method\), 58](#)
[getEntropiesOfReaction\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEntropiesOfReaction\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEntropiesOfReaction\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEntropiesOfReaction\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEntropiesOfReaction\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 144](#)
[getEntropy\(\) \(in module rmgpy.statmech.schrodinger\), 164](#)
[getEntropy\(\) \(rmgpy.pdep.Configuration method\), 103](#)
[getEntropy\(\) \(rmgpy.rmg.model.Species method\), 146](#)
[getEntropy\(\) \(rmgpy.species.Species method\), 151](#)
[getEntropy\(\) \(rmgpy.species.TransitionState method\), 152](#)
[getEntropy\(\) \(rmgpy.statmech.Conformer method\), 165](#)
[getEntropy\(\) \(rmgpy.statmech.HarmonicOscillator method\), 161](#)
[getEntropy\(\) \(rmgpy.statmech.HinderedRotor method\), 162](#)
[getEntropy\(\) \(rmgpy.statmech.IdealGasTranslation method\), 154](#)
[getEntropy\(\) \(rmgpy.statmech.KRotor method\), 158](#)
[getEntropy\(\) \(rmgpy.statmech.LinearRotor method\), 155](#)
[getEntropy\(\) \(rmgpy.statmech.NonlinearRotor method\), 157](#)
[getEntropy\(\) \(rmgpy.statmech.SphericalTopRotor method\), 159](#)
[getEntropy\(\) \(rmgpy.thermo.NASA method\), 172](#)
[getEntropy\(\) \(rmgpy.thermo.NASAPolynomial method\), 175](#)
[getEntropy\(\) \(rmgpy.thermo.ThermoData method\), 167](#)
[getEntropy\(\) \(rmgpy.thermo.Wilhoit method\), 170](#)
[getEntropyOfReaction\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEntropyOfReaction\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEntropyOfReaction\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEntropyOfReaction\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEntropyOfReaction\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 144](#)
[getEquilibriumConstant\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEquilibriumConstant\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEquilibriumConstant\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEquilibriumConstant\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEquilibriumConstant\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 144](#)
[getEquilibriumConstants\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 18](#)
[getEquilibriumConstants\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 35](#)
[getEquilibriumConstants\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 50](#)
[getEquilibriumConstants\(\) \(rmgpy.reaction.Reaction method\), 132](#)
[getEquilibriumConstants\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 144](#)
[getFilePath\(\) \(rmgpy.qm.gaussian.GaussianMol method\), 116](#)
[getFilePath\(\) \(rmgpy.qm.gaussian.GaussianMolPM3 method\), 117](#)
[getFilePath\(\) \(rmgpy.qm.gaussian.GaussianMolPM6 method\), 119](#)
[getFilePath\(\) \(rmgpy.qm.molecule.Geometry method\), 111](#)
[getFilePath\(\) \(rmgpy.qm.molecule.QMMolecule method\), 112](#)
[getFilePath\(\) \(rmgpy.qm.mopac.MopacMol method\), 121](#)
[getFilePath\(\) \(rmgpy.qm.mopac.MopacMolPM3 method\), 123](#)
[getFilePath\(\) \(rmgpy.qm.mopac.MopacMolPM6 method\), 124](#)
[getFilePath\(\) \(rmgpy.qm.mopac.MopacMolPM7 method\), 126](#)
[getFingerprint\(\) \(rmgpy.molecule.Molecule method\), 88](#)
[getFormula\(\) \(rmgpy.molecule.Molecule method\), 88](#)

`getForwardReactionForFamilyEntry()`
(`rmgpy.data.kinetics.KineticsDatabase` method), 21

`getFreeEnergiesOfReaction()`
(`rmgpy.data.kinetics.DepositoryReaction` method), 18

`getFreeEnergiesOfReaction()`
(`rmgpy.data.kinetics.LibraryReaction` method), 35

`getFreeEnergiesOfReaction()`
(`rmgpy.data.kinetics.TemplateReaction` method), 50

`getFreeEnergiesOfReaction()` (`rmgpy.reaction.Reaction` method), 133

`getFreeEnergiesOfReaction()`
(`rmgpy.rmg.pdep.PDepReaction` method), 144

`getFreeEnergy()` (`rmgpy.pdep.Configuration` method), 103

`getFreeEnergy()` (`rmgpy.rmg.model.Species` method), 147

`getFreeEnergy()` (`rmgpy.species.Species` method), 151

`getFreeEnergy()` (`rmgpy.species.TransitionState` method), 152

`getFreeEnergy()` (`rmgpy.statmech.Conformer` method), 165

`getFreeEnergy()` (`rmgpy.thermo.NASA` method), 172

`getFreeEnergy()` (`rmgpy.thermo.NASAPolynomial` method), 175

`getFreeEnergy()` (`rmgpy.thermo.ThermoData` method), 167

`getFreeEnergy()` (`rmgpy.thermo.Wilhoit` method), 170

`getFreeEnergyOfReaction()`
(`rmgpy.data.kinetics.DepositoryReaction` method), 19

`getFreeEnergyOfReaction()`
(`rmgpy.data.kinetics.LibraryReaction` method), 35

`getFreeEnergyOfReaction()`
(`rmgpy.data.kinetics.TemplateReaction` method), 50

`getFreeEnergyOfReaction()` (`rmgpy.reaction.Reaction` method), 133

`getFreeEnergyOfReaction()`
(`rmgpy.rmg.pdep.PDepReaction` method), 145

`getFrequency()` (`rmgpy.statmech.HinderedRotor` method), 162

`getFrequencyGroups()` (`rmgpy.data.statmech.StatmechGroup` method), 45

`getHamiltonian()` (`rmgpy.statmech.HinderedRotor` method), 162

`getHeatCapacity()` (in `rmgpy.statmech.schrodinger` module), 164

`getHeatCapacity()` (`rmgpy.pdep.Configuration` method), 103

`getHeatCapacity()` (`rmgpy.rmg.model.Species` method), 147

`getHeatCapacity()` (`rmgpy.species.Species` method), 151

`getHeatCapacity()` (`rmgpy.species.TransitionState` method), 152

`getHeatCapacity()` (`rmgpy.statmech.Conformer` method), 165

`getHeatCapacity()` (`rmgpy.statmech.HarmonicOscillator` method), 161

`getHeatCapacity()` (`rmgpy.statmech.HinderedRotor` method), 162

`getHeatCapacity()` (`rmgpy.statmech.IdealGasTranslation` method), 154

`getHeatCapacity()` (`rmgpy.statmech.KRotor` method), 158

`getHeatCapacity()` (`rmgpy.statmech.LinearRotor` method), 155

`getHeatCapacity()` (`rmgpy.statmech.NonlinearRotor` method), 157

`getHeatCapacity()` (`rmgpy.statmech.SphericalTopRotor` method), 159

`getHeatCapacity()` (`rmgpy.thermo.NASA` method), 172

`getHeatCapacity()` (`rmgpy.thermo.NASAPolynomial` method), 175

`getHeatCapacity()` (`rmgpy.thermo.ThermoData` method), 167

`getHeatCapacity()` (`rmgpy.thermo.Wilhoit` method), 170

`getInChiKeyAug()` (`rmgpy.qm.gaussian.GaussianMol` method), 116

`getInChiKeyAug()` (`rmgpy.qm.gaussian.GaussianMolPM3` method), 117

`getInChiKeyAug()` (`rmgpy.qm.gaussian.GaussianMolPM6` method), 119

`getInChiKeyAug()` (`rmgpy.qm.molecule.QMMolecule` method), 112

`getInChiKeyAug()` (`rmgpy.qm.mopac.MopacMol` method), 121

`getInChiKeyAug()` (`rmgpy.qm.mopac.MopacMolPM3` method), 123

`getInChiKeyAug()` (`rmgpy.qm.mopac.MopacMolPM6` method), 124

`getInChiKeyAug()` (`rmgpy.qm.mopac.MopacMolPM7` method), 126

`getInternalReducedMomentOfInertia()`
(`rmgpy.cantherm.geometry.Geometry` method), 5

`getInternalReducedMomentOfInertia()`
(`rmgpy.statmech.Conformer` method), 165

`getKekulizedResonanceIsomers()`
(`rmgpy.molecule.Molecule` method), 88

`getKinetics()` (`rmgpy.data.kinetics.KineticsFamily` method), 25

- getKineticsForTemplate()
(rmgpy.data.kinetics.KineticsFamily method), 25
- getKineticsFromDepository()
(rmgpy.data.kinetics.KineticsFamily method), 25
- getLabeledAtom() (rmgpy.molecule.Group method), 94
- getLabeledAtom() (rmgpy.molecule.Molecule method), 88
- getLabeledAtoms() (rmgpy.molecule.Group method), 94
- getLabeledAtoms() (rmgpy.molecule.Molecule method), 88
- getLeakBranchingRatios()
(rmgpy.rmg.pdep.PDepNetwork method), 141
- getLeakCoefficient() (rmgpy.rmg.pdep.PDepNetwork method), 141
- getLevelDegeneracy() (rmgpy.statmech.HinderedRotor method), 163
- getLevelDegeneracy() (rmgpy.statmech.KRotor method), 158
- getLevelDegeneracy() (rmgpy.statmech.LinearRotor method), 156
- getLevelDegeneracy() (rmgpy.statmech.SphericalTopRotor method), 159
- getLevelEnergy() (rmgpy.statmech.HinderedRotor method), 163
- getLevelEnergy() (rmgpy.statmech.KRotor method), 158
- getLevelEnergy() (rmgpy.statmech.LinearRotor method), 156
- getLevelEnergy() (rmgpy.statmech.SphericalTopRotor method), 159
- getLists() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- getLonePairRadicalResonanceIsomers()
(rmgpy.molecule.Molecule method), 88
- getMaximumLeakSpecies()
(rmgpy.rmg.pdep.PDepNetwork method), 142
- getModelSize() (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
- getMolecularWeight() (rmgpy.molecule.Molecule method), 88
- getMolFilePathForCalculation()
(rmgpy.qm.gaussian.GaussianMol method), 116
- getMolFilePathForCalculation()
(rmgpy.qm.gaussian.GaussianMolPM3 method), 117
- getMolFilePathForCalculation()
(rmgpy.qm.gaussian.GaussianMolPM6 method), 119
- getMolFilePathForCalculation()
(rmgpy.qm.molecule.QMMolecule method), 112
- getMolFilePathForCalculation()
(rmgpy.qm.mopac.MopacMol method), 121
- getMolFilePathForCalculation()
(rmgpy.qm.mopac.MopacMolPM3 method), 123
- getMolFilePathForCalculation()
(rmgpy.qm.mopac.MopacMolPM6 method), 124
- getMolFilePathForCalculation()
(rmgpy.qm.mopac.MopacMolPM7 method), 126
- getMomentOfInertiaTensor()
(rmgpy.cantherm.geometry.Geometry method), 5
- getMomentOfInertiaTensor()
(rmgpy.statmech.Conformer method), 165
- getN5dd_N5tsResonanceIsomers()
(rmgpy.molecule.Molecule method), 88
- getNetCharge() (rmgpy.molecule.Molecule method), 88
- getNumAtoms() (rmgpy.molecule.Molecule method), 88
- getNumberDegreesOfFreedom()
(rmgpy.statmech.Conformer method), 165
- getNumberOfAtoms() (rmgpy.cantherm.gaussian.GaussianLog method), 4
- getNumberOfRadicalElectrons()
(rmgpy.molecule.Molecule method), 88
- getOtherVertex() (rmgpy.molecule.Bond method), 85
- getOtherVertex() (rmgpy.molecule.graph.Edge method), 79
- getOtherVertex() (rmgpy.molecule.GroupBond method), 92
- getParser() (rmgpy.qm.gaussian.GaussianMol method), 116
- getParser() (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
- getParser() (rmgpy.qm.gaussian.GaussianMolPM6 method), 119
- getParser() (rmgpy.qm.mopac.Mopac method), 120
- getParser() (rmgpy.qm.mopac.MopacMol method), 121
- getParser() (rmgpy.qm.mopac.MopacMolPM3 method), 123
- getParser() (rmgpy.qm.mopac.MopacMolPM6 method), 125
- getParser() (rmgpy.qm.mopac.MopacMolPM7 method), 126
- getPartitionFunction() (in module rmgpy.statmech.schrodinger), 164
- getPartitionFunction() (rmgpy.rmg.model.Species method), 147
- getPartitionFunction() (rmgpy.species.Species method), 151
- getPartitionFunction() (rmgpy.species.TransitionState method), 152

<code>getPartitionFunction()</code> (rmgpy.statmech.Conformer method), 166	<code>getRateCoefficient()</code> (rmgpy.kinetics.MultiPDepArrhenius method), 67
<code>getPartitionFunction()</code> (rmgpy.statmech.HarmonicOscillator method), 161	<code>getRateCoefficient()</code> (rmgpy.kinetics.PDepArrhenius method), 66
<code>getPartitionFunction()</code> (rmgpy.statmech.HinderedRotor method), 163	<code>getRateCoefficient()</code> (rmgpy.kinetics.PDepKineticsData method), 64
<code>getPartitionFunction()</code> (rmgpy.statmech.IdealGasTranslation method), 154	<code>getRateCoefficient()</code> (rmgpy.kinetics.ThirdBody method), 70
<code>getPartitionFunction()</code> (rmgpy.statmech.KRotor method), 158	<code>getRateCoefficient()</code> (rmgpy.kinetics.Troe method), 74
<code>getPartitionFunction()</code> (rmgpy.statmech.LinearRotor method), 156	<code>getRateCoefficient()</code> (rmgpy.reaction.Reaction method), 133
<code>getPartitionFunction()</code> (rmgpy.statmech.NonlinearRotor method), 157	<code>getRateCoefficient()</code> (rmgpy.rmg.pdep.PDepReaction method), 145
<code>getPartitionFunction()</code> (rmgpy.statmech.SphericalTopRotor method), 159	<code>getRateCoefficientUnits()</code> (rmgpy.data.kinetics.KineticsFamily method), 25
<code>getPossibleStructures()</code> (rmgpy.data.base.LogicOr method), 36	<code>getRateRule()</code> (rmgpy.data.kinetics.KineticsFamily method), 25
<code>getPotential()</code> (rmgpy.statmech.HinderedRotor method), 163	<code>getReactionPairs()</code> (rmgpy.data.kinetics.KineticsFamily method), 25
<code>getPrincipalMomentsOfInertia()</code> (rmgpy.cantherm.geometry.Geometry method), 5	<code>getReactionRates()</code> (rmgpy.reaction.ReactionModel method), 134
<code>getPrincipalMomentsOfInertia()</code> (rmgpy.statmech.Conformer method), 166	<code>getReactionRates()</code> (rmgpy.rmg.model.CoreEdgeReactionModel method), 136
<code>getRadicalAtoms()</code> (rmgpy.molecule.Molecule method), 88	<code>getReactionTemplate()</code> (rmgpy.data.kinetics.KineticsFamily method), 25
<code>getRadicalCount()</code> (rmgpy.molecule.Molecule method), 89	<code>getReactionTemplate()</code> (rmgpy.data.kinetics.KineticsGroups method), 28
<code>getRate()</code> (rmgpy.data.kinetics.DepositoryReaction method), 19	<code>getRefinedMolFilePath()</code> (rmgpy.qm.molecule.Geometry method), 111
<code>getRate()</code> (rmgpy.data.kinetics.LibraryReaction method), 35	<code>getReverse()</code> (rmgpy.data.kinetics.ReactionRecipe method), 37
<code>getRate()</code> (rmgpy.data.kinetics.TemplateReaction method), 50	<code>getRootTemplate()</code> (rmgpy.data.kinetics.KineticsFamily method), 25
<code>getRate()</code> (rmgpy.reaction.Reaction method), 133	<code>getRule()</code> (rmgpy.data.kinetics.KineticsRules method), 32
<code>getRate()</code> (rmgpy.rmg.pdep.PDepReaction method), 145	<code>getSmallestSetOfSmallestRings()</code> (rmgpy.molecule.graph.Graph method), 80
<code>getRateCoefficient()</code> (rmgpy.data.kinetics.DepositoryReaction method), 19	<code>getSmallestSetOfSmallestRings()</code> (rmgpy.molecule.Group method), 94
<code>getRateCoefficient()</code> (rmgpy.data.kinetics.LibraryReaction method), 35	<code>getSmallestSetOfSmallestRings()</code> (rmgpy.molecule.Molecule method), 89
<code>getRateCoefficient()</code> (rmgpy.data.kinetics.TemplateReaction method), 50	<code>getSource()</code> (rmgpy.data.kinetics.DepositoryReaction method), 19
<code>getRateCoefficient()</code> (rmgpy.kinetics.Arrhenius method), 62	<code>getSource()</code> (rmgpy.data.kinetics.LibraryReaction method), 35
<code>getRateCoefficient()</code> (rmgpy.kinetics.Chebyshev method), 69	<code>getSource()</code> (rmgpy.data.kinetics.TemplateReaction method), 50
<code>getRateCoefficient()</code> (rmgpy.kinetics.KineticsData method), 60	<code>getSource()</code> (rmgpy.rmg.pdep.PDepReaction method), 145
<code>getRateCoefficient()</code> (rmgpy.kinetics.Lindemann method), 72	<code>getSpecies()</code> (rmgpy.data.base.Database method), 15
<code>getRateCoefficient()</code> (rmgpy.kinetics.MultiArrhenius method), 63	<code>getSpecies()</code> (rmgpy.data.kinetics.KineticsDepository method), 22
	<code>getSpecies()</code> (rmgpy.data.kinetics.KineticsFamily method), 25

- [getSpecies\(\)](#) (rmgpy.data.kinetics.KineticsGroups method), 28
[getSpecies\(\)](#) (rmgpy.data.kinetics.KineticsLibrary method), 30
[getSpecies\(\)](#) (rmgpy.data.kinetics.KineticsRules method), 32
[getSpecies\(\)](#) (rmgpy.data.statmech.StatmechDepository method), 39
[getSpecies\(\)](#) (rmgpy.data.statmech.StatmechGroups method), 45
[getSpecies\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), 47
[getSpecies\(\)](#) (rmgpy.data.thermo.ThermoDepository method), 54
[getSpecies\(\)](#) (rmgpy.data.thermo.ThermoGroups method), 56
[getSpecies\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), 58
[getSpeciesIdentifier\(\)](#) (in module rmgpy.chemkin), 12
[getStatmechData\(\)](#) (rmgpy.data.statmech.StatmechDatabase method), 38
[getStatmechData\(\)](#) (rmgpy.data.statmech.StatmechGroups method), 45
[getStatmechDataFromDepository\(\)](#) (rmgpy.data.statmech.StatmechDatabase method), 38
[getStatmechDataFromGroups\(\)](#) (rmgpy.data.statmech.StatmechDatabase method), 38
[getStatmechDataFromLibrary\(\)](#) (rmgpy.data.statmech.StatmechDatabase method), 38
[getStoichiometricCoefficient\(\)](#) (rmgpy.data.kinetics.DepositoryReaction method), 19
[getStoichiometricCoefficient\(\)](#) (rmgpy.data.kinetics.LibraryReaction method), 35
[getStoichiometricCoefficient\(\)](#) (rmgpy.data.kinetics.TemplateReaction method), 50
[getStoichiometricCoefficient\(\)](#) (rmgpy.reaction.Reaction method), 133
[getStoichiometricCoefficient\(\)](#) (rmgpy.rmg.pdep.PDepReaction method), 145
[getStoichiometryMatrix\(\)](#) (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
[getSumOfStates\(\)](#) (in module rmgpy.statmech.schrodinger), 164
[getSumOfStates\(\)](#) (rmgpy.rmg.model.Species method), 147
[getSumOfStates\(\)](#) (rmgpy.species.Species method), 151
[getSumOfStates\(\)](#) (rmgpy.species.TransitionState method), 152
[getSumOfStates\(\)](#) (rmgpy.statmech.Conformer method), 166
[getSumOfStates\(\)](#) (rmgpy.statmech.HarmonicOscillator method), 161
[getSumOfStates\(\)](#) (rmgpy.statmech.HinderedRotor method), 163
[getSumOfStates\(\)](#) (rmgpy.statmech.IdealGasTranslation method), 154
[getSumOfStates\(\)](#) (rmgpy.statmech.KRotor method), 158
[getSumOfStates\(\)](#) (rmgpy.statmech.LinearRotor method), 156
[getSumOfStates\(\)](#) (rmgpy.statmech.NonlinearRotor method), 157
[getSumOfStates\(\)](#) (rmgpy.statmech.SphericalTopRotor method), 159
[getSymmetricTopRotors\(\)](#) (rmgpy.statmech.Conformer method), 166
[getSymmetryNumber\(\)](#) (rmgpy.molecule.Molecule method), 89
[getSymmetryNumber\(\)](#) (rmgpy.rmg.model.Species method), 147
[getSymmetryNumber\(\)](#) (rmgpy.species.Species method), 151
[getThermoData\(\)](#) (rmgpy.data.thermo.ThermoDatabase method), 52
[getThermoData\(\)](#) (rmgpy.qm.main.QMCalculator method), 111
[getThermoDataFromDepository\(\)](#) (rmgpy.data.thermo.ThermoDatabase method), 52
[getThermoDataFromGroups\(\)](#) (rmgpy.data.thermo.ThermoDatabase method), 52
[getThermoDataFromLibraries\(\)](#) (rmgpy.data.thermo.ThermoDatabase method), 52
[getThermoDataFromLibrary\(\)](#) (rmgpy.data.thermo.ThermoDatabase method), 52
[getThermoFilePath\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), 116
[getThermoFilePath\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
[getThermoFilePath\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), 119
[getThermoFilePath\(\)](#) (rmgpy.qm.molecule.QMMolecule method), 113
[getThermoFilePath\(\)](#) (rmgpy.qm.mopac.MopacMol method), 121
[getThermoFilePath\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), 123

- [getThermoFilePath\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), 125
[getThermoFilePath\(\)](#) (rmgpy.qm.mopac.MopacMolPM7 method), 126
[getTotalMass\(\)](#) (rmgpy.cantherm.geometry.Geometry method), 5
[getTotalMass\(\)](#) (rmgpy.statmech.Conformer method), 166
[getUncertainty\(\)](#) (rmgpy.quantity.ScalarQuantity method), 129
[getUncertaintyType\(\)](#) (rmgpy.quantity.ScalarQuantity method), 129
[getURL\(\)](#) (rmgpy.data.kinetics.DepositoryReaction method), 19
[getURL\(\)](#) (rmgpy.data.kinetics.LibraryReaction method), 36
[getURL\(\)](#) (rmgpy.data.kinetics.TemplateReaction method), 51
[getURL\(\)](#) (rmgpy.molecule.Molecule method), 89
[getURL\(\)](#) (rmgpy.reaction.Reaction method), 133
[getURL\(\)](#) (rmgpy.rmg.pdep.PDepReaction method), 145
[getValue\(\)](#) (rmgpy.quantity.ScalarQuantity method), 129
[Graph](#) (class in rmgpy.molecule.graph), 79
[groundStateDegeneracy](#) (rmgpy.qm.qmdata.QMDData attribute), 113
[Group](#) (class in rmgpy.molecule), 93
[GroupAtom](#) (class in rmgpy.molecule), 92
[GroupBond](#) (class in rmgpy.molecule), 92
[GroupFrequencies](#) (class in rmgpy.data.statmech), 20
- ## H
- [H0](#) (rmgpy.thermo.Wilhoit attribute), 169
[H298](#) (rmgpy.thermo.ThermoData attribute), 167
[HarmonicOscillator](#) (class in rmgpy.statmech), 160
[harmonicOscillator_d_heatCapacity_d_freq\(\)](#) (in module rmgpy.data.statmechfit), 41
[harmonicOscillator_heatCapacity\(\)](#) (in module rmgpy.data.statmechfit), 41
[hasAtom\(\)](#) (rmgpy.molecule.Group method), 94
[hasAtom\(\)](#) (rmgpy.molecule.Molecule method), 89
[hasBond\(\)](#) (rmgpy.molecule.Group method), 94
[hasBond\(\)](#) (rmgpy.molecule.Molecule method), 89
[hasEdge\(\)](#) (rmgpy.molecule.graph.Graph method), 80
[hasEdge\(\)](#) (rmgpy.molecule.Group method), 94
[hasEdge\(\)](#) (rmgpy.molecule.Molecule method), 89
[hasRateRule\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), 25
[hasRule\(\)](#) (rmgpy.data.kinetics.KineticsRules method), 32
[hasStatMech\(\)](#) (rmgpy.pdep.Configuration method), 103
[hasStatMech\(\)](#) (rmgpy.rmg.model.Species method), 147
[hasStatMech\(\)](#) (rmgpy.species.Species method), 151
[hasTemplate\(\)](#) (rmgpy.data.kinetics.DepositoryReaction method), 19
[hasTemplate\(\)](#) (rmgpy.data.kinetics.LibraryReaction method), 36
[hasTemplate\(\)](#) (rmgpy.data.kinetics.TemplateReaction method), 51
[hasTemplate\(\)](#) (rmgpy.reaction.Reaction method), 133
[hasTemplate\(\)](#) (rmgpy.rmg.pdep.PDepReaction method), 145
[hasThermo\(\)](#) (rmgpy.pdep.Configuration method), 103
[hasThermo\(\)](#) (rmgpy.rmg.model.Species method), 147
[hasThermo\(\)](#) (rmgpy.species.Species method), 151
[hasVertex\(\)](#) (rmgpy.molecule.graph.Graph method), 80
[hasVertex\(\)](#) (rmgpy.molecule.Group method), 94
[hasVertex\(\)](#) (rmgpy.molecule.Molecule method), 89
[highPlimit](#) (rmgpy.kinetics.Chebyshev attribute), 69
[highPlimit](#) (rmgpy.kinetics.Lindemann attribute), 72
[highPlimit](#) (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
[highPlimit](#) (rmgpy.kinetics.PDepArrhenius attribute), 66
[highPlimit](#) (rmgpy.kinetics.PDepKineticsData attribute), 64
[highPlimit](#) (rmgpy.kinetics.ThirdBody attribute), 70
[highPlimit](#) (rmgpy.kinetics.Troe attribute), 74
[HinderedRotor](#) (class in rmgpy.statmech), 161
[hinderedRotor_d_heatCapacity_d_barr\(\)](#) (in module rmgpy.data.statmechfit), 41
[hinderedRotor_d_heatCapacity_d_freq\(\)](#) (in module rmgpy.data.statmechfit), 41
[hinderedRotor_heatCapacity\(\)](#) (in module rmgpy.data.statmechfit), 41
- ## I
- [IdealGasTranslation](#) (class in rmgpy.statmech), 153
[incrementLonePairs\(\)](#) (rmgpy.molecule.Atom method), 84
[incrementOrder\(\)](#) (rmgpy.molecule.Bond method), 85
[incrementRadical\(\)](#) (rmgpy.molecule.Atom method), 84
[inertia](#) (rmgpy.statmech.HinderedRotor attribute), 163
[inertia](#) (rmgpy.statmech.KRotor attribute), 158
[inertia](#) (rmgpy.statmech.LinearRotor attribute), 156
[inertia](#) (rmgpy.statmech.NonlinearRotor attribute), 157
[inertia](#) (rmgpy.statmech.SphericalTopRotor attribute), 159
[initialize\(\)](#) (rmgpy.data.statmechfit.DirectFit method), 42
[initialize\(\)](#) (rmgpy.data.statmechfit.PseudoFit method), 43
[initialize\(\)](#) (rmgpy.data.statmechfit.PseudoRotorFit method), 42
[initialize\(\)](#) (rmgpy.pdep.Network method), 105
[initialize\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), 116
[initialize\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
[initialize\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), 119

- [initialize\(\) \(rmgpy.qm.main.QMCalculator method\), 111](#)
[initialize\(\) \(rmgpy.qm.molecule.QMMolecule method\), 113](#)
[initialize\(\) \(rmgpy.qm.mopac.MopacMol method\), 121](#)
[initialize\(\) \(rmgpy.qm.mopac.MopacMolPM3 method\), 123](#)
[initialize\(\) \(rmgpy.qm.mopac.MopacMolPM6 method\), 125](#)
[initialize\(\) \(rmgpy.qm.mopac.MopacMolPM7 method\), 126](#)
[initialize\(\) \(rmgpy.rmg.main.RMG method\), 139](#)
[initialize\(\) \(rmgpy.rmg.pdep.PDepNetwork method\), 142](#)
[initialize\(\) \(rmgpy.solver.ReactionSystem method\), 148](#)
[initialize\(\) \(rmgpy.solver.SimpleReactor method\), 149](#)
[initializeLog\(\) \(in module rmgpy.rmg.main\), 140](#)
[initializeLog\(\) \(rmgpy.cantherm.CanTherm method\), 6](#)
[initializeModel\(\) \(rmgpy.solver.ReactionSystem method\), 148](#)
[initializeModel\(\) \(rmgpy.solver.SimpleReactor method\), 149](#)
[inputFileKeywords\(\) \(rmgpy.qm.gaussian.GaussianMol method\), 116](#)
[inputFileKeywords\(\) \(rmgpy.qm.gaussian.GaussianMolPM3 method\), 118](#)
[inputFileKeywords\(\) \(rmgpy.qm.gaussian.GaussianMolPM6 method\), 119](#)
[inputFileKeywords\(\) \(rmgpy.qm.mopac.MopacMol method\), 121](#)
[inputFileKeywords\(\) \(rmgpy.qm.mopac.MopacMolPM3 method\), 123](#)
[inputFileKeywords\(\) \(rmgpy.qm.mopac.MopacMolPM6 method\), 125](#)
[inputFileKeywords\(\) \(rmgpy.qm.mopac.MopacMolPM7 method\), 126](#)
[inputFilePath \(rmgpy.qm.gaussian.GaussianMol attribute\), 116](#)
[inputFilePath \(rmgpy.qm.gaussian.GaussianMolPM3 attribute\), 118](#)
[inputFilePath \(rmgpy.qm.gaussian.GaussianMolPM6 attribute\), 119](#)
[inputFilePath \(rmgpy.qm.molecule.QMMolecule attribute\), 113](#)
[inputFilePath \(rmgpy.qm.mopac.MopacMol attribute\), 122](#)
[inputFilePath \(rmgpy.qm.mopac.MopacMolPM3 attribute\), 123](#)
[inputFilePath \(rmgpy.qm.mopac.MopacMolPM6 attribute\), 125](#)
[inputFilePath \(rmgpy.qm.mopac.MopacMolPM7 attribute\), 126](#)
[inputFilePath \(rmgpy.qm.symmetry.SymmetryJob attribute\), 114](#)
[invalidate\(\) \(rmgpy.pdep.Network method\), 105](#)
[invalidate\(\) \(rmgpy.rmg.pdep.PDepNetwork method\), 142](#)
[isAromatic\(\) \(rmgpy.molecule.Molecule method\), 89](#)
[isAssociation\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 19](#)
[isAssociation\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 36](#)
[isAssociation\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 51](#)
[isAssociation\(\) \(rmgpy.reaction.Reaction method\), 133](#)
[isAssociation\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 145](#)
[isAtomInCycle\(\) \(rmgpy.molecule.Molecule method\), 89](#)
[isBalanced\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 19](#)
[isBalanced\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 36](#)
[isBalanced\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 51](#)
[isBalanced\(\) \(rmgpy.reaction.Reaction method\), 133](#)
[isBalanced\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 145](#)
[isBenzene\(\) \(rmgpy.molecule.Bond method\), 85](#)
[isBimolecular\(\) \(rmgpy.pdep.Configuration method\), 103](#)
[isBondInCycle\(\) \(rmgpy.molecule.Molecule method\), 89](#)
[isCarbon\(\) \(rmgpy.molecule.Atom method\), 84](#)
[isCyclic\(\) \(rmgpy.molecule.graph.Graph method\), 80](#)
[isCyclic\(\) \(rmgpy.molecule.Group method\), 94](#)
[isCyclic\(\) \(rmgpy.molecule.Molecule method\), 89](#)
[isDissociation\(\) \(rmgpy.data.kinetics.DepositoryReaction method\), 19](#)
[isDissociation\(\) \(rmgpy.data.kinetics.LibraryReaction method\), 36](#)
[isDissociation\(\) \(rmgpy.data.kinetics.TemplateReaction method\), 51](#)
[isDissociation\(\) \(rmgpy.reaction.Reaction method\), 133](#)
[isDissociation\(\) \(rmgpy.rmg.pdep.PDepReaction method\), 145](#)
[isDouble\(\) \(rmgpy.molecule.Bond method\), 85](#)
[isEdgeInCycle\(\) \(rmgpy.molecule.graph.Graph method\), 80](#)
[isEdgeInCycle\(\) \(rmgpy.molecule.Group method\), 94](#)
[isEdgeInCycle\(\) \(rmgpy.molecule.Molecule method\), 89](#)
[isHydrogen\(\) \(rmgpy.molecule.Atom method\), 84](#)
[isIdentical\(\) \(rmgpy.molecule.Group method\), 94](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.Arrhenius method\), 62](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.Chebyshev method\), 69](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.KineticsData method\), 61](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.Lindemann method\), 72](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.MultiArrhenius method\), 63](#)
[isIdenticalTo\(\) \(rmgpy.kinetics.MultiPDepArrhenius method\), 67](#)

- isIdenticalTo() (rmgpy.kinetics.PDepArrhenius method), 66
- isIdenticalTo() (rmgpy.kinetics.PDepKineticsData method), 64
- isIdenticalTo() (rmgpy.kinetics.ThirdBody method), 70
- isIdenticalTo() (rmgpy.kinetics.Troe method), 74
- isIdenticalTo() (rmgpy.thermo.NASA method), 172
- isIdenticalTo() (rmgpy.thermo.NASAPolynomial method), 175
- isIdenticalTo() (rmgpy.thermo.ThermoData method), 168
- isIdenticalTo() (rmgpy.thermo.Wilhoit method), 171
- isIsomerization() (rmgpy.data.kinetics.DepositoryReaction method), 19
- isIsomerization() (rmgpy.data.kinetics.LibraryReaction method), 36
- isIsomerization() (rmgpy.data.kinetics.TemplateReaction method), 51
- isIsomerization() (rmgpy.reaction.Reaction method), 133
- isIsomerization() (rmgpy.rmg.pdep.PDepReaction method), 145
- isIsomorphic() (rmgpy.data.kinetics.DepositoryReaction method), 19
- isIsomorphic() (rmgpy.data.kinetics.LibraryReaction method), 36
- isIsomorphic() (rmgpy.data.kinetics.TemplateReaction method), 51
- isIsomorphic() (rmgpy.molecule.graph.Graph method), 80
- isIsomorphic() (rmgpy.molecule.Group method), 94
- isIsomorphic() (rmgpy.molecule.Molecule method), 89
- isIsomorphic() (rmgpy.molecule.vf2.VF2 method), 81
- isIsomorphic() (rmgpy.reaction.Reaction method), 133
- isIsomorphic() (rmgpy.rmg.model.Species method), 147
- isIsomorphic() (rmgpy.rmg.pdep.PDepReaction method), 145
- isIsomorphic() (rmgpy.species.Species method), 151
- isLinear() (rmgpy.molecule.Molecule method), 89
- isMappingValid() (rmgpy.molecule.graph.Graph method), 80
- isMappingValid() (rmgpy.molecule.Group method), 94
- isMappingValid() (rmgpy.molecule.Molecule method), 89
- isMoleculeForbidden() (rmgpy.data.kinetics.KineticsFamily method), 25
- isNitrogen() (rmgpy.molecule.Atom method), 84
- isNonHydrogen() (rmgpy.molecule.Atom method), 84
- isOxygen() (rmgpy.molecule.Atom method), 84
- isPressureDependent() (rmgpy.kinetics.Arrhenius method), 62
- isPressureDependent() (rmgpy.kinetics.Chebyshev method), 69
- isPressureDependent() (rmgpy.kinetics.KineticsData method), 61
- isPressureDependent() (rmgpy.kinetics.Lindemann method), 72
- isPressureDependent() (rmgpy.kinetics.MultiArrhenius method), 63
- isPressureDependent() (rmgpy.kinetics.MultiPDepArrhenius method), 67
- isPressureDependent() (rmgpy.kinetics.PDepArrhenius method), 66
- isPressureDependent() (rmgpy.kinetics.PDepKineticsData method), 64
- isPressureDependent() (rmgpy.kinetics.ThirdBody method), 71
- isPressureDependent() (rmgpy.kinetics.Troe method), 74
- isPressureValid() (rmgpy.kinetics.Chebyshev method), 69
- isPressureValid() (rmgpy.kinetics.Lindemann method), 72
- isPressureValid() (rmgpy.kinetics.MultiPDepArrhenius method), 67
- isPressureValid() (rmgpy.kinetics.PDepArrhenius method), 66
- isPressureValid() (rmgpy.kinetics.PDepKineticsData method), 64
- isPressureValid() (rmgpy.kinetics.ThirdBody method), 71
- isPressureValid() (rmgpy.kinetics.Troe method), 74
- isRadical() (rmgpy.molecule.Molecule method), 90
- isSimilarTo() (rmgpy.kinetics.Arrhenius method), 62
- isSimilarTo() (rmgpy.kinetics.Chebyshev method), 69
- isSimilarTo() (rmgpy.kinetics.KineticsData method), 61
- isSimilarTo() (rmgpy.kinetics.Lindemann method), 72
- isSimilarTo() (rmgpy.kinetics.MultiArrhenius method), 63
- isSimilarTo() (rmgpy.kinetics.MultiPDepArrhenius method), 67
- isSimilarTo() (rmgpy.kinetics.PDepArrhenius method), 66
- isSimilarTo() (rmgpy.kinetics.PDepKineticsData method), 65
- isSimilarTo() (rmgpy.kinetics.ThirdBody method), 71
- isSimilarTo() (rmgpy.kinetics.Troe method), 74
- isSimilarTo() (rmgpy.thermo.NASA method), 172
- isSimilarTo() (rmgpy.thermo.NASAPolynomial method), 175
- isSimilarTo() (rmgpy.thermo.ThermoData method), 168
- isSimilarTo() (rmgpy.thermo.Wilhoit method), 171
- isSingle() (rmgpy.molecule.Bond method), 85
- isSpecificCaseOf() (rmgpy.molecule.Atom method), 84
- isSpecificCaseOf() (rmgpy.molecule.AtomType method), 82
- isSpecificCaseOf() (rmgpy.molecule.Bond method), 85
- isSpecificCaseOf() (rmgpy.molecule.graph.Edge method), 79
- isSpecificCaseOf() (rmgpy.molecule.graph.Vertex method), 78

- isSpecificCaseOf() (rmgpy.molecule.GroupAtom method), 92
- isSpecificCaseOf() (rmgpy.molecule.GroupBond method), 93
- isSubgraphIsomorphic() (rmgpy.molecule.graph.Graph method), 80
- isSubgraphIsomorphic() (rmgpy.molecule.Group method), 95
- isSubgraphIsomorphic() (rmgpy.molecule.Molecule method), 90
- isSubgraphIsomorphic() (rmgpy.molecule.vf2.VF2 method), 81
- isTemperatureValid() (rmgpy.kinetics.Arrhenius method), 62
- isTemperatureValid() (rmgpy.kinetics.Chebyshev method), 69
- isTemperatureValid() (rmgpy.kinetics.KineticsData method), 61
- isTemperatureValid() (rmgpy.kinetics.Lindemann method), 72
- isTemperatureValid() (rmgpy.kinetics.MultiArrhenius method), 63
- isTemperatureValid() (rmgpy.kinetics.MultiPDepArrhenius method), 67
- isTemperatureValid() (rmgpy.kinetics.PDepArrhenius method), 66
- isTemperatureValid() (rmgpy.kinetics.PDepKineticsData method), 65
- isTemperatureValid() (rmgpy.kinetics.ThirdBody method), 71
- isTemperatureValid() (rmgpy.kinetics.Troe method), 74
- isTemperatureValid() (rmgpy.thermo.NASA method), 172
- isTemperatureValid() (rmgpy.thermo.NASAPolynomial method), 175
- isTemperatureValid() (rmgpy.thermo.ThermoData method), 168
- isTemperatureValid() (rmgpy.thermo.Wilhoit method), 171
- isTransitionState() (rmgpy.pdep.Configuration method), 103
- isTriple() (rmgpy.molecule.Bond method), 85
- isUncertaintyAdditive() (rmgpy.quantity.ArrayQuantity method), 130
- isUncertaintyAdditive() (rmgpy.quantity.ScalarQuantity method), 129
- isUncertaintyMultiplicative() (rmgpy.quantity.ArrayQuantity method), 130
- isUncertaintyMultiplicative() (rmgpy.quantity.ScalarQuantity method), 129
- isUnimolecular() (rmgpy.data.kinetics.DepositoryReaction method), 19
- isUnimolecular() (rmgpy.data.kinetics.LibraryReaction method), 36
- isUnimolecular() (rmgpy.data.kinetics.TemplateReaction method), 51
- isUnimolecular() (rmgpy.pdep.Configuration method), 103
- isUnimolecular() (rmgpy.reaction.Reaction method), 133
- isUnimolecular() (rmgpy.rmg.pdep.PDepReaction method), 145
- isVertexInCycle() (rmgpy.molecule.graph.Graph method), 80
- isVertexInCycle() (rmgpy.molecule.Group method), 95
- isVertexInCycle() (rmgpy.molecule.Molecule method), 90
- ## J
- jacobian() (rmgpy.solver.SimpleReactor method), 149
- ## K
- kdata (rmgpy.kinetics.KineticsData attribute), 61
- kdata (rmgpy.kinetics.PDepKineticsData attribute), 65
- keywords (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 118
- keywords (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 119
- keywords (rmgpy.qm.mopac.MopacMol attribute), 122
- KineticsData (class in rmgpy.kinetics), 60
- KineticsDatabase (class in rmgpy.data.kinetics), 20
- KineticsDepository (class in rmgpy.data.kinetics), 22
- KineticsFamily (class in rmgpy.data.kinetics), 23
- KineticsGroups (class in rmgpy.data.kinetics), 27
- KineticsJob (class in rmgpy.cantherm), 6
- KineticsLibrary (class in rmgpy.data.kinetics), 29
- KineticsRules (class in rmgpy.data.kinetics), 31
- KRotor (class in rmgpy.statmech), 157
- kunits (rmgpy.kinetics.Chebyshev attribute), 69
- ## L
- label (rmgpy.rmg.pdep.PDepNetwork attribute), 142
- LibraryReaction (class in rmgpy.data.kinetics), 33
- Lindemann (class in rmgpy.kinetics), 71
- LinearRotor (class in rmgpy.statmech), 155
- load() (rmgpy.cantherm.StatMechJob method), 9
- load() (rmgpy.data.base.Database method), 15
- load() (rmgpy.data.kinetics.KineticsDatabase method), 21
- load() (rmgpy.data.kinetics.KineticsFamily method), 25
- load() (rmgpy.data.kinetics.KineticsGroups method), 28
- load() (rmgpy.data.kinetics.KineticsRules method), 32
- load() (rmgpy.data.statmech.StatmechDatabase method), 38
- load() (rmgpy.data.statmech.StatmechDepository method), 39
- load() (rmgpy.data.statmech.StatmechGroups method), 45

- `load()` (`rmgpy.data.statmech.StatmechLibrary` method), 47
- `load()` (`rmgpy.data.thermo.ThermoDatabase` method), 52
- `load()` (`rmgpy.data.thermo.ThermoDepository` method), 54
- `load()` (`rmgpy.data.thermo.ThermoGroups` method), 56
- `load()` (`rmgpy.data.thermo.ThermoLibrary` method), 58
- `loadChemkinFile()` (in module `rmgpy.chemkin`), 11
- `loadConformer()` (`rmgpy.cantherm.gaussian.GaussianLog` method), 4
- `loadDepository()` (`rmgpy.data.statmech.StatmechDatabase` method), 38
- `loadDepository()` (`rmgpy.data.thermo.ThermoDatabase` method), 52
- `loadEnergy()` (`rmgpy.cantherm.gaussian.GaussianLog` method), 4
- `loadFamilies()` (`rmgpy.data.kinetics.KineticsDatabase` method), 21
- `loadForbidden()` (`rmgpy.data.kinetics.KineticsFamily` method), 26
- `loadForceConstantMatrix()`
(`rmgpy.cantherm.gaussian.GaussianLog` method), 4
- `loadGeometry()` (`rmgpy.cantherm.gaussian.GaussianLog` method), 4
- `loadGroups()` (`rmgpy.data.statmech.StatmechDatabase` method), 38
- `loadGroups()` (`rmgpy.data.thermo.ThermoDatabase` method), 53
- `loadInput()` (`rmgpy.rmg.main.RMG` method), 139
- `loadInputFile()` (in module `rmgpy.cantherm.input`), 5
- `loadInputFile()` (`rmgpy.cantherm.CanTherm` method), 6
- `loadLibraries()` (`rmgpy.data.kinetics.KineticsDatabase` method), 21
- `loadLibraries()` (`rmgpy.data.statmech.StatmechDatabase` method), 38
- `loadLibraries()` (`rmgpy.data.thermo.ThermoDatabase` method), 53
- `loadNegativeFrequency()`
(`rmgpy.cantherm.gaussian.GaussianLog` method), 4
- `loadOld()` (`rmgpy.data.base.Database` method), 16
- `loadOld()` (`rmgpy.data.kinetics.KineticsDatabase` method), 21
- `loadOld()` (`rmgpy.data.kinetics.KineticsDepository` method), 22
- `loadOld()` (`rmgpy.data.kinetics.KineticsFamily` method), 26
- `loadOld()` (`rmgpy.data.kinetics.KineticsGroups` method), 28
- `loadOld()` (`rmgpy.data.kinetics.KineticsLibrary` method), 30
- `loadOld()` (`rmgpy.data.kinetics.KineticsRules` method), 32
- `loadOld()` (`rmgpy.data.statmech.StatmechDatabase` method), 38
- `loadOld()` (`rmgpy.data.statmech.StatmechDepository` method), 39
- `loadOld()` (`rmgpy.data.statmech.StatmechGroups` method), 45
- `loadOld()` (`rmgpy.data.statmech.StatmechLibrary` method), 47
- `loadOld()` (`rmgpy.data.thermo.ThermoDatabase` method), 53
- `loadOld()` (`rmgpy.data.thermo.ThermoDepository` method), 54
- `loadOld()` (`rmgpy.data.thermo.ThermoGroups` method), 56
- `loadOld()` (`rmgpy.data.thermo.ThermoLibrary` method), 58
- `loadOldDictionary()` (`rmgpy.data.base.Database` method), 16
- `loadOldDictionary()` (`rmgpy.data.kinetics.KineticsDepository` method), 22
- `loadOldDictionary()` (`rmgpy.data.kinetics.KineticsFamily` method), 26
- `loadOldDictionary()` (`rmgpy.data.kinetics.KineticsGroups` method), 28
- `loadOldDictionary()` (`rmgpy.data.kinetics.KineticsLibrary` method), 30
- `loadOldDictionary()` (`rmgpy.data.kinetics.KineticsRules` method), 32
- `loadOldDictionary()` (`rmgpy.data.statmech.StatmechDepository` method), 39
- `loadOldDictionary()` (`rmgpy.data.statmech.StatmechGroups` method), 45
- `loadOldDictionary()` (`rmgpy.data.statmech.StatmechLibrary` method), 47
- `loadOldDictionary()` (`rmgpy.data.thermo.ThermoDepository` method), 54
- `loadOldDictionary()` (`rmgpy.data.thermo.ThermoGroups` method), 56
- `loadOldDictionary()` (`rmgpy.data.thermo.ThermoLibrary` method), 58
- `loadOldLibrary()` (`rmgpy.data.base.Database` method), 16
- `loadOldLibrary()` (`rmgpy.data.kinetics.KineticsDepository` method), 22
- `loadOldLibrary()` (`rmgpy.data.kinetics.KineticsFamily` method), 26
- `loadOldLibrary()` (`rmgpy.data.kinetics.KineticsGroups` method), 28
- `loadOldLibrary()` (`rmgpy.data.kinetics.KineticsLibrary` method), 30
- `loadOldLibrary()` (`rmgpy.data.kinetics.KineticsRules` method), 32
- `loadOldLibrary()` (`rmgpy.data.statmech.StatmechDepository` method), 39

- loadOldLibrary() (rmgpy.data.statmech.StatmechGroups method), 45
- loadOldLibrary() (rmgpy.data.statmech.StatmechLibrary method), 47
- loadOldLibrary() (rmgpy.data.thermo.ThermoDepository method), 54
- loadOldLibrary() (rmgpy.data.thermo.ThermoGroups method), 56
- loadOldLibrary() (rmgpy.data.thermo.ThermoLibrary method), 58
- loadOldTemplate() (rmgpy.data.kinetics.KineticsFamily method), 26
- loadOldTree() (rmgpy.data.base.Database method), 16
- loadOldTree() (rmgpy.data.kinetics.KineticsDepository method), 22
- loadOldTree() (rmgpy.data.kinetics.KineticsFamily method), 26
- loadOldTree() (rmgpy.data.kinetics.KineticsGroups method), 28
- loadOldTree() (rmgpy.data.kinetics.KineticsLibrary method), 30
- loadOldTree() (rmgpy.data.kinetics.KineticsRules method), 32
- loadOldTree() (rmgpy.data.statmech.StatmechDepository method), 39
- loadOldTree() (rmgpy.data.statmech.StatmechGroups method), 45
- loadOldTree() (rmgpy.data.statmech.StatmechLibrary method), 47
- loadOldTree() (rmgpy.data.thermo.ThermoDepository method), 54
- loadOldTree() (rmgpy.data.thermo.ThermoGroups method), 56
- loadOldTree() (rmgpy.data.thermo.ThermoLibrary method), 58
- loadRecipe() (rmgpy.data.kinetics.KineticsFamily method), 26
- loadRecommendedFamiliesList() (rmgpy.data.kinetics.KineticsDatabase method), 21
- loadRestartFile() (rmgpy.rmg.main.RMG method), 139
- loadRMGJavaInput() (rmgpy.rmg.main.RMG method), 139
- loadScanEnergies() (rmgpy.cantherm.gaussian.GaussianLog method), 5
- loadSpeciesDictionary() (in module rmgpy.chemkin), 11
- loadTemplate() (rmgpy.data.kinetics.KineticsFamily method), 26
- loadThermoData() (rmgpy.qm.gaussian.GaussianMol method), 116
- loadThermoData() (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
- loadThermoData() (rmgpy.qm.gaussian.GaussianMolPM6 method), 119
- loadThermoData() (rmgpy.qm.molecule.QMMolecule method), 113
- loadThermoData() (rmgpy.qm.mopac.MopacMol method), 122
- loadThermoData() (rmgpy.qm.mopac.MopacMolPM3 method), 123
- loadThermoData() (rmgpy.qm.mopac.MopacMolPM6 method), 125
- loadThermoData() (rmgpy.qm.mopac.MopacMolPM7 method), 126
- loadThermoInput() (rmgpy.rmg.main.RMG method), 139
- loadTransportFile() (in module rmgpy.chemkin), 11
- loadZeroPointEnergy() (rmgpy.cantherm.gaussian.GaussianLog method), 5
- logConversions() (rmgpy.solver.ReactionSystem method), 148
- logConversions() (rmgpy.solver.SimpleReactor method), 149
- logFooter() (rmgpy.cantherm.CanTherm method), 6
- logHeader() (rmgpy.cantherm.CanTherm method), 6
- logHeader() (rmgpy.rmg.main.RMG method), 139
- LogicAnd (class in rmgpy.data.base), 36
- LogicNode (class in rmgpy.data.base), 36
- LogicOr (class in rmgpy.data.base), 36
- logRates() (rmgpy.solver.ReactionSystem method), 148
- logRates() (rmgpy.solver.SimpleReactor method), 149
- ## M
- makeLogicNode() (in module rmgpy.data.base), 37
- makeNewPDepReaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- makeNewReaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- makeNewSpecies() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- makeOutputSubdirectory() (rmgpy.rmg.main.RMG method), 139
- makeProfileGraph() (in module rmgpy.rmg.main), 140
- mapDensitiesOfStates() (rmgpy.pdep.Network method), 105
- mapDensitiesOfStates() (rmgpy.rmg.pdep.PDepNetwork method), 142
- mapDensityOfStates() (rmgpy.pdep.Configuration method), 103
- mapSumOfStates() (rmgpy.pdep.Configuration method), 104
- markChemkinDuplicates() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- markDuplicateReactions() (in module rmgpy.chemkin), 12
- markValidDuplicates() (rmgpy.data.kinetics.KineticsLibrary method), 30

mass (rmgpy.statmech.Conformer attribute), 166
mass (rmgpy.statmech.IdealGasTranslation attribute), 154
matchesMolecules() (rmgpy.data.kinetics.DepositoryReaction method), 19
matchesMolecules() (rmgpy.data.kinetics.LibraryReaction method), 36
matchesMolecules() (rmgpy.data.kinetics.TemplateReaction method), 51
matchesMolecules() (rmgpy.reaction.Reaction method), 133
matchesMolecules() (rmgpy.rmg.pdep.PDepReaction method), 145
matchLogicOr() (rmgpy.data.base.LogicOr method), 37
matchNodeToChild() (rmgpy.data.base.Database method), 16
matchNodeToChild() (rmgpy.data.kinetics.KineticsDepository method), 22
matchNodeToChild() (rmgpy.data.kinetics.KineticsFamily method), 26
matchNodeToChild() (rmgpy.data.kinetics.KineticsGroups method), 28
matchNodeToChild() (rmgpy.data.kinetics.KineticsLibrary method), 30
matchNodeToChild() (rmgpy.data.kinetics.KineticsRules method), 32
matchNodeToChild() (rmgpy.data.statmech.StatmechDepository method), 39
matchNodeToChild() (rmgpy.data.statmech.StatmechGroups method), 45
matchNodeToChild() (rmgpy.data.statmech.StatmechLibrary method), 47
matchNodeToChild() (rmgpy.data.thermo.ThermoDepository method), 54
matchNodeToChild() (rmgpy.data.thermo.ThermoGroups method), 56
matchNodeToChild() (rmgpy.data.thermo.ThermoLibrary method), 58
matchNodeToNode() (rmgpy.data.base.Database method), 16
matchNodeToNode() (rmgpy.data.kinetics.KineticsDepository method), 22
matchNodeToNode() (rmgpy.data.kinetics.KineticsFamily method), 26
matchNodeToNode() (rmgpy.data.kinetics.KineticsGroups method), 28
matchNodeToNode() (rmgpy.data.kinetics.KineticsLibrary method), 30
matchNodeToNode() (rmgpy.data.kinetics.KineticsRules method), 32
matchNodeToNode() (rmgpy.data.statmech.StatmechDepository method), 40
matchNodeToNode() (rmgpy.data.statmech.StatmechGroups method), 45
matchNodeToNode() (rmgpy.data.statmech.StatmechLibrary method), 47
matchNodeToNode() (rmgpy.data.thermo.ThermoDepository method), 54
matchNodeToNode() (rmgpy.data.thermo.ThermoGroups method), 56
matchNodeToNode() (rmgpy.data.thermo.ThermoLibrary method), 58
matchNodeToNode() (rmgpy.data.thermo.ThermoGroups method), 56
matchNodeToNode() (rmgpy.data.thermo.ThermoLibrary method), 58
matchNodeToStructure() (rmgpy.data.base.Database method), 16
matchNodeToStructure() (rmgpy.data.kinetics.KineticsDepository method), 23
matchNodeToStructure() (rmgpy.data.kinetics.KineticsFamily method), 26
matchNodeToStructure() (rmgpy.data.kinetics.KineticsGroups method), 29
matchNodeToStructure() (rmgpy.data.kinetics.KineticsLibrary method), 30
matchNodeToStructure() (rmgpy.data.kinetics.KineticsRules method), 32
matchNodeToStructure() (rmgpy.data.statmech.StatmechDepository method), 40
matchNodeToStructure() (rmgpy.data.statmech.StatmechGroups method), 46
matchNodeToStructure() (rmgpy.data.statmech.StatmechLibrary method), 47
matchNodeToStructure() (rmgpy.data.thermo.ThermoDepository method), 54
matchNodeToStructure() (rmgpy.data.thermo.ThermoGroups method), 56
matchNodeToStructure() (rmgpy.data.thermo.ThermoLibrary method), 58
matchToStructure() (rmgpy.data.base.LogicAnd method), 36
matchToStructure() (rmgpy.data.base.LogicOr method), 37
maxAttempts (rmgpy.qm.gaussian.GaussianMol attribute), 116
maxAttempts (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 118
maxAttempts (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 120

maxAttempts (rmgpy.qm.molecule.QMMolecule attribute), 113
 maxAttempts (rmgpy.qm.mopac.MopacMol attribute), 122
 maxAttempts (rmgpy.qm.mopac.MopacMolPM3 attribute), 123
 maxAttempts (rmgpy.qm.mopac.MopacMolPM6 attribute), 125
 maxAttempts (rmgpy.qm.mopac.MopacMolPM7 attribute), 126
 maximumGrainSize (rmgpy.cantherm.PressureDependenceJob attribute), 9
 merge() (rmgpy.molecule.graph.Graph method), 80
 merge() (rmgpy.molecule.Group method), 95
 merge() (rmgpy.molecule.Molecule method), 90
 merge() (rmgpy.reaction.ReactionModel method), 134
 merge() (rmgpy.rmg.pdep.PDepNetwork method), 142
 modes (rmgpy.statmech.Conformer attribute), 166
 Molecule (class in rmgpy.molecule), 85
 MoleculeDrawer (class in rmgpy.molecule.draw), 98
 Mopac (class in rmgpy.qm.mopac), 120
 MopacMol (class in rmgpy.qm.mopac), 121
 MopacMolPM3 (class in rmgpy.qm.mopac), 122
 MopacMolPM6 (class in rmgpy.qm.mopac), 124
 MopacMolPM7 (class in rmgpy.qm.mopac), 125
 MultiArrhenius (class in rmgpy.kinetics), 62
 MultiPDepArrhenius (class in rmgpy.kinetics), 66

N

n (rmgpy.kinetics.Arrhenius attribute), 62
 n (rmgpy.pdep.SingleExponentialDown attribute), 101
 NASA (class in rmgpy.thermo), 171
 NASAPolynomial (class in rmgpy.thermo), 174
 Network (class in rmgpy.pdep), 104
 NonlinearRotor (class in rmgpy.statmech), 156
 number (rmgpy.statmech.Conformer attribute), 166
 numberOfAtoms (rmgpy.qm.qmdata.QMData attribute), 113

O

opticalIsomers (rmgpy.statmech.Conformer attribute), 166
 outputFilePath (rmgpy.qm.gaussian.GaussianMol attribute), 116
 outputFilePath (rmgpy.qm.gaussian.GaussianMolPM3 attribute), 118
 outputFilePath (rmgpy.qm.gaussian.GaussianMolPM6 attribute), 120
 outputFilePath (rmgpy.qm.molecule.QMMolecule attribute), 113
 outputFilePath (rmgpy.qm.mopac.MopacMol attribute), 122
 outputFilePath (rmgpy.qm.mopac.MopacMolPM3 attribute), 123

outputFilePath (rmgpy.qm.mopac.MopacMolPM6 attribute), 125
 outputFilePath (rmgpy.qm.mopac.MopacMolPM7 attribute), 126

P

parse() (rmgpy.qm.gaussian.Gaussian method), 115
 parse() (rmgpy.qm.gaussian.GaussianMol method), 116
 parse() (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
 parse() (rmgpy.qm.gaussian.GaussianMolPM6 method), 120
 parse() (rmgpy.qm.molecule.QMMolecule method), 113
 parse() (rmgpy.qm.mopac.MopacMol method), 122
 parse() (rmgpy.qm.mopac.MopacMolPM3 method), 123
 parse() (rmgpy.qm.mopac.MopacMolPM6 method), 125
 parse() (rmgpy.qm.mopac.MopacMolPM7 method), 127
 parse() (rmgpy.qm.symmetry.SymmetryJob method), 115
 parseCommandLineArguments() (rmgpy.cantherm.CanTherm method), 7
 parseOldLibrary() (rmgpy.data.base.Database method), 16
 parseOldLibrary() (rmgpy.data.kinetics.KineticsDepository method), 23
 parseOldLibrary() (rmgpy.data.kinetics.KineticsFamily method), 27
 parseOldLibrary() (rmgpy.data.kinetics.KineticsGroups method), 29
 parseOldLibrary() (rmgpy.data.kinetics.KineticsLibrary method), 31
 parseOldLibrary() (rmgpy.data.kinetics.KineticsRules method), 33
 parseOldLibrary() (rmgpy.data.statmech.StatmechDepository method), 40
 parseOldLibrary() (rmgpy.data.statmech.StatmechGroups method), 46
 parseOldLibrary() (rmgpy.data.statmech.StatmechLibrary method), 48
 parseOldLibrary() (rmgpy.data.thermo.ThermoDepository method), 55
 parseOldLibrary() (rmgpy.data.thermo.ThermoGroups method), 57
 parseOldLibrary() (rmgpy.data.thermo.ThermoLibrary method), 59
 Pdata (rmgpy.kinetics.PDepKineticsData attribute), 64
 PDepArrhenius (class in rmgpy.kinetics), 65
 PDepKineticsData (class in rmgpy.kinetics), 64
 PDepNetwork (class in rmgpy.rmg.pdep), 141
 PDepReaction (class in rmgpy.rmg.pdep), 143
 Plist (rmgpy.cantherm.PressureDependenceJob attribute), 8
 plot() (rmgpy.cantherm.KineticsJob method), 6
 plot() (rmgpy.cantherm.ThermoJob method), 9

- plotHinderedRotor() (rmgpy.cantherm.StatMechJob method), 9
- Pmax (rmgpy.cantherm.PressureDependenceJob attribute), 8
- Pmax (rmgpy.kinetics.Arrhenius attribute), 61
- Pmax (rmgpy.kinetics.Chebyshev attribute), 68
- Pmax (rmgpy.kinetics.KineticsData attribute), 60
- Pmax (rmgpy.kinetics.Lindemann attribute), 71
- Pmax (rmgpy.kinetics.MultiArrhenius attribute), 63
- Pmax (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
- Pmax (rmgpy.kinetics.PDepArrhenius attribute), 65
- Pmax (rmgpy.kinetics.PDepKineticsData attribute), 64
- Pmax (rmgpy.kinetics.ThirdBody attribute), 70
- Pmax (rmgpy.kinetics.Troe attribute), 73
- Pmin (rmgpy.cantherm.PressureDependenceJob attribute), 8
- Pmin (rmgpy.kinetics.Arrhenius attribute), 61
- Pmin (rmgpy.kinetics.Chebyshev attribute), 68
- Pmin (rmgpy.kinetics.KineticsData attribute), 60
- Pmin (rmgpy.kinetics.Lindemann attribute), 71
- Pmin (rmgpy.kinetics.MultiArrhenius attribute), 63
- Pmin (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
- Pmin (rmgpy.kinetics.PDepArrhenius attribute), 65
- Pmin (rmgpy.kinetics.PDepKineticsData attribute), 64
- Pmin (rmgpy.kinetics.ThirdBody attribute), 70
- Pmin (rmgpy.kinetics.Troe attribute), 73
- PointGroup (class in rmgpy.qm.symmetry), 114
- PointGroupCalculator (class in rmgpy.qm.symmetry), 114
- poly1 (rmgpy.thermo.NASA attribute), 173
- poly2 (rmgpy.thermo.NASA attribute), 173
- poly3 (rmgpy.thermo.NASA attribute), 173
- polynomials (rmgpy.thermo.NASA attribute), 173
- PressureDependenceJob (class in rmgpy.cantherm), 7
- pressures (rmgpy.kinetics.PDepArrhenius attribute), 66
- prettify() (in module rmgpy.cantherm.output), 7
- PrettifyVisitor (class in rmgpy.cantherm.output), 7
- printEnlargeSummary() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- printSummary() (rmgpy.pdep.Network method), 105
- printSummary() (rmgpy.rmg.pdep.PDepNetwork method), 142
- processNewReactions() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- processOldLibraryEntry() (rmgpy.data.kinetics.KineticsRules method), 33
- processOldLibraryEntry() (rmgpy.data.statmech.StatmechGroups method), 46
- processOldLibraryEntry() (rmgpy.data.statmech.StatmechLibrary method), 48
- processOldLibraryEntry() (rmgpy.data.thermo.ThermoGroups method), 57
- processOldLibraryEntry() (rmgpy.data.thermo.ThermoLibrary method), 59
- processProfileStats() (in module rmgpy.rmg.main), 140
- processThermoData() (rmgpy.rmg.model.Species method), 147
- prune() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- pruneHeteroatoms() (rmgpy.data.thermo.ThermoDatabase method), 53
- PseudoFit (class in rmgpy.data.statmechfit), 43
- PseudoRotorFit (class in rmgpy.data.statmechfit), 42
- ## Q
- QMCALCULATOR (class in rmgpy.qm.main), 110
- QMData (class in rmgpy.qm.qmdata), 113
- QMMolecule (class in rmgpy.qm.molecule), 112
- QMSettings (class in rmgpy.qm.main), 110
- QMVerifier (class in rmgpy.qm.qmverifier), 114
- Quantity() (in module rmgpy.quantity), 130
- quantum (rmgpy.statmech.HarmonicOscillator attribute), 161
- quantum (rmgpy.statmech.HinderedRotor attribute), 163
- quantum (rmgpy.statmech.IdealGasTranslation attribute), 154
- quantum (rmgpy.statmech.KRRotor attribute), 158
- quantum (rmgpy.statmech.LinearRotor attribute), 156
- quantum (rmgpy.statmech.NonlinearRotor attribute), 157
- quantum (rmgpy.statmech.SphericalTopRotor attribute), 159
- ## R
- rd_build() (rmgpy.qm.molecule.Geometry method), 111
- rd_embed() (rmgpy.qm.molecule.Geometry method), 111
- Reaction (class in rmgpy.rmg.model.CoreEdgeReactionModel method), 137
- Reaction (class in rmgpy.reaction), 131
- ReactionDrawer (class in rmgpy.molecule.draw), 99
- ReactionModel (class in rmgpy.reaction), 134
- ReactionModel (class in rmgpy.rmg.model), 138
- ReactionRecipe (class in rmgpy.data.kinetics), 37
- ReactionSystem (class in rmgpy.solver), 148
- readInputFile() (in module rmgpy.rmg.input), 138
- readKineticsEntry() (in module rmgpy.chemkin), 11
- readMeaningfulLineJava() (rmgpy.rmg.main.RMG method), 139
- readReactionComments() (in module rmgpy.chemkin), 11
- readReactionsBlock() (in module rmgpy.chemkin), 11
- readThermoEntry() (in module rmgpy.chemkin), 11
- removeAtom() (rmgpy.molecule.Group method), 95
- removeAtom() (rmgpy.molecule.Molecule method), 90

- removeBond() (rmgpy.molecule.Group method), 95
 removeBond() (rmgpy.molecule.Molecule method), 90
 removeCommentFromLine() (in module rmgpy.chemkin), 11
 removeEdge() (rmgpy.molecule.graph.Graph method), 80
 removeEdge() (rmgpy.molecule.Group method), 95
 removeEdge() (rmgpy.molecule.Molecule method), 90
 removeSpeciesFromEdge() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
 removeVertex() (rmgpy.molecule.graph.Graph method), 80
 removeVertex() (rmgpy.molecule.Group method), 95
 removeVertex() (rmgpy.molecule.Molecule method), 90
 render() (rmgpy.molecule.draw.MoleculeDrawer method), 99
 resetConnectivityValues() (rmgpy.molecule.Atom method), 84
 resetConnectivityValues() (rmgpy.molecule.graph.Graph method), 80
 resetConnectivityValues() (rmgpy.molecule.graph.Vertex method), 78
 resetConnectivityValues() (rmgpy.molecule.Group method), 95
 resetConnectivityValues() (rmgpy.molecule.GroupAtom method), 92
 resetConnectivityValues() (rmgpy.molecule.Molecule method), 90
 residual() (rmgpy.solver.ReactionSystem method), 148
 residual() (rmgpy.solver.SimpleReactor method), 149
 RMG (class in rmgpy.rmg.main), 138
 rmgpy.cantherm (module), 3
 rmgpy.chemkin (module), 10
 rmgpy.constants (module), 13
 rmgpy.data (module), 13
 rmgpy.kinetics (module), 59
 rmgpy.molecule (module), 77
 rmgpy.molecule.adjlist (module), 95
 rmgpy.pdep (module), 99
 rmgpy.qm (module), 109
 rmgpy.quantity (module), 127
 rmgpy.reaction (module), 130
 rmgpy.rmg (module), 134
 rmgpy.solver (module), 147
 rmgpy.species (module), 149
 rmgpy.statmech (module), 152
 rmgpy.statmech.schrodinger (module), 163
 rmgpy.thermo (module), 166
 rotationalConstant (rmgpy.statmech.HinderedRotor attribute), 163
 rotationalConstant (rmgpy.statmech.KRotor attribute), 158
 rotationalConstant (rmgpy.statmech.LinearRotor attribute), 156
 rotationalConstant (rmgpy.statmech.NonlinearRotor attribute), 157
 rotationalConstant (rmgpy.statmech.SphericalTopRotor attribute), 159
 rotationalConstants (rmgpy.qm.qmdata.QMData attribute), 113
 run() (rmgpy.qm.symmetry.SymmetryJob method), 115
- ## S
- S0 (rmgpy.thermo.Wilhoit attribute), 170
 S298 (rmgpy.thermo.ThermoData attribute), 167
 saturate() (rmgpy.molecule.Molecule method), 90
 save() (rmgpy.cantherm.KineticsJob method), 6
 save() (rmgpy.cantherm.StatMechJob method), 9
 save() (rmgpy.cantherm.ThermoJob method), 10
 save() (rmgpy.data.base.Database method), 16
 save() (rmgpy.data.kinetics.KineticsDatabase method), 21
 save() (rmgpy.data.kinetics.KineticsDepository method), 23
 save() (rmgpy.data.kinetics.KineticsFamily method), 27
 save() (rmgpy.data.kinetics.KineticsGroups method), 29
 save() (rmgpy.data.kinetics.KineticsLibrary method), 31
 save() (rmgpy.data.kinetics.KineticsRules method), 33
 save() (rmgpy.data.statmech.StatmechDatabase method), 38
 save() (rmgpy.data.statmech.StatmechDepository method), 40
 save() (rmgpy.data.statmech.StatmechGroups method), 46
 save() (rmgpy.data.statmech.StatmechLibrary method), 48
 save() (rmgpy.data.thermo.ThermoDatabase method), 53
 save() (rmgpy.data.thermo.ThermoDepository method), 55
 save() (rmgpy.data.thermo.ThermoGroups method), 57
 save() (rmgpy.data.thermo.ThermoLibrary method), 59
 saveChemkinFile() (in module rmgpy.chemkin), 11
 saveChemkinFile() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
 saveChemkinFiles() (rmgpy.rmg.main.RMG method), 140
 saveCoordinatesFromQMData() (rmgpy.qm.molecule.Geometry method), 111
 saveDepository() (rmgpy.data.kinetics.KineticsFamily method), 27
 saveDepository() (rmgpy.data.statmech.StatmechDatabase method), 38
 saveDepository() (rmgpy.data.thermo.ThermoDatabase method), 53
 saveDictionary() (rmgpy.data.base.Database method), 16
 saveDictionary() (rmgpy.data.kinetics.KineticsDepository method), 23

`saveDictionary()` (rmgpy.data.kinetics.KineticsFamily method), 27

`saveDictionary()` (rmgpy.data.kinetics.KineticsGroups method), 29

`saveDictionary()` (rmgpy.data.kinetics.KineticsLibrary method), 31

`saveDictionary()` (rmgpy.data.kinetics.KineticsRules method), 33

`saveDictionary()` (rmgpy.data.statmech.StatmechDepository method), 40

`saveDictionary()` (rmgpy.data.statmech.StatmechGroups method), 46

`saveDictionary()` (rmgpy.data.statmech.StatmechLibrary method), 48

`saveDictionary()` (rmgpy.data.thermo.ThermoDepository method), 55

`saveDictionary()` (rmgpy.data.thermo.ThermoGroups method), 57

`saveDictionary()` (rmgpy.data.thermo.ThermoLibrary method), 59

`saveDiffHTML()` (in module rmgpy.rmg.output), 140

`saveEntry()` (rmgpy.data.kinetics.KineticsDepository method), 23

`saveEntry()` (rmgpy.data.kinetics.KineticsFamily method), 27

`saveEntry()` (rmgpy.data.kinetics.KineticsLibrary method), 31

`saveEntry()` (rmgpy.data.kinetics.KineticsRules method), 33

`saveEntry()` (rmgpy.data.statmech.StatmechDepository method), 40

`saveEntry()` (rmgpy.data.statmech.StatmechGroups method), 46

`saveEntry()` (rmgpy.data.statmech.StatmechLibrary method), 48

`saveEntry()` (rmgpy.data.thermo.ThermoDepository method), 55

`saveEntry()` (rmgpy.data.thermo.ThermoGroups method), 57

`saveEntry()` (rmgpy.data.thermo.ThermoLibrary method), 59

`saveEverything()` (rmgpy.rmg.main.RMG method), 140

`saveExecutionStatistics()` (rmgpy.rmg.main.RMG method), 140

`saveFamilies()` (rmgpy.data.kinetics.KineticsDatabase method), 21

`saveGroups()` (rmgpy.data.kinetics.KineticsFamily method), 27

`saveGroups()` (rmgpy.data.statmech.StatmechDatabase method), 38

`saveGroups()` (rmgpy.data.thermo.ThermoDatabase method), 53

`saveHTMLFile()` (in module rmgpy.chemkin), 12

`saveInput()` (rmgpy.rmg.main.RMG method), 140

`saveInputFile()` (in module rmgpy.rmg.input), 138

`saveInputFile()` (rmgpy.cantherm.PressureDependenceJob method), 9

`saveJavaKineticsLibrary()` (in module rmgpy.chemkin), 12

`saveLibraries()` (rmgpy.data.kinetics.KineticsDatabase method), 21

`saveLibraries()` (rmgpy.data.statmech.StatmechDatabase method), 38

`saveLibraries()` (rmgpy.data.thermo.ThermoDatabase method), 53

`saveOld()` (rmgpy.data.base.Database method), 16

`saveOld()` (rmgpy.data.kinetics.KineticsDatabase method), 21

`saveOld()` (rmgpy.data.kinetics.KineticsDepository method), 23

`saveOld()` (rmgpy.data.kinetics.KineticsFamily method), 27

`saveOld()` (rmgpy.data.kinetics.KineticsGroups method), 29

`saveOld()` (rmgpy.data.kinetics.KineticsLibrary method), 31

`saveOld()` (rmgpy.data.kinetics.KineticsRules method), 33

`saveOld()` (rmgpy.data.statmech.StatmechDatabase method), 38

`saveOld()` (rmgpy.data.statmech.StatmechDepository method), 40

`saveOld()` (rmgpy.data.statmech.StatmechGroups method), 46

`saveOld()` (rmgpy.data.statmech.StatmechLibrary method), 48

`saveOld()` (rmgpy.data.thermo.ThermoDatabase method), 53

`saveOld()` (rmgpy.data.thermo.ThermoDepository method), 55

`saveOld()` (rmgpy.data.thermo.ThermoGroups method), 57

`saveOld()` (rmgpy.data.thermo.ThermoLibrary method), 59

`saveOldDictionary()` (rmgpy.data.base.Database method), 17

`saveOldDictionary()` (rmgpy.data.kinetics.KineticsDepository method), 23

`saveOldDictionary()` (rmgpy.data.kinetics.KineticsFamily method), 27

`saveOldDictionary()` (rmgpy.data.kinetics.KineticsGroups method), 29

`saveOldDictionary()` (rmgpy.data.kinetics.KineticsLibrary method), 31

`saveOldDictionary()` (rmgpy.data.kinetics.KineticsRules method), 33

`saveOldDictionary()` (rmgpy.data.statmech.StatmechDepository method), 40

- [saveOldDictionary\(\)](#) (rmgpy.data.statmech.StatmechGroups method), [46](#)
[saveOldDictionary\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), [48](#)
[saveOldDictionary\(\)](#) (rmgpy.data.thermo.ThermoDepository method), [55](#)
[saveOldDictionary\(\)](#) (rmgpy.data.thermo.ThermoGroups method), [57](#)
[saveOldDictionary\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), [59](#)
[saveOldLibrary\(\)](#) (rmgpy.data.base.Database method), [17](#)
[saveOldLibrary\(\)](#) (rmgpy.data.kinetics.KineticsDepository method), [23](#)
[saveOldLibrary\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), [27](#)
[saveOldLibrary\(\)](#) (rmgpy.data.kinetics.KineticsGroups method), [29](#)
[saveOldLibrary\(\)](#) (rmgpy.data.kinetics.KineticsLibrary method), [31](#)
[saveOldLibrary\(\)](#) (rmgpy.data.kinetics.KineticsRules method), [33](#)
[saveOldLibrary\(\)](#) (rmgpy.data.statmech.StatmechDepository method), [40](#)
[saveOldLibrary\(\)](#) (rmgpy.data.statmech.StatmechGroups method), [46](#)
[saveOldLibrary\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), [48](#)
[saveOldLibrary\(\)](#) (rmgpy.data.thermo.ThermoDepository method), [55](#)
[saveOldLibrary\(\)](#) (rmgpy.data.thermo.ThermoGroups method), [57](#)
[saveOldLibrary\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), [59](#)
[saveOldTemplate\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), [27](#)
[saveOldTree\(\)](#) (rmgpy.data.base.Database method), [17](#)
[saveOldTree\(\)](#) (rmgpy.data.kinetics.KineticsDepository method), [23](#)
[saveOldTree\(\)](#) (rmgpy.data.kinetics.KineticsFamily method), [27](#)
[saveOldTree\(\)](#) (rmgpy.data.kinetics.KineticsGroups method), [29](#)
[saveOldTree\(\)](#) (rmgpy.data.kinetics.KineticsLibrary method), [31](#)
[saveOldTree\(\)](#) (rmgpy.data.kinetics.KineticsRules method), [33](#)
[saveOldTree\(\)](#) (rmgpy.data.statmech.StatmechDepository method), [40](#)
[saveOldTree\(\)](#) (rmgpy.data.statmech.StatmechGroups method), [46](#)
[saveOldTree\(\)](#) (rmgpy.data.statmech.StatmechLibrary method), [48](#)
[saveOldTree\(\)](#) (rmgpy.data.thermo.ThermoDepository method), [55](#)
[saveOldTree\(\)](#) (rmgpy.data.thermo.ThermoGroups method), [57](#)
[saveOldTree\(\)](#) (rmgpy.data.thermo.ThermoLibrary method), [59](#)
[saveOutputHTML\(\)](#) (in module rmgpy.rmg.output), [140](#)
[saveOutputHTML\(\)](#) (rmgpy.rmg.main.RMG method), [140](#)
[saveRecommendedFamilies\(\)](#) (rmgpy.data.kinetics.KineticsDatabase method), [22](#)
[saveRestartFile\(\)](#) (rmgpy.rmg.main.RMG method), [140](#)
[saveSpeciesDictionary\(\)](#) (in module rmgpy.chemkin), [11](#)
[saveThermoData\(\)](#) (rmgpy.qm.gaussian.GaussianMol method), [116](#)
[saveThermoData\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM3 method), [118](#)
[saveThermoData\(\)](#) (rmgpy.qm.gaussian.GaussianMolPM6 method), [120](#)
[saveThermoData\(\)](#) (rmgpy.qm.molecule.QMMolecule method), [113](#)
[saveThermoData\(\)](#) (rmgpy.qm.mopac.MopacMol method), [122](#)
[saveThermoData\(\)](#) (rmgpy.qm.mopac.MopacMolPM3 method), [123](#)
[saveThermoData\(\)](#) (rmgpy.qm.mopac.MopacMolPM6 method), [125](#)
[saveThermoData\(\)](#) (rmgpy.qm.mopac.MopacMolPM7 method), [127](#)
[saveTransportFile\(\)](#) (in module rmgpy.chemkin), [12](#)
[ScalarQuantity](#) (class in rmgpy.quantity), [128](#)
[scriptAttempts](#) (rmgpy.qm.gaussian.GaussianMol attribute), [116](#)
[scriptAttempts](#) (rmgpy.qm.gaussian.GaussianMolPM3 attribute), [118](#)
[scriptAttempts](#) (rmgpy.qm.gaussian.GaussianMolPM6 attribute), [120](#)
[scriptAttempts](#) (rmgpy.qm.molecule.QMMolecule attribute), [113](#)
[scriptAttempts](#) (rmgpy.qm.mopac.MopacMol attribute), [122](#)
[scriptAttempts](#) (rmgpy.qm.mopac.MopacMolPM3 attribute), [123](#)
[scriptAttempts](#) (rmgpy.qm.mopac.MopacMolPM6 attribute), [125](#)
[scriptAttempts](#) (rmgpy.qm.mopac.MopacMolPM7 attribute), [127](#)
[selectEnergyGrains\(\)](#) (rmgpy.pdep.Network method), [105](#)
[selectEnergyGrains\(\)](#) (rmgpy.rmg.pdep.PDepNetwork method), [142](#)
[selectPolynomial\(\)](#) (rmgpy.thermo.NASA method), [173](#)
[semiclassical](#) (rmgpy.statmech.HinderedRotor attribute), [163](#)
[setConditions\(\)](#) (rmgpy.pdep.Network method), [105](#)

- setConditions() (rmgpy.rmg.pdep.PDepNetwork method), 142
 setDefaultOutputDirectory() (rmgpy.qm.main.QMCalculator method), 111
 setLonePairs() (rmgpy.molecule.Atom method), 84
 setSpinMultiplicity() (rmgpy.molecule.Atom method), 84
 setUncertaintyType() (rmgpy.quantity.ScalarQuantity method), 129
 SimpleReactor (class in rmgpy.solver), 148
 simulate() (rmgpy.solver.ReactionSystem method), 148
 simulate() (rmgpy.solver.SimpleReactor method), 149
 SingleExponentialDown (class in rmgpy.pdep), 100
 solve() (rmgpy.data.statmechfit.DirectFit method), 42
 solve() (rmgpy.data.statmechfit.PseudoFit method), 44
 solve() (rmgpy.data.statmechfit.PseudoRotorFit method), 43
 solveFullME() (rmgpy.pdep.Network method), 105
 solveFullME() (rmgpy.rmg.pdep.PDepNetwork method), 142
 solveReducedME() (rmgpy.pdep.Network method), 105
 solveReducedME() (rmgpy.rmg.pdep.PDepNetwork method), 142
 solveSchrodingerEquation() (rmgpy.statmech.HinderedRotor method), 163
 sortAtoms() (rmgpy.molecule.Group method), 95
 sortAtoms() (rmgpy.molecule.Molecule method), 90
 sortVertices() (rmgpy.molecule.graph.Graph method), 80
 sortVertices() (rmgpy.molecule.Group method), 95
 sortVertices() (rmgpy.molecule.Molecule method), 90
 Species (class in rmgpy.rmg.model), 146
 Species (class in rmgpy.species), 150
 SphericalTopRotor (class in rmgpy.statmech), 158
 spinMultiplicity (rmgpy.statmech.Conformer attribute), 166
 split() (rmgpy.molecule.graph.Graph method), 80
 split() (rmgpy.molecule.Group method), 95
 split() (rmgpy.molecule.Molecule method), 90
 StatmechDatabase (class in rmgpy.data.statmech), 38
 StatmechDepository (class in rmgpy.data.statmech), 39
 StatmechGroups (class in rmgpy.data.statmech), 44
 StatMechJob (class in rmgpy.cantherm), 9
 StatmechLibrary (class in rmgpy.data.statmech), 46
 step() (rmgpy.solver.ReactionSystem method), 148
 step() (rmgpy.solver.SimpleReactor method), 149
 stericEnergy (rmgpy.qm.qmdata.QMData attribute), 113
 succesfulJobExists() (rmgpy.qm.qmverifier.QMVerifier method), 114
 successKeys (rmgpy.qm.gaussian.Gaussian attribute), 115
 successKeys (rmgpy.qm.mopac.Mopac attribute), 120
 symmetry (rmgpy.statmech.HinderedRotor attribute), 163
 symmetry (rmgpy.statmech.KRotor attribute), 158
 symmetry (rmgpy.statmech.LinearRotor attribute), 156
 symmetry (rmgpy.statmech.NonlinearRotor attribute), 157
 symmetry (rmgpy.statmech.SphericalTopRotor attribute), 159
 SymmetryJob (class in rmgpy.qm.symmetry), 114
- ## T
- T0 (rmgpy.kinetics.Arrhenius attribute), 62
 T0 (rmgpy.pdep.SingleExponentialDown attribute), 101
 T1 (rmgpy.kinetics.Troe attribute), 73
 T2 (rmgpy.kinetics.Troe attribute), 74
 T3 (rmgpy.kinetics.Troe attribute), 74
 Tdata (rmgpy.kinetics.KineticsData attribute), 60
 Tdata (rmgpy.kinetics.PDepKineticsData attribute), 64
 Tdata (rmgpy.thermo.ThermoData attribute), 167
 TemplateReaction (class in rmgpy.data.kinetics), 48
 TerminationConversion (class in rmgpy.solver), 149
 TerminationTime (class in rmgpy.solver), 149
 ThermoData (class in rmgpy.thermo), 167
 ThermoDatabase (class in rmgpy.data.thermo), 51
 ThermoDepository (class in rmgpy.data.thermo), 53
 ThermoGroups (class in rmgpy.data.thermo), 55
 ThermoJob (class in rmgpy.cantherm), 9
 ThermoLibrary (class in rmgpy.data.thermo), 57
 ThirdBody (class in rmgpy.kinetics), 70
 Tlist (rmgpy.cantherm.KineticsJob attribute), 6
 Tlist (rmgpy.cantherm.PressureDependenceJob attribute), 8
 Tmax (rmgpy.cantherm.KineticsJob attribute), 6
 Tmax (rmgpy.cantherm.PressureDependenceJob attribute), 8
 Tmax (rmgpy.kinetics.Arrhenius attribute), 62
 Tmax (rmgpy.kinetics.Chebyshev attribute), 68
 Tmax (rmgpy.kinetics.KineticsData attribute), 60
 Tmax (rmgpy.kinetics.Lindemann attribute), 72
 Tmax (rmgpy.kinetics.MultiArrhenius attribute), 63
 Tmax (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
 Tmax (rmgpy.kinetics.PDepArrhenius attribute), 65
 Tmax (rmgpy.kinetics.PDepKineticsData attribute), 64
 Tmax (rmgpy.kinetics.ThirdBody attribute), 70
 Tmax (rmgpy.kinetics.Troe attribute), 74
 Tmax (rmgpy.thermo.NASA attribute), 172
 Tmax (rmgpy.thermo.NASAPolynomial attribute), 174
 Tmax (rmgpy.thermo.ThermoData attribute), 167
 Tmax (rmgpy.thermo.Wilhoit attribute), 170
 Tmin (rmgpy.cantherm.KineticsJob attribute), 6
 Tmin (rmgpy.cantherm.PressureDependenceJob attribute), 8
 Tmin (rmgpy.kinetics.Arrhenius attribute), 62
 Tmin (rmgpy.kinetics.Chebyshev attribute), 68
 Tmin (rmgpy.kinetics.KineticsData attribute), 60
 Tmin (rmgpy.kinetics.Lindemann attribute), 72
 Tmin (rmgpy.kinetics.MultiArrhenius attribute), 63

- Tmin (rmgpy.kinetics.MultiPDepArrhenius attribute), 67
 Tmin (rmgpy.kinetics.PDepArrhenius attribute), 65
 Tmin (rmgpy.kinetics.PDepKineticsData attribute), 64
 Tmin (rmgpy.kinetics.ThirdBody attribute), 70
 Tmin (rmgpy.kinetics.Troe attribute), 74
 Tmin (rmgpy.thermo.NASA attribute), 172
 Tmin (rmgpy.thermo.NASAPolynomial attribute), 174
 Tmin (rmgpy.thermo.ThermoData attribute), 167
 Tmin (rmgpy.thermo.Wilhoit attribute), 170
 toAdjacencyList() (in module rmgpy.molecule.adjlist), 97
 toAdjacencyList() (rmgpy.molecule.Group method), 95
 toAdjacencyList() (rmgpy.molecule.Molecule method), 90
 toAdjacencyList() (rmgpy.rmg.model.Species method), 147
 toAdjacencyList() (rmgpy.species.Species method), 151
 toArrhenius() (rmgpy.kinetics.MultiArrhenius method), 63
 toAugmentedInChI() (rmgpy.molecule.Molecule method), 90
 toAugmentedInChIKey() (rmgpy.molecule.Molecule method), 90
 toChemkin() (rmgpy.data.kinetics.DepositoryReaction method), 19
 toChemkin() (rmgpy.data.kinetics.LibraryReaction method), 36
 toChemkin() (rmgpy.data.kinetics.TemplateReaction method), 51
 toChemkin() (rmgpy.reaction.Reaction method), 134
 toChemkin() (rmgpy.rmg.pdep.PDepReaction method), 145
 toHTML() (rmgpy.kinetics.Arrhenius method), 62
 toHTML() (rmgpy.kinetics.Chebyshev method), 69
 toHTML() (rmgpy.kinetics.KineticsData method), 61
 toHTML() (rmgpy.kinetics.Lindemann method), 72
 toHTML() (rmgpy.kinetics.MultiArrhenius method), 63
 toHTML() (rmgpy.kinetics.MultiPDepArrhenius method), 68
 toHTML() (rmgpy.kinetics.PDepArrhenius method), 66
 toHTML() (rmgpy.kinetics.PDepKineticsData method), 65
 toHTML() (rmgpy.kinetics.ThirdBody method), 71
 toHTML() (rmgpy.kinetics.Troe method), 74
 toInChI() (rmgpy.molecule.Molecule method), 91
 toInChIKey() (rmgpy.molecule.Molecule method), 91
 toNASA() (rmgpy.thermo.ThermoData method), 168
 toNASA() (rmgpy.thermo.Wilhoit method), 171
 toOBMol() (rmgpy.molecule.Molecule method), 91
 toRDKitMol() (rmgpy.molecule.Molecule method), 91
 toSingleBonds() (rmgpy.molecule.Molecule method), 91
 toSMARTS() (rmgpy.molecule.Molecule method), 91
 toSMILES() (rmgpy.molecule.Molecule method), 91
 toThermoData() (rmgpy.thermo.NASA method), 173
 toThermoData() (rmgpy.thermo.Wilhoit method), 171
 toWilhoit() (rmgpy.thermo.NASA method), 173
 toWilhoit() (rmgpy.thermo.ThermoData method), 168
 TransitionState (class in rmgpy.species), 151
 Troe (class in rmgpy.kinetics), 73
- ## U
- uncertainty (rmgpy.quantity.ScalarQuantity attribute), 129
 uncertaintyType (rmgpy.quantity.ScalarQuantity attribute), 129
 uniqueID (rmgpy.qm.molecule.Geometry attribute), 111
 uniqueID (rmgpy.qm.symmetry.SymmetryJob attribute), 115
 uniqueIDlong (rmgpy.qm.molecule.Geometry attribute), 112
 unitDegeneracy() (in module rmgpy.statmech.schrodinger), 164
 update() (rmgpy.rmg.pdep.PDepNetwork method), 143
 updateAtomTypes() (rmgpy.molecule.Molecule method), 91
 updateCharge() (rmgpy.molecule.Atom method), 84
 updateConfigurations() (rmgpy.rmg.pdep.PDepNetwork method), 143
 updateConnectivityValues() (rmgpy.molecule.graph.Graph method), 81
 updateConnectivityValues() (rmgpy.molecule.Group method), 95
 updateConnectivityValues() (rmgpy.molecule.Molecule method), 91
 updateFingerprint() (rmgpy.molecule.Group method), 95
 updateLonePairs() (rmgpy.molecule.Molecule method), 91
 updateMultiplicity() (rmgpy.molecule.Molecule method), 91
 updateUnimolecularReactionNetworks() (rmgpy.rmg.model.CoreEdgeReactionModel method), 137
 usePolar (rmgpy.qm.mopac.Mopac attribute), 120
- ## V
- value (rmgpy.quantity.ScalarQuantity attribute), 129
 verifyOutputFile() (rmgpy.qm.gaussian.Gaussian method), 115
 verifyOutputFile() (rmgpy.qm.gaussian.GaussianMol method), 116
 verifyOutputFile() (rmgpy.qm.gaussian.GaussianMolPM3 method), 118
 verifyOutputFile() (rmgpy.qm.gaussian.GaussianMolPM6 method), 120
 verifyOutputFile() (rmgpy.qm.mopac.Mopac method), 120
 verifyOutputFile() (rmgpy.qm.mopac.MopacMol method), 122

`verifyOutputFile()` (rmgpy.qm.mopac.MopacMolPM3 method), [123](#)
`verifyOutputFile()` (rmgpy.qm.mopac.MopacMolPM6 method), [125](#)
`verifyOutputFile()` (rmgpy.qm.mopac.MopacMolPM7 method), [127](#)
`Vertex` (class in rmgpy.molecule.graph), [78](#)
`VF2` (class in rmgpy.molecule.vf2), [81](#)
`visit()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_Call()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_Dict()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_List()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_Num()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_Str()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)
`visit_Tuple()` (rmgpy.cantherm.output.PrettifyVisitor method), [7](#)

W

`Wigner` (class in rmgpy.kinetics), [75](#)
`Wilhoit` (class in rmgpy.thermo), [168](#)
`writeInputFile()` (rmgpy.qm.gaussian.GaussianMol method), [117](#)
`writeInputFile()` (rmgpy.qm.gaussian.GaussianMolPM3 method), [118](#)
`writeInputFile()` (rmgpy.qm.gaussian.GaussianMolPM6 method), [120](#)
`writeInputFile()` (rmgpy.qm.mopac.MopacMol method), [122](#)
`writeInputFile()` (rmgpy.qm.mopac.MopacMolPM3 method), [124](#)
`writeInputFile()` (rmgpy.qm.mopac.MopacMolPM6 method), [125](#)
`writeInputFile()` (rmgpy.qm.mopac.MopacMolPM7 method), [127](#)
`writeInputFile()` (rmgpy.qm.symmetry.SymmetryJob method), [115](#)
`writeKineticsEntry()` (in module rmgpy.chemkin), [12](#)
`writeThermoEntry()` (in module rmgpy.chemkin), [12](#)