RMG-Py API Reference

Release 2.3.0

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

Dec 20, 2018

CONTENTS

1 RMG API Reference **3**
1.1 Arkane (`arkane`) 3
1.2 Chemkin files (`rmgpy.chemkin`) 11
1.3 Physical constants (`rmgpy.constants`) 14
1.4 Database (`rmgpy.data`) 14
1.5 Kinetics (`rmgpy.kinetics`) 76
1.6 Molecular representations (`rmgpy.molecule`) 97
1.7 Pressure dependence (`rmgpy.pdep`) 147
1.8 QMTP (`rmgpy.qm`) 156
1.9 Physical quantities (`rmgpy.quantity`) 174
1.10 Reactions (`rmgpy.reaction`) 179
1.11 Reaction mechanism generation (`rmgpy.rmg`) 185
1.12 Reaction system simulation (`rmgpy.solver`) 204
1.13 Species (`rmgpy.species`) 224
1.14 Statistical mechanics (`rmgpy.statmech`) 229
1.15 Thermodynamics (`rmgpy.thermo`) 245
1.16 RMG Exceptions (`rmgpy.exceptions`) 255

Bibliography **259**

Python Module Index **261**

Index **263**

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/

CHAPTER

ONE

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

1.1 Arkane (`arkane`)

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

1.1.1 Reading Gaussian log files

Class	Description
`GaussianLog`	Extract chemical parameters from Gaussian log files

1.1.2 Reading Q-Chem log files

Class	Description
`QchemLog`	Extract chemical parameters from Q-Chem log files

1.1.3 Reading Molpro log files

Class	Description
`MolproLog`	Extract chemical parameters from Molpro log files

1.1.4 Input

Function	Description
`loadInputFile()`	Load an Arkane job input file

1.1.5 Job classes

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

arkane.gaussian.GaussianLog

class arkane.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 Arkane 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=0*, *opticalIsomers=1*, *symfromlog=None*, *label=''*)

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

loadEnergy(*frequencyScaleFactor=1.0*)

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

loadForceConstantMatrix()

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

loadGeometry()

Return the optimum geometry of the molecular configuration from the Gaussian log file. If multiple such geometries are identified, only the last is returned.

loadNegativeFrequency()

Return the negative frequency from a transition state frequency calculation in cm^-1.

loadScanEnergies()

Extract the optimized energies in J/mol from a log file, e.g. the result of a Gaussian "Scan" quantum chemistry calculation.

loadZeroPointEnergy()

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

arkane.qchem.QchemLog

class arkane.qchem.**QChemLog**(*path*)

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

getNumberOfAtoms()

Return the number of atoms in the molecular configuration used in the QChem output file.

loadConformer(*symmetry=None*, *spinMultiplicity=0*, *opticalIsomers=1*, *symfromlog=None*, *label="*)

Load the molecular degree of freedom data from an output file created as the result of a QChem "Freq" calculation. As QChem'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 QChem output file will be adopted.

loadEnergy(*frequencyScaleFactor=1.0*)

Load the energy in J/mol from a QChem log file. Only the last energy in the file is returned. The zero-point energy is *not* included in the returned value.

loadForceConstantMatrix()

Return the force constant matrix (in Cartesian coordinates) from the QChem log file. If multiple such matrices are identified, only the last is returned. The units of the returned force constants are J/m^2. If no force constant matrix can be found in the log file, `None` is returned.

loadGeometry()

Return the optimum geometry of the molecular configuration from the QChem log file. If multiple such geometries are identified, only the last is returned.

loadNegativeFrequency()

Return the imaginary frequency from a transition state frequency calculation in cm^-1.

loadScanEnergies()

Extract the optimized energies in J/mol from a QChem log file, e.g. the result of a QChem "PES Scan" quantum chemistry calculation.

loadZeroPointEnergy(*frequencyScaleFactor=1.0*)

Load the unscaled zero-point energy in J/mol from a QChem output file.

arkane.molpro.MolproLog

class `arkane.molpro.`**`MolproLog`**(*path*)

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

`getNumberOfAtoms`()

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

`loadConformer`(*symmetry=None*, *spinMultiplicity=0*, *opticalIsomers=1*, *symfromlog=None*, *label=''*)

Load the molecular degree of freedom data from a log file created as the result of a MolPro "Freq" quantum chemistry calculation with the thermo printed.

`loadEnergy`(*frequencyScaleFactor=1.0*)

Return either the f12 or MRCI energy in J/mol from a Molpro Logfile. If the MRCI job outputted the MRCI+Davidson energy, the latter is returned. For CCSD(T)-f12, the function determines which energy (f12a or f12b) to use based on the basis set, which it will parse out of the Molpro file. For the vdz and vtz basis sets f12a is a better approximation, but for higher basis sets f12b is a better approximation.

`loadForceConstantMatrix`()

Print the force constant matrix by including the print, hessian command in the input file

`loadGeometry`()

Return the optimum geometry of the molecular configuration from the Molpro .out 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.

`loadZeroPointEnergy`()

Load the unscaled zero-point energy in J/mol from a MolPro log file.

Arkane input files

`arkane.input.`**`loadInputFile`**(*path*)

Load the Arkane input file located at *path* on disk, and return a list of the jobs defined in that file.

arkane.KineticsJob

class `arkane.`**`KineticsJob`**(*reaction*, *Tmin=None*, *Tmax=None*, *Tlist=None*, *Tcount=0*, *sensitivity_conditions=None*)

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

***usedTST* - a boolean representing if TST was used to calculate the kinetics** if kinetics is already given in the input, then it is False.

`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.

`draw`(*outputDirectory*, *format='pdf'*)

Generate a PDF drawing of the reaction. This requires that Cairo and its Python wrapper be available; if not, the drawing is not generated.

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

`execute``(`*outputFile=None*, *plot=True*`)`
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.

arkane.Arkane

`class arkane.`**`Arkane`**`(`*inputFile=None*, *outputDirectory=None*, *verbose=20*`)`
The `Arkane` class represents an instance of Arkane, a tool for computing properties of chemical species and reactions. The attributes are:

Attribute	Description
jobList	A list of the jobs to execute
inputFile	The path of the input file defining the jobs to execute
outputDirectory	The directory in which to write the output files
verbose	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 Arkane 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 Arkane to the log.

`parseCommandLineArguments``()`
Parse the command-line arguments being passed to Arkane. This uses the `argparse` module, which ensures that the command-line arguments are sensible, parses them, and returns them.

Saving Arkane output

`arkane.output.`**`prettify`**`(`*string*, *indent=4*`)`

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

`class` `arkane.output.`**`PrettifyVisitor`**`(`*level=0*, *indent=4*`)`

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

`generic_visit``(`*node*`)`

Called if no explicit visitor function exists for a node.

`visit``(`*node*`)`

Visit a node.

`visit_Call``(`*node*`)`

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

`visit_Dict``(`*node*`)`

Return a pretty representation of the dict represented by *node*.

`visit_List``(`*node*`)`

Return a pretty representation of the list represented by *node*.

`visit_Num``(`*node*`)`

Return a pretty representation of the number represented by *node*.

`visit_Str``(`*node*`)`

Return a pretty representation of the string represented by *node*.

`visit_Tuple``(`*node*`)`

Return a pretty representation of the tuple represented by *node*.

arkane.PressureDependenceJob

`class` `arkane.`**`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*, *sensitivity_conditions=None*`)`

A representation of a pressure dependence job. The attributes are:

Attribute	Description
Tmin	The minimum temperature at which to compute $k(T, P)$ values
Tmax	The maximum temperature at which to compute $k(T, P)$ values
Tcount	The number of temperatures at which to compute $k(T, P)$ values
Pmin	The minimum pressure at which to compute $k(T, P)$ values
Pmax	The maximum pressure at which to compute $k(T, P)$ values
Pcount	The number of pressures at which to compute $k(T, P)$ values
Emin	The minimum energy to use to compute $k(T, P)$ values
Emax	The maximum energy to use to compute $k(T, P)$ values
maximumGrainSize	The maximum energy grain size to use to compute $k(T, P)$ values
minimumGrainCount	The minimum number of energy grains to use to compute $k(T, P)$ values
method	The method to use to reduce the master equation to $k(T, P)$ values
interpolationModel	The interpolation model to fit to the computed $k(T, P)$ values
maximumAtoms	The maximum number of atoms to apply pressure dependence to (in RMG jobs)
activeKRotor	A flag indicating whether to treat the K-rotor as active or adiabatic
activeJRotor	A flag indicating whether to treat the J-rotor as active or adiabatic
rmgmode	A flag that toggles “RMG mode”, described below
network	The unimolecular reaction network
Tlist	An array of temperatures at which to compute $k(T, P)$ values
Plist	An array of pressures at which to compute $k(T, P)$ values
Elist	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 -> products, where A is always an explored isomer; the high-P kinetics are reversed if necessary for this purpose
- Thermodynamic parameters are always used to compute the reverse k(E) from the forward k(E) for each path reaction

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

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

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

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

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

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

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

`copy()`
Return a copy of the pressure dependence job.

draw(*outputDirectory*, *format='pdf'*)

Generate a PDF drawing of the pressure-dependent reaction network. This requires that Cairo and its Python wrapper be available; if not, the drawing is not generated.

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

generatePressureList()

Returns an array of pressures based on the interpolation *model*, minimum and maximum pressures *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 an Arkane input file for the pressure dependence job to *path* on disk.

arkane.StatMechJob

class arkane.**StatMechJob**(*species*, *path*)

A representation of a Arkane 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.

arkane.ThermoJob

class arkane.**ThermoJob**(*species*, *thermoClass*)

A representation of an Arkane 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, enthapy, 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
`loadChemkinFile()`	Load a reaction mechanism from a Chemkin file
`loadSpeciesDictionary()`	Load a species dictionary from a file
`loadTransportFile()`	Load a Chemkin transport properties file
`readKineticsEntry()`	Read a single reaction entry from a Chemkin file
`readReactionComments()`	Read the comments associated with a reaction entry
`readReactionsBlock()`	Read the reactions block of a Chemkin file
`readThermoEntry()`	Read a single thermodynamics entry from a Chemkin file
`removeCommentFromLine()`	Remove comment text from a line of a Chemkin file or species dictionary

1.2.2 Writing Chemkin files

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

Reading Chemkin files

Main functions

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

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*, *oldStyle=False*)

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

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

`rmgpy.chemkin.`**`saveTransportFile`**(*path*, *species*)

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

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

1. An index indicating whether the molecule has a monatomic, linear or nonlinear geometrical configuration. If the index is 0, the molecule is a single atom. If the index is 1 the molecule is linear, and if it is 2, the molecule is nonlinear.
2. The Lennard-Jones potential well depth 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 $10^{-18}cm^{3/2}erg^{1/2}$.
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, with species.txt containing the RMG-Java formatted dictionary.

Helper functions

`rmgpy.chemkin.`**`getSpeciesIdentifier`**(*species*)

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

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

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

`rmgpy.chemkin.`**`writeThermoEntry`**(*species*, *elementCounts=None*, *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: Physical constants defined in the `rmgpy.constants` module

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

1.4 Database (`rmgpy.data`)

1.4.1 General classes

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

1.4.2 Thermodynamics database

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

1.4.3 Kinetics database

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

1.4.4 Statistical mechanics database

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

1.4.5 Statistical mechanics fitting

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

rmgpy.data.base.Database

class `rmgpy.data.base.`**`Database`**(*entries=None*, *top=None*, *label=”*, *name=”*, *solvent=None*, *shortDesc=”*, *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.

`areSiblings`(*node*, *nodeOther*)

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

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

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

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

Returns None if there is no matching root.

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

`descendants`(*node*)

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

generateOldTree(*entries*, *level*)

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

getEntriesToSave()

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)

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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

`removeGroup`(*groupToRemove*)

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

Returns the removed group

`save`(*path*)

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

`saveDictionary`(*path*)

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

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

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

`saveOldDictionary`(*path*)

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

`saveOldLibrary`(*path*)

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

`saveOldTree`(*path*)

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

rmgpy.data.kinetics.DepositoryReaction

class `rmgpy.data.kinetics.`**`DepositoryReaction`**(*index=-1*, *reactants=None*, *products=None*, *specificCollider=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.

`allow_max_rate_violation`

'bool'

Type allow_max_rate_violation

`allow_pdep_route`

'bool'

Type allow_pdep_route

calculateMicrocanonicalRateCoefficient(*self*, *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.

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

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

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

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.

calculateTSTRateCoefficients(*self*, *ndarray Tlist*) → ndarray

calculate_coll_limit(*self*, *float temp*, *bool reverse=False*)

Calculate the collision limit rate for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

canTST(*self*) → bool

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.

check_collision_limit_violation(*self*, *float t_min*, *float t_max*, *float p_min*, *float p_max*) → list

Warn if a core reaction violates the collision limit rate in either the forward or reverse direction at the relevant extreme T/P conditions. Assuming a monotonic behaviour of the kinetics. Returns a list with the reaction object and the direction in which the violation was detected.

comment

str

Type comment

copy(*self*)

Create a deep copy of the current reaction.

degeneracy

Reaction.__getDegneneracy(self)

draw(*self*, *path*)

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.

duplicate
'bool'

Type duplicate

elementary_high_p
'bool'

Type elementary_high_p

ensure_species(*self*, *bool reactant_resonance=False*, *bool product_resonance=True*)
Ensure the reaction contains species objects in its reactant and product attributes. If the reaction is found to hold molecule objects, it modifies the reactant, product and pairs to hold Species objects.

Generates resonance structures for Molecules if the corresponding options, reactant_resonance and/or product_resonance, are True. Does not generate resonance for reactants or products that start as Species objects.

fixBarrierHeight(*self*, *bool forcePositive=False*)
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(*self*, *T*)
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

generate3dTS(*self*, *reactants*, *products*)
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(*self*)
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 (C/O/N/S at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

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

getEnthalpiesOfReaction(*self*, *ndarray Tlist*) → ndarray
Return the enthalpies of reaction in J/mol evaluated at temperatures *Tlist* in K.

getEnthalpyOfReaction(*self*, *double T*) → double
Return the enthalpy of reaction in J/mol evaluated at temperature *T* in K.

getEntropiesOfReaction(*self*, *ndarray Tlist*) → ndarray
Return the entropies of reaction in J/mol*K evaluated at temperatures *Tlist* in K.

getEntropyOfReaction(*self*, *double T*) → double
Return the entropy of reaction in J/mol*K evaluated at temperature *T* in K.

getEquilibriumConstant(*self*, *double T*, *str type='Kc'*) → double
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(*self*, *ndarray Tlist*, *str type='Kc'*) → ndarray
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(*self*, *ndarray Tlist*) → ndarray
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

getRateCoefficient(*self*, *double T*, *double P=0*) → double
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(*self*, *Species spec*) → int
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(*self*)
Get a URL to search for this reaction in the rmg website.

get_mean_sigma_and_epsilon(*self*, *bool reverse=False*)
Calculates the collision diameter (sigma) using an arithmetic mean Calculates the well depth (epsilon) using a geometric mean If reverse is `False` the above is calculated for the reactants, otherwise for the products

get_reduced_mass(*self*, *bool reverse=False*)
Returns the reduced mass of the reactants if reverse is `False` Returns the reduced mass of the products if reverse is `True`

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

index
'int'

Type index

isAssociation(*self*) → bool
Return `True` if the reaction represents an association reaction $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or `False` if not.

isBalanced(*self*) → bool

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

isDissociation(*self*) → bool

Return `True` if the reaction represents a dissociation reaction $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or `False` if not.

isIsomerization(*self*) → bool

Return `True` if the reaction represents an isomerization reaction $\mathrm{A} \rightleftharpoons \mathrm{B}$ or `False` if not.

isIsomorphic(*self*, *Reaction other*, *bool eitherDirection=True*, *bool checkIdentical=False*, *bool checkOnlyLabel=False*, *bool checkTemplateRxnProducts=False*) → bool

Return `True` if this reaction is the same as the *other* reaction, or `False` if they are different. The comparison involves comparing isomorphism of reactants and products, and doesn't use any kinetic information.

If *eitherDirection=False* then the directions must match.

***checkIdentical* indicates that atom ID's must match and is used in** checking degeneracy

***checkOnlyLabel* indicates that the string representation will be** checked, ignoring the molecular structure comparisons

***checkTemplateRxnProducts* indicates that only the products of the** reaction are checked for isomorphism. This is used when we know the reactants are identical, i.e. in generating reactions.

isUnimolecular(*self*) → bool

Return `True` if the reaction has a single molecule as either reactant or product (or both) $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or $\mathrm{A} \rightleftharpoons \mathrm{B}$, or `False` if not.

is_forward

'bool'

Type is_forward

k_effective_cache

dict

Type k_effective_cache

kinetics

rmgpy.kinetics.model.KineticsModel

Type kinetics

label

str

Type label

matchesSpecies(*self*, *list reactants*, *list products=None*) → bool

Compares the provided reactants and products against the reactants and products of this reaction. Both directions are checked.

Parameters

- **reactants** (*list*) – Species required on one side of the reaction
- **products** (*list, optional*) – Species required on the other side

network_kinetics

rmgpy.kinetics.arrhenius.Arrhenius

Type network_kinetics

pairs

list

Type pairs

`products`
list

Type products

`reactants`
list

Type reactants

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

`reversible`
'bool'

Type reversible

`specificCollider`
rmgpy.species.Species

Type specificCollider

`toCantera`(*self*, *speciesList=None*, *useChemkinIdentifier=False*)
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

`toChemkin`(*self*, *speciesList=None*, *kinetics=True*)
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.

`toLabeledStr`(*self*, *use_index=False*)
the same as __str__ except that the labels are assumed to exist and used for reactant and products rather than the labels plus the index in parentheses

`transitionState`
rmgpy.species.TransitionState

Type transitionState

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*, *nodalDistance=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
index	A unique nonnegative integer index for the entry
label	A unique string identifier for the entry (or ‘’ if not used)
item	The item that this entry represents
parent	The parent of the entry in the hierarchy (or `None` if not used)
children	A list of the children of the entry in the hierarchy (or `None` if not used)
data	The data to associate with the item
reference	A `Reference` object containing bibliographic reference information to the source of the data
referenceType	The way the data was determined: `'theoretical'`, `'experimental'`, or `'review'`
shortDesc	A brief (one-line) description of the data
longDesc	A long, verbose description of the data
rank	An integer indicating the degree of confidence in the entry data, or `None` if not used
nodalDistance	A float representing the distance of a given entry from it’s parent entry

rmgpy.data.statmech.GroupFrequencies

class `rmgpy.data.statmech.`**GroupFrequencies**(*frequencies=None*, *symmetry=1*)

Represent a set of characteristic frequencies for a group in the frequency database. These frequencies are stored in the *frequencies* attribute, which is a `list` of `tuples`, where each `tuple` defines a lower bound, upper bound, and degeneracy. Each group also has a *symmetry* correction.

generateFrequencies(*count=1*)

Generate a set of frequencies. For each characteristic frequency group, the number of frequencies returned is degeneracy * count, and these are distributed linearly between the lower and upper bounds.

rmgpy.data.kinetics.KineticsDatabase

class `rmgpy.data.kinetics.`**KineticsDatabase**

A class for working with the RMG kinetics database.

extractSourceFromComments(*reaction*)

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

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

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

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

generate_reactions(*reactants*, *products=None*, *only_families=None*, *resonance=True*)

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.

generate_reactions_from_families(*reactants*, *products=None*, *only_families=None*, *resonance=True*)

Generate all reactions between the provided list or tuple of one or two *reactants*, which can be either

`Molecule` objects or `Species` objects. This method can apply all kinetics families or a selected subset.

Parameters

- **`reactants`** – Molecules or Species to react
- **`products`** – List of Molecules or Species of desired product structures (optional)
- **`only_families`** – List of family labels to generate reactions from (optional) Default is to generate reactions from all families
- **`resonance`** – Flag to generate resonance structures for reactants and products (optional) Default is True, resonance structures will be generated

Returns List of reactions containing Species objects with the specified reactants and products.

`generate_reactions_from_libraries`(*reactants*, *products=None*)

Find all reactions from all loaded kinetics library involving the provided *reactants*, which can be either `Molecule` objects or `Species` objects.

`generate_reactions_from_library`(*library*, *reactants*, *products=None*)

Find all reactions from the specified kinetics library involving the provided *reactants*, which can be either `Molecule` objects or `Species` objects.

`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.

The *families* argument accepts a single item or list of the following:

- Specific kinetics family labels
- Names of family sets defined in recommended.py
- 'all'
- 'none'

If all items begin with a *!* (e.g. ['!H_Abstraction']), then the selection will be inverted to families NOT in the list.

`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 recommended families from the given file. The file is usually stored at 'kinetics/families/recommended.py'.

The old style was as a dictionary named *recommendedFamilies* containing all family names as keys with True/False values.

The new style is as multiple sets with unique names which can be used individually or in combination.

Both styles can be loaded by this method.

react_molecules(*molecules*, *products=None*, *only_families=None*, *prod_resonance=True*)
Generate reactions from all families for the input molecules.

reconstructKineticsFromSource(*reaction*, *source*, *fixBarrierHeight=False*, *forcePositiveBarrier=False*)
Reaction is the original reaction with original kinetics. Note that for Library and PDep reactions this function does not do anything other than return the original kinetics…

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

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

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

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

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

saveRecommendedFamilies(*path*)
Save the recommended families to [path]/recommended.py. The old style was as a dictionary named *recommendedFamilies*. The new style is as multiple sets with different labels.

rmgpy.data.kinetics.KineticsDepository

class rmgpy.data.kinetics.**KineticsDepository**(*label=''*, *name=''*, *shortDesc=''*, *longDesc=''*)
A class for working with an RMG kinetics depository. Each depository corresponds to a reaction family (a `KineticsFamily` object). Each entry in a kinetics depository involves a reaction defined either by a real reactant and product species (as in a kinetics library).

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

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

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

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

Returns None if there is no matching root.

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

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)
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.

Attribute	Description
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

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

Returns the removed group

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

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

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

saveOld(*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="*, *reversible=True*, *shortDesc="*, *longDesc="*, *forwardTemplate=None*, *forwardRecipe=None*, *reverseTemplate=None*, *reverseRecipe=None*, *forbidden=None*, *boundaryAtoms=None*, *treeDistances=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
reverse	`string`	The name of the reverse reaction family
reversible	*Boolean*	Is family reversible? (True by default)
forwardTemplate	`Reaction`	The forward reaction template
forwardRecipe	`ReactionRecipe`	The steps to take when applying the forward reaction to a set of reactants
reverseTemplate	`Reaction`	The reverse reaction template
reverseRecipe	`ReactionRecipe`	The steps to take when applying the reverse reaction to a set of reactants
forbidden	`ForbiddenStructures`	(Optional) Forbidden product structures in either direction
ownReverse	*Boolean*	It's its own reverse?
'boundaryAtoms'	list	Labels which define the boundaries of end groups in backbone/end families
treeDistances	dict	The default distance from parent along each tree, if not set default is 1 for every tree
groups	`KineticsGroups`	The set of kinetics group additivity values
rules	`KineticsRules`	The set of kinetics rate rules from RMG-Java
depositories	`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`.

addAtomLabelsForReaction(*reaction*, *output_with_resonance=True*)

Apply atom labels on a reaction using the appropriate atom labels from this reaction family.

The reaction is modified in place containing species objects with the atoms labeled. If output_with_resonance is True, all resonance structures are generated with labels. If false, only the first resonance structure sucessfully able to map to the reaction is used. None is returned.

addEntry(*parent*, *grp*, *name*)

Adds a group entry with parent parent group structure grp and group name name

addKineticsRulesFromTrainingSet(*thermoDatabase=None*)

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

addReverseAttribute(*rxn*, *react_non_reactive=True*)

For rxn (with species' objects) from families with ownReverse, this method adds a *reverse* attribute that contains the reverse reaction information (like degeneracy)

Returns *True* if successful and *False* if the reverse reaction is forbidden. Will raise a *KineticsError* if unsuccessful for other reasons.

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.

areSiblings(*node*, *nodeOther*)

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

calculateDegeneracy(*reaction*)

For a *reaction* with *Molecule* or *Species* objects given in the direction in which the kinetics are defined, compute the reaction-path degeneracy.

This method by default adjusts for double counting of identical reactants. This should only be adjusted once per reaction. To not adjust for identical reactants (since you will be reducing them later in the algorithm), add *ignoreSameReactants= True* to this method.

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.

distributeTreeDistances()

fills in nodalDistance (the distance between an entry and its parent) if not already entered with the value from treeDistances associated with the tree the entry comes from

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

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

Returns just the kinetics, or None.

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

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

Returns a tuple (kinetics, entry) where *entry* is the database entry used to determine the kinetics only if it is an exact match, and is None if some averaging or use of a parent node took place.

evalExt(*parent*, *ext*, *extname*, *obj=None*, *T=1000.0*)

evaluates the objective function obj for the extension ext with name extname to the parent entry parent

extendNode(*parent*, *thermoDatabase=None*, *obj=None*, *T=1000.0*)

Constructs an extension to the group parent based on evaluation of the objective function obj

extendRegularization(*node*, *inds*, *regs*, *typ*)

Applies a regularization down the tree from a given parent node

extractSourceFromComments(*reaction*)

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

For a training reaction, the Source_Data returns:

```
[Family_Label, Training_Reaction_Entry, Kinetics_In_Reverse?]
```

For a reaction from rate rules, the Source_Data is a tuple containing:

```
[Family_Label, {'template': originalTemplate,
                'degeneracy': degeneracy,
                'exact': boolean_exact?,
```

(continues on next page)

(continued from previous page)

```
                'rules': a list of (original rate rule entry, weight in average)
                'training': a list of (original rate rule entry associated with␣
↪training entry, original training entry, weight in average)}]
```

where Exact is a boolean of whether the rate is an exact match, Template is the reaction template used, RateRules is a list of the rate rule entries containing the kinetics used, and TrainingReactions are ones that have created rules used in the estimate.

fillKineticsRulesByAveragingUp(*verbose=False*)
Fill in gaps in the kinetics rate rules by averaging child nodes recursively starting from the top level root template.

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

generateProductTemplate(*reactants0*)
Generate the product structures by applying the reaction template to the top-level nodes. For reactants defined by multiple structures, only the first is used here; it is assumed to be the most generic.

generateReactions(*reactants*, *products=None*, *prod_resonance=True*)
Generate all reactions between the provided list of one, two, or three *reactants*, which should be either single `Molecule` objects or lists of same. Does not estimate the kinetics of these reactions at this time. Returns a list of *`TemplateReaction`* objects using `Molecule` objects for both reactants and products The reactions are constructed such that the forward direction is consistent with the template of this reaction family.

Parameters

- **reactants** (*list*) – List of Molecules to react.
- **products** (*list, optional*) – List of Molecules or Species of desired product structures.
- **prod_resonance** (*bool, optional*) – Flag to generate resonance structures for product checking. Defaults to True, resonance structures are compared.

Returns List of all reactions containing Molecule objects with the specified reactants and products within this family. Degenerate reactions are returned as separate reactions.

generateTree(*obj=None*, *thermoDatabase=None*, *T=1000.0*)
Generate a tree by greedy optimization based on the objective function obj the optimization is done by iterating through every group and if the group has more than one training reaction associated with it a set of potential more specific extensions are generated and the extension that optimizing the objective function combination is chosen and the iteration starts over at the beginning

additionally the tree structure is simplified on the fly by removing groups that have no kinetics data associated if their parent has no kinetics data associated and they either have only one child or have two children one of which has no kinetics data and no children (its parent becomes the parent of its only relevant child node)

getBackboneRoots()
Returns: the top level backbone node in a unimolecular family.

getEndRoots()
Returns: A list of top level end nodes in a unimolecular family

getEntriesReactions(*template*)
retrieves all training reactions whose kinetics are associated with the entry template

getEntriesToSave()

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getExtensionEdge(*parent*, *obj*, *T*)

finds the set of all extension groups to parent such that 1) the extension group divides the set of reactions under parent 2) No generalization of the extension group divides the set of reactions under parent

We find this by generating all possible extensions of the initial group. Extensions that split reactions are added to the list. All extensions that do not split reactions and do not create bonds are ignored (although those that match every reaction are labeled so we don't search them twice). Those that match all reactions and involve bond creation undergo this process again.

Principle: Say you have two elementary changes to a group ext1 and ext2 if applying ext1 and ext2 results in a split at least one of ext1 and ext2 must result in a split

Speed of this algorithm relies heavily on searching non bond creation dimensions once.

getKinetics(*reaction*, *templateLabels*, *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 of

1. the kinetics
2. the source - this will be *None* if from a template estimate
3. the entry - this will be *None* if from a template estimate
4. is_forward a boolean denoting whether the matched entry is in the same direction as the inputted reaction. This will always be True if using rates rules or group additivity. This can be *True* or *False* if using a depository

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 possible RMG-Py behavior) and 'rate rules' (old RMG-Java behavior, and default RMG-Py behavior).

Returns a tuple (kinetics, entry): If it's estimated via 'rate rules' and an exact match is found in the tree, then the entry is returned as the second element of the tuple. But if an average is used, or the 'group additivity' method, then the tuple returned is (kinetics, None).

getKineticsFromDepository(*depository*, *reaction*, *template*, *degeneracy*)

Search the given *depository* in this kinetics family for kinetics for the given *reaction*. Returns a list of all of the matching kinetics, the corresponding entries, and `True` if the kinetics match the forward direction or `False` if they match the reverse direction.

getLabeledReactantsAndProducts(*reactants*, *products*)

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

getRateRule(*template*)

Return the rate rule with the given *template*. Raises a `ValueError` if no corresponding entry exists.

getReactionPairs(*reaction*)

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

`getReactionTemplate`(*reaction*)

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

`getReactionTemplateLabels`(*reaction*)

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

`getRootTemplate`()

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

`getSourcesForTemplate`(*template*)

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

Returns a tuple of rules, training

where rules are a list of tuples containing the [(original_entry, weight_used_in_average), …]

and training is a list of tuples containing the [(rate_rule_entry, training_reaction_entry, weight_used_in_average),…]

`getSpecies`(*path*, *resonance=True*)

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

`getTemplateKinetics`(*template*)

retrives a list of all the kinetics objects associated with a given template

`getTopLevelGroups`(*root*)

Returns a list of group nodes that are the highest in the tree starting at node "root". If "root" is a group node, then it will return a single-element list with "root". Otherwise, for every child of root, we descend until we find no nodes with logic nodes. We then return a list of all group nodes found along the way.

`getTrainingDepository`()

Returns the *training* depository from self.depositories

`hasRateRule`(*template*)

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

`isMoleculeForbidden`(*molecule*)

Return `True` if the molecule is forbidden in this family, or `False` otherwise.

`load`(*path*, *local_context=None*, *global_context=None*, *depositoryLabels=None*)

Load a kinetics database from a file located at *path* on disk.

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

If depositoryLabels is None then load 'training' first then everything else. If depositoryLabels is not None then load in the order specified in depositoryLabels.

`loadForbidden`(*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.

Attribute	Description
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

prepareTreeForGeneration(*thermoDatabase=None*)
clears groups and rules in the tree, generates an appropriate root group to start from and then reads training reactions Note this only works if a single top node (not a logic node) can be generated

regularize(*regularization=<function simpleRegularization>*)
Regularizes the tree according to the regularization function regularization

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

Returns the removed group

`retrieveOriginalEntry`(*templateLabel*)
Retrieves the original entry, be it a rule or training reaction, given the template label in the form 'group1;group2' or 'group1;group2;group3'

Returns tuple in the form (RateRuleEntry, TrainingReactionEntry)

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

`retrieveTemplate`(*templateLabels*)
Reconstruct the groups associated with the labels of the reaction template and return a list.

`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*.

`saveGeneratedTree`(*path=None*)
clears the rules and saves the family to its current location in database

`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.

`saveTrainingReactions`(*reactions*, *reference=None*, *referenceType="*, *shortDesc="*, *longDesc="*, *rank=3*)
This function takes a list of reactions appends it to the training reactions file. It ignores the existence of duplicate reactions.

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

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

`simpleRegularization`(*node*)
Simplest regularization algorithm All nodes are made as specific as their descendant reactions Training reactions are assumed to not generalize For example if an particular atom at a node is Oxygen for all of its descendent reactions a reaction where it is Sulfur will never hit that node unless it is the top node even if the tree did not split on the identity of that atom

splitReactions(*rxns*, *oldlabel*, *newgrp*)
divides the reactions in rxns between the new group structure newgrp and the old structure with label oldlabel returns a list of reactions associated with the new group the list of reactions associated with the old group and a list of the indices of all of the reactions associated with the new group

rmgpy.data.kinetics.KineticsGroups

class rmgpy.data.kinetics.**KineticsGroups**(*entries=None*, *top=None*, *label=”*, *name=”*, *shortDesc=”*, *longDesc=”*, *forwardTemplate=None*, *forwardRecipe=None*, *reverseTemplate=None*, *reverseRecipe=None*, *forbidden=None*)
A class for working with an RMG kinetics family group additivity values.

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

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

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

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

Returns None if there is no matching root.

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

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

estimateKineticsUsingGroupAdditivity(*template*, *referenceKinetics*, *degeneracy=1*)
Determine the appropriate kinetics for a reaction with the given *template* using group additivity.

Returns just the kinetics.

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.

Then renumber the entry indexes so that we never have any duplicate indexes.

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*, *resonance=True*)
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.

loadEntry(*index*, *label*, *group*, *kinetics*, *reference=None*, *referenceType="*, *shortDesc="*, *longDesc="*, *nodalDistance=None*)

nodalDistance is the distance between a given entry and its parent specified by a float

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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

save(*path*)

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

saveDictionary(*path*)

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

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

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

saveOldDictionary(*path*)

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

saveOldLibrary(*path*)

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

saveOldTree(*path*)

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

rmgpy.data.kinetics.KineticsLibrary

class rmgpy.data.kinetics.**KineticsLibrary**(*label="*, *name="*, *solvent=None*, *shortDesc="*, *longDesc="*, *autoGenerated=False*)

A class for working with an RMG kinetics library.

ancestors(*node*)

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

areSiblings(*node*, *nodeOther*)

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

checkForDuplicates(*markDuplicates=False*)

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

convertDuplicatesToMulti()

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

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

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

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

Returns None if there is no matching root.

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

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getLibraryReactions()
makes library and template reactions as appropriate from the library comments and returns at list of all of these LibraryReaction and TemplateReaction objects

getSpecies(*path*, *resonance=True*)
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.

Attribute	Description
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

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

Returns the removed group

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

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

saveEntry(*f*, *entry*)
Write the given *entry* in the kinetics 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.

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

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

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

Returns None if there is no matching root.

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

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

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

Returns a tuple (kinetics, entry) where *entry* is the database entry used to determine the kinetics only if it is an exact match, and is None if some averaging or use of a parent node took place.

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

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

getAllRules(*template*)
Return all of the exact rate rules with the given *template*. Raises a `ValueError` if no corresponding entry exists.

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

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

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

getSpecies(*path*, *resonance=True*)
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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

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

Returns the removed group

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

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

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

saveOld(*path*, *groups*)
Save a set of old rate rules for kinetics groups from this depository.

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

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

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

rmgpy.data.kinetics.LibraryReaction

class rmgpy.data.kinetics.**LibraryReaction**(*index=-1*, *reactants=None*, *products=None*, *specificCollider=None*, *kinetics=None*, *network_kinetics=None*, *reversible=True*, *transitionState=None*, *duplicate=False*, *degeneracy=1*, *pairs=None*, *library=None*, *allow_pdep_route=False*, *elementary_high_p=False*, *allow_max_rate_violation=False*, *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.

allow_max_rate_violation
'bool'

Type allow_max_rate_violation

allow_pdep_route
'bool'

Type allow_pdep_route

calculateMicrocanonicalRateCoefficient(*self*, *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.

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

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

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

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.

calculateTSTRateCoefficients(*self*, *ndarray Tlist*) → ndarray

calculate_coll_limit(*self*, *float temp*, *bool reverse=False*)
Calculate the collision limit rate for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

canTST(*self*) → bool
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.

check_collision_limit_violation(*self*, *float t_min*, *float t_max*, *float p_min*, *float p_max*) → list
Warn if a core reaction violates the collision limit rate in either the forward or reverse direction at the relevant extreme T/P conditions. Assuming a monotonic behaviour of the kinetics. Returns a list with the reaction object and the direction in which the violation was detected.

comment
str

Type comment

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

degeneracy
Reaction.__getDegneneracy(self)

draw(*self*, *path*)
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.

duplicate
'bool'

Type duplicate

elementary_high_p
'bool'

Type elementary_high_p

ensure_species(*self*, *bool reactant_resonance=False*, *bool product_resonance=True*)
Ensure the reaction contains species objects in its reactant and product attributes. If the reaction is found to hold molecule objects, it modifies the reactant, product and pairs to hold Species objects.

Generates resonance structures for Molecules if the corresponding options, reactant_resonance and/or product_resonance, are True. Does not generate resonance for reactants or products that start as Species objects.

fixBarrierHeight(*self*, *bool forcePositive=False*)
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(*self*, *T*)

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

generate3dTS(*self*, *reactants*, *products*)

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(*self*)

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 (C/O/N/S at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generateReverseRateCoefficient(*self*, *bool network_kinetics=False*)

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

generate_high_p_limit_kinetics()

If the LibraryReactions represented by *self* has pressure dependent kinetics, try extracting the high pressure limit rate from it. Used for incorporating library reactions with pressure-dependent kinetics in PDep networks. Only reactions flagged as *elementary_high_p=True* should be processed here. If the kinetics is a :class:Lindemann or a :class:Troe, simply get the high pressure limit rate. If the kinetics is a :class:PDepArrhenius or a :class:Chebyshev, generate a :class:Arrhenius kinetics entry that represents the high pressure limit if Pmax >= 90 bar . This high pressure limit Arrhenius kinetics is assigned to the reaction network_kinetics attribute. If this method successfully generated the high pressure limit kinetics, return `True`, otherwise `False`.

getEnthalpiesOfReaction(*self*, *ndarray Tlist*) → ndarray

Return the enthalpies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

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

getEntropiesOfReaction(*self*, *ndarray Tlist*) → ndarray

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

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

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

getEquilibriumConstant(*self*, *double T*, *str type='Kc'*) → double

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(*self*, *ndarray Tlist*, *str type='Kc'*) → ndarray

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`(*self*, *ndarray Tlist*) → ndarray
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures *Tlist* in K.

`getFreeEnergyOfReaction`(*self*, *double T*) → double
Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K.

`getRateCoefficient`(*self*, *double T*, *double P=0*) → double
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`(*self*, *Species spec*) → int
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`(*self*)
Get a URL to search for this reaction in the rmg website.

`get_mean_sigma_and_epsilon`(*self*, *bool reverse=False*)
Calculates the collision diameter (sigma) using an arithmetic mean Calculates the well depth (epsilon) using a geometric mean If reverse is `False` the above is calculated for the reactants, otherwise for the products

`get_reduced_mass`(*self*, *bool reverse=False*)
Returns the reduced mass of the reactants if reverse is `False` Returns the reduced mass of the products if reverse is `True`

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

`index`
'int'

Type index

`isAssociation`(*self*) → bool
Return `True` if the reaction represents an association reaction $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or `False` if not.

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

`isDissociation`(*self*) → bool
Return `True` if the reaction represents a dissociation reaction $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or `False` if not.

`isIsomerization`(*self*) → bool
Return `True` if the reaction represents an isomerization reaction $\mathrm{A} \rightleftharpoons \mathrm{B}$ or `False` if not.

`isIsomorphic`(*self*, *Reaction other*, *bool eitherDirection=True*, *bool checkIdentical=False*, *bool checkOnlyLabel=False*, *bool checkTemplateRxnProducts=False*) → bool
Return `True` if this reaction is the same as the *other* reaction, or `False` if they are different. The comparison involves comparing isomorphism of reactants and products, and doesn't use any kinetic information.

If *eitherDirection=False* then the directions must match.

***checkIdentical* indicates that atom ID's must match and is used in** checking degeneracy

***checkOnlyLabel* indicates that the string representation will be** checked, ignoring the molecular structure comparisons

***checkTemplateRxnProducts* indicates that only the products of the** reaction are checked for isomorphism. This is used when we know the reactants are identical, i.e. in generating reactions.

`isUnimolecular`(*self*) → bool

Return `True` if the reaction has a single molecule as either reactant or product (or both) $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or $\mathrm{A} \rightleftharpoons \mathrm{B}$, or `False` if not.

`is_forward`

'bool'

Type is_forward

`k_effective_cache`

dict

Type k_effective_cache

`kinetics`

rmgpy.kinetics.model.KineticsModel

Type kinetics

`label`

str

Type label

`matchesSpecies`(*self*, *list reactants*, *list products=None*) → bool

Compares the provided reactants and products against the reactants and products of this reaction. Both directions are checked.

Parameters

- **`reactants`** (*`list`*) – Species required on one side of the reaction
- **`products`** (*`list, optional`*) – Species required on the other side

`network_kinetics`

rmgpy.kinetics.arrhenius.Arrhenius

Type network_kinetics

`pairs`

list

Type pairs

`products`

list

Type products

`reactants`

list

Type reactants

`reverseThisArrheniusRate`(*self*, *Arrhenius kForward*, *str reverseUnits*)

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

reversible
‘bool’

Type reversible

specificCollider
rmgpy.species.Species

Type specificCollider

toCantera(*self*, *speciesList=None*, *useChemkinIdentifier=False*)
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species’ labels are unique when setting it False.

toChemkin(*self*, *speciesList=None*, *kinetics=True*)
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.

toLabeledStr(*self*, *use_index=False*)
the same as __str__ except that the labels are assumed to exist and used for reactant and products rather than the labels plus the index in parentheses

transitionState
rmgpy.species.TransitionState

Type transitionState

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

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

`addAction``(`*action*`)`

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

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

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

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

Apply the reverse reaction recipe to *molecule*, a single `Molecule` object.

`getReverse``()`

Generate a reaction recipe that, when applied, does the opposite of what the current recipe does, i.e., it is the recipe for the reverse of the reaction that this is the recipe for.

rmgpy.data.statmech.StatmechDatabase

`class` `rmgpy.data.statmech.`**`StatmechDatabase`**

A class for working with the RMG statistical mechanics (frequencies) database.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

rmgpy.data.statmech.StatmechDepository

class `rmgpy.data.statmech.`**`StatmechDepository`**(*label=", name=", shortDesc=", longDesc="*)

A class for working with the RMG statistical mechanics (frequencies) depository.

`ancestors`(*node*)

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

`areSiblings`(*node, nodeOther*)

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

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

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

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

Returns None if there is no matching root.

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

`descendants`(*node*)

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

`generateOldTree`(*entries, level*)

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

`getEntriesToSave`()

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

Then renumber the entry indexes so that we never have any duplicate indexes.

`getSpecies`(*path, resonance=True*)

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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

save(*path*)

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

saveDictionary(*path*)

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

saveEntry(*f*, *entry*)

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

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

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

saveOldDictionary(*path*)

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

saveOldLibrary(*path*)

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

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

rmgpy.data.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.

`evaluate``(`*x*`)`

Evaluate the nonlinear equations and constraints for this system, and the corresponding Jacobian matrices, at the given value of the solution vector *x*. Return a tuple containing three items:

- A vector of the current values of the system of equations $\mathbf{f}(\mathbf{x})$.
- A matrix of the current values of the Jacobian of the system of equations: $J_{ij} = \frac{\partial f_i}{\partial x_j}$.
- A matrix of the current values of the Jacobian of the (linear) constrains: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

`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.

`evaluate`(*x*)

Evaluate the nonlinear equations and constraints for this system, and the corresponding Jacobian matrices, at the given value of the solution vector *x*. Return a tuple containing three items:

- A vector of the current values of the system of equations $\mathbf{f}(\mathbf{x})$.
- A matrix of the current values of the Jacobian of the system of equations: $J_{ij} = \frac{\partial f_i}{\partial x_j}$.
- A matrix of the current values of the Jacobian of the (linear) constrains: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

`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".

evaluate(x)

Evaluate the nonlinear equations and constraints for this system, and the corresponding Jacobian matrices, at the given value of the solution vector *x*. Return a tuple containing three items:

- A vector of the current values of the system of equations $\mathbf{f}(\mathbf{x})$.
- A matrix of the current values of the Jacobian of the system of equations: $J_{ij} = \frac{\partial f_i}{\partial x_j}$.
- A matrix of the current values of the Jacobian of the (linear) constrains: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

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. $\|\vec{f}\| \le \epsilon_f$.
- *told* - The tolerance used for stopping when changes to the unknown variables has absolute length less than *told*, i.e. $\|\Delta \vec{x}\| \le \epsilon_d$.
- *tolx* - The tolerance used for stopping when changes to the unknown variables has relative length less than *tolx*, i.e. $\|\Delta \vec{x}\| \le \epsilon_x \cdot \|\vec{x}\|$.

- *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.

`areSiblings`(*node, nodeOther*)

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

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

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

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

Returns None if there is no matching root.

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

`descendants`(*node*)

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

`generateOldLibraryEntry`(*data*)

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

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

getSpecies(*path*, *resonance=True*)
: 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 speficied *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.

Attribute	Description
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

save(*path*)

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

saveDictionary(*path*)

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

saveEntry(*f*, *entry*)

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

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

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

saveOldDictionary(*path*)

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

saveOldLibrary(*path*)

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

saveOldTree(*path*)

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

rmgpy.data.statmech.StatmechLibrary

class rmgpy.data.statmech.**StatmechLibrary**(*label=''*, *name=''*, *shortDesc=''*, *longDesc=''*)

A class for working with a RMG statistical mechanics (frequencies) library.

ancestors(*node*)

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

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

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

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

Returns None if there is no matching root.

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

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

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)
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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

save(*path*)

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

saveDictionary(*path*)

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

saveEntry(*f*, *entry*)

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

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

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

saveOldDictionary(*path*)

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

saveOldLibrary(*path*)

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

saveOldTree(*path*)

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

rmgpy.data.kinetics.TemplateReaction

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

A Reaction object generated from a reaction family template. In addition to attributes inherited from `Reaction`, this class includes the following attributes:

Attribute	Type	Description
family	`str`	The kinetics family that the reaction was created from.
estimator	`str`	Whether the kinetics came from rate rules or group additivity.
reverse	`TemplateReaction`	The reverse reaction, for families that are their own reverse.
is_forward	`bool`	Whether the reaction was generated in the forward direction of the family.

`allow_max_rate_violation`

'bool'

Type allow_max_rate_violation

`allow_pdep_route`

'bool'

Type allow_pdep_route

`calculateMicrocanonicalRateCoefficient`(*self*, *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.

`calculateTSTRateCoefficient`(*self*, *double T*) → double

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

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

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.

`calculateTSTRateCoefficients`(*self*, *ndarray Tlist*) → ndarray

`calculate_coll_limit`(*self*, *float temp*, *bool reverse=False*)

Calculate the collision limit rate for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

`canTST`(*self*) → bool

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.

`check_collision_limit_violation`(*self*, *float t_min*, *float t_max*, *float p_min*, *float p_max*) → list

Warn if a core reaction violates the collision limit rate in either the forward or reverse direction at the relevant extreme T/P conditions. Assuming a monotonic behaviour of the kinetics. Returns a list with the reaction object and the direction in which the violation was detected.

`comment`

str

Type comment

`copy`()

creates a new instance of TemplateReaction

`degeneracy`

Reaction.__getDegneneracy(self)

`draw`(*self*, *path*)

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.

`duplicate`

'bool'

Type duplicate

`elementary_high_p`

'bool'

Type elementary_high_p

`ensure_species`(*self*, *bool reactant_resonance=False*, *bool product_resonance=True*)

Ensure the reaction contains species objects in its reactant and product attributes. If the reaction is found to hold molecule objects, it modifies the reactant, product and pairs to hold Species objects.

Generates resonance structures for Molecules if the corresponding options, reactant_resonance and/or product_resonance, are True. Does not generate resonance for reactants or products that start as Species objects.

`fixBarrierHeight`(*self*, *bool forcePositive=False*)

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`(*self*, *T*)

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

`generate3dTS`(*self*, *reactants*, *products*)

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(*self*)

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 (C/O/N/S at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generateReverseRateCoefficient(*self*, *bool network_kinetics=False*)

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

getEnthalpiesOfReaction(*self*, *ndarray Tlist*) → ndarray

Return the enthalpies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

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

getEntropiesOfReaction(*self*, *ndarray Tlist*) → ndarray

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

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

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

getEquilibriumConstant(*self*, *double T*, *str type='Kc'*) → double

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(*self*, *ndarray Tlist*, *str type='Kc'*) → ndarray

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(*self*, *ndarray Tlist*) → ndarray

Return the Gibbs free energies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

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

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

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(*self*, *Species spec*) → int
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(*self*)
Get a URL to search for this reaction in the rmg website.

get_mean_sigma_and_epsilon(*self*, *bool reverse=False*)
Calculates the collision diameter (sigma) using an arithmetic mean Calculates the well depth (epsilon) using a geometric mean If reverse is `False` the above is calculated for the reactants, otherwise for the products

get_reduced_mass(*self*, *bool reverse=False*)
Returns the reduced mass of the reactants if reverse is `False` Returns the reduced mass of the products if reverse is `True`

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

index
'int'

Type index

isAssociation(*self*) → bool
Return `True` if the reaction represents an association reaction $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or `False` if not.

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

isDissociation(*self*) → bool
Return `True` if the reaction represents a dissociation reaction $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or `False` if not.

isIsomerization(*self*) → bool
Return `True` if the reaction represents an isomerization reaction $\mathrm{A} \rightleftharpoons \mathrm{B}$ or `False` if not.

isIsomorphic(*self*, *Reaction other*, *bool eitherDirection=True*, *bool checkIdentical=False*, *bool checkOnlyLabel=False*, *bool checkTemplateRxnProducts=False*) → bool
Return `True` if this reaction is the same as the *other* reaction, or `False` if they are different. The comparison involves comparing isomorphism of reactants and products, and doesn't use any kinetic information.

If *eitherDirection=False* then the directions must match.

***checkIdentical* indicates that atom ID's must match and is used in** checking degeneracy

***checkOnlyLabel* indicates that the string representation will be** checked, ignoring the molecular structure comparisons

***checkTemplateRxnProducts* indicates that only the products of the** reaction are checked for isomorphism. This is used when we know the reactants are identical, i.e. in generating reactions.

isUnimolecular(*self*) → bool
Return `True` if the reaction has a single molecule as either reactant or product (or both) $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or $\mathrm{A} \rightleftharpoons \mathrm{B}$, or `False` if not.

is_forward
'bool'

Type is_forward

k_effective_cache
dict

Type k_effective_cache

kinetics
rmgpy.kinetics.model.KineticsModel

Type kinetics

label
str

Type label

matchesSpecies(*self*, *list reactants*, *list products=None*) → bool
Compares the provided reactants and products against the reactants and products of this reaction. Both directions are checked.

Parameters

- **reactants** (*list*) – Species required on one side of the reaction
- **products** (*list, optional*) – Species required on the other side

network_kinetics
rmgpy.kinetics.arrhenius.Arrhenius

Type network_kinetics

pairs
list

Type pairs

products
list

Type products

reactants
list

Type reactants

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

reversible
'bool'

Type reversible

specificCollider
rmgpy.species.Species

Type specificCollider

toCantera(*self*, *speciesList=None*, *useChemkinIdentifier=False*)
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

`toChemkin`(*self*, *speciesList=None*, *kinetics=True*)

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.

`toLabeledStr`(*self*, *use_index=False*)

the same as __str__ except that the labels are assumed to exist and used for reactant and products rather than the labels plus the index in parentheses

`transitionState`

rmgpy.species.TransitionState

Type transitionState

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.

No entropy is included in the returning term. This should be done later by the calling function.

`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.

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

`extractSourceFromComments`(*species*)

species: A species object containing thermo data and thermo data comments

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

Returns a dictionary with keys of either 'Library', 'QM', and/or 'GAV'. Commonly, species thermo are estimated using only one of these sources. However, a radical can be estimated with more than one type of source, for instance a saturated library value and a GAV HBI correction, or a QM saturated value and a GAV HBI correction.

source = {'Library': String_Name_of_Library_Used, 'QM': String_of_Method_Used, 'GAV': Dictionary_of_Groups_Used }

The Dictionary_of_Groups_Used looks like {'groupType':[List of tuples containing (Entry, Weight)]

`getAllThermoData`(*species*)

Return all possible sets of thermodynamic parameters for a given `Species` object *species*. The hits from the depository come first, then the libraries (in order), and then the group additivity estimate. This method is useful for a generic search job.

Returns: a list of tuples (ThermoData, source, entry) (Source is a library or depository, or None)

getRingGroupsFromComments(*thermoData*)

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

getThermoData(*species*, *trainingSet=None*)

Return the thermodynamic parameters for a given `Species` object *species*. This function first searches the loaded libraries in order, returning the first match found, before falling back to estimation via machine learning and then group additivity.

The method corrects for symmetry when the molecule uses machine learning or group additivity. Libraries and direct QM calculations are already corrected.

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.

This does not account for symmetry. The method calling this sould correct for it.

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.

get_thermo_data_from_ml(*species*, *ml_estimator*, *ml_settings*)

Return the set of thermodynamic parameters corresponding to a given `Species` object *species* by estimation using the ML estimator. Also compare the estimated uncertainties to the user-defined cutoffs. If any of the uncertainties are larger than their corresponding cutoffs, return None. Also check all other options in *ml_settings*.

For HBI, the resonance isomer with the lowest H298 is used and the resonance isomers in species are sorted in ascending order.

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

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.

If no libraries are given, all are loaded.

loadOld(*path*)

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

prioritizeThermo(*species*, *thermoDataList*)

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

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

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

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

recordPolycylicGenericNodes()

Identify generic nodes in tree for polycyclic groups. Saves them as a list in the *genericNodes* attribute in the polycyclic `ThermoGroups` object, which must be pre-loaded.

Necessary for polycyclic heuristic.

recordRingGenericNodes()

Identify generic nodes in tree for ring groups. Saves them as a list in the *genericNodes* attribute in the ring `ThermoGroups` object, which must be pre-loaded.

Necessary for polycyclic heuristic.

save(*path*)

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

saveDepository(*path*)

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

saveGroups(*path*)

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

saveLibraries(*path*)

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

saveOld(*path*)

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

rmgpy.data.thermo.ThermoDepository

class `rmgpy.data.thermo.`**ThermoDepository**(*label=”*, *name=”*, *shortDesc=”*, *longDesc=”*)

A class for working with the RMG thermodynamics depository.

ancestors(*node*)

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

areSiblings(*node*, *nodeOther*)

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

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

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

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

Returns None if there is no matching root.

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

descendants(*node*)

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

generateOldTree(*entries*, *level*)

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

getEntriesToSave()

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)

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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

save(*path*)

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

saveDictionary(*path*)

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

saveEntry(*f*, *entry*)

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

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

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

saveOldDictionary(*path*)

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

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

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

rmgpy.data.thermo.ThermoGroups

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

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

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

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

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

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

Returns None if there is no matching root.

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)
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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

removeGroup(*groupToRemove*)

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

Returns the removed group

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

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

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

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

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

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

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

rmgpy.data.thermo.ThermoLibrary

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

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

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

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

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

Returns None if there is no matching root.

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

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

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

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

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

Then renumber the entry indexes so that we never have any duplicate indexes.

getSpecies(*path*, *resonance=True*)
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
node	Either an Entry or a key in the self.entries dictionary which has a Group or LogicNode as its Entry.item
structure	A Group or a Molecule
atoms	Dictionary of {label: atom} in the structure. A possible dictionary is the one produced by structure.getLabeledAtoms()
strict	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.

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

Returns the removed group

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

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

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

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

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

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

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

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
`KineticsData`	A kinetics model based on a set of discrete rate coefficient points in temperature
`Arrhenius`	A kinetics model based on the (modified) Arrhenius expression
`MultiArrhenius`	A kinetics model based on a sum of `Arrhenius` expressions

1.5.2 Pressure-dependent kinetics models

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

1.5.3 Tunneling models

Class	Description
Wigner	A one-dimensional tunneling model based on the Wigner expression
Eckart	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
Tdata	An array of temperatures at which rate coefficient values are known
kdata	An array of rate coefficient values
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
comment	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

str

Type comment

`discrepancy`(*self*, *KineticsModel otherKinetics*) → double

Returns some measure of the discrepancy based on two different reaction models.

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

Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K.

`isIdenticalTo`(*self*, *KineticsModel otherKinetics*) → bool

Returns `True` if the kdata and Tdata match. Returns `False` otherwise.

`isPressureDependent`(*self*) → bool

Return `False` since, by default, all objects derived from KineticsModel represent pressure-independent kinetics.

`isSimilarTo`(*self*, *KineticsModel otherKinetics*) → bool

Returns `True` if rates of reaction at temperatures 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.

`setCanteraKinetics`(*self*, *ctReaction*, *speciesList*)

Sets the kinetics for a cantera reaction object.

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

The Arrhenius equation, given below, accurately reproduces the kinetics of many reaction families:

$$k(T) = A \left(\frac{T}{T_0} \right)^n \exp \left(- \frac{E_\mathrm{a}}{RT} \right)$$

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`
str

Type comment

`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 threeParams=True*)
Fit the Arrhenius parameters to a set of rate coefficient data *klist* in units of *kunits* corresponding to a set of temperatures *Tlist* in K. A linear least-squares fit is used, which guarantees that the resulting parameters provide the best possible approximation to the data.

`getRateCoefficient`(*self*, *double T*, *double P=0.0*) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K.

`isIdenticalTo`(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if kinetics matches that of another kinetics model. Must match temperature and pressure range of kinetics model, as well as parameters: A, n, Ea, T0. (Shouldn't have pressure range if it's Arrhenius.) Otherwise returns `False`.

`isPressureDependent`(*self*) → bool
Return `False` since, by default, all objects derived from KineticsModel represent pressure-independent kinetics.

`isSimilarTo`(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

`isTemperatureValid`(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

`n`
The temperature exponent.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Passes in a cantera ElementaryReaction() object and sets its rate to a Cantera Arrhenius() object.

toArrheniusEP(*self*, *double alpha=0.0*, *double dHrxn=0.0*) → ArrheniusEP
Converts an Arrhenius object to ArrheniusEP

If setting alpha, you need to also input dHrxn, which must be given in J/mol (and vise versa).

toCanteraKinetics(*self*)
Converts the Arrhenius object to a cantera Arrhenius object

Arrhenius(A,b,E) where A is in units of m^3/kmol/s, b is dimensionless, and E is in J/kmol

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.MultiArrhenius

class rmgpy.kinetics.**MultiArrhenius**(*arrhenius=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *comment=”*)
A kinetics model based on a set of (modified) Arrhenius equations, which are summed to obtain the overall rate. The attributes are:

Attribute	Description
arrhenius	A list of the `Arrhenius` kinetics
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
comment	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
list

Type arrhenius

changeRate(*self*, *double factor*)
Change kinetics rate by a multiple `factor`.

comment
str

Type comment

discrepancy(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

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

Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K.

isIdenticalTo(*self*, *KineticsModel otherKinetics*) → bool

Returns `True` if kinetics matches that of another kinetics model. Each duplicate reaction must be matched and equal to that in the other MultiArrhenius model in the same order. Otherwise returns `False`

isPressureDependent(*self*) → bool

Return `False` since, by default, all objects derived from KineticsModel represent pressure-independent kinetics.

isSimilarTo(*self*, *KineticsModel otherKinetics*) → bool

Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(*self*, *double T*) → bool

Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)

Sets the kinetic rates for a list of cantera *Reaction* objects Here, ctReaction must be a list rather than a single cantera reaction.

toArrhenius(*self*, *double Tmin=-1*, *double Tmax=-1*) → Arrhenius

Return an `Arrhenius` instance of the kinetics model

Fit the Arrhenius parameters to a set of rate coefficient data generated from the MultiArrhenius kinetics, over the temperature range Tmin to Tmax, in Kelvin. If Tmin or Tmax are unspecified (or -1) then the MultiArrhenius's Tmin and Tmax are used. A linear least-squares fit is used, which guarantees that the resulting parameters provide the best possible approximation to the data.

toHTML(*self*)

Return an HTML rendering.

rmgpy.kinetics.PDepKineticsData

class rmgpy.kinetics.**PDepKineticsData**(*Tdata=None*, *Pdata=None*, *kdata=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *comment=''*)

A kinetics model based on an array of rate coefficient data vs. temperature and pressure. The attributes are:

Attribute	Description
Tdata	An array of temperatures at which rate coefficient values are known
Pdata	An array of pressures at which rate coefficient values are known
kdata	An array of rate coefficient values at each temperature and pressure
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
comment	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`
str

Type comment

`discrepancy`(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

`efficiencies`
dict

Type efficiencies

`getCanteraEfficiencies`(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

`getEffectiveColliderEfficiencies`(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

`getEffectivePressure`(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

`getRateCoefficient`(*self*, *double T*, *double P=0.0*) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K and pressure *P* in Pa.

`highPlimit`
rmgpy.kinetics.model.KineticsModel

Type highPlimit

`isIdenticalTo`(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if the kdata and Tdata match. Returns `False` otherwise.

`isPressureDependent`(*self*) → bool
Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

`isPressureValid`(*self*, *double P*) → bool
Return `True` if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

`isSimilarTo`(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

`isTemperatureValid`(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

kdata
An array of rate coefficient values at each temperature and pressure.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Sets the kinetics for a cantera reaction object.

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.PDepArrhenius

class `rmgpy.kinetics.`**PDepArrhenius**(*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
pressures	The list of pressures
arrhenius	The list of `Arrhenius` objects at each pressure
Tmin	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure in bar at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure in bar at which the model is valid, or zero if unknown or undefined
efficiencies	A dict associating chemical species with associated efficiencies
order	The reaction order (1 = first, 2 = second, etc.)
comment	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:

$$k(T,P) = A(P) \left(\frac{T}{T_0} \right)^{n(P)} \exp \left(-\frac{E_\mathrm{a}(P)}{RT} \right)$$

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
list

Type arrhenius

changeRate(*self*, *double factor*)
Changes kinetics rate by a multiple `factor`.

comment
str

Type comment

discrepancy(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

efficiencies
dict

Type efficiencies

fitToData(*self*, *ndarray Tlist*, *ndarray Plist*, *ndarray K*, *str kunits*, *double T0=1*)
Fit the pressure-dependent Arrhenius model to a matrix of rate coefficient data *K* with units of *kunits* corresponding to a set of temperatures *Tlist* in K and pressures *Plist* in Pa. An Arrhenius model is fit cpdef changeRate(self, double factor)at each pressure.

getCanteraEfficiencies(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

getEffectiveColliderEfficiencies(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0*) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K and pressure *P* in Pa.

highPlimit
rmgpy.kinetics.model.KineticsModel

Type highPlimit

isIdenticalTo(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if kinetics matches that of another kinetics model. Each duplicate reaction must be matched and equal to that in the other PDepArrhenius model in the same order. Otherwise returns `False`

isPressureDependent(*self*) → bool
Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

isPressureValid(*self*, *double P*) → bool
Return `True` if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

isSimilarTo(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

pressures
The list of pressures.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)

Sets a Cantera PlogReaction()'s *rates* attribute with A list of tuples containing [(pressure in Pa, cantera arrhenius object), (..)]

toHTML(*self*)

Return an HTML rendering.

rmgpy.kinetics.MultiPDepArrhenius

class rmgpy.kinetics.**MultiPDepArrhenius**(*arrhenius=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *comment="*)

A kinetic model of a phenomenological rate coefficient $k(T, P)$ where sets of Arrhenius kinetics are stored at a variety of pressures and interpolated between on a logarithmic scale. The attributes are:

Attribute	Description
arrhenius	A list of the `PDepArrhenius` kinetics at each temperature
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
comment	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

list

Type arrhenius

changeRate(*self*, *double factor*)

Change kinetic rate by a multiple `factor`.

comment

str

Type comment

discrepancy(*self*, *KineticsModel otherKinetics*) → double

Returns some measure of the discrepancy based on two different reaction models.

efficiencies

dict

Type efficiencies

getCanteraEfficiencies(*self*, *speciesList*)

Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

getEffectiveColliderEfficiencies(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0.0*) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K and pressure *P* in Pa.

highPlimit
rmgpy.kinetics.model.KineticsModel

Type highPlimit

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.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Sets the PLOG kinetics for multiple cantera *Reaction* objects, provided in a list. ctReaction is a list of cantera reaction objects.

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.Chebyshev

class rmgpy.kinetics.**Chebyshev**(*coeffs=None*, *kunits=”*, *highPlimit=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *comment=”*)
A model of a phenomenological rate coefficient $k(T,P)$ using a set of Chebyshev polynomials in temperature and pressure. The attributes are:

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

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

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

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

str

Type comment

degreeP

'int'

Type degreeP

degreeT

'int'

Type degreeT

`discrepancy`(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

`efficiencies`
dict

Type efficiencies

`fitToData`(*self*, *ndarray Tlist*, *ndarray Plist*, *ndarray K*, *str kunits*, *int degreeT*, *int degreeP*, *double Tmin*, *double Tmax*, *double Pmin*, *double Pmax*)
Fit a Chebyshev kinetic model to a set of rate coefficients *K*, which is a matrix corresponding to the temperatures *Tlist* in K and pressures *Plist* in Pa. *degreeT* and *degreeP* are the degree of the polynomials in temperature and pressure, while *Tmin*, *Tmax*, *Pmin*, and *Pmax* set the edges of the valid temperature and pressure ranges in K and bar, respectively.

`getCanteraEfficiencies`(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,‘ChemicallyActivatedReaction‘

`getEffectiveColliderEfficiencies`(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

`getEffectivePressure`(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

`getRateCoefficient`(*self*, *double T*, *double P=0*) → double
Return the rate coefficient in the appropriate combination of m^3, mol, and s at temperature *T* in K and pressure *P* in Pa by evaluating the Chebyshev expression.

`highPlimit`
rmgpy.kinetics.model.KineticsModel

Type highPlimit

`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`
str

Type kunits

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)

Sets the kinetics parameters for a Cantera ChebyshevReaction() object Uses set_parameters(self,Tmin,Tmax,Pmin,Pmax,coeffs) where T's are in units of K, P's in units of Pa, and coeffs is 2D array of (nTemperature, nPressure).

toHTML(*self*)

Return an HTML rendering.

rmgpy.kinetics.ThirdBody

class rmgpy.kinetics.**ThirdBody**(*arrheniusLow=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *efficiencies=None*, *comment="*)

A kinetic model of a phenomenological rate coefficient $k(T, P)$ using third-body kinetics. The attributes are:

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

Third-body kinetics simply introduce an inert third body to the rate expression:

$$k(T, P) = k_0(T)[\mathrm{M}]$$

Above, $[\mathrm{M}] \approx P/RT$ 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

rmgpy.kinetics.arrhenius.Arrhenius

Type arrheniusLow

changeRate(*self*, *double factor*)

Changes kinetics rate by a multiple `factor`.

comment

str

Type comment

discrepancy(*self*, *KineticsModel otherKinetics*) → double

Returns some measure of the discrepancy based on two different reaction models.

efficiencies
dict

Type efficiencies

getCanteraEfficiencies(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

getEffectiveColliderEfficiencies(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0.0*) → double
Return the value of the rate coefficient $k(T)$ in units of m^3, mol, and s at the specified temperature *T* in K and pressure *P* in Pa. If you wish to consider collision efficiencies, then you should first use `getEffectivePressure()` to compute the effective pressure, and pass that value as the pressure to this method.

highPlimit
rmgpy.kinetics.model.KineticsModel

Type highPlimit

isIdenticalTo(*self*, *KineticsModel otherKinetics*) → bool
Checks to see if kinetics matches that of other kinetics and returns `True` if coeffs, kunits, Tmin,

isPressureDependent(*self*) → bool
Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

isPressureValid(*self*, *double P*) → bool
Return `True` if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

isSimilarTo(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Sets the kinetics and efficiencies for a cantera *ThreeBodyReaction* object

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.Lindemann

class rmgpy.kinetics.**Lindemann**(*arrheniusHigh=None*, *arrheniusLow=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *efficiencies=None*, *comment=''*)
A kinetic model of a phenomenological rate coefficient $k(T, P)$ using the Lindemann formulation. The attributes are:

Attribute	Description
arrheniusHigh	The Arrhenius kinetics at the high-pressure limit
arrheniusLow	The Arrhenius kinetics at the low-pressure limit
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
efficiencies	A dict associating chemical species with associated efficiencies
comment	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:

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

where

$$P_\mathrm{r} = \frac{k_0(T)}{k_\infty(T)} [\mathrm{M}]$$

$$k_0(T) = A_0 T^{n_0} \exp \left(- \frac{E_0}{RT} \right)$$

$$k_\infty(T) = A_\infty T^{n_\infty} \exp \left(- \frac{E_\infty}{RT} \right)$$

and $[\mathrm{M}] \approx P/RT$ 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
rmgpy.kinetics.arrhenius.Arrhenius

Type arrheniusHigh

arrheniusLow
rmgpy.kinetics.arrhenius.Arrhenius

Type arrheniusLow

changeRate(*self*, *double factor*)
Changes kinetics rate by a multiple `factor`.

comment
str

Type comment

discrepancy(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

efficiencies
dict

Type efficiencies

getCanteraEfficiencies(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

getEffectiveColliderEfficiencies(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

getEffectivePressure(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

getRateCoefficient(*self*, *double T*, *double P=0.0*) → double
Return the value of the rate coefficient $k(T)$ in units of m^3, mol, and s at the specified temperature *T* in K and pressure *P* in Pa. If you wish to consider collision efficiencies, then you should first use `getEffectivePressure()` to compute the effective pressure, and pass that value as the pressure to this method.

highPlimit
rmgpy.kinetics.model.KineticsModel

Type highPlimit

isIdenticalTo(*self*, *KineticsModel otherKinetics*) → bool
Checks to see if kinetics matches that of other kinetics and returns `True` if coeffs, kunits, Tmin,

isPressureDependent(*self*) → bool
Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

isPressureValid(*self*, *double P*) → bool
Return `True` if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

isSimilarTo(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Sets the efficiencies and kinetics for a cantera reaction.

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.Troe

class rmgpy.kinetics.**Troe**(*arrheniusHigh=None*, *arrheniusLow=None*, *alpha=0.0*, *T3=None*, *T1=None*, *T2=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *efficiencies=None*, *comment="*)
A kinetic model of a phenomenological rate coefficient $k(T, P)$ using the Troe formulation. The attributes are:

Attribute	Description
arrheniusHigh	The Arrhenius kinetics at the high-pressure limit
arrheniusLow	The Arrhenius kinetics at the low-pressure limit
alpha	The α parameter
T1	The T_1 parameter
T2	The T_2 parameter
T3	The T_3 parameter
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
Pmin	The minimum pressure at which the model is valid, or zero if unknown or undefined
Pmax	The maximum pressure at which the model is valid, or zero if unknown or undefined
efficiencies	A dict associating chemical species with associated efficiencies
comment	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:

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

where

$$\begin{aligned} P_\mathrm{r} &= \frac{k_0(T)}{k_\infty(T)} [\mathrm{M}] \\ k_0(T) &= A_0 T^{n_0} \exp \left(- \frac{E_0}{RT} \right) \\ k_\infty(T) &= A_\infty T^{n_\infty} \exp \left(- \frac{E_\infty}{RT} \right) \end{aligned}$$

and $[\mathrm{M}] \approx P/RT$ 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

$$\begin{aligned} \log F &= \left\{ 1 + \left[\frac{\log P_\mathrm{r} + c}{n - d (\log P_\mathrm{r} + c)} \right]^2 \right\}^{-1} \log F_\mathrm{cent} \\ c &= -0.4 - 0.67 \log F_\mathrm{cent} \\ n &= 0.75 - 1.27 \log F_\mathrm{cent} \\ d &= 0.14 \\ F_\mathrm{cent} &= (1 - \alpha) \exp \left(-T/T_3 \right) + \alpha \exp \left(-T/T_1 \right) + \exp \left(-T_2/T \right) \end{aligned}$$

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`
'double'

Type alpha

`arrheniusHigh`
rmgpy.kinetics.arrhenius.Arrhenius

Type arrheniusHigh

`arrheniusLow`
rmgpy.kinetics.arrhenius.Arrhenius

Type arrheniusLow

`changeRate`(*self*, *double factor*)
Changes kinetics rate by a multiple `factor`.

`comment`
str

Type comment

`discrepancy`(*self*, *KineticsModel otherKinetics*) → double
Returns some measure of the discrepancy based on two different reaction models.

`efficiencies`
dict

Type efficiencies

`getCanteraEfficiencies`(*self*, *speciesList*)
Returns a dictionary containing the collider efficiencies for this PDepKineticsModel object suitable for setting the efficiencies in the following cantera reaction objects: *ThreeBodyReaction*, *FalloffReaction*,'ChemicallyActivatedReaction'

`getEffectiveColliderEfficiencies`(*self*, *list species*) → ndarray
Return the effective collider efficiencies for all species in the form of a numpy array. This function helps assist rapid effective pressure calculations in the solver.

`getEffectivePressure`(*self*, *double P*, *list species*, *ndarray fractions*) → double
Return the effective pressure in Pa for a system at a given pressure *P* in Pa composed of the given list of *species* (Species or Molecule objects) with the given *fractions*.

`getRateCoefficient`(*self*, *double T*, *double P=0.0*) → double
Return the value of the rate coefficient $k(T)$ in units of m^3, mol, and s at the specified temperature *T* in K and pressure *P* in Pa. If you wish to consider collision efficiencies, then you should first use `getEffectivePressure()` to compute the effective pressure, and pass that value as the pressure to this method.

`highPlimit`
rmgpy.kinetics.model.KineticsModel

Type highPlimit

`isIdenticalTo`(*self*, *KineticsModel otherKinetics*) → bool
Checks to see if kinetics matches that of other kinetics and returns `True` if coeffs, kunits, Tmin,

isPressureDependent(*self*) → bool
Return `True` since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

isPressureValid(*self*, *double P*) → bool
Return `True` if the pressure *P* in Pa is within the valid pressure range of the kinetic data, or `False` if not. If the minimum and maximum pressure are not defined, `True` is returned.

isSimilarTo(*self*, *KineticsModel otherKinetics*) → bool
Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within +/ .5 for log(k), in other words, within a factor of 3.

isTemperatureValid(*self*, *double T*) → bool
Return `True` if the temperature *T* in K is within the valid temperature range of the kinetic data, or `False` if not. If the minimum and maximum temperature are not defined, `True` is returned.

setCanteraKinetics(*self*, *ctReaction*, *speciesList*)
Sets the efficiencies, kinetics, and troe falloff parameters for a cantera FalloffReaction.

toHTML(*self*)
Return an HTML rendering.

rmgpy.kinetics.Wigner

class `rmgpy.kinetics.`**Wigner**(*frequency*)
A tunneling model based on the Wigner formula. The attributes are:

Attribute	Description
frequency	The imaginary frequency of the transition state

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

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

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
frequency	The imaginary frequency of the transition state
E0_reac	The ground-state energy of the reactants
E0_TS	The ground-state energy of the transition state
E0_prod	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

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

where x represents the reaction coordinate and A and B are parameters. The potential is symmetric if $A = 0$ and asymmetric if $A \neq 0$. If we add the constraint $|B| > |A|$ then the potential has a maximum at

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

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

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

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

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

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

$$\alpha_1 = 2 \pi \frac{\Delta V_1}{h \left| \nu_\mathrm{TS} \right|}$$

$$\alpha_2 = 2 \pi \frac{\Delta V_2}{h \left| \nu_\mathrm{TS} \right|}$$

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

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 $\beta \equiv 1/k_\mathrm{B} T$):

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

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
`Vertex`	A generic vertex (node) in a graph
`Edge`	A generic edge (arc) in a graph
`Graph`	A generic graph data type

1.6.2 Graph isomorphism

Class	Description
`VF2`	Graph isomorphism using the VF2 algorithm

1.6.3 Elements and atom types

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

1.6.4 Molecules

Class	Description
`Atom`	An atom in a molecule
`Bond`	A bond in a molecule
`Molecule`	A molecular structure represented using a chemical graph

1.6.5 Functional groups

Class	Description
`GroupAtom`	An atom in a functional group
`GroupBond`	A bond in a functional group
`Group`	A functional group structure represented using a chemical graph

1.6.6 Molecule Utilities

Class	Description
`rmgpy.molecule.resonance`	Resonance structure generation methods
`rmgpy.molecule.kekulize`	Kekule structure generation
`rmgpy.molecule.pathfinder`	Resonance path enumeration
`rmgpy.molecule.converter`	Molecule object converter (RDKit/OpenBabel)
`rmgpy.molecule.translator`	Molecule string representation translator

1.6.7 Adjacency lists

Function	Description
`fromAdjacencyList()`	Convert an adjacency list to a set of atoms and bonds
`toAdjacencyList()`	Convert a set of atoms and bonds to an adjacency list

1.6.8 Symmetry numbers

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

1.6.9 Molecule and reaction drawing

Class	Description
`MoleculeDrawer`	Draw the skeletal formula of a molecule
`ReactionDrawer`	Draw a chemical reaction

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
connectivity1	`int`	The number of nearest neighbors
connectivity2	`int`	The sum of the neighbors' *connectivity1* values
connectivity3	`int`	The sum of the neighbors' *connectivity2* values
edges	`dict`	Dictionary of edges with keys being neighboring vertices
sortingLabel	`int`	An integer label used to sort the vertices

`connectivity1`

'short'

Type connectivity1

`connectivity2`

'short'

Type connectivity2

`connectivity3`

'short'

Type connectivity3

`copy`(*self*) → Vertex

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.

`edges`

dict

Type edges

`equivalent`(*self*, *Vertex other*) → bool

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.

`ignore`

'bool'

Type ignore

`isSpecificCaseOf`(*self*, *Vertex other*) → bool

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.

`mapping`

rmgpy.molecule.graph.Vertex

Type mapping

`resetConnectivityValues`(*self*)
Reset the cached structure information for this vertex.

`sortingLabel`
'short'

Type sortingLabel

`terminal`
'bool'

Type terminal

rmgpy.molecule.graph.Edge

class `rmgpy.molecule.graph.`**`Edge`**(*vertex1*, *vertex2*)
A base class for edges in a graph. The vertices which comprise the edge can be accessed using the *vertex1* and *vertex2* attributes.

`copy`(*self*) → Edge
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`(*self*, *Edge other*) → bool
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`(*self*, *Vertex vertex*) → Vertex
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`(*self*, *Edge other*) → bool
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.

`vertex1`
rmgpy.molecule.graph.Vertex

Type vertex1

`vertex2`
rmgpy.molecule.graph.Vertex

Type vertex2

rmgpy.molecule.graph.Graph

class `rmgpy.molecule.graph.`**`Graph`**(*vertices=None*)
A graph data type. The vertices of the graph are stored in a list *vertices*; this provides a consistent traversal order. A single edge can be accessed using the *`getEdge()`* method or by accessing specific vertices using `vertex1.edges[vertex2]`; in either case, an exception will be raised if the edge does not exist. All edges of a vertex can be accessed using the *`getEdges()`* method or `vertex.edges`.

`addEdge`(*self*, *Edge edge*) → Edge
Add an *edge* to the graph. The two vertices in the edge must already exist in the graph, or a `ValueError` is raised.

addVertex(*self*, *Vertex vertex*) → Vertex
Add a *vertex* to the graph. The vertex is initialized with no edges.

copy(*self*, *bool deep=False*) → Graph
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.

copyAndMap(*self*) → dict
Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from Graph.copy() method

findIsomorphism(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
Returns `True` if *other* is subgraph isomorphic and `False` otherwise, and the matching mapping. Uses the VF2 algorithm of Vento and Foggia.

findSubgraphIsomorphisms(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
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(*self*, *Vertex startingVertex*) → list
Given a starting vertex, returns a list of all the cycles containing that vertex.

This function returns a duplicate of each cycle because [0,1,2,3] is counted as separate from [0,3,2,1]

getAllCyclesOfSize(*self*, *int size*) → list
Return a list of the all non-duplicate rings with length 'size'. 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).

getAllCyclicVertices(*self*) → list
Returns all vertices belonging to one or more cycles.

getAllEdges(*self*) → list
Returns a list of all edges in the graph.

getAllPolycyclicVertices(*self*) → list
Return all vertices belonging to two or more cycles, fused or spirocyclic.

getAllSimpleCyclesOfSize(*self*, *int size*) → list
Return a list of all non-duplicate monocyclic rings with length 'size'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

getDisparateRings(*self*) → tuple
Get all disjoint monocyclic and polycyclic cycle clusters in the molecule. Takes the RC and recursively merges all cycles which share vertices.

Returns: monocyclic_cycles, polycyclic_cycles

getEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → Edge
Returns the edge connecting vertices *vertex1* and *vertex2*.

getEdges(*self*, *Vertex vertex*) → dict
Return a dictionary of the edges involving the specified *vertex*.

getLargestRing(*self*, *Vertex vertex*) → list
returns the largest ring containing vertex. This is typically useful for finding the longest path in a polycyclic

ring, since the polycyclic rings returned from getPolycyclicRings are not necessarily in order in the ring structure.

getMaxCycleOverlap(*self*) → int

Return the maximum number of vertices that are shared between any two cycles in the graph. For example, if there are only disparate monocycles or no cycles, the maximum overlap is zero; if there are "spiro" cycles, it is one; if there are "fused" cycles, it is two; and if there are "bridged" cycles, it is three.

getMonocyclicRings(*self*) → list

Return a list of cycles that are monocyclic.

getPolycyclicRings(*self*) → list

Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

getRelevantCycles(*self*) → list

Returns the set of relevant cycles as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

getSmallestSetOfSmallestRings(*self*) → list

Returns the smallest set of smallest rings as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

get_edges_in_cycle(*self*, *list vertices*, *bool sort=False*) → list

For a given list of atoms comprising a ring, return the set of bonds connecting them, in order around the ring.

If *sort=True*, then sort the vertices to match their connectivity. Otherwise, assumes that they are already sorted, which is true for cycles returned by getRelevantCycles or getSmallestSetOfSmallestRings.

hasEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → bool

Returns `True` if vertices *vertex1* and *vertex2* are connected by an edge, or `False` if not.

hasVertex(*self*, *Vertex vertex*) → bool

Returns `True` if *vertex* is a vertex in the graph, or `False` if not.

isCyclic(*self*) → bool

Return `True` if one or more cycles are present in the graph or `False` otherwise.

isEdgeInCycle(*self*, *Edge edge*) → bool

Return `True` if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or `False` if not.

isIsomorphic(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → bool

Returns `True` if two graphs are isomorphic and `False` otherwise. Uses the VF2 algorithm of Vento and Foggia.

isMappingValid(*self*, *Graph other*, *dict mapping*, *bool equivalent=True*) → bool

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. If equivalent is true it checks if atoms and edges are equivalent, if false it checks if they are specific cases of each other.

isSubgraphIsomorphic(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → bool
Returns `True` if *other* is subgraph isomorphic and `False` otherwise. Uses the VF2 algorithm of Vento and Foggia.

isVertexInCycle(*self*, *Vertex vertex*) → bool
Return `True` if the given *vertex* is contained in one or more cycles in the graph, or `False` if not.

merge(*self*, *Graph other*) → Graph
Merge two graphs so as to store them in a single Graph object.

ordered_vertices
list

Type ordered_vertices

removeEdge(*self*, *Edge edge*)
Remove the specified *edge* from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

removeVertex(*self*, *Vertex vertex*)
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(*self*)
Reset any cached connectivity information. Call this method when you have modified the graph.

restore_vertex_order(*self*)
reorder the vertices to what they were before sorting if you saved the order

sortVertices(*self*, *bool saveOrder=False*)
Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

split(*self*) → list
Convert a single Graph object containing two or more unconnected graphs into separate graphs.

updateConnectivityValues(*self*)
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

vertices
list

Type vertices

rmgpy.molecule.vf2.VF2

class `rmgpy.molecule.vf2.`**VF2**(*graphA=None*, *graphB=None*)
An implementation of the second version of the Vento-Foggia (VF2) algorithm for graph and subgraph isomorphism.

feasible(*self*, *Vertex vertex1*, *Vertex vertex2*) → bool
Return `True` if vertex *vertex1* from the first graph is a feasible match for vertex *vertex2* from the second graph, or `False` if not. The semantic and structural relationship of the vertices is evaluated, including several structural "look-aheads" that cheaply eliminate many otherwise feasible pairs.

findIsomorphism(*self*, *Graph graph1*, *Graph graph2*, *dict initialMapping*, *bool saveOrder=False*) → list
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(*self*, *Graph graph1*, *Graph graph2*, *dict initialMapping*, *bool saveOrder=False*) → list
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(*self*, *Graph graph1*, *Graph graph2*, *dict initialMapping*, *bool saveOrder=False*) → bool
Return `True` if graph *graph1* is isomorphic to graph *graph2* with the optional initial mapping *initialMapping*, or `False` otherwise.

isSubgraphIsomorphic(*self*, *Graph graph1*, *Graph graph2*, *dict initialMapping*, *bool saveOrder=False*) → bool
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**(*number*, *symbol*, *name*, *mass*, *isotope=-1*, *chemkinName=None*)
A chemical element. The attributes are:

Attribute	Type	Description
number	`int`	The atomic number of the element
symbol	`str`	The symbol used for the element
name	`str`	The IUPAC name of the element
mass	`float`	The mass of the element in kg/mol
covRadius	`float`	Covalent bond radius in Angstrom
isotope	`int`	The isotope integer of the element
chemkinName	`str`	The chemkin compatible representation of the element

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.

chemkinName
str

Type chemkinName

covRadius
'float'

Type covRadius

isotope
'int'

Type isotope

mass
'float'

Type mass

name
str

Type name

number
'int'

Type number

`symbol`
str

Type symbol

`rmgpy.molecule.`**`getElement`**`(`*value*, *int isotope=-1*`)` → Element

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`**`(`*label=”*, *generic=None*, *specific=None*, *single=None*, *allDouble=None*, *rDouble=None*, *oDouble=None*, *sDouble=None*, *triple=None*, *benzene=None*, *lonePairs=None*, *charge=None*`)`

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
label	`str`	A unique label for the atom type
generic	`list`	The atom types that are more generic than this one
specific	`list`	The atom types that are more specific than this one
incrementBond	`list`	The atom type(s) that result when an adjacent bond’s order is incremented
decrementBond	`list`	The atom type(s) that result when an adjacent bond’s order is decremented
formBond	`list`	The atom type(s) that result when a new single bond is formed to this atom type
breakBond	`list`	The atom type(s) that result when an existing single bond to this atom type is broken
incrementRadical	`list`	The atom type(s) that result when the number of radical electrons is incremented
decrementRadical	`list`	The atom type(s) that result when the number of radical electrons is decremented
incrementLonePair	`list`	The atom type(s) that result when the number of lone electron pairs is incremented
decrementLonePair	`list`	The atom type(s) that result when the number of lone electron pairs is decremented
The following features are what are required in a given atomtype. Any int in the list is acceptable. An empty list is a wildcard		
‘single’	‘’list’‘	The total number of single bonds on the atom
‘allDouble’	‘’list’‘	The total number of double bonds on the atom
‘rDouble’	‘’list’‘	The number of double bonds to any non-oxygen, nonsulfur
‘oDouble’	‘’list’‘	The number of double bonds to oxygen
‘sDouble’	‘’list’‘	The number of double bonds to sulfur
‘triple’	‘’list’‘	The total number of triple bonds on the atom
‘benzene’	‘’list’‘	The total number of benzene bonds on the atom
‘lonePairs’	‘’list’‘	The number of lone pairs on the atom
‘charge’	‘’list’‘	The partial charge of the atom

allDouble
list

Type allDouble

benzene
list

Type benzene

breakBond
list

Type breakBond

charge
list

Type charge

decrementBond
list

Type decrementBond

decrementLonePair
list

Type decrementLonePair

decrementRadical
list

Type decrementRadical

equivalent(*self*, *AtomType other*) → bool
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`.

formBond
list

Type formBond

generic
list

Type generic

getFeatures(*self*) → list
Returns a list of the features that are checked to determine atomtype

incrementBond
list

Type incrementBond

incrementLonePair
list

Type incrementLonePair

incrementRadical
list

Type incrementRadical

isSpecificCaseOf(*self*, *AtomType other*) → bool
Returns `True` if atom type *atomType1* is a specific case of atom type *atomType2* or `False` otherwise.

label
str

Type label

lonePairs
list

Type lonePairs

oDouble
list

Type oDouble

rDouble
list

Type rDouble

sDouble
list

Type sDouble

setActions(*self*, *incrementBond*, *decrementBond*, *formBond*, *breakBond*, *incrementRadical*, *decrementRadical*, *incrementLonePair*, *decrementLonePair*)

single
list

Type single

specific
list

Type specific

triple
list

Type triple

`rmgpy.molecule.`**getAtomType**(*atom*, *dict bonds*) → AtomType
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
General atom types	
`R`	any atom with any local bond structure
`R!H`	any non-hydrogen atom with any local bond structure
Hydrogen atom types	
`H`	hydrogen atom with up to one single bond
Carbon atom types	

Table 2 – continued from previous page

Atom type	Description
`C`	carbon atom with any local bond structure
`Ca`	carbon atom with two lone pairs and no bonds
`Cs`	carbon atom with up to four single bonds
`Csc`	charged carbon atom with up to three single bonds
`Cd`	carbon atom with one double bond (not to O or S) and up to two single bon
`Cdc`	charged carbon atom with one double bond and up to one single bond
`CO`	carbon atom with one double bond to oxygen and up to two single bonds
`CS`	carbon atom with one double bond to sulfur and up to two single bonds
`Cdd`	carbon atom with two double bonds
`Ct`	carbon atom with one triple bond and up to one single bond
`Cb`	carbon atom with up to two benzene bonds and up to one single bond
`Cbf`	carbon atom with three benzene bonds
`C2s`	carbon atom with one lone pair (valance 2) and up to two single bonds
`C2sc`	charged carbon atom with one lone pair (valance 2) and up to three single b
`C2d`	carbon atom with one lone pair (valance 2) and one double bond
`C2dc`	charged carbon atom with one lone pair (valance 2), one double bond and u
`C2tc`	charged carbon atom with one lone pair (valance 2), one triple bond
Nitrogen atom types	
`N`	nitrogen atom with any local bond structure
`N0sc`	charged nitrogen atom with three lone pairs (valance 0) with up to one sing
`N1s`	nitrogen atom with two lone pairs (valance 1) and up to one single bond
`N1sc`	charged nitrogen atom with two lone pairs (valance 1) up to two single bon
`N1dc`	charged nitrogen atom with two lone pairs (valance 1), one double bond
`N3s`	nitrogen atom with one lone pair (valance 3) with up to three single bonds
`N3d`	nitrogen atom with one lone pair (valance 3), one double bond and up to on
`N3t`	nitrogen atom with one lone pair (valance 3) and one triple bond
`N3b`	nitrogen atom with one lone pair (valance 3) and two benzene bonds
`N5sc`	charged nitrogen atom with no lone pairs (valance 5) with up to four single
`N5dc`	charged nitrogen atom with no lone pairs (valance 5), one double bond and
`N5ddc`	charged nitrogen atom with with no lone pairs (valance 5) and two double b
`N5dddc`	charged nitrogen atom with with no lone pairs (valance 5) and three double
`N5tc`	charged nitrogen atom with with no lone pairs (valance 5), one triple bond
`N5b`	nitrogen atom with with no lone pairs (valance 5) and two benzene bonds
`N5bd`	nitrogen atom with with no lone pairs (valance 5), two benzene bonds, and
Oxygen atom types	
`O`	oxygen atom with any local bond structure
`Oa`	oxygen atom with three lone pairs and no bonds
`O0sc`	charged oxygen with three lone pairs (valance 0) and up to one single bond
`O0dc`	charged oxygen atom with three lone pairs (valance 0) and one double bon
`O2s`	oxygen atom with two lone pairs (valance 2) and up to two single bonds
`O2sc`	charged oxygen atom with two lone pairs (valance 2) and up to one single b
`O2d`	oxygen atom with two lone pairs (valance 2) and one doubel bond
`O4sc`	charged oxygen atom with one one pair (valance 4) and up to three single b
`O4dc`	charged oxygen atom with one one pair (valance 4), one double bond and u
`O4tc`	charged oxygen atom with one one pair (valance 4) and one triple bond
`O4b`	oxygen atom with one one pair (valance 4) and and two benzene bonds
Silicon atom types	
`Si`	silicon atom with any local bond structure
`Sis`	silicon atom with four single bonds

Table 2 – continued from previous page

Atom type	Description
`Sid`	silicon atom with one double bond (to carbon) and two single bonds
`SiO`	silicon atom with one double bond (to oxygen) and two single bonds
`Sidd`	silicon atom with two double bonds
`Sit`	silicon atom with one triple bond and one single bond
`Sib`	silicon atom with two benzene bonds and one single bond
`Sibf`	silicon atom with three benzene bonds
Sulfur atom types	
`S`	sulfur atom with any local bond structure
`Sa`	sulfur atom with three lone pairs and no bonds
`S0sc`	charged sulfur atom with three lone pairs (valance 0) and up to one single b
`S2s`	sulfur atom with two lone pairs (valance 2) and up to two single bonds
`S2sc`	charged sulfur atom with two lone pairs (valance 2) and up to three single b
`S2d`	sulfur atom with two lone pairs (valance 2) and one double bond
`S2dc`	charged sulfur atom with two lone pairs (valance 2), one double bond and u
`S2tc`	charged sulfur atom with two lone pairs (valance 2) and one triple bond
`S4s`	sulfur atom with one lone pair (valance 4) and up to four single bonds
`S4sc`	charged sulfur atom with one lone pair (valance 4) and up to five single bon
`S4d`	sulfur atom with one lone pair (valance 4), one double bond and up to two s
`S4dd`	sulfur atom with one lone pair (valance 4) and two double bonds
`S4dc`	charged sulfur atom with one lone pair (valance 4), one to three double bon
`S4b`	sulfur atom with one lone pair (valance 4) and two benzene bonds (one of th
`S4t`	sulfur atom with one lone pair (valance 4), one triple bond and up to one si
`S4tdc`	charged sulfur atom with one lone pair (valance 4) one to two triple bonds,
`S6s`	sulfur atom with no lone pairs (valance 6) and up to six single bonds
`S6sc`	charged sulfur atom with no lone pairs (valance 6) and up to seven single b
`S6d`	sulfur atom with no lone pairs (valance 6), one double bond and up to four
`S6dd`	sulfur atom with no lone pairs (valance 6), two double bonds and up to two
`S6ddd`	sulfur atom with no lone pairs (valance 6) and three double bonds
`S6dc`	charged sulfur atom with no lone pairs (valance 6), one to three double bon
`S6t`	sulfur atom with no lone pairs (valance 6), one triple bond and up to three s
`S6td`	sulfur atom with no lone pairs (valance 6), one triple bond, one double bon
`S6tt`	sulfur atom with no lone pairs (valance 6) and two triple bonds
`S6tdc`	charged sulfur atom with no lone pairs (valance 6), one to two triple bonds,
Chlorine atom types	
`Cl`	chlorine atom with any local bond structure
`Cl1s`	chlorine atom with three lone pairs and zero to one single bonds
Iodine atom types	
`I`	iodine atom with any local bond structure
`I1s`	iodine atom with three lone pairs and zero to one single 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	*center1*, *order*, *center2*	change the bond order of the bond between *center1* and *center2* by *order*; do not break or form bonds
FORM_BOND	*center1*, *order*, *center2*	form a new bond between *center1* and *center2* of type *order*
BREAK_BOND	*center1*, *order*, *center2*	break the bond between *center1* and *center2*, which should be of type *order*
GAIN_RADICAL	*center*, *radical*	increase the number of free electrons on *center* by *radical*
LOSE_RADICAL	*center*, *radical*	decrease the number of free electrons on *center* by *radical*

rmgpy.molecule.Atom

class `rmgpy.molecule.`**`Atom`**(*element=None*, *radicalElectrons=0*, *charge=0*, *label="*, *lonePairs=-100*, *coords=numpy.array([])*, *id=-1*, *props=None*)

An atom. The attributes are:

Attribute	Type	Description
atomType	`AtomType`	The *atom type*
element	`Element`	The chemical element the atom represents
radicalElectrons	`short`	The number of radical electrons
charge	`short`	The formal charge of the atom
label	`str`	A string label that can be used to tag individual atoms
coords	`numpy array`	The (x,y,z) coordinates in Angstrom
lonePairs	`short`	The number of lone electron pairs
id	`int`	Number assignment for atom tracking purposes
bonds	`dict`	Dictionary of bond objects with keys being neighboring atoms
props	`dict`	Dictionary for storing additional atom properties
mass	`int`	atomic mass of element (read only)
number	`int`	atomic number of element (read only)
symbol	`str`	atomic symbol of element (read only)

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`(*self*, *action*)

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*.

`atomType`

rmgpy.molecule.atomtype.AtomType

Type atomType

`charge`

'short'

Type charge

`connectivity1`

'short'

Type connectivity1

connectivity2
'short'

Type connectivity2

connectivity3
'short'

Type connectivity3

coords
numpy.ndarray

Type coords

copy(*self*) → Vertex
Generate a deep copy of the current atom. Modifying the attributes of the copy will not affect the original.

decrementLonePairs(*self*)
Update the lone electron pairs pattern as a result of applying a LOSE_PAIR action.

decrementRadical(*self*)
Update the atom pattern as a result of applying a LOSE_RADICAL action, where *radical* specifies the number of radical electrons to remove.

edges
dict

Type edges

element
rmgpy.molecule.element.Element

Type element

equivalent(*self*, *Vertex other*) → bool
Return `True` if *other* is indistinguishable from this atom, or `False` otherwise. If *other* is an *`Atom`* object, then all attributes except *label* and 'ID' must match exactly. If *other* is an *`GroupAtom`* object, then the atom must match any of the combinations in the atom pattern.

getBondOrdersForAtom(*self*)
This helper function is to help calculate total bond orders for an input atom.

Some special consideration for the order *B* bond. For atoms having three *B* bonds, the order for each is 4/3.0, while for atoms having other than three *B* bonds, the order for each is 3/2.0

get_descriptor(*self*)
Return a tuple used for sorting atoms. Currently uses atomic number, connectivity value, radical electrons, lone pairs, and charge

id
'int'

Type id

ignore
'bool'

Type ignore

incrementLonePairs(*self*)
Update the lone electron pairs pattern as a result of applying a GAIN_PAIR action.

`incrementRadical`(*self*)
Update the atom pattern as a result of applying a GAIN_RADICAL action, where *radical* specifies the number of radical electrons to add.

`isCarbon`(*self*) → bool
Return `True` if the atom represents a carbon atom or `False` if not.

`isChlorine`(*self*) → bool
Return `True` if the atom represents a chlorine atom or `False` if not.

`isFluorine`(*self*) → bool
Return `True` if the atom represents a fluorine atom or `False` if not.

`isHydrogen`(*self*) → bool
Return `True` if the atom represents a hydrogen atom or `False` if not.

`isIodine`(*self*) → bool
Return `True` if the atom represents an iodine atom or `False` if not.

`isNOS`(*self*) → bool
Return `True` if the atom represent either nitrogen, sulfur, or oxygen `False` if it does not.

`isNitrogen`(*self*)
Return `True` if the atom represents a nitrogen atom or `False` if not.

`isNonHydrogen`(*self*) → bool
Return `True` if the atom does not represent a hydrogen atom or `False` if it does.

`isOxygen`(*self*) → bool
Return `True` if the atom represents an oxygen atom or `False` if not.

`isSilicon`(*self*) → bool
Return `True` if the atom represents a silicon atom or `False` if not.

`isSpecificCaseOf`(*self*, *Vertex other*) → bool
Return `True` if *self* is a specific case of *other*, or `False` otherwise. If *other* is an `Atom` object, then this is the same as the `equivalent()` method. If *other* is an `GroupAtom` object, then the atom must match or be more specific than any of the combinations in the atom pattern.

`isSulfur`(*self*) → bool
Return `True` if the atom represents a sulfur atom or `False` if not.

`label`
str

Type label

`lonePairs`
'short'

Type lonePairs

`mapping`
rmgpy.molecule.graph.Vertex

Type mapping

`props`
dict

Type props

`radicalElectrons`
'short'

Type radicalElectrons

`resetConnectivityValues`(*self*)
Reset the cached structure information for this vertex.

`setLonePairs`(*self*, *int lonePairs*)
Set the number of lone electron pairs.

`sortingLabel`
'short'

Type sortingLabel

`terminal`
'bool'

Type terminal

`updateCharge`(*self*)
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`**(*atom1*, *atom2*, *order=1*)
A chemical bond. The attributes are:

Attribute	Type	Description
order	`float`	The *bond type*
atom1	`Atom`	An Atom object connecting to the bond
atom2	`Atom`	An Atom object connecting to the bond

`applyAction`(*self*, *action*)
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`(*self*) → Edge
Generate a deep copy of the current bond. Modifying the attributes of the copy will not affect the original.

`decrementOrder`(*self*)
Update the bond as a result of applying a CHANGE_BOND action to decrease the order by one.

`equivalent`(*self*, *Edge other*) → bool
Return `True` if *other* is indistinguishable from this bond, or `False` otherwise. *other* can be either a `Bond` or a `GroupBond` object.

`getOrderNum`(*self*) → float
returns the bond order as a number

`getOrderStr`(*self*) → str
returns a string representing the bond order

`getOtherVertex`(*self*, *Vertex vertex*) → Vertex
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`(*self*)
Update the bond as a result of applying a CHANGE_BOND action to increase the order by one.

`isBenzene`(*self*) → bool
Return `True` if the bond represents a benzene bond or `False` if not.

isDouble(*self*) → bool
Return `True` if the bond represents a double bond or `False` if not.

isHydrogenBond(*self*)
Return `True` if the bond represents a hydrogen bond or `False` if not.

isOrder(*self*, *float otherOrder*) → bool
Return `True` if the bond is of order otherOrder or `False` if not. This compares floats that takes into account floating point error

NOTE: we can replace the absolute value relation with math.isclose when we swtich to python 3.5+

isSingle(*self*) → bool
Return `True` if the bond represents a single bond or `False` if not.

isSpecificCaseOf(*self*, *Edge other*) → bool
Return `True` if *self* is a specific case of *other*, or `False` otherwise. *other* can be either a `Bond` or a `GroupBond` object.

isTriple(*self*) → bool
Return `True` if the bond represents a triple bond or `False` if not.

order
'float'

Type order

setOrderNum(*self*, *float newOrder*)
change the bond order with a number

setOrderStr(*self*, *str newOrder*)
set the bond order using a valid bond-order character

vertex1
rmgpy.molecule.graph.Vertex

Type vertex1

vertex2
rmgpy.molecule.graph.Vertex

Type vertex2

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**(*atoms=None*, *symmetry=-1*, *multiplicity=-187*, *reactive=True*, *props=None*, *SMILES=''*)
A representation of a molecular structure using a graph data type, extending the `Graph` class. Attributes are:

Attribute	Type	Description
symmetryNumber	`float`	The (estimated) external + internal symmetry number of the molecule, modified for chirality
multiplicity	`int`	The multiplicity of this species, multiplicity = 2*total_spin+1
reactive	`bool`	**`True` (by default) if the molecule participates in re** It is set to `False` by the filtration functions if a non representative resonance structure was generated by a template reaction
props	`dict`	A list of properties describing the state of the molecule.
InChI	`str`	A string representation of the molecule in InChI
atoms	`list`	A list of Atom objects in the molecule
fingerprint	`str`	A representation for fast comparison, set as molecular formula

A new molecule object can be easily instantiated by passing the *SMILES* or *InChI* string representing the molecular structure.

`InChI`
str

Type InChI

`addAtom`(*self*, *Atom atom*)
Add an *atom* to the graph. The atom is initialized with no bonds.

`addBond`(*self*, *Bond bond*)
Add a *bond* to the graph as an edge connecting the two atoms *atom1* and *atom2*.

`addEdge`(*self*, *Edge edge*) → Edge
Add an *edge* to the graph. The two vertices in the edge must already exist in the graph, or a `ValueError` is raised.

`addVertex`(*self*, *Vertex vertex*) → Vertex
Add a *vertex* to the graph. The vertex is initialized with no edges.

`assignAtomIDs`(*self*)
Assigns an index to every atom in the molecule for tracking purposes. Uses entire range of cython's integer values to reduce chance of duplicates

`atomIDValid`(*self*) → bool
Checks to see if the atom IDs are valid in this structure

`atoms`
Molecule.__getAtoms(self)

`calculateCp0`(*self*) → double
Return the value of the heat capacity at zero temperature in J/mol*K.

calculateCpInf(*self*) → double
Return the value of the heat capacity at infinite temperature in J/mol*K.

calculateSymmetryNumber(*self*) → float
Return the symmetry number for the structure. The symmetry number includes both external and internal modes.

clearLabeledAtoms(*self*)
Remove the labels from all atoms in the molecule.

connectTheDots(*self*)
Delete all bonds, and set them again based on the Atoms' coords. Does not detect bond type.

containsLabeledAtom(*self*, *str label*) → bool
Return `True` if the molecule contains an atom with the label *label* and `False` otherwise.

copy(*self*, *bool deep=False*) → Graph
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.

copyAndMap(*self*) → dict
Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from Graph.copy() method

countInternalRotors(*self*) → int
Determine the number of internal rotors in the structure. Any single bond not in a cycle and between two atoms that also have other bonds are considered to be internal rotors.

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

findIsomorphism(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
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(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
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.

find_H_bonds(*self*)
generates a list of (new-existing H bonds ignored) possible Hbond coordinates [(i1,j1),(i2,j2),. . .] where i and j values correspond to the indexes of the atoms involved, Hbonds are allowed if they meet the following constraints:

1) between a H and [O,N] atoms
2) the hydrogen is covalently bonded to an O or N

3) the Hydrogen bond must complete a ring with at least 5 members
4) An atom can only be hydrogen bonded to one other atom

`fingerprint`
Molecule.__getFingerprint(self)

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.

`fromAdjacencyList`(*self*, *str adjlist*, *bool saturateH=False*)
Convert a string adjacency list *adjlist* to a molecular structure. Skips the first line (assuming it's a label) unless *withLabel* is `False`.

`fromAugmentedInChI`(*self*, *aug_inchi*)
Convert an Augmented InChI string *aug_inchi* to a molecular structure.

`fromInChI`(*self*, *str inchistr*, *backend='try-all'*)
Convert an InChI string *inchistr* to a molecular structure.

`fromSMARTS`(*self*, *smartsstr*)
Convert a SMARTS string *smartsstr* to a molecular structure. Uses RDKit to perform the conversion. This Kekulizes everything, removing all aromatic atom types.

`fromSMILES`(*self*, *str smilesstr*, *backend='try-all'*)
Convert a SMILES string *smilesstr* to a molecular structure.

`fromXYZ`(*self*, *ndarray atomicNums*, *ndarray coordinates*)
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.

`generate_H_bonded_structures`(*self*)
generates a list of Hbonded molecular structures in addition to the constraints on Hydrogen bonds applied in the find_H_Bonds function the generated structures are constrained to:

1) An atom can only be hydrogen bonded to one other atom
2) Only two H-bonds can exist in a given molecule

the second is done to avoid explosive growth in the number of structures as without this constraint the number of possible structures grows 2^n where n is the number of possible H-bonds

`generate_resonance_structures`(*self*, *bool keep_isomorphic=False*, *bool filter_structures=True*) → list
Returns a list of resonance structures of the molecule.

`getAllCycles`(*self*, *Vertex startingVertex*) → list
Given a starting vertex, returns a list of all the cycles containing that vertex.

This function returns a duplicate of each cycle because [0,1,2,3] is counted as separate from [0,3,2,1]

`getAllCyclesOfSize`(*self*, *int size*) → list
Return a list of the all non-duplicate rings with length 'size'. 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).

`getAllCyclicVertices`(*self*) → list
Returns all vertices belonging to one or more cycles.

getAllEdges(*self*) → list
Returns a list of all edges in the graph.

getAllPolycyclicVertices(*self*) → list
Return all vertices belonging to two or more cycles, fused or spirocyclic.

getAllSimpleCyclesOfSize(*self*, *int size*) → list
Return a list of all non-duplicate monocyclic rings with length 'size'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

getAromaticRings(*self*, *list rings=None*) → tuple
Returns all aromatic rings as a list of atoms and a list of bonds.

Identifies rings using *Graph.getSmallestSetOfSmallestRings()*, then uses RDKit to perceive aromaticity. RDKit uses an atom-based pi-electron counting algorithm to check aromaticity based on Huckel's Rule. Therefore, this method identifies "true" aromaticity, rather than simply the RMG bond type.

The method currently restricts aromaticity to six-membered carbon-only rings. This is a limitation imposed by RMG, and not by RDKit.

getBond(*self*, *Atom atom1*, *Atom atom2*) → Bond
Returns the bond connecting atoms *atom1* and *atom2*.

getBonds(*self*, *Atom atom*) → dict
Return a dictionary of the bonds involving the specified *atom*.

getChargeSpan(*self*)
Iterate through the atoms in the structure and calculate the charge span on the overall molecule. The charge span is a measure of the number of charge separations in a molecule.

getDeterministicSmallestSetOfSmallestRings(*self*) → list
Modified *Graph* method *getSmallestSetOfSmallestRings* by sorting calculated cycles by short lenth and then high atomic number instead of just short length (for cases where multiple cycles with same length are found, *getSmallestSetOfSmallestRings* outputs non-determinstically).

For instance, molecule with this SMILES: C1CC2C3CSC(CO3)C2C1, will have non-deterministic output from *getSmallestSetOfSmallestRings*, which leads to non-deterministic bycyclic decomposition Using this new method can effectively prevent this situation.

Important Note: This method returns an incorrect set of SSSR in certain molecules (such as cubane). It is recommended to use the main *Graph.getSmallestSetOfSmallestRings* method in new applications. Alternatively, consider using *Graph.getRelevantCycles* for deterministic output.

In future development, this method should ideally be replaced by some method to select a deterministic set of SSSR from the set of Relevant Cycles, as that would be a more robust solution.

getDisparateRings(*self*) → tuple
Get all disjoint monocyclic and polycyclic cycle clusters in the molecule. Takes the RC and recursively merges all cycles which share vertices.

Returns: monocyclic_cycles, polycyclic_cycles

getEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → Edge
Returns the edge connecting vertices *vertex1* and *vertex2*.

getEdges(*self*, *Vertex vertex*) → dict
Return a dictionary of the edges involving the specified *vertex*.

getFormula(*self*) → str
Return the molecular formula for the molecule.

getLabeledAtom(*self*, *str label*) → Atom
Return the atoms in the molecule that are labeled.

getLabeledAtoms(*self*) → dict
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.

getLargestRing(*self*, *Vertex vertex*) → list
returns the largest ring containing vertex. This is typically useful for finding the longest path in a polycyclic ring, since the polycyclic rings returned from getPolycyclicRings are not necessarily in order in the ring structure.

getMaxCycleOverlap(*self*) → int
Return the maximum number of vertices that are shared between any two cycles in the graph. For example, if there are only disparate monocycles or no cycles, the maximum overlap is zero; if there are “spiro” cycles, it is one; if there are “fused” cycles, it is two; and if there are “bridged” cycles, it is three.

getMolecularWeight(*self*) → double
Return the molecular weight of the molecule in kg/mol.

getMonocyclicRings(*self*) → list
Return a list of cycles that are monocyclic.

getNetCharge(*self*)
Iterate through the atoms in the structure and calculate the net charge on the overall molecule.

getNthNeighbor(*self*, *startingAtoms*, *distanceList*, *ignoreList=None*, *n=1*)
Recursively get the Nth nonHydrogen neighbors of the startingAtoms, and return them in a list. *startingAtoms* is a list of :class:Atom for which we will get the nth neighbor. *distanceList* is a list of intergers, corresponding to the desired neighbor distances. *ignoreList* is a list of :class:Atom that have been counted in (n-1)th neighbor, and will not be returned. *n* is an interger, corresponding to the distance to be calculated in the current iteration.

getNumAtoms(*self*, *str element=None*) → int
Return the number of atoms in molecule. If element is given, ie. “H” or “C”, the number of atoms of that element is returned.

getPolycyclicRings(*self*) → list
Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

getRadicalAtoms(*self*)
Return the atoms in the molecule that have unpaired electrons.

getRadicalCount(*self*) → short
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.

getRelevantCycles(*self*) → list
Returns the set of relevant cycles as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

getSingletCarbeneCount(*self*) → short
Return the total number of singlet carbenes (lone pair on a carbon atom) in the molecule. Counts the number of carbon atoms with a lone pair. In the case of [C] with two lone pairs, this method will return 1.

getSmallestSetOfSmallestRings(*self*) → list
Returns the smallest set of smallest rings as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

getSymmetryNumber(*self*)
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(*self*)
Get a URL to the molecule's info page on the RMG website.

get_edges_in_cycle(*self*, *list vertices*, *bool sort=False*) → list
For a given list of atoms comprising a ring, return the set of bonds connecting them, in order around the ring.

If *sort=True*, then sort the vertices to match their connectivity. Otherwise, assumes that they are already sorted, which is true for cycles returned by getRelevantCycles or getSmallestSetOfSmallestRings.

get_element_count(*self*) → dict
Returns the element count for the molecule as a dictionary.

hasAtom(*self*, *Atom atom*) → bool
Returns `True` if *atom* is an atom in the graph, or `False` if not.

hasBond(*self*, *Atom atom1*, *Atom atom2*) → bool
Returns `True` if atoms *atom1* and *atom2* are connected by an bond, or `False` if not.

hasEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → bool
Returns `True` if vertices *vertex1* and *vertex2* are connected by an edge, or `False` if not.

hasVertex(*self*, *Vertex vertex*) → bool
Returns `True` if *vertex* is a vertex in the graph, or `False` if not.

has_lone_pairs(*self*) → bool
Return `True` if the molecule contains at least one lone electron pair, or `False` otherwise.

identifyRingMembership(*self*)
Performs ring perception and saves ring membership information to the Atom.props attribute.

implicitHydrogens
'bool'

Type implicitHydrogens

isAromatic(*self*)
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.

isArylRadical(*self*, *list aromaticRings=None*) → bool
Return `True` if the molecule only contains aryl radicals, ie. radical on an aromatic ring, or `False` otherwise.

isAtomInCycle(*self*, *Atom atom*) → bool
Return `True` if *atom* is in one or more cycles in the structure, and `False` if not.

isBondInCycle(*self*, *Bond bond*) → bool
Return `True` if the bond between atoms *atom1* and *atom2* is in one or more cycles in the graph, or `False` if not.

isCyclic(*self*) → bool
Return `True` if one or more cycles are present in the graph or `False` otherwise.

isEdgeInCycle(*self*, *Edge edge*) → bool
Return `True` if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or `False` if not.

isIdentical(*self*, *Molecule other*) → bool
Performs isomorphism checking, with the added constraint that atom IDs must match.

Primary use case is tracking atoms in reactions for reaction degeneracy determination.

Returns `True` if two graphs are identical and `False` otherwise.

isIsomorphic(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → bool
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(*self*) → bool
Return `True` if the structure is linear and `False` otherwise.

isMappingValid(*self*, *Graph other*, *dict mapping*, *bool equivalent=True*) → bool
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. If equivalent is true it checks if atoms and edges are equivalent, if false it checks if they are specific cases of each other.

isRadical(*self*) → bool
Return `True` if the molecule contains at least one radical electron, or `False` otherwise.

isSubgraphIsomorphic(*self*, *Graph other*, *dict initialMap=None*, *bool generateInitialMap=False*, *bool saveOrder=False*) → bool
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(*self*, *Vertex vertex*) → bool
Return `True` if the given *vertex* is contained in one or more cycles in the graph, or `False` if not.

is_equal(*self*, *other*)
Method to test equality of two Molecule objects.

kekulize(*self*)
Kekulizes an aromatic molecule.

merge(*self*, *Graph other*) → Graph
Merge two molecules so as to store them in a single `Molecule` object. The merged `Molecule` object is returned.

multiplicity
'int'

Type multiplicity

ordered_vertices
list

Type ordered_vertices

props
dict

Type props

rdMol
object

Type rdMol

rdMolConfId
'int'

Type rdMolConfId

reactive
'bool'

Type reactive

removeAtom(*self*, *Atom atom*)
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(*self*, *Bond bond*)
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(*self*, *Edge edge*)
Remove the specified *edge* from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

removeVertex(*self*, *Vertex vertex*)
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.

remove_H_bonds(*self*)
removes any present hydrogen bonds from the molecule

resetConnectivityValues(*self*)
Reset any cached connectivity information. Call this method when you have modified the graph.

restore_vertex_order(*self*)
reorder the vertices to what they were before sorting if you saved the order

saturate_radicals(*self*)
Saturate the molecule by replacing all radicals with bonds to hydrogen atoms. Changes self molecule object.

saturate_unfilled_valence(*self*, *update=True*)
Saturate the molecule by adding H atoms to any unfilled valence

sortAtoms(*self*)
Sort the atoms in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

This function orders atoms using several attributes in atom.getDescriptor(). Currently it sorts by placing heaviest atoms first and hydrogen atoms last. Placing hydrogens last during sorting ensures that functions with hydrogen removal work properly.

sortVertices(*self*, *bool saveOrder=False*)
Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

`split`(*self*) → list
Convert a single `Molecule` object containing two or more unconnected molecules into separate class:*Molecule* objects.

`symmetryNumber`
'float'

Type symmetryNumber

`toAdjacencyList`(*self*, *str label=''*, *bool removeH=False*, *bool removeLonePairs=False*, *bool oldStyle=False*)
Convert the molecular structure to a string adjacency list.

`toAugmentedInChI`(*self*) → str
Adds an extra layer to the InChI denoting the multiplicity of the molecule.

Separate layer with a forward slash character.

`toAugmentedInChIKey`(*self*) → str
Adds an extra layer to the InChIKey denoting the multiplicity of the molecule.

Simply append the multiplicity string, do not separate by a character like forward slash.

`toGroup`(*self*)
This method converts a list of atoms in a Molecule to a Group object.

`toInChI`(*self*) → str
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`(*self*) → str
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.

`toRDKitMol`(*self*, **args*, ***kwargs*)
Convert a molecular structure to a RDKit rdmol object.

`toSMARTS`(*self*)
Convert a molecular structure to an SMARTS string. Uses RDKit to perform the conversion. Perceives aromaticity and removes Hydrogen atoms.

`toSMILES`(*self*) → str
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`(*self*)
Returns a copy of the current molecule, consisting of only single bonds.

This is useful for isomorphism comparison against something that was made via fromXYZ, which does not attempt to perceive bond orders

update(*self*, *log_species=True*)
Update connectivity values, atom types of atoms. Update multiplicity, and sort atoms using the new connectivity values.

updateAtomTypes(*self*, *bool logSpecies=True*, *bool raiseException=True*)
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).

If *raiseException* is *False*, then the generic atomType 'R' will be prescribed to any atom when getAtomType fails. Currently used for resonance hybrid atom types.

updateConnectivityValues(*self*)
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

updateLonePairs(*self*)
Iterate through the atoms in the structure and calculate the number of lone electron pairs, assuming a neutral molecule.

updateMultiplicity(*self*)
Update the multiplicity of a newly formed molecule.

vertices
list

Type vertices

rmgpy.molecule.GroupAtom

class rmgpy.molecule.**GroupAtom**(*atomType=None*, *radicalElectrons=None*, *charge=None*, *label="*, *lonePairs=None*, *props=None*)
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
atomType	`list`	The allowed atom types (as `AtomType` objects)
radicalElectrons	`list`	The allowed numbers of radical electrons (as short integers)
charge	`list`	The allowed formal charges (as short integers)
label	`str`	A string label that can be used to tag individual atoms
lonePairs	`list`	The number of lone electron pairs
'charge'	''list''	The partial charge of the atom
props	`dict`	Dictionary for storing additional atom properties
reg_dim_atm	`list`	List of atom types that are free dimensions in tree optimization
reg_dim_u	`list`	List of unpaired electron numbers that are free dimensions in tree optimization

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(*self*, *list action*)
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*.

atomType
list

Type atomType

charge
list

Type charge

connectivity1
'short'

Type connectivity1

connectivity2
'short'

Type connectivity2

connectivity3
'short'

Type connectivity3

copy(*self*) → Vertex
Return a deep copy of the `GroupAtom` object. Modifying the attributes of the copy will not affect the original.

countBonds(*self*, *wildcards=False*) → list
Returns: list of the number of bonds currently on the :class:GroupAtom

If the argument wildcards is turned off then any bonds with multiple options for bond orders will not be counted

edges
dict

Type edges

equivalent(*self*, *Vertex other*) → bool
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`.

hasWildcards(*self*) → bool
Return `True` if the atom has wildcards in any of the attributes: atomtype, electronpairs, lone pairs, charge, and bond order. Returns ''False" if no attribute has wildcards.

ignore
'bool'

Type ignore

isOxygen(*self*) → bool
Return `True` if the atom represents an oxygen atom or `False` if not.

isSpecificCaseOf(*self*, *Vertex other*) → bool
Returns `True` if *self* is the same as *other* or is a more specific case of *other*. Returns `False` if some of *self* is not included in *other* or they are mutually exclusive.

isSulfur(*self*) → bool
Return `True` if the atom represents an sulfur atom or `False` if not.

label
str

Type label

lonePairs
list

Type lonePairs

makeSampleAtom(*self*) → Atom
Returns: a class :Atom: object analagous to the GroupAtom

This makes a sample, so it takes the first element when there are multiple options inside of self.atomtype, self.radicalElectrons, self.lonePairs, and self.charge

mapping
rmgpy.molecule.graph.Vertex

Type mapping

props
dict

Type props

radicalElectrons
list

Type radicalElectrons

reg_dim_atm
list

Type reg_dim_atm

reg_dim_u
list

Type reg_dim_u

resetConnectivityValues(*self*)
Reset the cached structure information for this vertex.

sortingLabel
'short'

Type sortingLabel

terminal
'bool'

Type terminal

rmgpy.molecule.GroupBond

class rmgpy.molecule.**GroupBond**(*atom1*, *atom2*, *order=None*)
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
order	`list`	The allowed bond orders (as character strings)
reg_dim	`Boolean`	Indicates if this is a regularization dimension during tree generation

Each list represents a logical OR construct, i.e. a bond will match the group if it matches *any* item in the list.

applyAction(*self*, *list action*)
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(*self*) → Edge
Return a deep copy of the `GroupBond` object. Modifying the attributes of the copy will not affect the original.

equivalent(*self*, *Edge other*) → bool
Returns `True` if *other* is equivalent to *self* or `False` if not, where *other* can be either an `Bond` or an `GroupBond` object.

getOrderNum(*self*) → list
returns the bond order as a list of numbers

getOrderStr(*self*) → list
returns a list of strings representing the bond order

getOtherVertex(*self*, *Vertex vertex*) → Vertex
Given a vertex that makes up part of the edge, return the other vertex. Raise a `ValueError` if the given vertex is not part of the edge.

isBenzene(*self*, *bool wildcards=False*) → bool
Return `True` if the bond represents a benzene bond or `False` if not. If *wildcards* is `False` we return False anytime there is more than one bond order, otherwise we return `True` if any of the options are benzene

isDouble(*self*, *bool wildcards=False*) → bool
Return `True` if the bond represents a double bond or `False` if not. If *wildcards* is `False` we return False anytime there is more than one bond order, otherwise we return `True` if any of the options are double.

isHydrogenBond(*self*, *wildcards=False*)
Return `True` if the bond represents a hydrogen bond or `False` if not. If *wildcards* is `False` we return False anytime there is more than one bond order, otherwise we return `True` if any of the options are hydrogen bonds.

isSingle(*self*, *bool wildcards=False*) → bool
Return `True` if the bond represents a single bond or `False` if not. If *wildcards* is `False` we return False anytime there is more than one bond order, otherwise we return `True` if any of the options are single.

NOTE: we can replace the absolute value relation with math.isclose when we swtich to python 3.5+

isSpecificCaseOf(*self*, *Edge other*) → bool
Returns `True` if *other* is the same as *self* or is a more specific case of *self*. Returns `False` if some of *self* is not included in *other* or they are mutually exclusive.

isTriple(*self*, *bool wildcards=False*) → bool
Return `True` if the bond represents a triple bond or `False` if not. If *wildcards* is `False` we return False anytime there is more than one bond order, otherwise we return `True` if any of the options are triple.

makeBond(*self*, *Molecule molecule*, *Atom atom1*, *Atom atom2*)
Creates a :class: Bond between atom1 and atom2 analogous to self

The intended input arguments should be class :Atom: not class :GroupAtom: :param atom1: First :class: Atom the bond connects :param atom2: Second :class: Atom the bond connects

order
list

Type order

reg_dim
list

Type reg_dim

setOrderNum(*self*, *list newOrder*)
change the bond order with a list of numbers

setOrderStr(*self*, *list newOrder*)
set the bond order using a valid bond-order character list

vertex1
rmgpy.molecule.graph.Vertex

Type vertex1

vertex2
rmgpy.molecule.graph.Vertex

Type vertex2

rmgpy.molecule.Group

class rmgpy.molecule.**Group**(*atoms=None*, *props=None*, *multiplicity=None*)
A representation of a molecular substructure group using a graph data type, extending the `Graph` class. The attributes are:

Attribute	Type	Description
atoms	`list`	Aliases for the *vertices* storing `GroupAtom`
multiplicity	`list`	Range of multiplicities accepted for the group
props	`dict`	Dictionary of arbitrary properties/flags classifying state of Group object

Corresponding alias methods to Molecule have also been provided.

addAtom(*self*, *GroupAtom atom*)
Add an *atom* to the graph. The atom is initialized with no bonds.

addBond(*self*, *GroupBond bond*)
Add a *bond* to the graph as an edge connecting the two atoms *atom1* and *atom2*.

addEdge(*self*, *Edge edge*) → Edge
Add an *edge* to the graph. The two vertices in the edge must already exist in the graph, or a `ValueError` is raised.

addExplicitLigands(*self*) → bool
This function O2d/S2d ligand to CO or CS atomtypes if they are not already there.

Returns a 'True' if the group was modified otherwise returns 'False'

addImplicitAtomsFromAtomType(*self*) → Group
Returns: a modified group with implicit atoms added Add implicit double/triple bonded atoms O, S or R, for which we will use a C

Not designed to work with wildcards

addImplicitBenzene(*self*) → Group
Returns: A modified group with any implicit benzene rings added

This method currently does not if there are wildcards in atomtypes or bond orders The current algorithm also requires that all Cb and Cbf are atomtyped

There are other cases where the algorithm doesn't work. For example whenever there are many dangling Cb or Cbf atoms not in a ring, it is likely fail. In the database test (the only use thus far), we will require that any group with more than 3 Cbfs have complete rings. This is much stricter than this method can handle, but right now this method cannot handle very general cases, so it is better to be conservative.

addVertex(*self*, *Vertex vertex*) → Vertex
Add a *vertex* to the graph. The vertex is initialized with no edges.

atoms
Group.__getAtoms(self)

classifyBenzeneCarbons(*self*, *dict partners=None*) → tuple

Parameters

- **group** – :class:Group with atoms to classify
- **partners** – dictionary of partnered up atoms, which must be a cbf atom

Returns: tuple with lists of each atom classification

clearLabeledAtoms(*self*)
Remove the labels from all atoms in the molecular group.

containsLabeledAtom(*self*, *str label*) → bool
Return `True` if the group contains an atom with the label *label* and `False` otherwise.

copy(*self*, *bool deep=False*) → Graph
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.

copyAndMap(*self*) → dict
Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from Graph.copy() method

createAndConnectAtom(*self*, *list atomtypes*, *GroupAtom connectingAtom*, *list bondOrders*) → GroupAtom
This method creates an non-radical, uncharged, :class:GroupAtom with specified list of atomtypes and connects it to one atom of the group, 'connectingAtom'. This is useful for making sample atoms.

Parameters

- **atomtypes** – list of atomtype labels (strs)
- **connectingAtom** – :class:GroupAtom that is connected to the new benzene atom
- **bondOrders** – list of bond Orders connecting newAtom and connectingAtom

Returns: the newly created atom

draw(*self*, *format*)
Use pydot to draw a basic graph of the group.

Use format to specify the desired output format, eg. 'png', 'svg', 'ps', 'pdf', 'plain', etc.

elementCount
dict

Type elementCount

findIsomorphism(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
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(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → list
Returns `True` if *other* is subgraph isomorphic and `False` otherwise. In other words, return `True` is self is more specific than other. Also returns the lists all of valid mappings. The *initialMap* attribute can be used to specify a required mapping from *self* to *other* (i.e. the atoms of *self* are the keys, while the atoms of *other* are the values). The returned mappings also use the atoms of *self* for the keys and the atoms of *other* for the values. The *other* parameter must be a `Group` object, or a `TypeError` is raised.

fromAdjacencyList(*self*, *str adjlist*)
Convert a string adjacency list *adjlist* to a molecular structure. Skips the first line (assuming it's a label) unless *withLabel* is `False`.

getAllCycles(*self*, *Vertex startingVertex*) → list
Given a starting vertex, returns a list of all the cycles containing that vertex.

This function returns a duplicate of each cycle because [0,1,2,3] is counted as separate from [0,3,2,1]

getAllCyclesOfSize(*self*, *int size*) → list
Return a list of the all non-duplicate rings with length 'size'. 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).

getAllCyclicVertices(*self*) → list
Returns all vertices belonging to one or more cycles.

getAllEdges(*self*) → list
Returns a list of all edges in the graph.

getAllPolycyclicVertices(*self*) → list
Return all vertices belonging to two or more cycles, fused or spirocyclic.

getAllSimpleCyclesOfSize(*self*, *int size*) → list
Return a list of all non-duplicate monocyclic rings with length 'size'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

getBond(*self*, *GroupAtom atom1*, *GroupAtom atom2*) → GroupBond
Returns the bond connecting atoms *atom1* and *atom2*.

getBonds(*self*, *GroupAtom atom*) → dict
Return a list of the bonds involving the specified *atom*.

getDisparateRings(*self*) → tuple
Get all disjoint monocyclic and polycyclic cycle clusters in the molecule. Takes the RC and recursively merges all cycles which share vertices.

Returns: monocyclic_cycles, polycyclic_cycles

getEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → Edge
Returns the edge connecting vertices *vertex1* and *vertex2*.

getEdges(*self*, *Vertex vertex*) → dict
Return a dictionary of the edges involving the specified *vertex*.

getExtensions(*self*, *R=None*, *basename="*, *atmInd=None*, *atmInd2=None*)
generate all allowed group extensions and their complements note all atomtypes except for elements and R/R!H's must be removed

getLabeledAtom(*self*, *str label*) → GroupAtom
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(*self*) → dict
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.

getLargestRing(*self*, *Vertex vertex*) → list
returns the largest ring containing vertex. This is typically useful for finding the longest path in a polycyclic ring, since the polycyclic rings returned from getPolycyclicRings are not necessarily in order in the ring structure.

getMaxCycleOverlap(*self*) → int
Return the maximum number of vertices that are shared between any two cycles in the graph. For example, if there are only disparate monocycles or no cycles, the maximum overlap is zero; if there are "spiro" cycles, it is one; if there are "fused" cycles, it is two; and if there are "bridged" cycles, it is three.

getMonocyclicRings(*self*) → list
Return a list of cycles that are monocyclic.

getNetCharge(*self*)
Iterate through the atoms in the group and calculate the net charge

getPolycyclicRings(*self*) → list
Return a list of cycles that are polycyclic. In other words, merge the cycles which are fused or spirocyclic into a single polycyclic cycle, and return only those cycles. Cycles which are not polycyclic are not returned.

getRelevantCycles(*self*) → list
Returns the set of relevant cycles as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

getSmallestSetOfSmallestRings(*self*) → list
Returns the smallest set of smallest rings as a list of lists. Uses RingDecomposerLib for ring perception.

Kolodzik, A.; Urbaczek, S.; Rarey, M. Unique Ring Families: A Chemically Meaningful Description of Molecular Ring Topologies. J. Chem. Inf. Model., 2012, 52 (8), pp 2013-2021

Flachsenberg, F.; Andresen, N.; Rarey, M. RingDecomposerLib: An Open-Source Implementation of Unique Ring Families and Other Cycle Bases. J. Chem. Inf. Model., 2017, 57 (2), pp 122-126

get_edges_in_cycle(*self*, *list vertices*, *bool sort=False*) → list
For a given list of atoms comprising a ring, return the set of bonds connecting them, in order around the ring.

If *sort=True*, then sort the vertices to match their connectivity. Otherwise, assumes that they are already sorted, which is true for cycles returned by getRelevantCycles or getSmallestSetOfSmallestRings.

get_element_count(*self*) → dict
Returns the element count for the molecule as a dictionary. Wildcards are not counted as any particular element.

hasAtom(*self*, *GroupAtom atom*) → bool
Returns `True` if *atom* is an atom in the graph, or `False` if not.

hasBond(*self*, *GroupAtom atom1*, *GroupAtom atom2*) → bool
Returns `True` if atoms *atom1* and *atom2* are connected by an bond, or `False` if not.

hasEdge(*self*, *Vertex vertex1*, *Vertex vertex2*) → bool
Returns `True` if vertices *vertex1* and *vertex2* are connected by an edge, or `False` if not.

hasVertex(*self*, *Vertex vertex*) → bool
Returns `True` if *vertex* is a vertex in the graph, or `False` if not.

isAromaticRing(*self*) → bool
This method returns a boolean telling if the group has a 5 or 6 cyclic with benzene bonds exclusively

isBenzeneExplicit(*self*) → bool
Returns: 'True' if all Cb, Cbf atoms are in completely explicitly stated benzene rings.

Otherwise return 'False'

isCyclic(*self*) → bool
Return `True` if one or more cycles are present in the graph or `False` otherwise.

isEdgeInCycle(*self*, *Edge edge*) → bool
Return `True` if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or `False` if not.

isIdentical(*self*, *Graph other*, *bool saveOrder=False*) → bool
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(*self*, *Graph other*, *dict initialMap=None*, *bool saveOrder=False*) → bool
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(*self*, *Graph other*, *dict mapping*, *bool equivalent=True*) → bool
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. If equivalent is true it checks if atoms and edges are equivalent, if false it checks if they are specific cases of each other.

isSubgraphIsomorphic(*self*, *Graph other*, *dict initialMap=None*, *bool generateInitialMap=False*, *bool saveOrder=False*) → bool
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(*self*, *Vertex vertex*) → bool
Return `True` if the given *vertex* is contained in one or more cycles in the graph, or `False` if not.

makeSampleMolecule(*self*) → Molecule
Returns: A sample class :Molecule: from the group

merge(*self*, *Graph other*) → Graph
Merge two groups so as to store them in a single `Group` object. The merged `Group` object is returned.

mergeGroups(*self*, *Group other*) → Group
This function takes *other* :class:Group object and returns a merged :class:Group object based on overlapping labeled atoms between self and other

Currently assumes *other* can be merged at the closest labelled atom

multiplicity
list

Type multiplicity

ordered_vertices
list

Type ordered_vertices

`pickWildcards`(*self*)

Returns: the :class:Group object without wildcards in either atomtype or bonding

This function will naively pick the first atomtype for each atom, but will try to pick bond orders that make sense given the selected atomtypes

`props`

dict

Type props

`radicalCount`

'short'

Type radicalCount

`removeAtom`(*self*, *GroupAtom atom*)

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`(*self*, *GroupBond bond*)

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`(*self*, *Edge edge*)

Remove the specified *edge* from the graph. Does not remove vertices that no longer have any edges as a result of this removal.

`removeVertex`(*self*, *Vertex vertex*)

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`(*self*)

Reset any cached connectivity information. Call this method when you have modified the graph.

`resetRingMembership`(*self*)

Resets ring membership information in the GroupAtom.props attribute.

`restore_vertex_order`(*self*)

reorder the vertices to what they were before sorting if you saved the order

`sortAtoms`(*self*)

Sort the atoms in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

`sortByConnectivity`(*self*, *list atomList*) → list

Parameters **`atomList`** – input list of atoms

Returns: a sorted list of atoms where each atom is connected to a previous atom in the list if possible

`sortVertices`(*self*, *bool saveOrder=False*)

Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

`specifyAtomExtensions`(*self*, *i*, *basename*, *R*)

generates extensions for specification of the type of atom defined by a given atomtype or set of atomtypes

`specifyBondExtensions`(*self*, *i*, *j*, *basename*, *Rbonds*)

generates extensions for the specification of bond order for a given bond

specifyExternalNewBondExtensions(*self*, *i*, *basename*, *Rbonds*)
generates extensions for the creation of a bond (of undefined order) between an atom and a new atom that is not H

specifyInternalNewBondExtensions(*self*, *i*, *j*, *Nsplits*, *basename*, *Rbonds*)
generates extensions for creation of a bond (of undefined order) between two atoms indexed i,j that already exist in the group and are unbonded

specifyUnpairedExtensions(*self*, *i*, *basename*, *Run*)
generates extensions for specification of the number of electrons on a given atom

split(*self*) → list
Convert a single `Group` object containing two or more unconnected groups into separate class:*Group* objects.

standardizeAtomType(*self*) → bool
This function changes the atomTypes in a group if the atom must be a specific atomType based on its bonds and valency.

Currently only standardizes oxygen, carbon and sulfur atomTypes

We also only check when there is exactly one atomType, one bondType, one radical setting. For any group where there are wildcards or multiple attributes, we cannot apply this check.

In the case where the atomType is ambigious based on bonds and valency, this function will not change the type.

Returns a 'True' if the group was modified otherwise returns 'False'

standardizeGroup(*self*) → bool
This function modifies groups to make them have a standard AdjList form.

Currently it makes atomtypes as specific as possible and makes CO/CS atomtypes have explicit O2d/S2d ligands. Other functions can be added as necessary

Returns a 'True' if the group was modified otherwise returns 'False'

toAdjacencyList(*self*, *str label="*)
Convert the molecular structure to a string adjacency list.

update(*self*)

updateConnectivityValues(*self*)
Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

updateFingerprint(*self*)
Update the molecular fingerprint used to accelerate the subgraph isomorphism checks.

update_charge(*self*)
Update the partial charge according to the valence electron, total bond order, lone pairs and radical electrons. This method is used for products of specific families with recipes that modify charges.

vertices
list

Type vertices

rmgpy.molecule.resonance

This module contains methods for generation of resonance structures of molecules.

The main function to generate all relevant resonance structures for a given Molecule object is `generate_resonance_structures`. It calls the necessary functions for generating each type of resonance structure.

Currently supported resonance types:

- **All species:**
  - `generate_allyl_delocalization_resonance_structures`: single radical shift with double or triple bond
  - `generate_lone_pair_multiple_bond_resonance_structures`: lone pair shift with double or triple bond in a 3-atom system (between nonadjacent atoms)
  - `generate_adj_lone_pair_radical_resonance_structures`: single radical shift with lone pair between adjacent atoms
  - `generate_adj_lone_pair_multiple_bond_resonance_structures`: multiple bond shift with lone pair between adjacent atoms
  - `generate_adj_lone_pair_radical_multiple_bond_resonance_structures`: multiple bond and radical shift with lone pair and radical between adjacent atoms
  - `generate_N5dc_radical_resonance_structures`: shift between radical and lone pair mediated by an N5dc atom
  - `generate_aryne_resonance_structures`: shift between cumulene and alkyne forms of arynes, which are not considered aromatic in RMG
- **Aromatic species only:**
  - `generate_optimal_aromatic_resonance_structures`: fully delocalized structure, where all aromatic rings have benzene bonds
  - `generate_kekule_structure`: generate a single Kekule structure for an aromatic compound (single/double bond form)
  - `generate_opposite_kekule_structure`: for monocyclic aromatic species, rotate the double bond assignment
  - `generate_clar_structures`: generate all structures with the maximum number of pi-sextet assignments

`rmgpy.molecule.resonance.`**`analyze_molecule`**

Identify key features of molecule important for resonance structure generation.

Returns a dictionary of features.

`rmgpy.molecule.resonance.`**`generate_N5dc_radical_resonance_structures`**

Generate all of the resonance structures formed by radical and lone pair shifts mediated by an N5dc atom.

`rmgpy.molecule.resonance.`**`generate_adj_lone_pair_multiple_bond_resonance_structures`**

Generate all of the resonance structures formed by lone electron pair - multiple bond shifts between adjacent atoms. Example: [:NH]=[CH2] <=> [::NH-]-[CH2+] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge) Here atom1 refers to the N/S/O atom, atom 2 refers to the any R!H (atom2's lonePairs aren't affected) (In direction 1 atom1 <losses> a lone pair, in direction 2 atom1 <gains> a lone pair)

`rmgpy.molecule.resonance.`**`generate_adj_lone_pair_radical_multiple_bond_resonance_structures`**

Generate all of the resonance structures formed by lone electron pair - radical - multiple bond shifts between adjacent atoms. Example: [:N.]=[CH2] <=> [::N]-[.CH2] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge) Here atom1 refers to the N/S/O atom, atom 2 refers to the any R!H (atom2's lonePairs aren't affected) This function is similar to generate_adj_lone_pair_multiple_bond_resonance_structures() except for dealing with the radical transformations. (In direction 1 atom1 <losses> a lone pair, gains a radical, and atom2 looses a radical. In direction 2 atom1 <gains> a lone pair, looses a radical, and atom2 gains a radical)

`rmgpy.molecule.resonance.`**`generate_adj_lone_pair_radical_resonance_structures`**

Generate all of the resonance structures formed by lone electron pair - radical shifts between adjacent atoms. These resonance transformations do not involve changing bond orders. NO2 example: O=[:N]-[::O.] <=> O=[N.+]-[:::O-] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge)

`rmgpy.molecule.resonance.`**`generate_allyl_delocalization_resonance_structures`**

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

Biradicals on a single atom are not supported.

`rmgpy.molecule.resonance.`**`generate_aromatic_resonance_structure`**

Generate the aromatic form of the molecule in place without considering other resonance.

Parameters

- **`mol`** – `Molecule` object to modify
- **`aromatic_bonds`** (*`optional`*) – list of previously identified aromatic bonds
- **`copy`** (*`optional`*) – copy the molecule if `True`, otherwise modify in place

Returns List of one molecule if successful, empty list otherwise

`rmgpy.molecule.resonance.`**`generate_aryne_resonance_structures`**

Generate aryne resonance structures, including the cumulene and alkyne forms.

For all 6-membered rings, check for the following bond patterns:

- DDDSDS
- STSDSD

This does NOT cover all possible aryne resonance forms, only the simplest ones. Especially for polycyclic arynes, enumeration of all resonance forms is related to enumeration of all Kekule structures, which is very difficult.

`rmgpy.molecule.resonance.`**`generate_clar_structures`**

Generate Clar structures for a given molecule.

Returns a list of `Molecule` objects corresponding to the Clar structures.

`rmgpy.molecule.resonance.`**`generate_isomorphic_resonance_structures`**

Select the resonance isomer that is isomorphic to the parameter isomer, with the lowest unpaired electrons descriptor.

We generate over all resonance isomers (non-isomorphic as well as isomorphic) and retain isomorphic isomers.

If *saturate_h* is *True*, then saturate *mol* with hydrogens before generating the resonance structures, and remove the hydrogens before returning *isomorphic_isomers*. This is useful when resonance structures are generated for molecules in which all hydrogens were intentionally removed as in generating augInChI. Otherwise, RMG will probably get many of the lonePairs and partial charges in a molecule wrong.

WIP: do not generate aromatic resonance isomers.

`rmgpy.molecule.resonance.`**`generate_kekule_structure`**

Generate a kekulized (single-double bond) form of the molecule. The specific arrangement of double bonds is non-deterministic, and depends on RDKit.

Returns a single Kekule structure as an element of a list of length 1. If there's an error (eg. in RDKit) then it just returns an empty list.

`rmgpy.molecule.resonance.`**`generate_lone_pair_multiple_bond_resonance_structures`**

Generate all of the resonance structures formed by lone electron pair - multiple bond shifts in 3-atom systems. Examples: aniline (Nc1ccccc1), azide, [:NH2]C=[::O] <=> [NH2+]=C[:::O-] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge)

`rmgpy.molecule.resonance.`**`generate_optimal_aromatic_resonance_structures`**

Generate the aromatic form of the molecule. For radicals, generates the form with the most aromatic rings.

Returns result as a list. In most cases, only one structure will be returned. In certain cases where multiple forms have the same number of aromatic rings, multiple structures will be returned. If there's an error (eg. in RDKit) it just returns an empty list.

`rmgpy.molecule.resonance.`**`generate_resonance_structures`**

Generate and return all of the resonance structures for the input molecule.

Most of the complexity of this method goes into handling aromatic species, particularly to generate an accurate set of resonance structures that is consistent regardless of the input structure. The following considerations are made:

1. False positives from RDKit aromaticity detection can occur if a molecule has exocyclic double bonds
2. False negatives from RDKit aromaticity detection can occur if a radical is delocalized into an aromatic ring
3. sp2 hybridized radicals in the plane of an aromatic ring do not participate in hyperconjugation
4. Non-aromatic resonance structures of PAHs are not important resonance contributors (assumption)

Aromatic species are broken into the following categories for resonance treatment:

- Radical polycyclic aromatic species: Kekule structures are generated in order to generate adjacent resonance structures. The resulting structures are then used for Clar structure generation. After all three steps, any non-aromatic structures are removed, under the assumption that they are not important resonance contributors.
- Radical monocyclic aromatic species: Kekule structures are generated along with adjacent resonance structures. All are kept regardless of aromaticity because the radical is more likely to delocalize into the ring.
- Stable polycyclic aromatic species: Clar structures are generated
- Stable monocyclic aromatic species: Kekule structures are generated

`rmgpy.molecule.resonance.`**`populate_resonance_algorithms`**

Generate list of resonance structure algorithms relevant to the current molecule.

Takes a dictionary of features generated by analyze_molecule(). Returns a list of resonance algorithms.

rmgpy.molecule.kekulize

This module contains functions for kekulization of a aromatic molecule. The only function that should be used outside of this module is the main *kekulize()* function. The remaining functions and classes are designed only to support the kekulization algorithm, and should not be used on their own.

The basic algorithm is as follows: 1. Identify all aromatic rings in the molecule, based on bond types. 2. For each ring, identify endocyclic and exocyclic bonds. 3. Determine if any bonds in the ring are already defined (not benzene bonds). 4. For the remaining bonds, determine whether or not they can be double bonds. 5. If a clear determination cannot be made, make heuristic based assumption. 6. Continue until all bonds in the ring are determined. 7. Continue until all rings in the molecule are determined.

Here, *endo* refers to bonds that comprise a given ring, while *exo* refers to bonds that are connected to atoms in the ring, but not part of the ring itself.

A key part of the algorithm is use of degree of freedom (DOF) analysis in order to determine the optimal order to solve the system. Rings and bonds with fewer DOFs have fewer ways to be to be kekulized, and are generally easier to solve. Each ring or bond that is fixed reduces the DOF of adjacent rings and bonds, and the process continues until the entire molecule can be solved.

class `rmgpy.molecule.kekulize.`**AromaticBond**(*bond=None*, *ring_bonds=None*, *endo_dof=-1*, *exo_dof=-1*, *double_possible=True*, *double_required=False*)

Helper class containing information about a single aromatic bond in a molecule.

DO NOT use outside of this module. This class does not do any aromaticity perception.

bond

rmgpy.molecule.molecule.Bond

Type bond

double_possible

‘bool’

Type double_possible

double_required

‘bool’

Type double_required

endo_dof

‘int’

Type endo_dof

exo_dof

‘int’

Type exo_dof

ring_bonds

set

Type ring_bonds

update(*self*)

Update the local degree of freedom information for this aromatic bond. The DOF counts do not include the bond itself, only its adjacent bonds.

endo_dof refers to the number of adjacent bonds in the ring without fixed bond orders. *exo_dof* refers to the number of adjacent bonds outside the ring without fixed bond orders.

class `rmgpy.molecule.kekulize.`**AromaticRing**(*atoms=None*, *endo_bonds=None*, *exo_bonds=None*, *endo_dof=-1*, *exo_dof=-1*)

Helper class containing information about a single aromatic ring in a molecule.

DO NOT use outside of this module. This class does not do any aromaticity perception.

atoms

list

Type atoms

endo_dof

‘int’

Type endo_dof

exo_dof
'int'

Type exo_dof

kekulize(*self*) → bool
Attempts to kekulize a single aromatic ring in a molecule.

Returns True if successful, and False otherwise.

process_bonds(*self*) → tuple
Create AromaticBond objects for each endocyclic bond.

resolved
list

Type resolved

unresolved
list

Type unresolved

update(*self*)
Update the degree of freedom information for this aromatic ring.

endo_dof refers to the number of bonds in the ring without fixed bond orders. *exo_dof* refers to the number of bonds outside the ring without fixed bond orders.

`rmgpy.molecule.kekulize.`**kekulize**(*Molecule mol*)
Kekulize an aromatic molecule in place. If the molecule cannot be kekulized, a KekulizationError will be raised. However, the molecule will be left in a semi-kekulized state. Therefore, if the original molecule needs to be kept, it is advisable to create a copy before kekulizing.

Args: `Molecule` object to be kekulized

rmgpy.molecule.pathfinder

This module provides functions for searching paths within a molecule. The paths generally consist of alternating atoms and bonds.

`rmgpy.molecule.pathfinder.`**add_allyls**
Find all the (3-atom, 2-bond) patterns "X=X-X" starting from the last atom of the existing path.

The bond attached to the starting atom should be non single. The second bond should be single.

`rmgpy.molecule.pathfinder.`**add_inverse_allyls**
Find all the (3-atom, 2-bond) patterns "start~atom2=atom3" starting from the last atom of the existing path.

The second bond should be non-single.

`rmgpy.molecule.pathfinder.`**add_unsaturated_bonds**
Find all the (2-atom, 1-bond) patterns "X=X" starting from the last atom of the existing path.

The bond attached to the starting atom should be non single.

`rmgpy.molecule.pathfinder.`**compute_atom_distance**
Compute the distances between each pair of atoms in the atom_indices.

The distance between two atoms is defined as the length of the shortest path between the two atoms minus 1, because the start atom is part of the path.

The distance between multiple atoms is defined by generating all possible combinations between two atoms and storing the distance between each combination of atoms in a dictionary.

The parameter 'atom_indices' is a list of 1-based atom indices.

`rmgpy.molecule.pathfinder.`**`find_N5dc_radical_delocalization_paths`**

Find all the resonance structures of an N5dc nitrogen atom with a single bond to a radical N/O/S site, another single bond to a negatively charged N/O/S site, and one double bond (not participating in this transformation)

Example:

- N=[N+]([O])([O-]) <=> N=[N+]([O-])([O]), these structures are isomorphic but not identical, the transition is important for correct degeneracy calculations

In this transition atom1 is the middle N+ (N5dc), atom2 is the radical site, and atom3 is negatively charged A "if atom1.atomType.label == 'N5dc'" check should be done before calling this function

`rmgpy.molecule.pathfinder.`**`find_adj_lone_pair_multiple_bond_delocalization_paths`**

Find all the delocalization paths of atom1 which either

- Has a lonePair and is bonded by a single/double bond (e.g., [::NH-]-[CH2+], [::N-]=[CH+]) – direction 1
- Can obtain a lonePair and is bonded by a double/triple bond (e.g., [:NH]=[CH2], [:N]#[CH]) – direction 2

Giving the following resonance transitions, for example:

- [::NH-]-[CH2+] <=> [:NH]=[CH2]
- [:N]#[CH] <=> [::N-]=[CH+]
- other examples: S#N, N#[S], O=S([O])=O

Direction "1" is the direction <increasing> the bond order as in [::NH-]-[CH2+] <=> [:NH]=[CH2] Direction "2" is the direction <decreasing> the bond order as in [:NH]=[CH2] <=> [::NH-]-[CH2+] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge) (In direction 1 atom1 <losses> a lone pair, in direction 2 atom1 <gains> a lone pair)

`rmgpy.molecule.pathfinder.`**`find_adj_lone_pair_radical_delocalization_paths`**

Find all the delocalization paths of lone electron pairs next to the radical center indicated by *atom1*. Used to generate resonance isomers in adjacent N/O/S atoms. Two adjacent O atoms are not allowed since (a) currently RMG has no good thermo/kinetics for R[:O+.][:::O-] which could have been generated as a resonance structure of R[::O][::O.].

The radical site (atom1) could be either:

- *N u1 p0*, eg O=[N.+][:::O-]
- *N u1 p1*, eg R[:NH][:NH.]
- *O u1 p1*, eg [:O.+]=[::N-]; not allowed when adjacent to another O atom
- *O u1 p2*, eg O=N[::O.]; not allowed when adjacent to another O atom
- *S u1 p0*, eg O[S.+]([O-])=O
- *S u1 p1*, eg O[:S.+][O-]
- *S u1 p2*, eg O=N[::S.]
- any of the above with more than 1 radical where possible

The non-radical site (atom2) could respectively be:

- *N u0 p1*
- *N u0 p2*

- *O u0 p2*
- *O u0 p3*
- *S u0 p1*
- *S u0 p2*
- *S u0 p3*

(where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge) The bond between the sites does not have to be single, e.g.: [:O.+]=[::N-] <=> [::O]=[:N.]

`rmgpy.molecule.pathfinder.`**`find_adj_lone_pair_radical_multiple_bond_delocalization_paths`**

Find all the delocalization paths of atom1 which either

- Has a lonePair and is bonded by a single/double bond to a radical atom (e.g., [::N]-[.CH2])
- Can obtain a lonePair, has a radical, and is bonded by a double/triple bond (e.g., [:N.]=[CH2])

Giving the following resonance transitions, for example:

- [::N]-[.CH2] <=> [:N.]=[CH2]
- O[:S](=O)[::O.] <=> O[S.](=O)=[::O]

Direction "1" is the direction <increasing> the bond order as in [::N]-[.CH2] <=> [:N.]=[CH2] Direction "2" is the direction <decreasing> the bond order as in [:N.]=[CH2] <=> [::N]-[.CH2] (where ':' denotes a lone pair, '.' denotes a radical, '-' not in [] denotes a single bond, '-'/'+' denote charge) (In direction 1 atom1 <losses> a lone pair, gains a radical, and atom2 looses a radical. In direction 2 atom1 <gains> a lone pair, looses a radical, and atom2 gains a radical)

`rmgpy.molecule.pathfinder.`**`find_allyl_delocalization_paths`**

Find all the delocalization paths allyl to the radical center indicated by *atom1*.

`rmgpy.molecule.pathfinder.`**`find_allyl_end_with_charge`**

Search for a (3-atom, 2-bond) path between start and end atom that consists of alternating non-single and single bonds and ends with a charged atom.

Returns a list with atom and bond elements from start to end, or an empty list if nothing was found.

`rmgpy.molecule.pathfinder.`**`find_butadiene`**

Search for a path between start and end atom that consists of alternating non-single and single bonds.

Returns a list with atom and bond elements from start to end, or None if nothing was found.

`rmgpy.molecule.pathfinder.`**`find_butadiene_end_with_charge`**

Search for a (4-atom, 3-bond) path between start and end atom that consists of alternating non-single and single bonds and ends with a charged atom.

Returns a list with atom and bond elements from start to end, or None if nothing was found.

`rmgpy.molecule.pathfinder.`**`find_lone_pair_multiple_bond_paths`**

Find all the delocalization paths between lone electron pair and multiple bond in a 3-atom system *atom1* indicates the localized lone pair site. Currently carbenes are excluded from this path.

Examples:

- N2O (N#[N+][O-] <-> [N-]=[N+]=O)
- Azide (N#[N+][NH-] <-> [N-]=[N+]=N <-> [N-2][N+]#[NH+])
- N#N group on sulfur (O[S-](O)[N+]#N <-> OS(O)=[N+]=[N-] <-> O[S+](O)#[N+][N-2])
- N[N+]([O-])=O <=> N[N+](=O)[O-], these structures are isomorphic but not identical, this transition is important for correct degeneracy calculations

`rmgpy.molecule.pathfinder.`**`find_shortest_path`**

`rmgpy.molecule.pathfinder.`**`is_atom_able_to_gain_lone_pair`**

Helper function Returns True if atom is N/O/S and is able to <gain> an additional lone pair, False otherwise We don't allow O to remain with no lone pairs

`rmgpy.molecule.pathfinder.`**`is_atom_able_to_lose_lone_pair`**

Helper function Returns True if atom is N/O/S and is able to <loose> a lone pair, False otherwise We don't allow O to remain with no lone pairs

rmgpy.molecule.converter

This module provides methods for converting molecules between RMG, RDKit, and OpenBabel.

`rmgpy.molecule.converter.`**`debugRDKitMol`**

Takes an rdkit molecule object and logs some debugging information equivalent to calling rdmol.Debug() but uses our logging framework. Default logging level is INFO but can be controlled with the *level* parameter. Also returns the message as a string, should you want it for something.

`rmgpy.molecule.converter.`**`fromOBMol`**

Convert a OpenBabel Mol object *obmol* to a molecular structure. Uses OpenBabel to perform the conversion.

`rmgpy.molecule.converter.`**`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.

`rmgpy.molecule.converter.`**`toOBMol`**

Convert a molecular structure to an OpenBabel OBMol object. Uses OpenBabel to perform the conversion.

`rmgpy.molecule.converter.`**`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.

rmgpy.molecule.translator

This module provides methods for translating to and from common molecule representation formats, e.g. SMILES, InChI, SMARTS.

`rmgpy.molecule.translator.`**`fromAugmentedInChI`**

Creates a Molecule object from the augmented inchi.

First, the inchi is converted into a Molecule using the backend parsers.

Next, the multiplicity and unpaired electron information is used to fix a number of parsing errors made by the backends.

Finally, the atom types of the corrected molecule are perceived.

Returns a Molecule object

`rmgpy.molecule.translator.`**`fromInChI`**

Convert an InChI string *inchistr* to a molecular structure. Uses a user-specified backend for conversion, currently supporting rdkit (default) and openbabel.

`rmgpy.molecule.translator.`**`fromSMARTS`**

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

`rmgpy.molecule.translator.`**`fromSMILES`**

Convert a SMILES string *smilesstr* to a molecular structure. Uses a user-specified backend for conversion, currently supporting rdkit (default) and openbabel.

`rmgpy.molecule.translator.`**`toInChI`**

Convert a molecular structure to an InChI string. For aug_level=0, generates the canonical InChI. For aug_level=1, appends the molecule multiplicity. For aug_level=2, appends positions of unpaired and paired electrons.

Uses RDKit or OpenBabel for conversion.

Parameters

- **`choice of backend, 'try-all', 'rdkit', or 'openbabel'`** (*backend*) –
- **`level of augmentation, 0, 1, or 2`** (*aug_level*) –

`rmgpy.molecule.translator.`**`toInChIKey`**

Convert a molecular structure to an InChI Key string. For aug_level=0, generates the canonical InChI. For aug_level=1, appends the molecule multiplicity. For aug_level=2, appends positions of unpaired and paired electrons.

Uses RDKit or OpenBabel for conversion.

Parameters

- **`choice of backend, 'try-all', 'rdkit', or 'openbabel'`** (*backend*) –
- **`level of augmentation, 0, 1, or 2`** (*aug_level*) –

`rmgpy.molecule.translator.`**`toSMARTS`**

Convert a molecular structure to an SMARTS string. Uses RDKit to perform the conversion. Perceives aromaticity and removes Hydrogen atoms.

`rmgpy.molecule.translator.`**`toSMILES`**

Convert a molecular structure to an SMILES string.

If there is a Nitrogen/Sulfur 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.

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 is square brackets, and separated by commas, without spaces. If lone pairs or formal charges are omitted from a group definition, the wildcard is assumed.

The last set of tokens is the list of bonds. To indicate a bond, place the number of the atom at the other end of the bond and the bond type within curly braces and separated by a comma, e.g. `{2,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`**`(`*Molecule molecule*, *Atom atom*`)` → float
Return the symmetry number centered at *atom* in the structure. The *atom* of interest must not be in a cycle.

`rmgpy.molecule.symmetry.`**`calculateBondSymmetryNumber`**`(`*Molecule molecule*, *Atom atom1*, *Atom atom2*`)` → float

Return the symmetry number centered at *bond* in the structure.

`rmgpy.molecule.symmetry.`**`calculateAxisSymmetryNumber`**`(`*Molecule molecule*`)` → float

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:

```
A=C=C=C..        A-C=C=C=C-A

  s=1                s=1
```

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:

```
A\               A\         /A
  T=C=C=C=C-A      T=C=C=C=T
B/               A/         \B
      s=1              s=1
```

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:

```
A\         /B      A\
  C=C=C=C=C          C=C=C=C-B
A/         \B      A/
      s=2                s=2
```

`rmgpy.molecule.symmetry.`**`calculateCyclicSymmetryNumber`**`(`*Molecule molecule*`)` → float

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`**`(`*Molecule molecule*`)` → float

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*, *target=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 calcuating 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
`SingleExponentialDown`	A collisional energy transfer model based on the single exponential down model

1.7.2 Reaction events

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

1.7.3 Pressure-dependent reaction networks

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

1.7.4 The master equation

Function	Description
`generateFullMEMatrix()`	Return the full master equation matrix for a network

1.7.5 Master equation reduction methods

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

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
alpha0	The average energy transferred in a deactivating collision at the reference temperature
T0	The reference temperature
n	The temperature exponent

Based around the collisional energy transfer probability function

$$P(E, E') = C(E') \exp\left(-\frac{E' - E}{\alpha}\right) \qquad E < E'$$

where the parameter $\alpha = \langle \Delta E_\mathrm{d} \rangle$ 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 [**?**].

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

Generate and return the collision matrix $\mathbf{M}_\mathrm{coll} / \omega = \mathbf{P} - \mathbf{I}$ 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*) $\rightarrow$ 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`

‘double’

Type n

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 microcanonial 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 microcanonial 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:

$$\mathcal{L}\left[k(E)\rho(E)\right] = \int_0^\infty k(E)\rho(E) e^{-E/k_\mathrm{B}T} \, dE = k_\infty(T) Q(T)$$

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:

$$k(E) = A \frac{\rho(E - E_\mathrm{a})}{\rho(E)} \qquad (n = 0)$$

For $n > 0$ the transform is defined numerically. For $n < 0$ or $E_\mathrm{a} < 0$ the transform is not defined; in this case we approximate by simply lumping the T^n or $e^{-E_\mathrm{a}/RT}$ terms into the preexponential factor, and use a different $k(E)$ at each temperature.

The ILT method does not required 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`**`(`**species*`)`

A representation of a molecular configuration on a potential energy surface.

`E0`

The ground-state energy of the configuration in J/mol.

`Elist`

numpy.ndarray

Type Elist

`activeJRotor`

'bool'

Type activeJRotor

`activeKRotor`

'bool'

Type activeKRotor

`calculateCollisionFrequency``(`*self*, *double T*, *double P*, *dict bathGas*`)` → double

Return the value of the collision frequency in Hz at the given temperature *T* in K and pressure *P* in Pa. If a dictionary *bathGas* of bath gas species and corresponding mole fractions is given, the collision parameters of the bas gas species will be averaged with those of the species before computing the collision frequency.

Only the Lennard-Jones collision model is currently supported.

`calculateDensityOfStates``(`*self*, *ndarray Elist*, *bool activeJRotor=True*, *bool activeKRotor=True*, *bool rmgmode=False*`)`

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``(`*self*`)`

Delete intermediate arrays used in computing k(T,P) values.

`densStates`

numpy.ndarray

Type densStates

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

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``(`*self*, *double T*`)` → double

Return the enthalpy in kJ/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 kJ/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.

hasStatMech(*self*) → bool

Return `True` if all species in the configuration have statistical mechanics parameters, or `False` otherwise.

hasThermo(*self*) → bool

Return `True` if all species in the configuration have thermodynamics parameters, or `False` otherwise.

isBimolecular(*self*) → bool

Return `True` if the configuration represents a bimolecular reactant or product channel, or `False` otherwise.

isTransitionState(*self*) → bool

Return `True` if the configuration represents a transition state, or `False` otherwise.

isUnimolecular(*self*) → bool

Return `True` if the configuration represents a unimolecular isomer, or `False` otherwise.

mapDensityOfStates(*self*, *ndarray Elist*, *ndarray Jlist=None*)

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(*self*, *ndarray Elist*, *ndarray Jlist=None*)

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*.

species

list

Type species

sumStates

numpy.ndarray

Type sumStates

rmgpy.pdep.Network

class rmgpy.pdep.**Network**(*label=”*, *isomers=None*, *reactants=None*, *products=None*, *pathReactions=None*, *bathGas=None*, *netReactions=None*, *T=0.0*, *P=0.0*, *Elist=None*, *Jlist=None*, *Ngrains=0*, *NJ=0*, *activeKRotor=True*, *activeJRotor=True*, *grainSize=0.0*, *grainCount=0*, *E0=None*)

A representation of a unimolecular reaction network. The attributes are:

Attribute	Description
isomers	A list of the unimolecular isomers in the network
reactants	A list of the bimolecular reactant channels (Configuration objects) in the network
products	A list of the bimolecular product channels (Configuration objects) in the network
pathReactions	A list of “path” reaction objects that connect adjacent isomers (the high-pressure-limit)
bathGas	A dictionary of the bath gas species (keys) and their mole fractions (values)
netReactions	A list of “net” reaction objects that connect any pair of isomers
T	The current temperature in K
P	The current pressure in bar
Elist	The current array of energy grains in kJ/mol
Jlist	The current array of total angular momentum quantum numbers
Nisom	The number of unimolecular isomers in the network
Nreac	The number of bimolecular reactant channels in the network
Nprod	The number of bimolecular product channels in the network
Ngrains	The number of energy grains
NJ	The number of angular momentum grains
grainSize	Maximum size of separation between energies
grainCount	Minimum number of descrete energies separated
E0	A list of ground state energies of isomers, reactants, and products
activeKRotor	`True` if the K-rotor is treated as active, `False` if treated as adiabatic
activeJRotor	`True` if the J-rotor is treated as active, `False` if treated as adiabatic
rmgmode	`True` if in RMG mode, `False` otherwise
eqRatios	An array containing concentration of each isomer and reactant channel present at equilibrium
collFreq	An array of the frequency of collision between
Mcoll	Matrix of first-order rate coefficients for collisional population transfer between grains for each isomer
densStates	3D np array of stable configurations, number of grains, and number of J

`applyChemicallySignificantEigenvaluesMethod`(*lumpingOrder=None*)

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the chemically-significant eigenvalues method. If a *lumpingOrder* is provided, the algorithm will attempt to lump the configurations (given by index) in the order provided, and return a reduced set of $k(T, P)$ values.

`applyModifiedStrongCollisionMethod`(*efficiencyModel='default'*)

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the modified strong collision method.

`applyReservoirStateMethod`()

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the reservoir state method.

`calculateCollisionModel`()

Calculate the matrix of first-order rate coefficients for collisional population transfer between grains for each isomer, including the corresponding collision frequencies.

`calculateDensitiesOfStates`()

Calculate the densities of states of each configuration that has states data. The densities of states are computed such that they can be applied to each temperature in the range of interest by interpolation.

`calculateEquilibriumRatios`()

Return an array containing the fraction of each isomer and reactant channel present at equilibrium, as determined from the Gibbs free energy and using the concentration equilibrium constant K_c. These values are ratios, and the absolute magnitude is not guaranteed; however, the implementation scales the elements of the array so that they sum to unity.

calculateMicrocanonicalRates()

Calculate and return arrays containing the microcanonical rate coefficients $k(E)$ for the isomerization, dissociation, and association path reactions in the network.

getAllSpecies()

Return a list of all unique species in the network, including all isomers, reactant and product channels, and bath gas species.

initialize(*Tmin*, *Tmax*, *Pmin*, *Pmax*, *maximumGrainSize=0.0*, *minimumGrainCount=0*, *activeJRotor=True*, *activeKRotor=True*, *rmgmode=False*)

Initialize a pressure dependence calculation by computing several quantities that are independent of the conditions. You must specify the temperature and pressure ranges of interesting using *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 $k_\mathrm{B}T$ 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.

The master equation

`rmgpy.pdep.me.`**`generateFullMEMatrix`**(*network*, *bool products=True*)

Generate the full master equation matrix for the network.

An in-depth explanation can be found in the Master Equation section of the theory guide.

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`**(*network*, *str efficiencyModel='default'*)

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 is 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`**(*network*)

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`**(*network*, *list lumpingOrder=None*)

In the chemically-significant eigenvalues method, the master equation matrix is diagonized to determine its eigenmodes. Only the slowest of these modes are relevant to the chemistry; the rest involve internal energy relaxation due to collisions. Keeping only these "chemically-significant" eigenmodes allows for reduction to $k(T, P)$ values.

The chemically-significant eigenvalues method is the most accurate method, and is considered to be exact as long as the chemically-significant eigenmodes are separable and distinct from the internal energy relaxation eigenmodes. However, this is often only the case near the high-pressure limit, even for networks of only modest size. The chemically-significant eigenvalues method is also substantially more expensive to apply than the other methods.

1.8 QMTP (`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
`QMSettings`	A class to store settings related to quantum mechanics calculations
`QMCalculator`	An object to store settings and previous calculations

1.8.2 Molecule

Class	Description
`Geometry`	A geometry, used for quantum calculations
`QMMolecule`	A base class for QM Molecule calculations

1.8.3 QM Data

Class/Function	Description
`QMData`	General class for data extracted from a QM calculation

1.8.4 QM Verifier

Class/Function	Description
`QMVerifier`	Verifies whether a QM job was succesfully completed

1.8.5 Symmetry

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

1.8.6 Gaussian

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

1.8.7 Mopac

Class/Function	Description
Mopac	A base class for all QM calculations that use Mopac
MopacMol	A base Class for calculations of molecules using Mopac.
MopacMolPM3	A base Class for calculations of molecules using Mopac at PM3.
MopacMolPM6	A base Class for calculations of molecules using Mopac at PM6.
MopacMolPM7	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
software	`str`	Quantum chemical package name in common letters
method	`str`	Semi-empirical method
fileStore	`str`	The path to the QMfiles directory
scratchDirectory	`str`	The path to the scratch directory
onlyCyclics	`bool`	`True` if to run QM only on ringed species
maxRadicalNumber	`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
settings	`QMSettings`	Settings for QM calculations
database	`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
settings	`QMSettings`	Settings for QM calculations
uniqueID	`str`	A short ID such as an augmented InChI Key
molecule	`Molecule`	RMG Molecule object
uniqueIDlong	`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*, *scratch=True*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getRefinedMolFilePath()

Returns the path the the refined mol file.

rd_build()

Import rmg molecule and create rdkit molecule with the same atom labeling.

rd_embed(*rdmol*, *numConfAttempts*)

Embed the RDKit molecule and create the crude molecule file.

saveCoordinatesFromQMData(*qmdata*)
Save geometry info from QMData (eg CCLibData)

uniqueID = None
A short unique ID such as an augmented InChI Key.

uniqueIDlong = None
Long, truly unique, ID, such as the augmented InChI.

class `rmgpy.qm.molecule.`**QMMolecule**(*molecule*, *settings*)
A base class for QM Molecule calculations.

Specific programs and methods should inherit from this and define some extra attributes and methods:

- outputFileExtension
- inputFileExtension
- generateQMData() … and whatever else is needed to make this method work.

The attributes are:

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

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()
Check the paths in the settings are OK. Make folders as necessary.

checkReady()
Check that it's ready to run calculations.

createGeometry()
Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()
Calculate the QM data somehow and return a CCLibData object, or None if it fails.

generateThermoData()
Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*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 QMData 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=-1*, *numberOfAtoms=None*, *stericEnergy=None*, *molecularMass=None*, *energy=0*, *atomicNumbers=None*, *rotationalConstants=None*, *atomCoords=None*, *frequencies=None*, *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.

rmgpy.qm.qmverifier

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

Verifies whether a QM job (externalized) was succesfully 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.

succesfulJobExists()

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 QMData object.

successKeys = ['Normal termination of Gaussian']

List of phrases to indicate success. ALL of these must be present in a successful job.

verifyOutputFile()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that 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*ln(2) if chiral) in J/mol/K.

calculateThermoData()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()

Check the paths in the settings are OK. Make folders as necessary.

checkReady()

Check that it's ready to run calculations.

createGeometry()

Creates self.geometry with RDKit geometries

determinePointGroup()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()
Calculate the QM data and return a QMData object.

generateThermoData()
Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)
Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(*outputFile*)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFileKeywords(*attempt*)
Return the top keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

inputFilePath
Get the input file name.

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

writeInputFile(*attempt*)

Using the `Geometry` object, write the input file for the *attempt*.

class `rmgpy.qm.gaussian.`**GaussianMolPM3**(*molecule*, *settings*)

Gaussian PM3 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it's only the 'pm3' in the keywords that differs.

calculateChiralityCorrection()

Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()

Check the paths in the settings are OK. Make folders as necessary.

checkReady()

Check that it's ready to run calculations.

createGeometry()

Creates self.geometry with RDKit geometries

determinePointGroup()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()

Calculate the QM data and return a QMData object.

generateThermoData()

Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()

Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(*outputFile*)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFileKeywords(*attempt*)
Return the top keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

inputFilePath
Get the input file name.

keywords = ['# pm3 opt=(verytight,gdiis) freq IOP(2/16=3)', '# pm3 opt=(verytight,gdiis) freq IO
Keywords that will be added at the top of the qm input file

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

writeInputFile(*attempt*)
Using the `Geometry` object, write the input file for the *attempt*.

class `rmgpy.qm.gaussian.`**GaussianMolPM6**(*molecule*, *settings*)
Gaussian PM6 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it's only the 'pm6' in the keywords that differs.

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()

Check the paths in the settings are OK. Make folders as necessary.

checkReady()

Check that it's ready to run calculations.

createGeometry()

Creates self.geometry with RDKit geometries

determinePointGroup()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()

Calculate the QM data and return a QMData object.

generateThermoData()

Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()

Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude 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 I

Keywords that will be added at the top of the qm input file

loadThermoData()

Try loading a thermo data from a previous run.

maxAttempts

The total number of attempts to try

outputFilePath

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()

Save the generated thermo data.

scriptAttempts

The number of attempts with different script keywords

verifyOutputFile()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

writeInputFile(*attempt*)

Using the `Geometry` object, write the input file for the *attempt*.

rmgpy.qm.mopac

class rmgpy.qm.mopac.**Mopac**

A base class for all QM calculations that use MOPAC.

Classes such as *`MopacMol`* will inherit from this class.

failureKeys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME

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*ln(2) if chiral) in J/mol/K.

`calculateThermoData`()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

`checkPaths`()

Check the paths in the settings are OK. Make folders as necessary.

`checkReady`()

Check that it's ready to run calculations.

`createGeometry`()

Creates self.geometry with RDKit geometries

`determinePointGroup`()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

`generateQMData`()

Calculate the QM data and return a QMData object, or None if it fails.

`generateThermoData`()

Generate Thermo Data via a QM calc.

Returns None if it fails.

`getFilePath`(*extension*, *scratch=True*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

`getInChiKeyAug`()

Returns the augmented InChI from self.molecule

`getMolFilePathForCalculation`(*attempt*)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

`getParser`(*outputFile*)

Returns the appropriate cclib parser.

`getThermoFilePath`()

Returns the path the thermo data file.

`initialize`()

Do any startup tasks.

`inputFileKeywords`(*attempt*)

Return the top, bottom, and polar keywords.

inputFilePath

Get the input file name.

keywords = [{'top': 'precise nosym THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=

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 QMData object.

saveThermoData()

Save the generated thermo data.

scriptAttempts

The number of attempts with different script keywords

verifyOutputFile()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

writeInputFile(*attempt*)

Using the `Geometry` object, write the input file for the *attempt*.

class rmgpy.qm.mopac.**MopacMolPM3**(*molecule*, *settings*)

Mopac PM3 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it's the same as all the MOPAC PMn calculations, only pm3

calculateChiralityCorrection()

Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()

Check the paths in the settings are OK. Make folders as necessary.

checkReady()

Check that it's ready to run calculations.

createGeometry()

Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()
Calculate the QM data and return a QMData object, or None if it fails.

generateThermoData()
Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)
Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(*outputFile*)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFileKeywords(*attempt*)
Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

inputFilePath
Get the input file name.

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

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*ln(2) if chiral) in J/mol/K.

calculateThermoData()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()

Check the paths in the settings are OK. Make folders as necessary.

checkReady()

Check that it's ready to run calculations.

createGeometry()

Creates self.geometry with RDKit geometries

determinePointGroup()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()

Calculate the QM data and return a QMData object, or None if it fails.

generateThermoData()

Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)

Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()

Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)
Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(*outputFile*)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFileKeywords(*attempt*)
Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

inputFilePath
Get the input file name.

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

writeInputFile(*attempt*)
Using the `Geometry` object, write the input file for the *attempt*.

class `rmgpy.qm.mopac.`**MopacMolPM7**(*molecule*, *settings*)
Mopac PM7 calculations for molecules

This is a class of its own in case you wish to do anything differently, but for now it's the same as all the MOPAC PMn calculations, only pm7

calculateChiralityCorrection()
Returns the chirality correction to entropy (R*ln(2) if chiral) in J/mol/K.

calculateThermoData()
Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qmData and self.pointGroup need to be generated before this method is called.

checkPaths()
Check the paths in the settings are OK. Make folders as necessary.

checkReady()
Check that it's ready to run calculations.

createGeometry()
Creates self.geometry with RDKit geometries

determinePointGroup()
Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in self.pointGroup

generateQMData()
Calculate the QM data and return a QMData object, or None if it fails.

generateThermoData()
Generate Thermo Data via a QM calc.

Returns None if it fails.

getFilePath(*extension*, *scratch=True*)
Returns the path to the file with the given extension.

The provided extension should include the leading dot. If called with *scratch=False* then it will be in the *fileStore* directory, else *scratch=True* is assumed and it will be in the *scratchDirectory* directory.

getInChiKeyAug()
Returns the augmented InChI from self.molecule

getMolFilePathForCalculation(*attempt*)
Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.scriptAttempts then we use the refined coordinates, then we start to use the crude coordinates.

getParser(*outputFile*)
Returns the appropriate cclib parser.

getThermoFilePath()
Returns the path the thermo data file.

initialize()
Do any startup tasks.

inputFileKeywords(*attempt*)
Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

inputFilePath
Get the input file name.

loadThermoData()
Try loading a thermo data from a previous run.

maxAttempts
The total number of attempts to try

outputFilePath
Get the output file name.

parse()
Parses the results of the Mopac calculation, and returns a QMData object.

saveThermoData()
Save the generated thermo data.

scriptAttempts
The number of attempts with different script keywords

verifyOutputFile()
Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful MOPAC simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that are denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InchI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all succeed, then it will return True.

writeInputFile(*attempt*)
Using the `Geometry` object, write the input file for the *attempt*.

1.9 Physical quantities (`rmgpy.quantity`)

A physical quantity is defined by a numerical value and a unit of measurement.

The *`rmgpy.quantity`* module contains classes and methods for working with physical quantities. Physical quantities are represented by either the *`ScalarQuantity`* or *`ArrayQuantity`* class depending on whether a scalar or vector (or tensor) value is used. The *`Quantity`* function automatically chooses the appropriate class based on the input value. In both cases, the value of a physical quantity is available from the `value` attribute, and the units from the `units` attribute.

For efficient computation, the value is stored internally in the SI equivalent units. The SI value can be accessed directly using the `value_si` attribute. Usually it is good practice to read the `value_si` attribute into a local variable and then use it for computations, especially if it is referred to multiple times in the calculation.

Physical quantities also allow for storing of uncertainty values for both scalars and arrays. The `uncertaintyType` attribute indicates whether the given uncertainties are additive (`"+|-"`) or multiplicative (`"*|/"`), and the `uncertainty` attribute contains the stored uncertainties. For additive uncertainties these are stored in the given units (not the SI equivalent), since they are generally not needed for efficient computations. For multiplicative uncertainties, the uncertainty values are by definition dimensionless.

1.9.1 Quantity objects

Class	Description
ScalarQuantity	A scalar physical quantity, with units and uncertainty
ArrayQuantity	An array physical quantity, with units and uncertainty
Quantity()	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	$\mathrm{m/s^2}$
`Area()`	area	$\mathrm{m^2}$
`Concentration()`	concentration	$\mathrm{mol/cm^3}$
`Dimensionless()`	dimensionless	
`Energy()`	energy	$\mathrm{J/mol}$
`Entropy()`	entropy	$\mathrm{J/mol \cdot K}$
`Flux()`	flux	$\mathrm{mol/cm^2 \cdot s}$
`Frequency()`	frequency	$\mathrm{cm^{-1}}$
`Force()`	force	N
`Inertia()`	inertia	$\mathrm{kg \cdot m^2}$
`Length()`	length	m
`Mass()`	mass	kg
`Momentum()`	momentum	$\mathrm{kg \cdot m/s^2}$
`Power()`	power	W
`Pressure()`	pressure	Pa
`RateCoefficient()`	rate coefficient	$\mathrm{s^{-1}}$, $\mathrm{m^3/mol \cdot s}$, $\mathrm{m^6/mol^2 \cdot s}$, $\mathrm{m^9/mol^3 \cdot s}$
`Temperature()`	temperature	K
`Time()`	time	s
`Velocity()`	velocity	$\mathrm{m/s}$
`Volume()`	volume	$\mathrm{m^3}$

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 $\mathrm{amu*angstrom^2}$ instead of $\mathrm{kg*m^2}$. The recommended rule of thumb is to use prefixed SI units (or aliases thereof) in the output; for example, use $\mathrm{kJ/mol}$ instead of $\mathrm{kcal/mol}$ for energy values.

rmgpy.quantity.ScalarQuantity

class `rmgpy.quantity.`**`ScalarQuantity`**(*value*, *units=''*, *uncertainty=None*, *uncertaintyType='+|-'*)

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

Attribute	Description
value	The numeric value of the quantity in the given units
units	The units the value was specified in
uncertainty	The numeric uncertainty in the value in the given units (unitless if multiplicative)
uncertaintyType	The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative
value_si	The numeric value of the quantity in the corresponding SI units
uncertainty_si	The numeric value of the uncertainty in the corresponding SI units (unitless if multiplicative)

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`(*self*) → ScalarQuantity

Return a copy of the quantity.

`equals`(*self*, *quantity*)

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`(*self*) → double

Return the conversion factor for converting a quantity to a given set of *units* from the SI equivalent units.

`getConversionFactorToSI`(*self*) → double

Return the conversion factor for converting a quantity in a given set of'units' to the SI equivalent units.

`getUncertainty`(*self*)

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

`getUncertaintyType`(*self*) → str

The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative

`getValue`(*self*)

The numeric value of the quantity, in the given units

`isUncertaintyAdditive`(*self*) → bool

Return `True` if the uncertainty is specified in additive format and `False` otherwise.

`isUncertaintyMultiplicative`(*self*) → bool

Return `True` if the uncertainty is specified in multiplicative format and `False` otherwise.

`setUncertainty`(*self*, *v*)

`setUncertaintyType`(*self*, *str v*)

Check the uncertainty type is valid, then set it.

`setValue`(*self*, *v*)

`uncertainty`

ScalarQuantity.getUncertainty(self)

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

`uncertaintyType`
ScalarQuantity.getUncertaintyType(self) -> str

The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative

`uncertainty_si`
'double'

Type uncertainty_si

`units`
str

Type units

`value`
ScalarQuantity.getValue(self)

The numeric value of the quantity, in the given units

`value_si`
'double'

Type value_si

rmgpy.quantity.ArrayQuantity

class `rmgpy.quantity.`**`ArrayQuantity`**(*value*, *units=''*, *uncertainty=None*, *uncertaintyType='+|-'*)

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

Attribute	Description
value	The numeric value of the quantity in the given units
units	The units the value was specified in
uncertainty	The numeric uncertainty in the value (unitless if multiplicative)
uncertaintyType	The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative
value_si	The numeric value of the quantity in the corresponding SI units
uncertainty_si	The numeric value of the uncertainty in the corresponding SI units (unitless if multiplicative)

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`(*self*) → ArrayQuantity
Return a copy of the quantity.

`equals`(*self*, *quantity*)
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`(*self*) → double
Return the conversion factor for converting a quantity to a given set of *units* from the SI equivalent units.

getConversionFactorToSI(*self*) → double
Return the conversion factor for converting a quantity in a given set of‘units‘ to the SI equivalent units.

getUncertainty(*self*)
The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

getUncertaintyType(*self*) → str
The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative

getValue(*self*)
The numeric value of the array quantity, in the given units.

isUncertaintyAdditive(*self*) → bool
Return `True` if the uncertainty is specified in additive format and `False` otherwise.

isUncertaintyMultiplicative(*self*) → bool
Return `True` if the uncertainty is specified in multiplicative format and `False` otherwise.

setUncertainty(*self*, *v*)

setUncertaintyType(*self*, *str v*)
Check the uncertainty type is valid, then set it.

If you set the uncertainty then change the type, we have no idea what to do with the units. This ensures you set the type first.

setValue(*self*, *v*)

uncertainty
ArrayQuantity.getUncertainty(self)

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

uncertaintyType
ArrayQuantity.getUncertaintyType(self) -> str

The type of uncertainty: `'+|-'` for additive, `'*|/'` for multiplicative

uncertainty_si
numpy.ndarray

Type uncertainty_si

units
str

Type units

value
ArrayQuantity.getValue(self)

The numeric value of the array quantity, in the given units.

value_si
numpy.ndarray

Type value_si

rmgpy.quantity.Quantity

`rmgpy.quantity.`**Quantity**(**args*, ***kwargs*)
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
`Reaction`	A chemical reaction

rmgpy.reaction.Reaction

class `rmgpy.reaction.`**`Reaction`**(*index=-1*, *label=”*, *reactants=None*, *products=None*, *specificCollider=None*, *kinetics=None*, *network_kinetics=None*, *reversible=True*, *transitionState=None*, *duplicate=False*, *degeneracy=1*, *pairs=None*, *allow_pdep_route=False*, *elementary_high_p=False*, *allow_max_rate_violation=False*, *comment=”*, *is_forward=None*)

A chemical reaction. The attributes are:

Attribute	Type	Description
index	`int`	A unique nonnegative integer index
label	`str`	A descriptive string label
reactants	`list`	The reactant species (as `Species` objects)
products	`list`	The product species (as `Species` objects)
‘specificCollider’	`Species`	The collider species (as a `Species` object)
kinetics	`KineticsModel`	The kinetics model to use for the reaction
network_kinetics	`Arrhenius`	The kinetics model to use for PDep network exploration if the *kinetics* attribute is :class:PDepKineticsModel:
reversible	`bool`	`True` if the reaction is reversible, `False` if not
transitionState	`TransitionState`	The transition state
duplicate	`bool`	`True` if the reaction is known to be a duplicate, `False` if not
degeneracy	`double`	The reaction path degeneracy for the reaction
pairs	`list`	Reactant-product pairings to use in converting reaction flux to species flux
allow_pdep_route	`bool`	`True` if the reaction has an additional PDep pathway, `False` if not (by default), used for LibraryReactions
elementary_high_p	`bool`	If `True`, pressure dependent kinetics will be generated (relevant only for unimolecular library reactions) If `False` (by default), this library reaction will not be explored. Only unimolecular library reactions with high pressure limit kinetics should be flagged (not if the kinetics were measured at some relatively low pressure)
comment	`str`	A description of the reaction source (optional)
is_forward	`bool`	Indicates if the reaction was generated in the forward (true) or reverse (false)

allow_max_rate_violation
‘bool’

Type allow_max_rate_violation

allow_pdep_route
‘bool’

Type allow_pdep_route

calculateMicrocanonicalRateCoefficient(*self*, *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.

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

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

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

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.

calculateTSTRateCoefficients(*self*, *ndarray Tlist*) → ndarray

calculate_coll_limit(*self*, *float temp*, *bool reverse=False*)

Calculate the collision limit rate for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

canTST(*self*) → bool

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.

check_collision_limit_violation(*self*, *float t_min*, *float t_max*, *float p_min*, *float p_max*) → list

Warn if a core reaction violates the collision limit rate in either the forward or reverse direction at the relevant extreme T/P conditions. Assuming a monotonic behaviour of the kinetics. Returns a list with the reaction object and the direction in which the violation was detected.

comment

str

Type comment

copy(*self*)

Create a deep copy of the current reaction.

degeneracy

Reaction.__getDegneneracy(self)

draw(*self*, *path*)

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.

duplicate

'bool'

Type duplicate

`elementary_high_p`
'bool'

Type elementary_high_p

`ensure_species`(*self*, *bool reactant_resonance=False*, *bool product_resonance=True*)
Ensure the reaction contains species objects in its reactant and product attributes. If the reaction is found to hold molecule objects, it modifies the reactant, product and pairs to hold Species objects.

Generates resonance structures for Molecules if the corresponding options, reactant_resonance and/or product_resonance, are True. Does not generate resonance for reactants or products that start as Species objects.

`fixBarrierHeight`(*self*, *bool forcePositive=False*)
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`(*self*, *T*)
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

`generate3dTS`(*self*, *reactants*, *products*)
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`(*self*)
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 (C/O/N/S at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

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

`getEnthalpiesOfReaction`(*self*, *ndarray Tlist*) → ndarray
Return the enthalpies of reaction in J/mol evaluated at temperatures *Tlist* in K.

`getEnthalpyOfReaction`(*self*, *double T*) → double
Return the enthalpy of reaction in J/mol evaluated at temperature *T* in K.

`getEntropiesOfReaction`(*self*, *ndarray Tlist*) → ndarray
Return the entropies of reaction in J/mol*K evaluated at temperatures *Tlist* in K.

`getEntropyOfReaction`(*self*, *double T*) → double
Return the entropy of reaction in J/mol*K evaluated at temperature *T* in K.

getEquilibriumConstant(*self*, *double T*, *str type='Kc'*) → double
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(*self*, *ndarray Tlist*, *str type='Kc'*) → ndarray
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(*self*, *ndarray Tlist*) → ndarray
Return the Gibbs free energies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

getRateCoefficient(*self*, *double T*, *double P=0*) → double
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(*self*, *Species spec*) → int
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(*self*)
Get a URL to search for this reaction in the rmg website.

get_mean_sigma_and_epsilon(*self*, *bool reverse=False*)
Calculates the collision diameter (sigma) using an arithmetic mean Calculates the well depth (epsilon) using a geometric mean If reverse is `False` the above is calculated for the reactants, otherwise for the products

get_reduced_mass(*self*, *bool reverse=False*)
Returns the reduced mass of the reactants if reverse is `False` Returns the reduced mass of the products if reverse is `True`

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

index
'int'

Type index

isAssociation(*self*) → bool
Return `True` if the reaction represents an association reaction $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or `False` if not.

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

isDissociation(*self*) → bool
Return `True` if the reaction represents a dissociation reaction $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or `False` if not.

isIsomerization(*self*) → bool
Return `True` if the reaction represents an isomerization reaction $\mathrm{A} \rightleftharpoons \mathrm{B}$ or `False` if not.

isIsomorphic(*self*, *Reaction other*, *bool eitherDirection=True*, *bool checkIdentical=False*, *bool checkOnlyLabel=False*, *bool checkTemplateRxnProducts=False*) → bool

Return `True` if this reaction is the same as the *other* reaction, or `False` if they are different. The comparison involves comparing isomorphism of reactants and products, and doesn't use any kinetic information.

If *eitherDirection=False* then the directions must match.

***checkIdentical* indicates that atom ID's must match and is used in** checking degeneracy

***checkOnlyLabel* indicates that the string representation will be** checked, ignoring the molecular structure comparisons

***checkTemplateRxnProducts* indicates that only the products of the** reaction are checked for isomorphism. This is used when we know the reactants are identical, i.e. in generating reactions.

isUnimolecular(*self*) → bool

Return `True` if the reaction has a single molecule as either reactant or product (or both) $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or $\mathrm{A} \rightleftharpoons \mathrm{B}$, or `False` if not.

is_forward

'bool'

Type is_forward

k_effective_cache

dict

Type k_effective_cache

kinetics

rmgpy.kinetics.model.KineticsModel

Type kinetics

label

str

Type label

matchesSpecies(*self*, *list reactants*, *list products=None*) → bool

Compares the provided reactants and products against the reactants and products of this reaction. Both directions are checked.

Parameters

- **reactants** (*list*) – Species required on one side of the reaction
- **products** (*list, optional*) – Species required on the other side

network_kinetics

rmgpy.kinetics.arrhenius.Arrhenius

Type network_kinetics

pairs

list

Type pairs

products

list

Type products

reactants

list

Type reactants

`reverseThisArrheniusRate`(*self*, *Arrhenius kForward*, *str reverseUnits*)

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

`reversible`

'bool'

Type reversible

`specificCollider`

rmgpy.species.Species

Type specificCollider

`toCantera`(*self*, *speciesList=None*, *useChemkinIdentifier=False*)

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

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

`toChemkin`(*self*, *speciesList=None*, *kinetics=True*)

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.

`toLabeledStr`(*self*, *use_index=False*)

the same as __str__ except that the labels are assumed to exist and used for reactant and products rather than the labels plus the index in parentheses

`transitionState`

rmgpy.species.TransitionState

Type transitionState

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
`Species`	A chemical species, with RMG-specific functionality
`CoreEdgeReactionModel`	A reaction model comprised of core and edge species and reactions

1.11.2 Input

Function	Description
`readInputFile()`	Load an RMG job input file
`saveInputFile()`	Save an RMG job input file

1.11.3 Output

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

1.11.4 Job classes

Class	Description
`RMG`	Main class for RMG jobs

1.11.5 Pressure dependence

Class	Description
`PDepReaction`	A pressure-dependent "net" reaction
`PDepNetwork`	A pressure-dependent unimolecular reaction network, with RMG-specific functionality

rmgpy.rmg.model.CoreEdgeReactionModel

class `rmgpy.rmg.model.`**`CoreEdgeReactionModel`**(*core=None*, *edge=None*, *surface=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
core	The species and reactions of the current model core
edge	The species and reactions of the current model edge
networkDict	A dictionary of pressure-dependent reaction networks (`Network` objects) indexed by source.
networkList	A list of pressure-dependent reaction networks (`Network` objects)
networkCount	A counter for the number of pressure-dependent networks created
indexSpeciesDict	A dictionary with a unique index pointing to the species objects
solventName	String describing solvent name for liquid reactions. Empty for non-liquid estimation

`addNewSurfaceObjects`(*obj*, *newSurfaceSpecies*, *newSurfaceReactions*, *reactionSystem*)

obj is the list of objects for enlargement coming from simulate newSurfaceSpecies and newSurfaceReactions are the current lists of surface species and surface reactions following simulation reactionSystem is the current reactor manages surface species and reactions being moved to and from the surface moves them to appropriate newSurfaceSpc/RxnsAdd/loss sets returns false if the surface has changed

`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.

adjustSurface()
Here we add species intended to be added and remove any species that need to be moved out of the core. For now we remove reactions from the surface that have become part of a PDepNetwork by intersecting the set of surface reactions with the core so that all surface reactions are in the core thus the surface algorithm currently (June 2017) is not implemented for pdep networks (however it will function fine for non-pdep reactions on a pdep run)

applyKineticsToReaction(*reaction*)
retrieve the best kinetics for the reaction and apply it towards the forward or reverse direction (if reverse, flip the direaction).

checkForExistingReaction(*rxn*)
Check to see if an existing reaction has the same reactants, products, and family as *rxn*. Returns `True` or `False` and the matched reaction (if found).

First, a shortlist of reaction is retrieved that have the same reaction keys as the parameter reaction.

Next, the reaction ID containing an identifier (e.g. label) of the reactants and products is compared between the parameter reaction and the each of the reactions in the shortlist. If a match is found, the discovered reaction is returned.

If a match is not yet found, the Library (seed mechs, reaction libs) in the reaction database are iterated over to check if a reaction was overlooked (a reaction with a different "family" key as the parameter reaction).

checkForExistingSpecies(*molecule*)
Check to see if an existing species contains the same `molecule.Molecule` as *molecule*. Returns `True`, *reactive*, and the matched species (if found) or `False`, `False`, and `None` (if not found). *reactive* is a boolean argument which is `False` if this molecule is an unrepresentative resonance structure of an existing species (i.e., was found to be isomorphic only by generating its unfiltered resonance structures) and True otherwise. It is emphasized that *reactive* relates to the `Molecule` attribute.

clearSurfaceAdjustments()
empties surface tracking varaibles

enlarge(*newObject=None*, *reactEdge=False*, *unimolecularReact=None*, *bimolecularReact=None*, *trimolecularReact=None*)
Enlarge a reaction model by processing the objects in the list *newObject*. If *newObject* is a `rmg.species.Species` object, then the species is moved from the edge to the core and reactions generated for that species, reacting with itself and with all other species in the model core. If *newObject* is a `rmg.unirxn.network.Network` object, then reactions are generated for the species in the network with the largest leak flux.

If the *reactEdge* flag is *True*, then no newObject is needed, and instead the algorithm proceeds to react the core species together to form edge reactions.

generateKinetics(*reaction*)
Generate best possible kinetics for the given *reaction* using the kinetics database.

getLists()
Return lists of all of the species and reactions in the core and the edge.

getModelSize()
Return the numbers of species and reactions in the model core and edge. Note that this is not necessarily equal to the lengths of the corresponding species and reaction lists.

getSpecies(*obj*)
Retrieve species object, by polling the index species dictionary.

getStoichiometryMatrix()
Return the stoichiometry matrix for all generated species and reactions. The id of each species and reaction is the corresponding row and column, respectively, in the matrix.

inflate(*rxn*)
Convert reactions from reactants/products that are referring to the core species index, to the respective Species objects.

initializeIndexSpeciesDict()
Populates the core species dictionary

integer -> core Species

with the species that are currently in the core.

makeNewPDepReaction(*forward*)
Make a new pressure-dependent reaction based on a list of *reactants* and a list of *products*. The reaction belongs to the specified *network* and has pressure-dependent kinetics given by *kinetics*.

No checking for existing reactions is made here. The returned PDepReaction object is not added to the global list of reactions, as that is intended to represent only the high-pressure-limit set. The reactionCounter 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. It is emphasized that *reactive* relates to the `Species` attribute, while *reactive_structure* relates to the `Molecule` attribute.

`markChemkinDuplicates`()

Check that all reactions that will appear the chemkin output have been checked as duplicates.

Call this if you've done something that may have introduced undetected duplicate reactions, like add a reaction library or seed mechanism. Anything added via the `expand()` method should already be detected.

`printEnlargeSummary`(*newCoreSpecies*, *newCoreReactions*, *newEdgeSpecies*, *newEdgeReactions*, *reactionsMovedFromEdge=None*, *reactEdge=False*)

Output a summary of a model enlargement step to the log. The details of the enlargement are passed in the *newCoreSpecies*, *newCoreReactions*, *newEdgeSpecies*, and *newEdgeReactions* objects.

`processNewReactions`(*newReactions*, *newSpecies*, *pdepNetwork=None*)

Process a list of newly-generated reactions involving the new core species or explored isomer *newSpecies* in network *pdepNetwork*.

Makes a reaction and decides where to put it: core, edge, or PDepNetwork.

`prune`(*reactionSystems*, *toleranceKeepInEdge*, *toleranceMoveToCore*, *maximumEdgeSpecies*, *minSpeciesExistIterationsForPrune*)

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

`registerReaction`(*rxn*)

Adds the reaction to the reaction database.

The reaction database is structured as a multi-level dictionary, for efficient search and retrieval of existing reactions.

The database has two types of dictionary keys: - reaction family - reactant(s) keys

First, the keys are generated for the parameter reaction.

Next, it is checked whether the reaction database already contains similar keys. If not, a new container is created, either a dictionary for the family key and first reactant key, or a list for the second reactant key.

Finally, the reaction is inserted as the first element in the list.

`removeEmptyPdepNetworks`()

searches for and deletes any empty pdep networks

`removeSpeciesFromEdge`(*reactionSystems*, *spec*)

Remove species *spec* from the reaction model edge.

`retrieve`(*family_label*, *key1*, *key2*)

Returns a list of reactions from the reaction database with the same keys as the parameters.

Returns an empty list when one of the keys could not be found.

`retrieveNewSpecies`(*deflatedRxn*)

Searches for the first reactant or product in the deflated reaction that is represented by an integer.

Such an object refers to a core species that was used to generate the reaction in the first place. Reactants or products represented by an object that is not an integer will be a newly-generated structure.

`searchRetrieveReactions`(*rxn*)

Searches through the reaction database for reactions with an identical reaction key as the key of the parameter reaction.

Both the reaction key based on the reactants as well as on the products is used to search for possible candidate reactions.

setThermodynamicFilteringParameters(*Tmax*, *toleranceThermoKeepSpeciesInEdge*, *minCoreSizeForPrune*, *maximumEdgeSpecies*, *reactionSystems*)

sets parameters for thermodynamic filtering based on the current core Tmax is the maximum reactor temperature in K toleranceThermoKeepSpeciesInEdge is the Gibbs number above which species will be filtered minCoreSizeForPrune is the core size at which thermodynamic filtering will start maximumEdgeSpecies is the maximum allowed number of edge species reactionSystems is a list of reactionSystem objects

thermoFilterDown(*maximumEdgeSpecies*, *minSpeciesExistIterationsForPrune=0*)

removes species from the edge based on their Gibbs energy until maximumEdgeSpecies is reached under the constraint that all removed species are older than minSpeciesExistIterationsForPrune iterations maximumEdgeSpecies is the maximum allowed number of edge species minSpeciesExistIterationsForPrune is the number of iterations a species must be in the edge before it is eligible for thermo filtering

thermoFilterSpecies(*spcs*)

checks Gibbs energy of the species in species against the maximum allowed Gibbs energy

updateUnimolecularReactionNetworks()

Iterate through all of the currently-existing unimolecular reaction networks, updating those that have been marked as invalid. In each update, the phenomonological rate coefficients $k(T, P)$ are computed for each net reaction in the network, and the resulting reactions added or updated.

class `rmgpy.rmg.model.`**ReactionModel**(*species=None*, *reactions=None*)

Represent a generic reaction model. A reaction model consists of *species*, a list of species, and *reactions*, a list of reactions.

merge(*other*)

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

RMG input files

`rmgpy.rmg.input.`**readInputFile**(*path*, *rmg0*)

Read an RMG input file at *path* on disk into the `RMG` object *rmg*.

`rmgpy.rmg.input.`**saveInputFile**(*path*, *rmg*)

Save an RMG input file at *path* on disk from the `RMG` object *rmg*.

rmgpy.rmg.main.RMG

class `rmgpy.rmg.main.`**RMG**(*inputFile=None*, *outputDirectory=None*)

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

Attribute	Description
inputFile	The path to the input file
databaseDirectory	The directory containing the RMG database
thermoLibraries	The thermodynamics libraries to load
reactionLibraries	The kinetics libraries to load
statmechLibraries	The statistical mechanics libraries to load
seedMechanisms	The seed mechanisms included in the model
kineticsFamilies	The kinetics families to use for reaction generation
kineticsDepositories	The kinetics depositories to use for looking up kinetics in each family
kineticsEstimator	The method to use to estimate kinetics: 'group additivity' or 'rate rules'
solvent	If solvation estimates are required, the name of the solvent.
reactionModel	The core-edge reaction model generated by this job

Table 3 – continued from previous page

Attribute	Description
reactionSystems	A list of the reaction systems used in this job
database	The RMG database used in this job
modelSettingsList	List of ModelSettings objects containing information related to how to manage species/reaction mov
simulatorSettingsList	List of SimulatorSettings objects containing information on how to run simulations
trimolecular	`True` to consider reactions between three species (i.e., if trimolecular reaction families are present)
unimolecularThreshold	Array of flags indicating whether a species is above the unimolecular reaction threshold
bimolecularThreshold	Array of flags indicating whether two species are above the bimolecular reaction threshold
trimolecularThreshold	Array of flags indicating whether three species are above the trimolecular reaction threshold
unimolecularReact	Array of flags indicating whether a species should react unimolecularly in the enlarge step
bimolecularReact	Array of flags indicating whether two species should react in the enlarge step
trimolecularReact	Array of flags indicating whether three species should react in the enlarge step
termination	A list of termination targets (i.e `TerminationTime` and `TerminationConversion` objects)
speciesConstraints	Dictates the maximum number of atoms, carbons, electrons, etc. generated by RMG
outputDirectory	The directory used to save output files
verbosity	The level of logging verbosity for console output
loadRestart	`True` if restarting a previous job, `False` otherwise
saveRestartPeriod	The time period to periodically save a restart file (`Quantity`), or `None` for never.
units	The unit system to use to save output files (currently must be ‘si’)
generateOutputHTML	`True` to draw pictures of the species and reactions, saving a visualized model in an output HTML fi
generatePlots	`True` to generate plots of the job execution statistics after each iteration, `False` otherwise
verboseComments	`True` to keep the verbose comments for database estimates, `False` otherwise
saveEdgeSpecies	`True` to save chemkin and HTML files of the edge species, `False` otherwise
keepIrreversible	`True` to keep ireversibility of library reactions as is (‘<=>’ or ‘=>’). `False` (default) to force all libr
trimolecularProductReversible	`True` (default) to allow families with trimolecular products to react in the reverse direction, `False`
pressureDependence	Whether to process unimolecular (pressure-dependent) reaction networks
quantumMechanics	Whether to apply quantum mechanical calculations instead of group additivity to certain molecular
ml_estimator	To use thermo estimation with machine learning
ml_settings	Settings for ML estimation
wallTime	The maximum amount of CPU time in the form DD:HH:MM:SS to expend on this job; used to stop
kineticsdatastore	`True` if storing details of each kinetic database entry in text file, `False` otherwise
initializationTime	The time at which the job was initiated, in seconds since the epoch (i.e. from time.time())
done	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.

check_model()

Run checks on the RMG model

clear()

Clear all loaded information about the job (except the file paths).

execute(***kwargs*)

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

finish()

Complete the model generation.

generateCanteraFiles(*chemkinFile*, ***kwargs*)

Convert a chemkin mechanism chem.inp file to a cantera mechanism file chem.cti and save it in the cantera directory

`initialize`(***kwargs*)
Initialize an RMG job using the command-line arguments *args* as returned by the `argparse` package.

`loadInput`(*path=None*)
Load an RMG job from the input file located at *inputFile*, or from the *inputFile* attribute if not given as a parameter.

`loadRMGJavaInput`(*path*)
Load an RMG-Java job from the input file located at *inputFile*, or from the *inputFile* attribute if not given as a parameter.

`loadRestartFile`(*path*)
Load a restart file at *path* on disk.

`loadThermoInput`(*path=None*)
Load an Thermo Estimation job from a thermo input file located at *inputFile*, or from the *inputFile* attribute if not given as a parameter.

`logHeader`(*level=20*)
Output a header containing identifying information about RMG to the log.

`makeSeedMech`(*firstTime=False*)
causes RMG to make a seed mechanism out of the current chem_annotated.inp and species_dictionary.txt this seed mechanism is outputted in a seed folder within the run directory and automatically added to as the (or replaces the current) 'Seed' thermo and kinetics libraries in database

if run with firstTime=True it will change self.name to be unique within the thermo/kinetics libraries by adding integers to the end of the name to prevent overwritting

`makeSpeciesLabelsIndependent`(*species*)
This method looks at the core species labels and makes sure none of them conflict If a conflict occurs, the second occurance will have '-2' added returns a list of the old labels

`processPdepNetworks`(*obj*)
properly processes PDepNetwork objects and lists of PDepNetwork objects returned from simulate

`processReactionsToSpecies`(*obj*)
properly processes Reaction objects and lists of Reaction objects returned from simulate

`processToSpeciesNetworks`(*obj*)
breaks down the objects returned by simulate into Species and PDepNetwork components

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

`register_listeners`()
Attaches listener classes depending on the options found in the RMG input file.

`saveEverything`()
Saves the output HTML, the Chemkin file, and the Restart file (if appropriate).

The restart file is only saved if self.saveRestartPeriod or self.done.

`saveInput`(*path=None*)
Save an RMG job to the input file located at *path*, or from the *outputFile* attribute if not given as a parameter.

updateReactionThresholdAndReactFlags(*rxnSysUnimolecularThreshold=None*, *rxnSysBimolecularThreshold=None*, *rxnSysTrimolecularThreshold=None*, *skipUpdate=False*)
updates the length and boolean value of the unimolecular and bimolecular react and threshold flags

`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
source	`list`	The isomer or reactant channel that acts as the source
explored	`list`	A list of the unimolecular isomers whose reactions have been fully explored

addPathReaction(*newReaction*)
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 intermedate arrays used to compute k(T,P) values.

exploreIsomer(*isomer*)

Explore a previously-unexplored unimolecular *isomer* in this partial network using the provided core-edge reaction model *reactionModel*, returning the new reactions and new species.

getAllSpecies()

Return a list of all unique species in the network, including all isomers, reactant and product channels, and bath gas species.

getLeakBranchingRatios(*T*, *P*)

Return a dict with the unexplored isomers in the partial network as the keys and the fraction of the total leak coefficient as the values.

getLeakCoefficient(*T*, *P*)

Return the pressure-dependent rate coefficient $k(T, P)$ describing the total rate of "leak" from this network. This is defined as the sum of the $k(T, P)$ values for all net reactions to 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.

get_energy_filtered_reactions(*T*, *tol*)

Returns a list of products and isomers that are greater in Free Energy than a*R*T + Gfsource(T)

get_rate_filtered_reactions(*T*, *P*, *tol*)

determines the set of pathReactions that have fluxes less than tol at steady state where all A => B + C reactions are irreversible and there is a constant flux from/to the source configuration of 1.0

initialize(*Tmin*, *Tmax*, *Pmin*, *Pmax*, *maximumGrainSize=0.0*, *minimumGrainCount=0*, *activeJRotor=True*, *activeKRotor=True*, *rmgmode=False*)

Initialize a pressure dependence calculation by computing several quantities that are independent of the conditions. You must specify the temperature and pressure ranges of interesting using *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.

remove_disconnected_reactions()
gets rid of reactions/isomers/products not connected to the source by a reaction sequence

remove_reactions(*reactionModel*, *rxns*)
removes a list of reactions from the network and all reactions/products left disconnected by removing those reactions

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 $k_\mathrm{B}T$ 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.

solve_SS_network(*T*, *P*)
calculates the steady state concentrations if all A => B + C reactions are irreversible and the flux from/to the source configuration is 1.0

update(*reactionModel*, *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*, *specificCollider=None*, *network=None*, *kinetics=None*, *network_kinetics=None*, *reversible=True*, *transitionState=None*, *duplicate=False*, *degeneracy=1*, *pairs=None*)

allow_max_rate_violation
'bool'

Type allow_max_rate_violation

allow_pdep_route
'bool'

Type allow_pdep_route

calculateMicrocanonicalRateCoefficient(*self*, *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.

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

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

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.

calculateTSTRateCoefficients(*self*, *ndarray Tlist*) → ndarray

calculate_coll_limit(*self*, *float temp*, *bool reverse=False*)
Calculate the collision limit rate for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

canTST(*self*) → bool
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.

check_collision_limit_violation(*self*, *float t_min*, *float t_max*, *float p_min*, *float p_max*) → list
Warn if a core reaction violates the collision limit rate in either the forward or reverse direction at the relevant extreme T/P conditions. Assuming a monotonic behaviour of the kinetics. Returns a list with the reaction object and the direction in which the violation was detected.

comment
str

Type comment

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

degeneracy
Reaction.__getDegneneracy(self)

draw(*self*, *path*)
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.

duplicate
'bool'

Type duplicate

elementary_high_p
'bool'

Type elementary_high_p

ensure_species(*self*, *bool reactant_resonance=False*, *bool product_resonance=True*)
Ensure the reaction contains species objects in its reactant and product attributes. If the reaction is found to hold molecule objects, it modifies the reactant, product and pairs to hold Species objects.

Generates resonance structures for Molecules if the corresponding options, reactant_resonance and/or product_resonance, are True. Does not generate resonance for reactants or products that start as Species objects.

fixBarrierHeight(*self*, *bool forcePositive=False*)
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(*self*, *T*)
Decrease the pre-exponential factor (A) by the diffusion factor to account for the diffusion limit at the specified temperature.

generate3dTS(*self*, *reactants*, *products*)
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(*self*)

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 (C/O/N/S at the moment). This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generateReverseRateCoefficient(*self*, *bool network_kinetics=False*)

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

getEnthalpiesOfReaction(*self*, *ndarray Tlist*) → ndarray

Return the enthalpies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

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

getEntropiesOfReaction(*self*, *ndarray Tlist*) → ndarray

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

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

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

getEquilibriumConstant(*self*, *double T*, *str type='Kc'*) → double

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(*self*, *ndarray Tlist*, *str type='Kc'*) → ndarray

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(*self*, *ndarray Tlist*) → ndarray

Return the Gibbs free energies of reaction in J/mol evaluated at temperatures *Tlist* in K.

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

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

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

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(*self*, *Species spec*) → int

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

get_mean_sigma_and_epsilon(*self*, *bool reverse=False*)
Calculates the collision diameter (sigma) using an arithmetic mean Calculates the well depth (epsilon) using a geometric mean If reverse is `False` the above is calculated for the reactants, otherwise for the products

get_reduced_mass(*self*, *bool reverse=False*)
Returns the reduced mass of the reactants if reverse is `False` Returns the reduced mass of the products if reverse is `True`

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

index
'int'

Type index

isAssociation(*self*) → bool
Return `True` if the reaction represents an association reaction $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or `False` if not.

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

isDissociation(*self*) → bool
Return `True` if the reaction represents a dissociation reaction $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or `False` if not.

isIsomerization(*self*) → bool
Return `True` if the reaction represents an isomerization reaction $\mathrm{A} \rightleftharpoons \mathrm{B}$ or `False` if not.

isIsomorphic(*self*, *Reaction other*, *bool eitherDirection=True*, *bool checkIdentical=False*, *bool checkOnlyLabel=False*, *bool checkTemplateRxnProducts=False*) → bool
Return `True` if this reaction is the same as the *other* reaction, or `False` if they are different. The comparison involves comparing isomorphism of reactants and products, and doesn't use any kinetic information.

If *eitherDirection=False* then the directions must match.

***checkIdentical* indicates that atom ID's must match and is used in** checking degeneracy

***checkOnlyLabel* indicates that the string representation will be** checked, ignoring the molecular structure comparisons

***checkTemplateRxnProducts* indicates that only the products of the** reaction are checked for isomorphism. This is used when we know the reactants are identical, i.e. in generating reactions.

isUnimolecular(*self*) → bool
Return `True` if the reaction has a single molecule as either reactant or product (or both) $\mathrm{A} \rightleftharpoons \mathrm{B} + \mathrm{C}$ or $\mathrm{A} + \mathrm{B} \rightleftharpoons \mathrm{C}$ or $\mathrm{A} \rightleftharpoons \mathrm{B}$, or `False` if not.

is_forward
'bool'

Type is_forward

k_effective_cache
dict

Type k_effective_cache

kinetics
rmgpy.kinetics.model.KineticsModel

Type kinetics

label
str

Type label

matchesSpecies(*self*, *list reactants*, *list products=None*) → bool
Compares the provided reactants and products against the reactants and products of this reaction. Both directions are checked.

Parameters

- **reactants** (*list*) – Species required on one side of the reaction
- **products** (*list, optional*) – Species required on the other side

network_kinetics
rmgpy.kinetics.arrhenius.Arrhenius

Type network_kinetics

pairs
list

Type pairs

products
list

Type products

reactants
list

Type reactants

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

reversible
'bool'

Type reversible

specificCollider
rmgpy.species.Species

Type specificCollider

toCantera(*self*, *speciesList=None*, *useChemkinIdentifier=False*)
Converts the RMG Reaction object to a Cantera Reaction object with the appropriate reaction class.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

toChemkin(*self*, *speciesList=None*, *kinetics=True*)
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.

toLabeledStr(*self*, *use_index=False*)

the same as __str__ except that the labels are assumed to exist and used for reactant and products rather than the labels plus the index in parentheses

transitionState

rmgpy.species.TransitionState

Type transitionState

rmgpy.rmg.model.Species

class rmgpy.rmg.model.**Species**(*index=-1*, *label=''*, *thermo=None*, *conformer=None*, *molecule=None*, *transportData=None*, *molecularWeight=None*, *energyTransferModel=None*, *reactive=True*, *props=None*, *aug_inchi=None*, *symmetryNumber=-1*, *creationIteration=0*, *explicitlyAllowed=False*)

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

Attribute	Description
index	A unique nonnegative integer index
label	A descriptive string label
thermo	The heat capacity model for the species
conformer	The molecular conformer for the species
molecule	A list of the `Molecule` objects describing the molecular structure
transportData	A set of transport collision parameters
molecularWeight	The molecular weight of the species
energyTransferModel	The collisional energy transfer model to use
reactive	**`True` if the species participates in reaction families, `False` if not** Reaction libraries and seed mechanisms that include the species are always considered regardless of this variable
props	A generic 'properties' dictionary to store user-defined flags
aug_inchi	Unique augmented inchi
isSolvent	Boolean describing whether this species is the solvent
creationIteration	Iteration which the species is created within the reaction mechanism generation algorithm

aug_inchi

str

Type aug_inchi

calculateCp0(*self*) → double

Return the value of the heat capacity at zero temperature in J/mol*K.

calculateCpInf(*self*) → double

Return the value of the heat capacity at infinite temperature in J/mol*K.

conformer

rmgpy.statmech.conformer.Conformer

Type conformer

copy(*self*, *bool deep=False*) → Species
Create a copy of the current species. If the kw argument 'deep' is True, then a deep copy will be made of the Molecule objects in self.molecule.

For other complex attributes, a deep copy will always be made.

creationIteration
'int'

Type creationIteration

energyTransferModel
object

Type energyTransferModel

explicitlyAllowed
'bool'

Type explicitlyAllowed

fromAdjacencyList(*self*, *adjlist*)
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(*self*, *smiles*)
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(*self*)
Generate the collisional energy transfer model parameters for the species. This "algorithm" is *very* much in need of improvement.

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

generateTransportData(*self*)
Generate the transportData parameters for the species.

generate_aug_inchi(*self*)

generate_resonance_structures(*self*, *bool keep_isomorphic=True*, *bool filter_structures=True*)
Generate all of the resonance structures 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 structures have already been generated.

getAugmentedInChI(*self*)

getDensityOfStates(*self*, *ndarray Elist*) → ndarray
Return the density of states $\rho(E)\ dE$ at the specified energies *Elist* in J/mol above the ground state.

getEnthalpy(*self*, *double T*) → double
Return the enthalpy in J/mol for the species at the specified temperature *T* in K.

getEntropy(*self*, *double T*) → double
Return the entropy in J/mol*K for the species at the specified temperature *T* in K.

getFreeEnergy(*self*, *double T*) → double
Return the Gibbs free energy in J/mol for the species at the specified temperature *T* in K.

getHeatCapacity(*self*, *double T*) → double
Return the heat capacity in J/mol*K for the species at the specified temperature *T* in K.

getPartitionFunction(*self*, *double T*) → double
Return the partition function for the species at the specified temperature T in K.

getResonanceHybrid(*self*)
Returns a molecule object with bond orders that are the average of all the resonance structures.

getSumOfStates(*self*, *ndarray Elist*) → ndarray
Return the sum of states $N(E)$ at the specified energies *Elist* in J/mol.

getSymmetryNumber(*self*)
Get the symmetry number for the species, which is the highest symmetry number amongst its resonance isomers and the resonance hybrid. This function is currently used for website purposes and testing only as it requires additional calculateSymmetryNumber calls.

getThermoData(*self*, *solventName=''*)
Returns a *thermoData* object of the current Species object.

If the thermo object already exists, it is either of the (Wilhoit, ThermoData) type, or it is a Future.

If the type of the thermo attribute is Wilhoit, or ThermoData, then it is converted into a NASA format.

If it is a Future, then a blocking call is made to retrieve the NASA object. If the thermo object did not exist yet, the thermo object is generated.

getTransportData(*self*)
Returns the transport data associated with this species, and calculates it if it is not yet available.

hasStatMech(*self*) → bool
Return `True` if the species has statistical mechanical parameters, or `False` otherwise.

hasThermo(*self*) → bool
Return `True` if the species has thermodynamic parameters, or `False` otherwise.

has_reactive_molecule(*self*) → bool
True if the species has at least one reactive molecule, *False* otherwise

index
'int'

Type index

isIdentical(*self*, *other*) → bool
Return `True` if at least one molecule of the species is identical to *other*, which can be either a `Molecule` object or a `Species` object.

isIsomorphic(*self*, *other*, *bool generate_res=False*) → bool
Return `True` if the species is isomorphic to *other*, which can be either a `Molecule` object or a `Species` object. If generate_res is `True` and other is a `Species` object, the resonance structures of other will be generated and isomorphically compared against self. This is useful for situations where a "non-representative" resonance structure of self is generated, and it should be identified as the same Species, and be assigned a reactive=False flag.

isSolvent
'bool'

Type isSolvent

is_structure_in_list(*self*, *list species_list*) → bool
Return `True` if at least one Molecule in self is isomorphic with at least one other Molecule in at least one Species in species list.

label
str

Type label

`molecularWeight`
value_si is in kg/molecule not kg/mol)

Type The molecular weight of the species. (Note

`molecule`
list

Type molecule

`props`
dict

Type props

`reactive`
'bool'

Type reactive

`setE0WithThermo`(*self*)
Helper method that sets species' E0 using the species' thermo data

`symmetryNumber`
'float'

Type symmetryNumber

`thermo`
object

Type thermo

`toAdjacencyList`(*self*)
Return a string containing each of the molecules' adjacency lists.

`toCantera`(*self*, *useChemkinIdentifier=False*)
Converts the RMG Species object to a Cantera Species object with the appropriate thermo data.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

`toChemkin`(*self*)
Return the chemkin-formatted string for this species.

`transportData`
object

Type transportData

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
ReactionSystem	Base class for all reaction systems
SimpleReactor	A simple isothermal, isobaric, well-mixed batch reactor
LiquidReactor	A homogeneous, isothermal, isobaric liquid batch reactor

1.12.2 Termination criteria

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

rmgpy.solver.ReactionSystem

class rmgpy.solver.**ReactionSystem**(*termination=None*, *sensitiveSpecies=None*, *sensitivityThreshold=1e-3*)

A base class for all RMG reaction systems.

Keq

numpy.ndarray

Type Keq

addReactionsToSurface(*self*, *list newSurfaceReactions*, *list newSurfaceReactionInds*, *list surfaceSpecies*, *list surfaceReactions*, *list edgeSpecies*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

atol_array

numpy.ndarray

Type atol_array

bimolecularThreshold

numpy.ndarray

Type bimolecularThreshold

computeRateDerivative(*self*)

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

compute_network_variables(*self*, *pdepNetworks=None*)

Initialize the arrays containing network information:

- **NetworkLeakCoefficients is a n x 1 array with** n the number of pressure-dependent networks.
- **NetworkIndices is a n x 3 matrix with** n the number of pressure-dependent networks and 3 the maximum number of molecules allowed in either the reactant or product side of a reaction.

coreReactionRates

numpy.ndarray

Type coreReactionRates

coreSpeciesConcentrations
numpy.ndarray

Type coreSpeciesConcentrations

coreSpeciesConsumptionRates
numpy.ndarray

Type coreSpeciesConsumptionRates

coreSpeciesProductionRates
numpy.ndarray

Type coreSpeciesProductionRates

coreSpeciesRates
numpy.ndarray

Type coreSpeciesRates

dydt0
numpy.ndarray

Type dydt0

edgeReactionRates
numpy.ndarray

Type edgeReactionRates

edgeSpeciesRates
numpy.ndarray

Type edgeSpeciesRates

generate_reactant_product_indices(*self*, *coreReactions*, *edgeReactions*)
Creates a matrix for the reactants and products.

generate_reaction_indices(*self*, *coreReactions*, *edgeReactions*)
Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*self*, *coreSpecies*, *edgeSpecies*)
Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

getLayeringIndices(*self*)
determines the edge reaction indices that indicate reactions that are valid for movement from edge to surface based on the layering constraint

get_species_index(*self*, *spc*)
Retrieves the index that is associated with the parameter species from the species index dictionary.

initialize()
Initialize the DASPK solver by setting the initial values of the independent variable *t0*, dependent variables *y0*, and first derivatives *dydt0*. If provided, the derivatives must be consistent with the other initial conditions; if not provided, DASPK will attempt to estimate a consistent set of initial values for the derivatives. You can also set the absolute and relative tolerances *atol* and *rtol*, respectively, either as single values for all dependent variables or individual values for each dependent variable.

initializeModel(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies=None*, *list surfaceReactions=None*, *list pdepNetworks=None*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*, *filterReactions=False*, *dict conditions=None*)
Initialize a simulation of the reaction system using the provided kinetic model. You will probably want to

create your own version of this method in the derived class; don't forget to also call the base class version, too.

`initialize_solver`(*self*)

`initialize_surface`(*self*, *list coreSpecies*, *list coreReactions*, *list surfaceSpecies*, *list surfaceReactions*)

removes surfaceSpecies and surfaceReactions from until they are self consistent:

1) every reaction has one species in the surface

2) every species participates in a surface reaction

`initiate_tolerances`(*self*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*)

Computes the number of differential equations and initializes the tolerance arrays.

`jacobianMatrix`

numpy.ndarray

Type jacobianMatrix

`kb`

numpy.ndarray

Type kb

`kf`

numpy.ndarray

Type kf

`logConversions`(*self*, *speciesIndex*, *y0*)

Log information about the current conversion values.

`logRates`(*self*, *double charRate*, *species*, *double speciesRate*, *double maxDifLnAccumNum*, *network*, *double networkRate*)

Log information about the current maximum species and network rates.

`maxEdgeSpeciesRateRatios`

numpy.ndarray

Type maxEdgeSpeciesRateRatios

`maxNetworkLeakRateRatios`

numpy.ndarray

Type maxNetworkLeakRateRatios

`neq`

'int'

Type neq

`networkIndices`

numpy.ndarray

Type networkIndices

`networkLeakCoefficients`

numpy.ndarray

Type networkLeakCoefficients

`networkLeakRates`

numpy.ndarray

Type networkLeakRates

numCoreReactions
'int'

Type numCoreReactions

numCoreSpecies
'int'

Type numCoreSpecies

numEdgeReactions
'int'

Type numEdgeReactions

numEdgeSpecies
'int'

Type numEdgeSpecies

numPdepNetworks
'int'

Type numPdepNetworks

productIndices
numpy.ndarray

Type productIndices

prunableNetworkIndices
numpy.ndarray

Type prunableNetworkIndices

prunableNetworks
list

Type prunableNetworks

prunableSpecies
list

Type prunableSpecies

prunableSpeciesIndices
numpy.ndarray

Type prunableSpeciesIndices

reactantIndices
numpy.ndarray

Type reactantIndices

reactionIndex
dict

Type reactionIndex

reset_max_edge_species_rate_ratios(*self*)
This function sets maxEdgeSpeciesRateRatios back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

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).

rtol_array

numpy.ndarray

Type rtol_array

sensitiveSpecies

list

Type sensitiveSpecies

sensitivityCoefficients

numpy.ndarray

Type sensitivityCoefficients

sensitivityThreshold

'double'

Type sensitivityThreshold

set_initial_conditions(*self*)

Sets the common initial conditions of the rate equations that represent the reaction system.

- Sets the initial time of the reaction system to 0
- Initializes the species moles to a n x 1 array with zeros

set_initial_derivative(*self*)

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

set_initial_reaction_thresholds(*self*)

set_prunable_indices(*self*, *edgeSpecies*, *pdepNetworks*)

simulate(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies*, *list surfaceReactions*, *list pdepNetworks=None*, *bool prune=False*, *bool sensitivity=False*, *list sensWorksheet=None*, *modelSettings=None*, *simulatorSettings=None*, *dict conditions=None*)

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.

snapshots

list

Type snapshots

speciesIndex

dict

Type speciesIndex

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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

surfaceReactionIndices
numpy.ndarray

Type surfaceReactionIndices

surfaceSpeciesIndices
numpy.ndarray

Type surfaceSpeciesIndices

t0
'float'

Type t0

termination
list

Type termination

trimolecular
'bool'

Type trimolecular

trimolecularThreshold
numpy.ndarray

Type trimolecularThreshold

unimolecularThreshold
numpy.ndarray

Type unimolecularThreshold

validLayeringIndices
numpy.ndarray

Type validLayeringIndices

y0
numpy.ndarray

Type y0

rmgpy.solver.SimpleReactor

class rmgpy.solver.**SimpleReactor**(*T*, *P*, *initialMoleFractions*, *nSims=1*, *termination=None*, *sensitiveSpecies=None*, *sensitivityThreshold=1e-3*, *sensConditions=None*)

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.

Keq
numpy.ndarray

Type Keq

P
rmgpy.quantity.ScalarQuantity

Type P

Prange
list

Type Prange

T
rmgpy.quantity.ScalarQuantity

Type T

Trange
list

Type Trange

V
'double'

Type V

addReactionsToSurface(*self*, *list newSurfaceReactions*, *list newSurfaceReactionInds*, *list surfaceSpecies*, *list surfaceReactions*, *list edgeSpecies*)
moves new surface reactions to the surface done after the while loop before the simulate call ends

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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

atol_array
numpy.ndarray

Type atol_array

bimolecularThreshold
numpy.ndarray

Type bimolecularThreshold

calculate_effective_pressure(*self*, *rxn*)
Computes the effective pressure for a reaction as:

$$P_{eff} = P * \sum_i \frac{y_i * eff_i}{\sum_j y_j}$$

with:

- P the pressure of the reactor,
- y the array of initial moles of the core species

or as:

$$P_{eff} = \frac{P * y_{specificCollider}}{\sum_j y_j}$$

if a specificCollider is mentioned.

colliderEfficiencies
numpy.ndarray

Type colliderEfficiencies

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

compute_network_variables(*self*, *pdepNetworks=None*)

Initialize the arrays containing network information:

- **NetworkLeakCoefficients is a n x 1 array with** n the number of pressure-dependent networks.
- **NetworkIndices is a n x 3 matrix with** n the number of pressure-dependent networks and 3 the maximum number of molecules allowed in either the reactant or product side of a reaction.

constantVolume

'bool'

Type constantVolume

convertInitialKeysToSpeciesObjects(*self*, *speciesDict*)

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

coreReactionRates

numpy.ndarray

Type coreReactionRates

coreSpeciesConcentrations

numpy.ndarray

Type coreSpeciesConcentrations

coreSpeciesConsumptionRates

numpy.ndarray

Type coreSpeciesConsumptionRates

coreSpeciesProductionRates

numpy.ndarray

Type coreSpeciesProductionRates

coreSpeciesRates

numpy.ndarray

Type coreSpeciesRates

dydt0

numpy.ndarray

Type dydt0

edgeReactionRates

numpy.ndarray

Type edgeReactionRates

edgeSpeciesRates

numpy.ndarray

Type edgeSpeciesRates

generate_rate_coefficients(*self*, *coreReactions*, *edgeReactions*)

Populates the forward rate coefficients (kf), reverse rate coefficients (kb) and equilibrium constants (Keq) arrays with the values computed at the temperature and (effective) pressure of the reaction system.

generate_reactant_product_indices(*self*, *coreReactions*, *edgeReactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*self*, *coreReactions*, *edgeReactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*self*, *coreSpecies*, *edgeSpecies*)
Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

getLayeringIndices(*self*)
determines the edge reaction indices that indicate reactions that are valid for movement from edge to surface based on the layering constraint

get_species_index(*self*, *spc*)
Retrieves the index that is associated with the parameter species from the species index dictionary.

get_threshold_rate_constants(*self*, *modelSettings*)
Get the threshold rate constants for reaction filtering.

initialMoleFractions
dict

Type initialMoleFractions

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(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies=None*, *list surfaceReactions=None*, *list pdepNetworks=None*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*, *filterReactions=False*, *dict conditions=None*)
Initialize a simulation of the simple reactor using the provided kinetic model.

initialize_solver(*self*)

initialize_surface(*self*, *list coreSpecies*, *list coreReactions*, *list surfaceSpecies*, *list surfaceReactions*)

removes surfaceSpecies and surfaceReactions from until they are self consistent:

1) every reaction has one species in the surface
2) every species participates in a surface reaction

initiate_tolerances(*self*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*)
Computes the number of differential equations and initializes the tolerance arrays.

jacobian(*self*, *double t*, *ndarray y*, *ndarray dydt*, *double cj*, *ndarray senpar=numpy.zeros(1, numpy.float64)*)
Return the analytical Jacobian for the reaction system.

jacobianMatrix
numpy.ndarray

Type jacobianMatrix

kb
numpy.ndarray

Type kb

kf
numpy.ndarray

Type kf

logConversions(*self*, *speciesIndex*, *y0*)
Log information about the current conversion values.

logRates(*self*, *double charRate*, *species*, *double speciesRate*, *double maxDifLnAccumNum*, *network*, *double networkRate*)
Log information about the current maximum species and network rates.

maxEdgeSpeciesRateRatios
numpy.ndarray

Type maxEdgeSpeciesRateRatios

maxNetworkLeakRateRatios
numpy.ndarray

Type maxNetworkLeakRateRatios

nSims
'int'

Type nSims

neq
'int'

Type neq

networkIndices
numpy.ndarray

Type networkIndices

networkLeakCoefficients
numpy.ndarray

Type networkLeakCoefficients

networkLeakRates
numpy.ndarray

Type networkLeakRates

numCoreReactions
'int'

Type numCoreReactions

numCoreSpecies
'int'

Type numCoreSpecies

numEdgeReactions
'int'

Type numEdgeReactions

numEdgeSpecies
'int'

Type numEdgeSpecies

numPdepNetworks
'int'

Type numPdepNetworks

pdepColliderKinetics
list

Type pdepColliderKinetics

pdepColliderReactionIndices
numpy.ndarray

Type pdepColliderReactionIndices

pdepSpecificColliderKinetics
list

Type pdepSpecificColliderKinetics

pdepSpecificColliderReactionIndices
numpy.ndarray

Type pdepSpecificColliderReactionIndices

productIndices
numpy.ndarray

Type productIndices

prunableNetworkIndices
numpy.ndarray

Type prunableNetworkIndices

prunableNetworks
list

Type prunableNetworks

prunableSpecies
list

Type prunableSpecies

prunableSpeciesIndices
numpy.ndarray

Type prunableSpeciesIndices

reactantIndices
numpy.ndarray

Type reactantIndices

reactionIndex
dict

Type reactionIndex

reset_max_edge_species_rate_ratios(*self*)
This function sets maxEdgeSpeciesRateRatios back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*self*, *double t*, *ndarray y*, *ndarray dydt*, *ndarray senpar=numpy.zeros(1*, *numpy.float64)*)
Return the residual function for the governing DAE system for the simple reaction system.

rtol_array
numpy.ndarray

Type rtol_array

sensConditions
dict

Type sensConditions

sensitiveSpecies
list

Type sensitiveSpecies

sensitivityCoefficients
numpy.ndarray

Type sensitivityCoefficients

sensitivityThreshold
'double'

Type sensitivityThreshold

set_colliders(*self*, *coreReactions*, *edgeReactions*, *coreSpecies*)
Store collider efficiencies and reaction indices for pdep reactions that have collider efficiencies, and store specific collider indices

set_initial_conditions(*self*)
Sets the initial conditions of the rate equations that represent the current reactor model.

The volume is set to the value derived from the ideal gas law, using the user-defined pressure, temperature, and the number of moles of initial species.

The species moles array (y0) is set to the values stored in the initial mole fractions dictionary.

The initial species concentration is computed and stored in the coreSpeciesConcentrations array.

set_initial_derivative(*self*)
Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

set_initial_reaction_thresholds(*self*)

set_prunable_indices(*self*, *edgeSpecies*, *pdepNetworks*)

simulate(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies*, *list surfaceReactions*, *list pdepNetworks=None*, *bool prune=False*, *bool sensitivity=False*, *list sensWorksheet=None*, *modelSettings=None*, *simulatorSettings=None*, *dict conditions=None*)
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.

snapshots
list

Type snapshots

speciesIndex
dict

Type speciesIndex

specificColliderSpecies
list

Type specificColliderSpecies

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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

surfaceReactionIndices

numpy.ndarray

Type surfaceReactionIndices

surfaceSpeciesIndices

numpy.ndarray

Type surfaceSpeciesIndices

t0

'float'

Type t0

termination

list

Type termination

trimolecular

'bool'

Type trimolecular

trimolecularThreshold

numpy.ndarray

Type trimolecularThreshold

unimolecularThreshold

numpy.ndarray

Type unimolecularThreshold

validLayeringIndices

numpy.ndarray

Type validLayeringIndices

y0

numpy.ndarray

Type y0

rmgpy.solver.LiquidReactor

class rmgpy.solver.**LiquidReactor**(*T*, *initialConcentrations*, *nSims=1*, *termination=None*, *sensitiveSpecies=None*, *sensitivityThreshold=1e-3*, *sensConditions=None*, *constSPCNames=None*)

A reaction system consisting of a homogeneous, isothermal, constant volume batch reactor. These assumptions allow for a number of optimizations that enable this solver to complete very rapidly, even for large kinetic models.

Keq

numpy.ndarray

Type Keq

P
rmgpy.quantity.ScalarQuantity

Type P

T
rmgpy.quantity.ScalarQuantity

Type T

Trange
list

Type Trange

V
'double'

Type V

addReactionsToSurface(*self*, *list newSurfaceReactions*, *list newSurfaceReactionInds*, *list surfaceSpecies*, *list surfaceReactions*, *list edgeSpecies*)
moves new surface reactions to the surface done after the while loop before the simulate call ends

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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

atol_array
numpy.ndarray

Type atol_array

bimolecularThreshold
numpy.ndarray

Type bimolecularThreshold

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

compute_network_variables(*self*, *pdepNetworks=None*)
Initialize the arrays containing network information:

- **NetworkLeakCoefficients is a n x 1 array with** n the number of pressure-dependent networks.
- **NetworkIndices is a n x 3 matrix with** n the number of pressure-dependent networks and 3 the maximum number of molecules allowed in either the reactant or product side of a reaction.

constSPCIndices
list

Type constSPCIndices

constSPCNames
list

Type constSPCNames

constantVolume
'bool'

Type constantVolume

convertInitialKeysToSpeciesObjects(*self*, *speciesDict*)
Convert the initialConcentrations dictionary from species names into species objects, using the given dictionary of species.

coreReactionRates
numpy.ndarray

Type coreReactionRates

coreSpeciesConcentrations
numpy.ndarray

Type coreSpeciesConcentrations

coreSpeciesConsumptionRates
numpy.ndarray

Type coreSpeciesConsumptionRates

coreSpeciesProductionRates
numpy.ndarray

Type coreSpeciesProductionRates

coreSpeciesRates
numpy.ndarray

Type coreSpeciesRates

dydt0
numpy.ndarray

Type dydt0

edgeReactionRates
numpy.ndarray

Type edgeReactionRates

edgeSpeciesRates
numpy.ndarray

Type edgeSpeciesRates

generate_rate_coefficients(*self*, *coreReactions*, *edgeReactions*)
Populates the forwardRateCoefficients, reverseRateCoefficients and equilibriumConstants arrays with the values computed at the temperature and (effective) pressure of the reacion system.

generate_reactant_product_indices(*self*, *coreReactions*, *edgeReactions*)
Creates a matrix for the reactants and products.

generate_reaction_indices(*self*, *coreReactions*, *edgeReactions*)
Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*self*, *coreSpecies*, *edgeSpecies*)
Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

getLayeringIndices(*self*)
determines the edge reaction indices that indicate reactions that are valid for movement from edge to surface based on the layering constraint

get_constSPCIndices(*self*, *coreSpecies*)
Allow to identify constant Species position in solver

get_species_index(*self*, *spc*)
Retrieves the index that is associated with the parameter species from the species index dictionary.

get_threshold_rate_constants(*self*, *modelSettings*)
Get the threshold rate constants for reaction filtering.

modelSettings is not used here, but is needed so that the method matches the one in simpleReactor.

initialConcentrations
dict

Type initialConcentrations

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(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies=None*, *list surfaceReactions=None*, *list pdepNetworks=None*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*, *filterReactions=False*, *dict conditions=None*)
Initialize a simulation of the liquid reactor using the provided kinetic model.

initialize_solver(*self*)

initialize_surface(*self*, *list coreSpecies*, *list coreReactions*, *list surfaceSpecies*, *list surfaceReactions*)

removes surfaceSpecies and surfaceReactions from until they are self consistent:

1) every reaction has one species in the surface
2) every species participates in a surface reaction

initiate_tolerances(*self*, *atol=1e-16*, *rtol=1e-8*, *sensitivity=False*, *sens_atol=1e-6*, *sens_rtol=1e-4*)
Computes the number of differential equations and initializes the tolerance arrays.

jacobian(*self*, *double t*, *ndarray y*, *ndarray dydt*, *double cj*, *ndarray senpar=numpy.zeros(1, numpy.float64)*)
Return the analytical Jacobian for the reaction system.

jacobianMatrix
numpy.ndarray

Type jacobianMatrix

kb
numpy.ndarray

Type kb

kf
numpy.ndarray

Type kf

logConversions(*self*, *speciesIndex*, *y0*)
Log information about the current conversion values.

logRates(*self*, *double charRate*, *species*, *double speciesRate*, *double maxDifLnAccumNum*, *network*, *double networkRate*)
Log information about the current maximum species and network rates.

`maxEdgeSpeciesRateRatios`
numpy.ndarray

Type maxEdgeSpeciesRateRatios

`maxNetworkLeakRateRatios`
numpy.ndarray

Type maxNetworkLeakRateRatios

`nSims`
'int'

Type nSims

`neq`
'int'

Type neq

`networkIndices`
numpy.ndarray

Type networkIndices

`networkLeakCoefficients`
numpy.ndarray

Type networkLeakCoefficients

`networkLeakRates`
numpy.ndarray

Type networkLeakRates

`numCoreReactions`
'int'

Type numCoreReactions

`numCoreSpecies`
'int'

Type numCoreSpecies

`numEdgeReactions`
'int'

Type numEdgeReactions

`numEdgeSpecies`
'int'

Type numEdgeSpecies

`numPdepNetworks`
'int'

Type numPdepNetworks

`productIndices`
numpy.ndarray

Type productIndices

`prunableNetworkIndices`
numpy.ndarray

Type prunableNetworkIndices

prunableNetworks
list

Type prunableNetworks

prunableSpecies
list

Type prunableSpecies

prunableSpeciesIndices
numpy.ndarray

Type prunableSpeciesIndices

reactantIndices
numpy.ndarray

Type reactantIndices

reactionIndex
dict

Type reactionIndex

reset_max_edge_species_rate_ratios(*self*)
This function sets maxEdgeSpeciesRateRatios back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*self*, *double t*, *ndarray y*, *ndarray dydt*, *ndarray senpar=numpy.zeros(1*, *numpy.float64)*)
Return the residual function for the governing DAE system for the liquid reaction system.

rtol_array
numpy.ndarray

Type rtol_array

sensConditions
dict

Type sensConditions

sensitiveSpecies
list

Type sensitiveSpecies

sensitivityCoefficients
numpy.ndarray

Type sensitivityCoefficients

sensitivityThreshold
'double'

Type sensitivityThreshold

set_initial_conditions(*self*)
Sets the initial conditions of the rate equations that represent the current reactor model.

The volume is set to the value in m3 required to contain one mole total of core species at start.

The coreSpeciesConcentrations array is set to the values stored in the initial concentrations dictionary.

The initial number of moles of a species j is computed and stored in the y0 instance attribute.

`set_initial_derivative`(*self*)

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

`set_initial_reaction_thresholds`(*self*)

`set_prunable_indices`(*self*, *edgeSpecies*, *pdepNetworks*)

`simulate`(*self*, *list coreSpecies*, *list coreReactions*, *list edgeSpecies*, *list edgeReactions*, *list surfaceSpecies*, *list surfaceReactions*, *list pdepNetworks=None*, *bool prune=False*, *bool sensitivity=False*, *list sensWorksheet=None*, *modelSettings=None*, *simulatorSettings=None*, *dict conditions=None*)

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.

`snapshots`

list

Type snapshots

`speciesIndex`

dict

Type speciesIndex

`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, $\mathbf{y}$, and $\frac{d\mathbf{y}}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

`surfaceReactionIndices`

numpy.ndarray

Type surfaceReactionIndices

`surfaceSpeciesIndices`

numpy.ndarray

Type surfaceSpeciesIndices

`t0`

‘float’

Type t0

`termination`

list

Type termination

`trimolecular`

‘bool’

Type trimolecular

`trimolecularThreshold`

numpy.ndarray

Type trimolecularThreshold

`unimolecularThreshold`

numpy.ndarray

Type unimolecularThreshold

validLayeringIndices
numpy.ndarray

Type validLayeringIndices

viscosity
'double'

Type viscosity

y0
numpy.ndarray

Type y0

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
`Species`	A chemical species

1.13.2 Transition state

Class	Description
`TransitionState`	A transition state

rmgpy.species.Species

class `rmgpy.species.`**Species**(*index=-1*, *label=''*, *thermo=None*, *conformer=None*, *molecule=None*, *transportData=None*, *molecularWeight=None*, *energyTransferModel=None*, *reactive=True*, *props=None*, *aug_inchi=None*, *symmetryNumber=-1*, *creationIteration=0*, *explicitlyAllowed=False*)
A chemical species, representing a local minimum on a potential energy surface. The attributes are:

Attribute	Description
index	A unique nonnegative integer index
label	A descriptive string label
thermo	The heat capacity model for the species
conformer	The molecular conformer for the species
molecule	A list of the `Molecule` objects describing the molecular structure
transportData	A set of transport collision parameters
molecularWeight	The molecular weight of the species
energyTransferModel	The collisional energy transfer model to use
reactive	**`True` if the species participates in reaction families, `False` if not** Reaction libraries and seed mechanisms that include the species are always considered regardless of this variable
props	A generic ‘properties’ dictionary to store user-defined flags
aug_inchi	Unique augmented inchi
isSolvent	Boolean describing whether this species is the solvent
creationIteration	Iteration which the species is created within the reaction mechanism generation algorithm

`aug_inchi`
str

Type aug_inchi

`calculateCp0`(*self*) → double
Return the value of the heat capacity at zero temperature in J/mol*K.

`calculateCpInf`(*self*) → double
Return the value of the heat capacity at infinite temperature in J/mol*K.

`conformer`
rmgpy.statmech.conformer.Conformer

Type conformer

`copy`(*self*, *bool deep=False*) → Species
Create a copy of the current species. If the kw argument ‘deep’ is True, then a deep copy will be made of the Molecule objects in self.molecule.

For other complex attributes, a deep copy will always be made.

`creationIteration`
‘int’

Type creationIteration

`energyTransferModel`
object

Type energyTransferModel

`explicitlyAllowed`
‘bool’

Type explicitlyAllowed

`fromAdjacencyList`(*self*, *adjlist*)
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`(*self*, *smiles*)
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`(*self*)
Generate the collisional energy transfer model parameters for the species. This "algorithm" is *very* much in need of improvement.

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

`generateTransportData`(*self*)
Generate the transportData parameters for the species.

`generate_aug_inchi`(*self*)

`generate_resonance_structures`(*self*, *bool keep_isomorphic=True*, *bool filter_structures=True*)
Generate all of the resonance structures 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 structures have already been generated.

`getAugmentedInChI`(*self*)

`getDensityOfStates`(*self*, *ndarray Elist*) → ndarray
Return the density of states $\rho(E)\ dE$ at the specified energies *Elist* in J/mol above the ground state.

`getEnthalpy`(*self*, *double T*) → double
Return the enthalpy in J/mol for the species at the specified temperature *T* in K.

`getEntropy`(*self*, *double T*) → double
Return the entropy in J/mol*K for the species at the specified temperature *T* in K.

`getFreeEnergy`(*self*, *double T*) → double
Return the Gibbs free energy in J/mol for the species at the specified temperature *T* in K.

`getHeatCapacity`(*self*, *double T*) → double
Return the heat capacity in J/mol*K for the species at the specified temperature *T* in K.

`getPartitionFunction`(*self*, *double T*) → double
Return the partition function for the species at the specified temperature *T* in K.

`getResonanceHybrid`(*self*)
Returns a molecule object with bond orders that are the average of all the resonance structures.

`getSumOfStates`(*self*, *ndarray Elist*) → ndarray
Return the sum of states $N(E)$ at the specified energies *Elist* in J/mol.

`getSymmetryNumber`(*self*)
Get the symmetry number for the species, which is the highest symmetry number amongst its resonance isomers and the resonance hybrid. This function is currently used for website purposes and testing only as it requires additional calculateSymmetryNumber calls.

`getThermoData`(*self*, *solventName="*)
Returns a *thermoData* object of the current Species object.

If the thermo object already exists, it is either of the (Wilhoit, ThermoData) type, or it is a Future.

If the type of the thermo attribute is Wilhoit, or ThermoData, then it is converted into a NASA format.

If it is a Future, then a blocking call is made to retrieve the NASA object. If the thermo object did not exist yet, the thermo object is generated.

`getTransportData`(*self*)
Returns the transport data associated with this species, and calculates it if it is not yet available.

`hasStatMech`(*self*) → bool
Return `True` if the species has statistical mechanical parameters, or `False` otherwise.

`hasThermo`(*self*) → bool
Return `True` if the species has thermodynamic parameters, or `False` otherwise.

`has_reactive_molecule`(*self*) → bool
True if the species has at least one reactive molecule, *False* otherwise

`index`
'int'

Type index

`isIdentical`(*self*, *other*) → bool
Return `True` if at least one molecule of the species is identical to *other*, which can be either a `Molecule` object or a `Species` object.

`isIsomorphic`(*self*, *other*, *bool generate_res=False*) → bool
Return `True` if the species is isomorphic to *other*, which can be either a `Molecule` object or a `Species` object. If generate_res is `True` and other is a `Species` object, the resonance structures of other will be generated and isomorphically compared against self. This is useful for situations where a "non-representative" resonance structure of self is generated, and it should be identified as the same Species, and be assigned a reactive=False flag.

`isSolvent`
'bool'

Type isSolvent

`is_structure_in_list`(*self*, *list species_list*) → bool
Return `True` if at least one Molecule in self is isomorphic with at least one other Molecule in at least one Species in species list.

`label`
str

Type label

`molecularWeight`
value_si is in kg/molecule not kg/mol)

Type The molecular weight of the species. (Note

`molecule`
list

Type molecule

`props`
dict

Type props

reactive
'bool'

Type reactive

setE0WithThermo(*self*)
Helper method that sets species' E0 using the species' thermo data

symmetryNumber
'float'

Type symmetryNumber

thermo
object

Type thermo

toAdjacencyList(*self*)
Return a string containing each of the molecules' adjacency lists.

toCantera(*self*, *useChemkinIdentifier=False*)
Converts the RMG Species object to a Cantera Species object with the appropriate thermo data.

If useChemkinIdentifier is set to False, the species label is used instead. Be sure that species' labels are unique when setting it False.

toChemkin(*self*)
Return the chemkin-formatted string for this species.

transportData
object

Type transportData

rmgpy.species.TransitionState

class rmgpy.species.**TransitionState**(*label=''*, *conformer=None*, *frequency=None*, *tunneling=None*, *degeneracy=1*)
A chemical transition state, representing a first-order saddle point on a potential energy surface. The attributes are:

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

calculateTunnelingFactor(*self*, *double T*) → double
Calculate and return the value of the canonical tunneling correction factor for the reaction at the given temperature *T* in K.

calculateTunnelingFunction(*self*, *ndarray Elist*) → ndarray
Calculate and return the value of the microcanonical tunneling correction for the reaction at the given energies *Elist* in J/mol.

conformer
rmgpy.statmech.conformer.Conformer

Type conformer

degeneracy
'int'

Type degeneracy

frequency
TransitionState.getFrequency(self)

getDensityOfStates(*self*, *ndarray Elist*) → ndarray
Return the density of states $\rho(E)\ dE$ at the specified energies *Elist* in J/mol above the ground state.

getEnthalpy(*self*, *double T*) → double
Return the enthalpy in J/mol for the transition state at the specified temperature *T* in K.

getEntropy(*self*, *double T*) → double
Return the entropy in J/mol*K for the transition state at the specified temperature *T* in K.

getFreeEnergy(*self*, *double T*) → double
Return the Gibbs free energy in J/mol for the transition state at the specified temperature *T* in K.

getFrequency(*self*)

getHeatCapacity(*self*, *double T*) → double
Return the heat capacity in J/mol*K for the transition state at the specified temperature *T* in K.

getPartitionFunction(*self*, *double T*) → double
Return the partition function for the transition state at the specified temperature *T* in K.

getSumOfStates(*self*, *ndarray Elist*) → ndarray
Return the sum of states $N(E)$ at the specified energies *Elist* in J/mol.

label
str

Type label

setFrequency(*self*, *value*)

tunneling
rmgpy.kinetics.model.TunnelingModel

Type tunneling

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
`IdealGasTranslation`	A model of three-dimensional translation of an ideal gas

1.14.2 Rotational degrees of freedom

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

1.14.3 Vibrational degrees of freedom

Class	Description
`HarmonicOscillator`	A model of a set of one-dimensional harmonic oscillators

1.14.4 Torsional degrees of freedom

Class	Description
`HinderedRotor`	A model of a one-dimensional hindered rotation

1.14.5 The Schrodinger equation

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

1.14.6 Convolution

Class	Description
`convolve()`	Return the convolution of two arrays
`convolveBS()`	Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart (BS) direct count algorithm
`convolveBSSR()`	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
`Conformer`	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
mass	The mass of the translating object
quantum	`True` to use the quantum mechanical model, `False` to use the classical model

Translational energies are much smaller than $k_\mathrm{B} T$ 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:

$$V(x,y,z) = \begin{cases} 0 & 0 \le x \le L_x, 0 \le y \le L_y, 0 \le z \le L_z \\ \infty & \text{otherwise} \end{cases}$$

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:

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

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:

$$Q_\mathrm{trans}(T) = \sum_{n_x=1}^\infty \sum_{n_y=1}^\infty \sum_{n_z=1}^\infty \exp \left(-\frac{E_{n_x,n_y,n_z}}{k_\mathrm{B} T} \right)$$

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:

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

For a constant-pressure problem we can use the ideal gas law to replace V with $k_\mathrm{B} T/P$. 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
'bool'

Type quantum

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
inertia	The moment of inertia of the rotor
rotationalConstant	The rotational constant of the rotor
symmetry	The symmetry number of the rotor
quantum	`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

$$\begin{aligned} E_J &= B J (J+1) \qquad J = 0, 1, 2, \ldots \\ g_J &= 2J + 1 \end{aligned}$$

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:

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

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:

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

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
‘bool’

Type quantum

rotationalConstant
The rotational constant of the rotor.

symmetry
'int'

Type symmetry

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
inertia	The moments of inertia of the rotor
rotationalConstant	The rotational constants of the rotor
symmetry	The symmetry number of the rotor
quantum	`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:

$$Q_\mathrm{rot}^\mathrm{cl}(T) = \frac{\pi^{1/2}}{\sigma} \left(\frac{8 k_\mathrm{B} T}{h^2} \right)^{3/2} \sqrt{I_\mathrm{A} I_\mathrm{B} I_\mathrm{C}}$$

`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
'bool'

Type quantum

rotationalConstant
The rotational constant of the rotor.

symmetry
'int'

Type symmetry

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
inertia	The moment of inertia of the rotor in amu*angstrom^2
rotationalConstant	The rotational constant of the rotor in cm^-1
symmetry	The symmetry number of the rotor
quantum	`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

$$E_K = BK^2 \qquad K = 0, \pm 1, \pm 2, \ldots$$

where K is the quantum number for the rotor and $B \equiv \hbar^2/2I$ is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition function for the K-rotor:

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

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:

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

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

'bool'

Type quantum

rotationalConstant

The rotational constant of the rotor.

symmetry

'int'

Type symmetry

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
inertia	The moment of inertia of the rotor
rotationalConstant	The rotational constant of the rotor
symmetry	The symmetry number of the rotor
quantum	`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. $I_\mathrm{A} = I_\mathrm{B} = I_\mathrm{C} = I$. The energy levels E_J and corresponding degeneracies g_J of the spherial top rotor are given by

$$\begin{aligned} E_J &= B J (J+1) \qquad J = 0, 1, 2, \ldots \\ g_J &= (2J+1)^2 \end{aligned}$$

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:

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

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:

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

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
‘bool’

Type quantum

rotationalConstant
The rotational constant of the rotor.

symmetry
‘int’

Type symmetry

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
frequencies	The vibrational frequencies of the oscillators
quantum	`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 $k_\mathrm{B}T$, 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:

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

Above we have introduced n as the quantum number. Note that, even in the ground state ($n = 0$), the harmonic oscillator has an energy that is not zero; this energy is called the *zero-point energy*.

The harmonic oscillator partition function is obtained by summing over the above energy levels:

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

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

$$Q_\mathrm{vib}(T) = \frac{e^{-\hbar \omega / 2 k_\mathrm{B} T}}{1 - e^{-\hbar \omega / k_\mathrm{B} T}}$$

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

$$Q_\mathrm{vib}(T) = \frac{1}{1 - e^{-\hbar \omega / k_\mathrm{B} T}}$$

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:

$$Q_\mathrm{vib}^\mathrm{cl}(T) = \frac{k_\mathrm{B} T}{\hbar \omega}$$

`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
'bool'

Type quantum

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
inertia	The moment of inertia of the rotor
rotationalConstant	The rotational constant of the rotor
symmetry	The symmetry number of the rotor
fourier	The $2xN$ array of Fourier series coefficients
barrier	The barrier height of the cosine potential
quantum	`True` to use the quantum mechanical model, `False` to use the classical model
semiclassical	`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

$$V(\phi) = \frac{1}{2}V_0\left(1 - \cos\sigma\phi\right)$$

where V_0 is the barrier height and σ is the symmetry number, or a more general Fourier series of the form

$$V(\phi) = A + \sum_{k=1}^{C}\left(a_k\cos k\phi + b_k\sin k\phi\right)$$

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:

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

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

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

Above we have defined ν as the vibrational frequency of the hindered rotor:

$$\nu \equiv \frac{\sigma}{2 \pi} \sqrt{\frac{V_0}{2 I}}$$

barrier
The barrier height of the cosine potential.

energies
numpy.ndarray

Type energies

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
'double'

Type frequency

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
'bool'

Type quantum

rotationalConstant
The rotational constant of the rotor.

semiclassical
'bool'

Type semiclassical

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
'int'

Type symmetry

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 $C_\mathrm{v}(T)/R$ 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
E0	The ground-state energy (including zero-point energy) of the conformer
modes	A list of the molecular degrees of freedom
spinMultiplicity	The degeneracy of the electronic ground state
opticalIsomers	The number of optical isomers
number	An array of atomic numbers of each atom in the conformer
mass	An array of masses of each atom in the conformer
coordinates	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*, *option=3*`)` → double

Calculate and return the reduced moment of inertia for an internal torsional rotation around the axis defined by the two atoms in *pivots*. The list *top1* contains the atoms that should be considered as part of the rotating top; this list should contain the pivot atom connecting the top to the rest of the molecule. The procedure used is that of Pitzer[1], which is described as $I^{(2,option)}$ by East and Radom[2]. In this procedure, the molecule is divided into two tops: those at either end of the hindered rotor bond. The moment of inertia of each top is evaluated using an axis determined by option. Finally, the reduced moment of inertia is evaluated from the moment of inertia of each top via the formula (I1*I2)/(I1+I2).

Option corresponds to 3 possible ways of calculating the internal reduced moment of inertia as discussed in East and Radom [2]

option = 1	moments of inertia of each rotating group calculated about the axis containing the twisting bond
option = 2	(unimplemented) each moment of inertia of each rotating group is calculated about an axis parallel to the twisting bond and passing through its center of mass
option = 3	moments of inertia of each rotating group calculated about the axis passing through the centers of mass of both groups

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

`getMomentOfInertiaTensor`(*self*) → ndarray

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

`getNumberDegreesOfFreedom`(*self*)

Return the number of degrees of freedom in a species object, which should be 3N, and raises an exception if it is not.

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

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

`getPrincipalMomentsOfInertia`(*self*)

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

`getSumOfStates`(*self*, *ndarray Elist*) → ndarray

Return the sum of states $N(E)$ at the specified energies *Elist* in kJ/mol above the ground state.

`getSymmetricTopRotors`(*self*)

Return objects representing the external J-rotor and K-rotor under the symmetric top approximation. For nonlinear molecules, the J-rotor is a 2D rigid rotor with a rotational constant B determined as the geometric mean of the two most similar rotational constants. The K-rotor is a 1D rigid rotor with a rotational constant $A - B$ determined by the difference between the remaining molecular rotational constant and the J-rotor rotational constant.

`getTotalMass`(*self*, *atoms=None*) → double

Calculate and return the total mass of the atoms in the conformer in kg. If a list *atoms* of atoms is specified, only those atoms will be used to calculate the center of mass. Otherwise, all atoms will be used.

`mass`

An array of masses of each atom in the conformer.

---

[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).

`modes`

list

Type modes

`number`

An array of atomic numbers of each atom in the conformer.

`opticalIsomers`

'int'

Type opticalIsomers

`spinMultiplicity`

'int'

Type spinMultiplicity

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
`ThermoData`	A heat capacity model based on a set of discrete heat capacity points
`Wilhoit`	A heat capacity model based on the Wilhoit polynomial
`NASA`	A heat capacity model based on a set of NASA polynomials
`NASAPolynomial`	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*, *label=”*, *comment=”*)

A heat capacity model based on a set of discrete heat capacity data points. The attributes are:

Attribute	Description
Tdata	An array of temperatures at which the heat capacity is known
Cpdata	An array of heat capacities at the given temperatures
H298	The standard enthalpy of formation at 298 K
S298	The standard entropy at 298 K
Tmin	The minimum temperature at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature at which the model is valid, or zero if unknown or undefined
E0	The energy at zero Kelvin (including zero point energy)
comment	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`
str

Type comment

`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.

`label`
str

Type label

`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 $C_\mathrm{p}(T)$ at *Tint*
  - 1: constrain $C_\mathrm{p}(T)$ to be continous at *Tint*
  - 2: constrain $C_\mathrm{p}(T)$ and $\frac{dC_\mathrm{p}}{dT}$ to be continuous at *Tint*
  - 3: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, and $\frac{d^2C_\mathrm{p}}{dT^2}$ to be continuous at *Tint*
  - 4: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, $\frac{d^2C_\mathrm{p}}{dT^2}$, and $\frac{d^3C_\mathrm{p}}{dT^3}$ to be continuous at *Tint*
  - 5: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, $\frac{d^2C_\mathrm{p}}{dT^2}$, $\frac{d^3C_\mathrm{p}}{dT^3}$, and $\frac{d^4C_\mathrm{p}}{dT^4}$ 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*, *B=None*) → Wilhoit

Convert the Benson model to a Wilhoit model. For the conversion to succeed, you must have set the *Cp0* and *CpInf* attributes of the Benson model.

B: the characteristic temperature in Kelvin.

rmgpy.thermo.Wilhoit

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*, *label=''*, *comment=''*)

A heat capacity model based on the Wilhoit equation. The attributes are:

Attribute	Description
a0	The zeroth-order Wilhoit polynomial coefficient
a1	The first-order Wilhoit polynomial coefficient
a2	The second-order Wilhoit polynomial coefficient
a3	The third-order Wilhoit polynomial coefficient
H0	The integration constant for enthalpy (not H at T=0)
S0	The integration constant for entropy (not S at T=0)
E0	The energy at zero Kelvin (including zero point energy)
B	The Wilhoit scaled temperature coefficient in K
Tmin	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
comment	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:

$$C_\mathrm{p}(T) = C_\mathrm{p}(0) + \left[C_\mathrm{p}(\infty) - C_\mathrm{p}(0) \right] y^2 \left[1 + (y - 1) \sum_{i=0}^3 a_i y^i \right]$$

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

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

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

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

The low-temperature limit $C_\mathrm{p}(0)$ is $3.5R$ for linear molecules and $4R$ for nonlinear molecules. The high-temperature limit $C_\mathrm{p}(\infty)$ is taken to be $\left[3 N_\mathrm{atoms} - 1.5 \right] R$ for linear molecules and $\left[3 N_\mathrm{atoms} - (2 + 0.5 N_\mathrm{rotors}) \right] R$ 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 enthlapy 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

'double'

Type a0

a1

'double'

Type a1

a2

'double'

Type a2

a3
'double'

Type a3

comment
str

Type comment

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.

label
str

Type label

`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 $C_\mathrm{p}(T)$ at *Tint*
  - 1: constrain $C_\mathrm{p}(T)$ to be continous at *Tint*
  - 2: constrain $C_\mathrm{p}(T)$ and $\frac{dC_\mathrm{p}}{dT}$ to be continuous at *Tint*
  - 3: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, and $\frac{d^2C_\mathrm{p}}{dT^2}$ to be continuous at *Tint*
  - 4: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, $\frac{d^2C_\mathrm{p}}{dT^2}$, and $\frac{d^3C_\mathrm{p}}{dT^3}$ to be continuous at *Tint*
  - 5: constrain $C_\mathrm{p}(T)$, $\frac{dC_\mathrm{p}}{dT}$, $\frac{d^2C_\mathrm{p}}{dT^2}$, $\frac{d^3C_\mathrm{p}}{dT^3}$, and $\frac{d^4C_\mathrm{p}}{dT^4}$ 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*, *Cp0=None*, *CpInf=None*, *label="*, *comment="*)

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

Attribute	Description
polynomials	The list of NASA polynomials to use in this model
Tmin	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
E0	The energy at zero Kelvin (including zero point energy)
comment	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

$$\frac{C_\mathrm{p}(T)}{R} = a_{-2}T^{-2} + a_{-1}T^{-1} + a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4$$

$$\frac{H(T)}{RT} = -a_{-2}T^{-2} + a_{-1}T^{-1}\ln T + a_0 + \frac{1}{2}a_1 T + \frac{1}{3}a_2 T^2 + \frac{1}{4}a_3 T^3 + \frac{1}{5}a_4 T^4 + \frac{a_5}{T}$$

$$\frac{S(T)}{R} = -\frac{1}{2}a_{-2}T^{-2} - a_{-1}T^{-1} + a_0 \ln T + a_1 T + \frac{1}{2}a_2 T^2 + \frac{1}{3}a_3 T^3 + \frac{1}{4}a_4 T^4 + a_6$$

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.

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, 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.

changeBaseEntropy(*self*, *double deltaS*) → NASA
Add deltaS in J/molK to the base entropy of formation S298 and return the modified NASA object

comment
str

Type comment

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 $H(T)$ in J/mol at the specified temperature *T* in K.

getEntropy(*self*, *double T*) → double
Return the entropy $S(T)$ in J/mol*K at the specified temperature *T* in K.

getFreeEnergy(*self*, *double T*) → double
Return the Gibbs free energy $G(T)$ in J/mol at the specified temperature *T* in K.

getHeatCapacity(*self*, *double T*) → double
Return the constant-pressure heat capacity $C_\mathrm{p}(T)$ 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.

`label`
str

Type label

`poly1`
rmgpy.thermo.nasa.NASAPolynomial

Type poly1

`poly2`
rmgpy.thermo.nasa.NASAPolynomial

Type poly2

`poly3`
rmgpy.thermo.nasa.NASAPolynomial

Type poly3

`polynomials`
The set of one, two, or three NASA polynomials.

`selectPolynomial`(*self*, *double T*) → NASAPolynomial

`toCantera`(*self*)
Return the cantera equivalent NasaPoly2 object from this NASA object.

`toThermoData`(*self*) → ThermoData
Convert the Wilhoit model to a `ThermoData` object.

`toWilhoit`(*self*) → Wilhoit
Convert a `MultiNASA` object *multiNASA* to a `Wilhoit` object. You must specify the linearity of the molecule *linear*, the number of vibrational modes *Nfreq*, and the number of hindered rotor modes *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,-2.98434398E+04,-1.97109989E+01], Tmin=(200,'K'), Tmax=(1000,'K')),
        NASAPolynomial(coeffs=[2.09430708E+01,4.41691018E-02,-1.53261633E-05,2.30544803E-09,-1.
↪29765727E-13,-3.55755088E+04,-8.10637726E+01], Tmin=(1000,'K'), Tmax=(6000,'K')),
    ],
    Tmin = (200,'K'),
    Tmax = (6000,'K'),
),
```

(continues on next page)

(continued from previous page)

```
reference = Reference(authors=["check on burcat"], title='burcat', year="1999", url="http://www.
↪me.berkeley.edu/gri-mech/version30/text30.html"),
referenceType = "review",
shortDesc = u"""""",
longDesc =
    u"""

    """,
    )
```

rmgpy.thermo.NASAPolynomial

class `rmgpy.thermo.`**`NASAPolynomial`**(*coeffs=None*, *Tmin=None*, *Tmax=None*, *E0=None*, *label=”*, *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
coeffs	The seven or nine NASA polynomial coefficients
Tmin	The minimum temperature in K at which the model is valid, or zero if unknown or undefined
Tmax	The maximum temperature in K at which the model is valid, or zero if unknown or undefined
E0	The energy at zero Kelvin (including zero point energy)
comment	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

$$\frac{C_\mathrm{p}(T)}{R} = a_{-2} T^{-2} + a_{-1} T^{-1} + a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4$$

$$\frac{H(T)}{RT} = - a_{-2} T^{-2} + a_{-1} T^{-1} \ln T + a_0 + \frac{1}{2} a_1 T + \frac{1}{3} a_2 T^2 + \frac{1}{4} a_3 T^3 + \frac{1}{5} a_4 T^4 + \frac{a_5}{T}$$

$$\frac{S(T)}{R} = -\frac{1}{2} a_{-2} T^{-2} - a_{-1} T^{-1} + a_0 \ln T + a_1 T + \frac{1}{2} a_2 T^2 + \frac{1}{3} a_3 T^3 + \frac{1}{4} a_4 T^4 + a_6$$

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.

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, 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`
'double'

Type c0

`c1`
'double'

Type c1

`c2`
'double'

Type c2

`c3`
'double'

Type c3

`c4`
'double'

Type c4

`c5`
'double'

Type c5

`c6`
'double'

Type c6

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

`changeBaseEntropy`(*self*, *double deltaS*)
Add deltaS in J/molK to the base entropy of formation S298.

`cm1`
'double'

Type cm1

`cm2`
'double'

Type cm2

`coeffs`
The set of seven or nine NASA polynomial coefficients.

`comment`
str

Type comment

`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.

label
str

Type label

1.16 RMG Exceptions (`rmgpy.exceptions`)

This module contains classes which extend Exception for usage in the RMG module

exception `rmgpy.exceptions.`**ActionError**
An exception class for errors that occur while applying reaction recipe actions. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**AtomTypeError**
An exception to be raised when an error occurs while working with atom types. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**ChemicallySignificantEigenvaluesError**
An exception raised when the chemically significant eigenvalue method is unsuccessful for any reason. Pass a string describing the cause of the exceptional behavior.

exception `rmgpy.exceptions.`**ChemkinError**
An exception class for exceptional behavior involving Chemkin files. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**CollisionError**
An exception class for when RMG is unable to calculate collision efficiencies for the single exponential down pressure dependent solver. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**CoreError**
An exception raised if there is a problem within the model core

exception `rmgpy.exceptions.`**`DatabaseError`**

A exception that occurs when working with an RMG database. Pass a string giving specifics about the exceptional behavior.

exception `rmgpy.exceptions.`**`DependencyError`**

An exception that occurs when an error is encountered with a dependency. Pass a string describing the circumstances that caused the exception.

exception `rmgpy.exceptions.`**`ElementError`**

An exception class for errors that occur while working with elements. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**`ForbiddenStructureException`**

An exception passed when RMG encounters a forbidden structure. These are usually caught and the reaction that created it is ignored.

exception `rmgpy.exceptions.`**`ILPSolutionError`**

An exception to be raised when solving an integer linear programming problem if a solution could not be found or the solution is not valid. Can pass a string to indicate the reason that the solution is invalid.

exception `rmgpy.exceptions.`**`ImplicitBenzeneError`**

An exception class when encountering a group with too many implicit benzene atoms. These groups are hard to create sample molecules and hard for users to interpret. Pass a string describing the limitation.

exception `rmgpy.exceptions.`**`InchiException`**

An exception used when encountering a non-valid Inchi expression are encountered. Pass a string describing the error.

exception `rmgpy.exceptions.`**`InputError`**

An exception raised when parsing an input file for any module in RMG: mechanism generation, Arkane, conformer creation, etc. Pass a string describing the error.

exception `rmgpy.exceptions.`**`InvalidActionError`**

An exception to be raised when an invalid action is encountered in a reaction recipe.

exception `rmgpy.exceptions.`**`InvalidAdjacencyListError`**

An exception used to indicate that an RMG-style adjacency list is invalid. Pass a string describing the reason the adjacency list is invalid

exception `rmgpy.exceptions.`**`InvalidMicrocanonicalRateError`**(*message*, *k_ratio=1.0*, *Keq_ratio=1.0*)

Used in pressure dependence when the k(E) calculation does not give the correct kf(T) or Kc(T)

`badness`()

How bad is the error?

Returns the max of the absolute logarithmic errors of kf and Kc

exception `rmgpy.exceptions.`**`KekulizationError`**

An exception to be raised when encountering an error while kekulizing an aromatic molecule. Can pass a string to indicate the reason for failure.

exception `rmgpy.exceptions.`**`KineticsError`**

An exception class for problems with kinetics. This can be used when finding degeneracy in reaction generation, modifying KineticsData objects, or finding the kinetics of reactions. Unable Pass a string describing the problem.

exception `rmgpy.exceptions.`**`ModifiedStrongCollisionError`**

An exception raised when the modified strong collision method is unsuccessful for any reason. Pass a string describing the cause of the exceptional behavior.

exception `rmgpy.exceptions.`**`NegativeBarrierException`**

This Exception occurs when the energy barrier for a hindered Rotor is negative. This can occur if the scan or fourier fit is poor.

exception `rmgpy.exceptions.`**`NetworkError`**

Raised when an error occurs while working with a pressure-dependent reaction network

exception `rmgpy.exceptions.`**`OutputError`**

This exception is raised whenever an error occurs while saving output information. Pass a string describing the circumstances of the exceptional behavior.

exception `rmgpy.exceptions.`**`PressureDependenceError`**

An exception class to use when an error involving pressure dependence is encountered. Pass a string describing the circumstances of the exceptional behavior.

exception `rmgpy.exceptions.`**`QuantityError`**

An exception to be raised when an error occurs while working with physical quantities in RMG. Pass a string describing the circumstances of the exceptional behavior.

exception `rmgpy.exceptions.`**`ReactionError`**

An exception class for exceptional behavior involving `Reaction` objects. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**`ReactionPairsError`**

An exception to be raised when an error occurs while working with reaction pairs.

exception `rmgpy.exceptions.`**`ReservoirStateError`**

An exception raised when the reservoir state method is unsuccessful for any reason. Pass a string describing the cause of the exceptional behavior.

exception `rmgpy.exceptions.`**`ResonanceError`**

An exception class for when RMG is unable to generate resonance structures.

exception `rmgpy.exceptions.`**`SettingsError`**

An exception raised when dealing with settings.

exception `rmgpy.exceptions.`**`SpeciesError`**

An exception class for exceptional behavior that occurs while working with chemical species. Pass a string describing the circumstances that caused the exceptional behavior.

exception `rmgpy.exceptions.`**`StatmechError`**

An exception used when an error occurs in estimating Statmech.

exception `rmgpy.exceptions.`**`StatmechFitError`**

An exception used when attempting to fit molecular degrees of freedom to heat capacity data. Pass a string describing the circumstances of the exceptional behavior.

exception `rmgpy.exceptions.`**`UndeterminableKineticsError`**(*reaction*, *message=''*)

An exception raised when attempts to estimate appropriate kinetic parameters for a chemical reaction are unsuccessful.

exception `rmgpy.exceptions.`**`UnexpectedChargeError`**(*graph*)

An exception class when encountering a group/molecule with unexpected charge Curently in RMG, we never expect to see -2/+2 or greater magnitude charge, we only except +1/-1 charges on nitrogen, oxygen, sulfur or specifically carbon monoxide/monosulfide.

Attributes: *graph* is the molecule or group object with the unexpected charge

exception `rmgpy.exceptions.`**`VF2Error`**

An exception raised if an error occurs within the VF2 graph isomorphism algorithm. Pass a string describing the error.

BIBLIOGRAPHY

[1932Wigner] E.Wigner. *Phys. Rev.* **40**, p. 749-759 (1932). doi:10.1103/PhysRev.40.749

[1959Bell] R. P. Bell. *Trans. Faraday Soc.* **55**, p. 1-4 (1959). doi: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

PYTHON MODULE INDEX

a

arkane, 3

r

rmgpy.chemkin, 11
rmgpy.constants, 14
rmgpy.data, 14
rmgpy.exceptions, 255
rmgpy.kinetics, 76
rmgpy.molecule, 97
rmgpy.molecule.adjlist, 143
rmgpy.molecule.converter, 142
rmgpy.molecule.kekulize, 137
rmgpy.molecule.pathfinder, 139
rmgpy.molecule.resonance, 134
rmgpy.molecule.translator, 142
rmgpy.pdep, 147
rmgpy.qm, 156
rmgpy.quantity, 174
rmgpy.reaction, 179
rmgpy.rmg, 185
rmgpy.solver, 204
rmgpy.species, 224
rmgpy.statmech, 229
rmgpy.statmech.schrodinger, 241
rmgpy.thermo, 245

INDEX

A

`A` (*rmgpy.kinetics.Arrhenius attribute*), 78
`a0` (*rmgpy.thermo.Wilhoit attribute*), 248
`a1` (*rmgpy.thermo.Wilhoit attribute*), 248
`a2` (*rmgpy.thermo.Wilhoit attribute*), 248
`a3` (*rmgpy.thermo.Wilhoit attribute*), 249
`ActionError`, 255
`activeJRotor` (*rmgpy.pdep.Configuration attribute*), 151
`activeKRotor` (*rmgpy.pdep.Configuration attribute*), 151
`add_allyls` (*in module rmgpy.molecule.pathfinder*), 139
`add_inverse_allyls` (*in module rmgpy.molecule.pathfinder*), 139
`add_unsaturated_bonds` (*in module rmgpy.molecule.pathfinder*), 139
`addAction()` (*rmgpy.data.kinetics.ReactionRecipe method*), 49
`addAtom()` (*rmgpy.molecule.Group method*), 128
`addAtom()` (*rmgpy.molecule.Molecule method*), 115
`addAtomLabelsForReaction()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`addBond()` (*rmgpy.molecule.Group method*), 128
`addBond()` (*rmgpy.molecule.Molecule method*), 115
`addEdge()` (*rmgpy.molecule.graph.Graph method*), 100
`addEdge()` (*rmgpy.molecule.Group method*), 128
`addEdge()` (*rmgpy.molecule.Molecule method*), 115
`addEntry()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`addExplicitLigands()` (*rmgpy.molecule.Group method*), 128
`addImplicitAtomsFromAtomType()` (*rmgpy.molecule.Group method*), 128
`addImplicitBenzene()` (*rmgpy.molecule.Group method*), 128
`addKineticsRulesFromTrainingSet()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`addNewSurfaceObjects()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 186
`addPathReaction()` (*rmgpy.rmg.pdep.PDepNetwork method*), 193
`addReactionLibraryToEdge()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 186
`addReactionLibraryToOutput()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 186
`addReactionsToSurface()` (*rmgpy.solver.LiquidReactor method*), 218
`addReactionsToSurface()` (*rmgpy.solver.ReactionSystem method*), 205
`addReactionsToSurface()` (*rmgpy.solver.SimpleReactor method*), 211
`addReactionToCore()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 186
`addReactionToEdge()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`addReactionToUnimolecularNetworks()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`addReverseAttribute()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`addSeedMechanismToCore()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`addSpeciesToCore()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`addSpeciesToEdge()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`addVertex()` (*rmgpy.molecule.graph.Graph method*), 100
`addVertex()` (*rmgpy.molecule.Group method*), 129
`addVertex()` (*rmgpy.molecule.Molecule method*), 115
`adjustSurface()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`advance()` (*rmgpy.solver.LiquidReactor method*), 218
`advance()` (*rmgpy.solver.ReactionSystem method*), 205

`advance()` (*rmgpy.solver.SimpleReactor method*), 211
`allDouble` (*rmgpy.molecule.AtomType attribute*), 106
`allow_max_rate_violation` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 18
`allow_max_rate_violation` (*rmgpy.data.kinetics.LibraryReaction attribute*), 43
`allow_max_rate_violation` (*rmgpy.data.kinetics.TemplateReaction attribute*), 62
`allow_max_rate_violation` (*rmgpy.reaction.Reaction attribute*), 180
`allow_max_rate_violation` (*rmgpy.rmg.pdep.PDepReaction attribute*), 196
`allow_pdep_route` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 18
`allow_pdep_route` (*rmgpy.data.kinetics.LibraryReaction attribute*), 43
`allow_pdep_route` (*rmgpy.data.kinetics.TemplateReaction attribute*), 62
`allow_pdep_route` (*rmgpy.reaction.Reaction attribute*), 180
`allow_pdep_route` (*rmgpy.rmg.pdep.PDepReaction attribute*), 196
`alpha` (*rmgpy.kinetics.Troe attribute*), 94
`alpha0` (*rmgpy.pdep.SingleExponentialDown attribute*), 149
`analyze_molecule` (*in module rmgpy.molecule.resonance*), 135
`ancestors()` (*rmgpy.data.base.Database method*), 16
`ancestors()` (*rmgpy.data.kinetics.KineticsDepository method*), 26
`ancestors()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`ancestors()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`ancestors()` (*rmgpy.data.kinetics.KineticsLibrary method*), 38
`ancestors()` (*rmgpy.data.kinetics.KineticsRules method*), 40
`ancestors()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`ancestors()` (*rmgpy.data.statmech.StatmechGroups method*), 57
`ancestors()` (*rmgpy.data.statmech.StatmechLibrary method*), 59
`ancestors()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`ancestors()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`ancestors()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`applyAction()` (*rmgpy.molecule.Atom method*), 110
`applyAction()` (*rmgpy.molecule.Bond method*), 113
`applyAction()` (*rmgpy.molecule.GroupAtom method*), 124
`applyAction()` (*rmgpy.molecule.GroupBond method*), 126
`applyChemicallySignificantEigenvaluesMethod()` (*in module rmgpy.pdep.cse*), 156
`applyChemicallySignificantEigenvaluesMethod()` (*rmgpy.pdep.Network method*), 153
`applyChemicallySignificantEigenvaluesMethod()` (*rmgpy.rmg.pdep.PDepNetwork method*), 193
`applyForward()` (*rmgpy.data.kinetics.ReactionRecipe method*), 49
`applyInverseLaplaceTransformMethod()` (*in module rmgpy.pdep*), 150
`applyKineticsToReaction()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
`applyModifiedStrongCollisionMethod()` (*in module rmgpy.pdep.msc*), 155
`applyModifiedStrongCollisionMethod()` (*rmgpy.pdep.Network method*), 153
`applyModifiedStrongCollisionMethod()` (*rmgpy.rmg.pdep.PDepNetwork method*), 193
`applyRecipe()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`applyReservoirStateMethod()` (*in module rmgpy.pdep.rs*), 155
`applyReservoirStateMethod()` (*rmgpy.pdep.Network method*), 153
`applyReservoirStateMethod()` (*rmgpy.rmg.pdep.PDepNetwork method*), 193
`applyReverse()` (*rmgpy.data.kinetics.ReactionRecipe method*), 49
`applyRRKMTheory()` (*in module rmgpy.pdep*), 150
`areSiblings()` (*rmgpy.data.base.Database method*), 16
`areSiblings()` (*rmgpy.data.kinetics.KineticsDepository method*), 26
`areSiblings()` (*rmgpy.data.kinetics.KineticsFamily method*), 29
`areSiblings()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`areSiblings()` (*rmgpy.data.kinetics.KineticsLibrary method*), 38
`areSiblings()` (*rmgpy.data.kinetics.KineticsRules method*), 40
`areSiblings()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`areSiblings()` (*rmgpy.data.statmech.StatmechGroups method*), 57

`areSiblings()` (*rmgpy.data.statmech.StatmechLibrary method*), 59
`areSiblings()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`areSiblings()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`areSiblings()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`Arkane` (*class in arkane*), 7
`arkane` (*module*), 3
`AromaticBond` (*class in rmgpy.molecule.kekulize*), 138
`AromaticRing` (*class in rmgpy.molecule.kekulize*), 138
`ArrayQuantity` (*class in rmgpy.quantity*), 177
`Arrhenius` (*class in rmgpy.kinetics*), 78
`arrhenius` (*rmgpy.kinetics.MultiArrhenius attribute*), 80
`arrhenius` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
`arrhenius` (*rmgpy.kinetics.PDepArrhenius attribute*), 83
`arrheniusHigh` (*rmgpy.kinetics.Lindemann attribute*), 91
`arrheniusHigh` (*rmgpy.kinetics.Troe attribute*), 94
`arrheniusLow` (*rmgpy.kinetics.Lindemann attribute*), 91
`arrheniusLow` (*rmgpy.kinetics.ThirdBody attribute*), 89
`arrheniusLow` (*rmgpy.kinetics.Troe attribute*), 94
`assignAtomIDs()` (*rmgpy.molecule.Molecule method*), 115
`atol_array` (*rmgpy.solver.LiquidReactor attribute*), 218
`atol_array` (*rmgpy.solver.ReactionSystem attribute*), 205
`atol_array` (*rmgpy.solver.SimpleReactor attribute*), 211
`Atom` (*class in rmgpy.molecule*), 110
`atomIDValid()` (*rmgpy.molecule.Molecule method*), 115
`atoms` (*rmgpy.molecule.Group attribute*), 129
`atoms` (*rmgpy.molecule.kekulize.AromaticRing attribute*), 138
`atoms` (*rmgpy.molecule.Molecule attribute*), 115
`AtomType` (*class in rmgpy.molecule*), 105
`atomType` (*rmgpy.molecule.Atom attribute*), 110
`atomType` (*rmgpy.molecule.GroupAtom attribute*), 124
`AtomTypeError`, 255
`aug_inchi` (*rmgpy.rmg.model.Species attribute*), 201
`aug_inchi` (*rmgpy.species.Species attribute*), 225

B

`B` (*rmgpy.thermo.Wilhoit attribute*), 248
`badness()` (*rmgpy.exceptions.InvalidMicrocanonicalRateError method*), 256
`barrier` (*rmgpy.statmech.HinderedRotor attribute*), 240
`benzene` (*rmgpy.molecule.AtomType attribute*), 106
`bimolecularThreshold` (*rmgpy.solver.LiquidReactor attribute*), 218
`bimolecularThreshold` (*rmgpy.solver.ReactionSystem attribute*), 205
`bimolecularThreshold` (*rmgpy.solver.SimpleReactor attribute*), 211
`Bond` (*class in rmgpy.molecule*), 113
`bond` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138
`breakBond` (*rmgpy.molecule.AtomType attribute*), 106

C

`c0` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c1` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c2` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c3` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c4` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c5` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`c6` (*rmgpy.thermo.NASAPolynomial attribute*), 254
`calculate()` (*rmgpy.qm.symmetry.SymmetryJob method*), 161
`calculate_coll_limit()` (*rmgpy.data.kinetics.DepositoryReaction method*), 19
`calculate_coll_limit()` (*rmgpy.data.kinetics.LibraryReaction method*), 44
`calculate_coll_limit()` (*rmgpy.data.kinetics.TemplateReaction method*), 63
`calculate_coll_limit()` (*rmgpy.reaction.Reaction method*), 181
`calculate_coll_limit()` (*rmgpy.rmg.pdep.PDepReaction method*), 196
`calculate_effective_pressure()` (*rmgpy.solver.SimpleReactor method*), 211
`calculateAtomSymmetryNumber()` (*in module rmgpy.molecule.symmetry*), 145
`calculateAxisSymmetryNumber()` (*in module rmgpy.molecule.symmetry*), 146
`calculateBondSymmetryNumber()` (*in module rmgpy.molecule.symmetry*), 145
`calculateChiralityCorrection()` (*rmgpy.qm.gaussian.GaussianMol method*), 162
`calculateChiralityCorrection()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`calculateChiralityCorrection()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 165

calculateChiralityCorrection() (*rmgpy.qm.molecule.QMMolecule method*), 159
calculateChiralityCorrection() (*rmgpy.qm.mopac.MopacMol method*), 168
calculateChiralityCorrection() (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
calculateChiralityCorrection() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
calculateChiralityCorrection() (*rmgpy.qm.mopac.MopacMolPM7 method*), 172
calculateCollisionEfficiency() (*rmgpy.pdep.SingleExponentialDown method*), 149
calculateCollisionFrequency() (*rmgpy.pdep.Configuration method*), 151
calculateCollisionModel() (*rmgpy.pdep.Network method*), 153
calculateCollisionModel() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
calculateCp0() (*rmgpy.molecule.Molecule method*), 115
calculateCp0() (*rmgpy.rmg.model.Species method*), 201
calculateCp0() (*rmgpy.species.Species method*), 225
calculateCpInf() (*rmgpy.molecule.Molecule method*), 115
calculateCpInf() (*rmgpy.rmg.model.Species method*), 201
calculateCpInf() (*rmgpy.species.Species method*), 225
calculateCyclicSymmetryNumber() (*in module rmgpy.molecule.symmetry*), 146
calculateDegeneracy() (*rmgpy.data.kinetics.KineticsFamily method*), 29
calculateDensitiesOfStates() (*rmgpy.pdep.Network method*), 153
calculateDensitiesOfStates() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
calculateDensityOfStates() (*rmgpy.pdep.Configuration method*), 151
calculateEquilibriumRatios() (*rmgpy.pdep.Network method*), 153
calculateEquilibriumRatios() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
calculateMicrocanonicalRateCoefficient() (*in module rmgpy.pdep*), 149
calculateMicrocanonicalRateCoefficient() (*rmgpy.data.kinetics.DepositoryReaction method*), 18
calculateMicrocanonicalRateCoefficient() (*rmgpy.data.kinetics.LibraryReaction method*), 43
calculateMicrocanonicalRateCoefficient() (*rmgpy.data.kinetics.TemplateReaction method*), 62
calculateMicrocanonicalRateCoefficient() (*rmgpy.reaction.Reaction method*), 180
calculateMicrocanonicalRateCoefficient() (*rmgpy.rmg.pdep.PDepReaction method*), 196
calculateMicrocanonicalRates() (*rmgpy.pdep.Network method*), 154
calculateMicrocanonicalRates() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
calculateSymmetryNumber() (*in module rmgpy.molecule.symmetry*), 146
calculateSymmetryNumber() (*rmgpy.molecule.Molecule method*), 116
calculateThermoData() (*rmgpy.qm.gaussian.GaussianMol method*), 162
calculateThermoData() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
calculateThermoData() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 165
calculateThermoData() (*rmgpy.qm.molecule.QMMolecule method*), 159
calculateThermoData() (*rmgpy.qm.mopac.MopacMol method*), 168
calculateThermoData() (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
calculateThermoData() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
calculateThermoData() (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
calculateTSTRateCoefficient() (*rmgpy.data.kinetics.DepositoryReaction method*), 19
calculateTSTRateCoefficient() (*rmgpy.data.kinetics.LibraryReaction method*), 43
calculateTSTRateCoefficient() (*rmgpy.data.kinetics.TemplateReaction method*), 62

calculateTSTRateCoefficient() (*rmgpy.reaction.Reaction method*), 181
calculateTSTRateCoefficient() (*rmgpy.rmg.pdep.PDepReaction method*), 196
calculateTSTRateCoefficients() (*rmgpy.data.kinetics.DepositoryReaction method*), 19
calculateTSTRateCoefficients() (*rmgpy.data.kinetics.LibraryReaction method*), 44
calculateTSTRateCoefficients() (*rmgpy.data.kinetics.TemplateReaction method*), 62
calculateTSTRateCoefficients() (*rmgpy.reaction.Reaction method*), 181
calculateTSTRateCoefficients() (*rmgpy.rmg.pdep.PDepReaction method*), 196
calculateTunnelingFactor() (*rmgpy.kinetics.Eckart method*), 97
calculateTunnelingFactor() (*rmgpy.kinetics.Wigner method*), 95
calculateTunnelingFactor() (*rmgpy.species.TransitionState method*), 228
calculateTunnelingFunction() (*rmgpy.kinetics.Eckart method*), 97
calculateTunnelingFunction() (*rmgpy.kinetics.Wigner method*), 95
calculateTunnelingFunction() (*rmgpy.species.TransitionState method*), 228
canTST() (*rmgpy.data.kinetics.DepositoryReaction method*), 19
canTST() (*rmgpy.data.kinetics.LibraryReaction method*), 44
canTST() (*rmgpy.data.kinetics.TemplateReaction method*), 63
canTST() (*rmgpy.reaction.Reaction method*), 181
canTST() (*rmgpy.rmg.pdep.PDepReaction method*), 197
changeBaseEnthalpy() (*rmgpy.thermo.NASA method*), 251
changeBaseEnthalpy() (*rmgpy.thermo.NASAPolynomial method*), 254
changeBaseEntropy() (*rmgpy.thermo.NASA method*), 251
changeBaseEntropy() (*rmgpy.thermo.NASAPolynomial method*), 254
changeRate() (*rmgpy.kinetics.Arrhenius method*), 79
changeRate() (*rmgpy.kinetics.Chebyshev method*), 87
changeRate() (*rmgpy.kinetics.Lindemann method*), 91
changeRate() (*rmgpy.kinetics.MultiArrhenius method*), 80
changeRate() (*rmgpy.kinetics.MultiPDepArrhenius method*), 85
changeRate() (*rmgpy.kinetics.PDepArrhenius method*), 83
changeRate() (*rmgpy.kinetics.ThirdBody method*), 89
changeRate() (*rmgpy.kinetics.Troe method*), 94
changeT0() (*rmgpy.kinetics.Arrhenius method*), 79
charge (*rmgpy.molecule.Atom attribute*), 110
charge (*rmgpy.molecule.AtomType attribute*), 106
charge (*rmgpy.molecule.GroupAtom attribute*), 124
Chebyshev (*class in rmgpy.kinetics*), 86
check_collision_limit_violation() (*rmgpy.data.kinetics.DepositoryReaction method*), 19
check_collision_limit_violation() (*rmgpy.data.kinetics.LibraryReaction method*), 44
check_collision_limit_violation() (*rmgpy.data.kinetics.TemplateReaction method*), 63
check_collision_limit_violation() (*rmgpy.reaction.Reaction method*), 181
check_collision_limit_violation() (*rmgpy.rmg.pdep.PDepReaction method*), 197
check_model() (*rmgpy.rmg.main.RMG method*), 191
checkAllSet() (*rmgpy.qm.main.QMSettings method*), 157
checkForDuplicates() (*rmgpy.data.kinetics.KineticsLibrary method*), 38
checkForExistingReaction() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
checkForExistingSpecies() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
checkForInChiKeyCollision() (*rmgpy.qm.qmverifier.QMVerifier method*), 161
checkInput() (*rmgpy.rmg.main.RMG method*), 191
checkLibraries() (*rmgpy.rmg.main.RMG method*), 191
checkPaths() (*rmgpy.qm.gaussian.GaussianMol method*), 162
checkPaths() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
checkPaths() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
checkPaths() (*rmgpy.qm.main.QMCalculator method*), 158
checkPaths() (*rmgpy.qm.molecule.QMMolecule

method), 159
checkPaths() (*rmgpy.qm.mopac.MopacMol method*), 168
checkPaths() (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
checkPaths() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
checkPaths() (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
checkReady() (*rmgpy.qm.gaussian.GaussianMol method*), 162
checkReady() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
checkReady() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
checkReady() (*rmgpy.qm.main.QMCalculator method*), 158
checkReady() (*rmgpy.qm.molecule.QMMolecule method*), 159
checkReady() (*rmgpy.qm.mopac.MopacMol method*), 168
checkReady() (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
checkReady() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
checkReady() (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
ChemicallySignificantEigenvaluesError, 255
ChemkinError, 255
chemkinName (*rmgpy.molecule.Element attribute*), 104
classifyBenzeneCarbons() (*rmgpy.molecule.Group method*), 129
cleanup() (*rmgpy.pdep.Configuration method*), 151
cleanup() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
clear() (*rmgpy.rmg.main.RMG method*), 191
clearLabeledAtoms() (*rmgpy.molecule.Group method*), 129
clearLabeledAtoms() (*rmgpy.molecule.Molecule method*), 116
clearSurfaceAdjustments() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 187
cm1 (*rmgpy.thermo.NASAPolynomial attribute*), 254
cm2 (*rmgpy.thermo.NASAPolynomial attribute*), 254
coeffs (*rmgpy.kinetics.Chebyshev attribute*), 87
coeffs (*rmgpy.thermo.NASAPolynomial attribute*), 254
colliderEfficiencies (*rmgpy.solver.SimpleReactor attribute*), 211
CollisionError, 255
comment (*rmgpy.data.kinetics.DepositoryReaction attribute*), 19
comment (*rmgpy.data.kinetics.LibraryReaction attribute*), 44
comment (*rmgpy.data.kinetics.TemplateReaction attribute*), 63
comment (*rmgpy.kinetics.Arrhenius attribute*), 79
comment (*rmgpy.kinetics.Chebyshev attribute*), 87
comment (*rmgpy.kinetics.KineticsData attribute*), 77
comment (*rmgpy.kinetics.Lindemann attribute*), 91
comment (*rmgpy.kinetics.MultiArrhenius attribute*), 80
comment (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
comment (*rmgpy.kinetics.PDepArrhenius attribute*), 83
comment (*rmgpy.kinetics.PDepKineticsData attribute*), 82
comment (*rmgpy.kinetics.ThirdBody attribute*), 89
comment (*rmgpy.kinetics.Troe attribute*), 94
comment (*rmgpy.reaction.Reaction attribute*), 181
comment (*rmgpy.rmg.pdep.PDepReaction attribute*), 197
comment (*rmgpy.thermo.NASA attribute*), 251
comment (*rmgpy.thermo.NASAPolynomial attribute*), 254
comment (*rmgpy.thermo.ThermoData attribute*), 246
comment (*rmgpy.thermo.Wilhoit attribute*), 249
compute_atom_distance (*in module rmgpy.molecule.pathfinder*), 139
compute_network_variables() (*rmgpy.solver.LiquidReactor method*), 218
compute_network_variables() (*rmgpy.solver.ReactionSystem method*), 205
compute_network_variables() (*rmgpy.solver.SimpleReactor method*), 211
computeGroupAdditivityThermo() (*rmgpy.data.thermo.ThermoDatabase method*), 67
computeRateDerivative() (*rmgpy.solver.LiquidReactor method*), 218
computeRateDerivative() (*rmgpy.solver.ReactionSystem method*), 205
computeRateDerivative() (*rmgpy.solver.SimpleReactor method*), 211
Configuration (*class in rmgpy.pdep*), 151
Conformer (*class in rmgpy.statmech*), 243
conformer (*rmgpy.rmg.model.Species attribute*), 201
conformer (*rmgpy.species.Species attribute*), 225
conformer (*rmgpy.species.TransitionState attribute*), 228
connectivity1 (*rmgpy.molecule.Atom attribute*), 110
connectivity1 (*rmgpy.molecule.graph.Vertex attribute*), 99
connectivity1 (*rmgpy.molecule.GroupAtom attribute*), 125
connectivity2 (*rmgpy.molecule.Atom attribute*), 110
connectivity2 (*rmgpy.molecule.graph.Vertex attribute*), 99
connectivity2 (*rmgpy.molecule.GroupAtom attribute*), 125
connectivity3 (*rmgpy.molecule.Atom attribute*), 111

`connectivity3` (*rmgpy.molecule.graph.Vertex attribute*), 99
`connectivity3` (*rmgpy.molecule.GroupAtom attribute*), 125
`connectTheDots()` (*rmgpy.molecule.Molecule method*), 116
`constantVolume` (*rmgpy.solver.LiquidReactor attribute*), 218
`constantVolume` (*rmgpy.solver.SimpleReactor attribute*), 212
`constSPCIndices` (*rmgpy.solver.LiquidReactor attribute*), 218
`constSPCNames` (*rmgpy.solver.LiquidReactor attribute*), 218
`containsLabeledAtom()` (*rmgpy.molecule.Group method*), 129
`containsLabeledAtom()` (*rmgpy.molecule.Molecule method*), 116
`convertDuplicatesToMulti()` (*rmgpy.data.kinetics.KineticsLibrary method*), 38
`convertInitialKeysToSpeciesObjects()` (*rmgpy.solver.LiquidReactor method*), 218
`convertInitialKeysToSpeciesObjects()` (*rmgpy.solver.SimpleReactor method*), 212
`convolve()` (*in module rmgpy.statmech.schrodinger*), 241
`convolveBS()` (*in module rmgpy.statmech.schrodinger*), 241
`convolveBSSR()` (*in module rmgpy.statmech.schrodinger*), 242
`coordinates` (*rmgpy.statmech.Conformer attribute*), 243
`coords` (*rmgpy.molecule.Atom attribute*), 111
`copy()` (*arkane.PressureDependenceJob method*), 9
`copy()` (*rmgpy.data.kinetics.DepositoryReaction method*), 19
`copy()` (*rmgpy.data.kinetics.LibraryReaction method*), 44
`copy()` (*rmgpy.data.kinetics.TemplateReaction method*), 63
`copy()` (*rmgpy.molecule.Atom method*), 111
`copy()` (*rmgpy.molecule.Bond method*), 113
`copy()` (*rmgpy.molecule.graph.Edge method*), 100
`copy()` (*rmgpy.molecule.graph.Graph method*), 101
`copy()` (*rmgpy.molecule.graph.Vertex method*), 99
`copy()` (*rmgpy.molecule.Group method*), 129
`copy()` (*rmgpy.molecule.GroupAtom method*), 125
`copy()` (*rmgpy.molecule.GroupBond method*), 127
`copy()` (*rmgpy.molecule.Molecule method*), 116
`copy()` (*rmgpy.quantity.ArrayQuantity method*), 177
`copy()` (*rmgpy.quantity.ScalarQuantity method*), 176
`copy()` (*rmgpy.reaction.Reaction method*), 181
`copy()` (*rmgpy.rmg.model.Species method*), 201
`copy()` (*rmgpy.rmg.pdep.PDepReaction method*), 197
`copy()` (*rmgpy.species.Species method*), 225
`copy()` (*rmgpy.thermo.Wilhoit method*), 249
`copyAndMap()` (*rmgpy.molecule.graph.Graph method*), 101
`copyAndMap()` (*rmgpy.molecule.Group method*), 129
`copyAndMap()` (*rmgpy.molecule.Molecule method*), 116
`copyData()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`CoreEdgeReactionModel` (*class in rmgpy.rmg.model*), 186
`CoreError`, 255
`coreReactionRates` (*rmgpy.solver.LiquidReactor attribute*), 219
`coreReactionRates` (*rmgpy.solver.ReactionSystem attribute*), 205
`coreReactionRates` (*rmgpy.solver.SimpleReactor attribute*), 212
`coreSpeciesConcentrations` (*rmgpy.solver.LiquidReactor attribute*), 219
`coreSpeciesConcentrations` (*rmgpy.solver.ReactionSystem attribute*), 206
`coreSpeciesConcentrations` (*rmgpy.solver.SimpleReactor attribute*), 212
`coreSpeciesConsumptionRates` (*rmgpy.solver.LiquidReactor attribute*), 219
`coreSpeciesConsumptionRates` (*rmgpy.solver.ReactionSystem attribute*), 206
`coreSpeciesConsumptionRates` (*rmgpy.solver.SimpleReactor attribute*), 212
`coreSpeciesProductionRates` (*rmgpy.solver.LiquidReactor attribute*), 219
`coreSpeciesProductionRates` (*rmgpy.solver.ReactionSystem attribute*), 206
`coreSpeciesProductionRates` (*rmgpy.solver.SimpleReactor attribute*), 212
`coreSpeciesRates` (*rmgpy.solver.LiquidReactor attribute*), 219
`coreSpeciesRates` (*rmgpy.solver.ReactionSystem attribute*), 206
`coreSpeciesRates` (*rmgpy.solver.SimpleReactor attribute*), 212
`countBonds()` (*rmgpy.molecule.GroupAtom method*), 125
`countInternalRotors()` (*rmgpy.molecule.Molecule method*), 116
`covRadius` (*rmgpy.molecule.Element attribute*), 104
`Cp0` (*rmgpy.thermo.NASA attribute*), 251
`Cp0` (*rmgpy.thermo.NASAPolynomial attribute*), 253
`Cp0` (*rmgpy.thermo.ThermoData attribute*), 245
`Cp0` (*rmgpy.thermo.Wilhoit attribute*), 248

`Cpdata` (*rmgpy.thermo.ThermoData attribute*), 245
`CpInf` (*rmgpy.thermo.NASA attribute*), 251
`CpInf` (*rmgpy.thermo.NASAPolynomial attribute*), 253
`CpInf` (*rmgpy.thermo.ThermoData attribute*), 245
`CpInf` (*rmgpy.thermo.Wilhoit attribute*), 248
`createAndConnectAtom()` (*rmgpy.molecule.Group method*), 129
`createGeometry()` (*rmgpy.qm.gaussian.GaussianMol method*), 162
`createGeometry()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`createGeometry()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`createGeometry()` (*rmgpy.qm.molecule.QMMolecule method*), 159
`createGeometry()` (*rmgpy.qm.mopac.MopacMol method*), 168
`createGeometry()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
`createGeometry()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
`createGeometry()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`creationIteration` (*rmgpy.rmg.model.Species attribute*), 202
`creationIteration` (*rmgpy.species.Species attribute*), 225

D

`Database` (*class in rmgpy.data.base*), 16
`DatabaseError`, 255
`debugRDKitMol` (*in module rmgpy.molecule.converter*), 142
`decrementBond` (*rmgpy.molecule.AtomType attribute*), 106
`decrementLonePair` (*rmgpy.molecule.AtomType attribute*), 106
`decrementLonePairs()` (*rmgpy.molecule.Atom method*), 111
`decrementOrder()` (*rmgpy.molecule.Bond method*), 113
`decrementRadical` (*rmgpy.molecule.AtomType attribute*), 106
`decrementRadical()` (*rmgpy.molecule.Atom method*), 111
`degeneracy` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 19
`degeneracy` (*rmgpy.data.kinetics.LibraryReaction attribute*), 44
`degeneracy` (*rmgpy.data.kinetics.TemplateReaction attribute*), 63
`degeneracy` (*rmgpy.reaction.Reaction attribute*), 181
`degeneracy` (*rmgpy.rmg.pdep.PDepReaction attribute*), 197
`degeneracy` (*rmgpy.species.TransitionState attribute*), 229
`degreeP` (*rmgpy.kinetics.Chebyshev attribute*), 87
`degreeT` (*rmgpy.kinetics.Chebyshev attribute*), 87
`deleteHydrogens()` (*rmgpy.molecule.Molecule method*), 116
`densStates` (*rmgpy.pdep.Configuration attribute*), 151
`DependencyError`, 256
`DepositoryReaction` (*class in rmgpy.data.kinetics*), 18
`descendants()` (*rmgpy.data.base.Database method*), 16
`descendants()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`descendants()` (*rmgpy.data.kinetics.KineticsFamily method*), 30
`descendants()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`descendants()` (*rmgpy.data.kinetics.KineticsLibrary method*), 38
`descendants()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`descendants()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`descendants()` (*rmgpy.data.statmech.StatmechGroups method*), 57
`descendants()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`descendants()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`descendants()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`descendants()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`descendTree()` (*rmgpy.data.base.Database method*), 16
`descendTree()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`descendTree()` (*rmgpy.data.kinetics.KineticsFamily method*), 30
`descendTree()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`descendTree()` (*rmgpy.data.kinetics.KineticsLibrary method*), 38
`descendTree()` (*rmgpy.data.kinetics.KineticsRules method*), 40
`descendTree()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`descendTree()` (*rmgpy.data.statmech.StatmechGroups method*), 57
`descendTree()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`descendTree()` (*rmgpy.data.thermo.ThermoDepository method*), 70

`descendTree()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`descendTree()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`determinePointGroup()` (*rmgpy.qm.gaussian.GaussianMol method*), 162
`determinePointGroup()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`determinePointGroup()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`determinePointGroup()` (*rmgpy.qm.molecule.QMMolecule method*), 159
`determinePointGroup()` (*rmgpy.qm.mopac.MopacMol method*), 168
`determinePointGroup()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 169
`determinePointGroup()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
`determinePointGroup()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`DirectFit` (*class in rmgpy.data.statmechfit*), 54
`discrepancy()` (*rmgpy.kinetics.Arrhenius method*), 79
`discrepancy()` (*rmgpy.kinetics.Chebyshev method*), 88
`discrepancy()` (*rmgpy.kinetics.KineticsData method*), 78
`discrepancy()` (*rmgpy.kinetics.Lindemann method*), 91
`discrepancy()` (*rmgpy.kinetics.MultiArrhenius method*), 80
`discrepancy()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 85
`discrepancy()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`discrepancy()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`discrepancy()` (*rmgpy.kinetics.ThirdBody method*), 89
`discrepancy()` (*rmgpy.kinetics.Troe method*), 94
`discrepancy()` (*rmgpy.thermo.NASA method*), 251
`discrepancy()` (*rmgpy.thermo.NASAPolynomial method*), 254
`discrepancy()` (*rmgpy.thermo.ThermoData method*), 246
`discrepancy()` (*rmgpy.thermo.Wilhoit method*), 249
`distributeTreeDistances()` (*rmgpy.data.kinetics.KineticsFamily method*), 30
`double_possible` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138
`double_required` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138
`draw()` (*arkane.KineticsJob method*), 6
`draw()` (*arkane.PressureDependenceJob method*), 9
`draw()` (*rmgpy.data.kinetics.DepositoryReaction method*), 19
`draw()` (*rmgpy.data.kinetics.LibraryReaction method*), 44
`draw()` (*rmgpy.data.kinetics.TemplateReaction method*), 63
`draw()` (*rmgpy.molecule.draw.MoleculeDrawer method*), 146
`draw()` (*rmgpy.molecule.draw.ReactionDrawer method*), 147
`draw()` (*rmgpy.molecule.Group method*), 129
`draw()` (*rmgpy.molecule.Molecule method*), 116
`draw()` (*rmgpy.reaction.Reaction method*), 181
`draw()` (*rmgpy.rmg.pdep.PDepReaction method*), 197
`duplicate` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 20
`duplicate` (*rmgpy.data.kinetics.LibraryReaction attribute*), 44
`duplicate` (*rmgpy.data.kinetics.TemplateReaction attribute*), 63
`duplicate` (*rmgpy.reaction.Reaction attribute*), 181
`duplicate` (*rmgpy.rmg.pdep.PDepReaction attribute*), 197
`dydt0` (*rmgpy.solver.LiquidReactor attribute*), 219
`dydt0` (*rmgpy.solver.ReactionSystem attribute*), 206
`dydt0` (*rmgpy.solver.SimpleReactor attribute*), 212

E

`E0` (*rmgpy.pdep.Configuration attribute*), 151
`E0` (*rmgpy.statmech.Conformer attribute*), 243
`E0` (*rmgpy.thermo.NASA attribute*), 251
`E0` (*rmgpy.thermo.NASAPolynomial attribute*), 253
`E0` (*rmgpy.thermo.ThermoData attribute*), 246
`E0` (*rmgpy.thermo.Wilhoit attribute*), 248
`E0_prod` (*rmgpy.kinetics.Eckart attribute*), 97
`E0_reac` (*rmgpy.kinetics.Eckart attribute*), 97
`E0_TS` (*rmgpy.kinetics.Eckart attribute*), 97
`Ea` (*rmgpy.kinetics.Arrhenius attribute*), 79
`Eckart` (*class in rmgpy.kinetics*), 95
`Edge` (*class in rmgpy.molecule.graph*), 100
`edgeReactionRates` (*rmgpy.solver.LiquidReactor attribute*), 219
`edgeReactionRates` (*rmgpy.solver.ReactionSystem attribute*), 206
`edgeReactionRates` (*rmgpy.solver.SimpleReactor attribute*), 212
`edges` (*rmgpy.molecule.Atom attribute*), 111
`edges` (*rmgpy.molecule.graph.Vertex attribute*), 99
`edges` (*rmgpy.molecule.GroupAtom attribute*), 125

`edgeSpeciesRates` (*rmgpy.solver.LiquidReactor attribute*), 219

`edgeSpeciesRates` (*rmgpy.solver.ReactionSystem attribute*), 206

`edgeSpeciesRates` (*rmgpy.solver.SimpleReactor attribute*), 212

`efficiencies` (*rmgpy.kinetics.Chebyshev attribute*), 88

`efficiencies` (*rmgpy.kinetics.Lindemann attribute*), 92

`efficiencies` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85

`efficiencies` (*rmgpy.kinetics.PDepArrhenius attribute*), 84

`efficiencies` (*rmgpy.kinetics.PDepKineticsData attribute*), 82

`efficiencies` (*rmgpy.kinetics.ThirdBody attribute*), 89

`efficiencies` (*rmgpy.kinetics.Troe attribute*), 94

`Element` (*class in rmgpy.molecule*), 104

`element` (*rmgpy.molecule.Atom attribute*), 111

`elementary_high_p` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 20

`elementary_high_p` (*rmgpy.data.kinetics.LibraryReaction attribute*), 44

`elementary_high_p` (*rmgpy.data.kinetics.TemplateReaction attribute*), 63

`elementary_high_p` (*rmgpy.reaction.Reaction attribute*), 182

`elementary_high_p` (*rmgpy.rmg.pdep.PDepReaction attribute*), 197

`elementCount` (*rmgpy.molecule.Group attribute*), 129

`ElementError`, 256

`Elist` (*rmgpy.pdep.Configuration attribute*), 151

`endo_dof` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138

`endo_dof` (*rmgpy.molecule.kekulize.AromaticRing attribute*), 138

`energies` (*rmgpy.statmech.HinderedRotor attribute*), 240

`energyTransferModel` (*rmgpy.rmg.model.Species attribute*), 202

`energyTransferModel` (*rmgpy.species.Species attribute*), 225

`enlarge()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188

`ensure_species()` (*rmgpy.data.kinetics.DepositoryReaction method*), 20

`ensure_species()` (*rmgpy.data.kinetics.LibraryReaction method*), 44

`ensure_species()` (*rmgpy.data.kinetics.TemplateReaction method*), 63

`ensure_species()` (*rmgpy.reaction.Reaction method*), 182

`ensure_species()` (*rmgpy.rmg.pdep.PDepReaction method*), 197

`Entry` (*class in rmgpy.data.base*), 23

`equals()` (*rmgpy.quantity.ArrayQuantity method*), 177

`equals()` (*rmgpy.quantity.ScalarQuantity method*), 176

`equivalent()` (*rmgpy.molecule.Atom method*), 111

`equivalent()` (*rmgpy.molecule.AtomType method*), 106

`equivalent()` (*rmgpy.molecule.Bond method*), 113

`equivalent()` (*rmgpy.molecule.graph.Edge method*), 100

`equivalent()` (*rmgpy.molecule.graph.Vertex method*), 99

`equivalent()` (*rmgpy.molecule.GroupAtom method*), 125

`equivalent()` (*rmgpy.molecule.GroupBond method*), 127

`estimateKinetics()` (*rmgpy.data.kinetics.KineticsRules method*), 41

`estimateKineticsUsingGroupAdditivity()` (*rmgpy.data.kinetics.KineticsFamily method*), 30

`estimateKineticsUsingGroupAdditivity()` (*rmgpy.data.kinetics.KineticsGroups method*), 36

`estimateKineticsUsingRateRules()` (*rmgpy.data.kinetics.KineticsFamily method*), 30

`estimateRadicalThermoViaHBI()` (*rmgpy.data.thermo.ThermoDatabase method*), 67

`estimateThermoViaGroupAdditivity()` (*rmgpy.data.thermo.ThermoDatabase method*), 67

`evalExt()` (*rmgpy.data.kinetics.KineticsFamily method*), 30

`evaluate()` (*rmgpy.data.statmechfit.DirectFit method*), 54

`evaluate()` (*rmgpy.data.statmechfit.PseudoFit method*), 56

`evaluate()` (*rmgpy.data.statmechfit.PseudoRotorFit method*), 55

`execute()` (*arkane.Arkane method*), 7

`execute()` (*arkane.KineticsJob method*), 7

`execute()` (*arkane.StatMechJob method*), 10

`execute()` (*arkane.ThermoJob method*), 10

`execute()` (*rmgpy.rmg.main.RMG method*), 191

`exo_dof` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138

`exo_dof` (*rmgpy.molecule.kekulize.AromaticRing attribute*), 139

`explicitlyAllowed` (*rmgpy.rmg.model.Species attribute*), 202

`explicitlyAllowed` (*rmgpy.species.Species attribute*), 225

`exploreIsomer()` (*rmgpy.rmg.pdep.PDepNetwork method*), 194

extendNode() (*rmgpy.data.kinetics.KineticsFamily method*), 30
extendRegularization() (*rmgpy.data.kinetics.KineticsFamily method*), 30
extractSourceFromComments() (*rmgpy.data.kinetics.KineticsDatabase method*), 24
extractSourceFromComments() (*rmgpy.data.kinetics.KineticsFamily method*), 30
extractSourceFromComments() (*rmgpy.data.thermo.ThermoDatabase method*), 67

F

failureKeys (*rmgpy.qm.gaussian.Gaussian attribute*), 162
failureKeys (*rmgpy.qm.mopac.Mopac attribute*), 167
feasible() (*rmgpy.molecule.vf2.VF2 method*), 103
fillKineticsRulesByAveragingUp() (*rmgpy.data.kinetics.KineticsFamily method*), 31
fillRulesByAveragingUp() (*rmgpy.data.kinetics.KineticsRules method*), 41
find_adj_lone_pair_multiple_bond_delocalization_paths (*in module rmgpy.molecule.pathfinder*), 140
find_adj_lone_pair_radical_delocalization_paths (*in module rmgpy.molecule.pathfinder*), 140
find_adj_lone_pair_radical_multiple_bond_delocalization_paths (*in module rmgpy.molecule.pathfinder*), 141
find_allyl_delocalization_paths (*in module rmgpy.molecule.pathfinder*), 141
find_allyl_end_with_charge (*in module rmgpy.molecule.pathfinder*), 141
find_butadiene (*in module rmgpy.molecule.pathfinder*), 141
find_butadiene_end_with_charge (*in module rmgpy.molecule.pathfinder*), 141
find_H_bonds() (*rmgpy.molecule.Molecule method*), 116
find_lone_pair_multiple_bond_paths (*in module rmgpy.molecule.pathfinder*), 141
find_N5dc_radical_delocalization_paths (*in module rmgpy.molecule.pathfinder*), 140
find_shortest_path (*in module rmgpy.molecule.pathfinder*), 141
findIsomorphism() (*rmgpy.molecule.graph.Graph method*), 101
findIsomorphism() (*rmgpy.molecule.Group method*), 129
findIsomorphism() (*rmgpy.molecule.Molecule method*), 116
findIsomorphism() (*rmgpy.molecule.vf2.VF2 method*), 103
findSubgraphIsomorphisms() (*rmgpy.molecule.graph.Graph method*), 101
findSubgraphIsomorphisms() (*rmgpy.molecule.Group method*), 129
findSubgraphIsomorphisms() (*rmgpy.molecule.Molecule method*), 116
findSubgraphIsomorphisms() (*rmgpy.molecule.vf2.VF2 method*), 103
fingerprint (*rmgpy.molecule.Molecule attribute*), 117
finish() (*rmgpy.rmg.main.RMG method*), 191
fitCosinePotentialToData() (*rmgpy.statmech.HinderedRotor method*), 240
fitFourierPotentialToData() (*rmgpy.statmech.HinderedRotor method*), 240
fitStatmechDirect() (*in module rmgpy.data.statmechfit*), 53
fitStatmechPseudo() (*in module rmgpy.data.statmechfit*), 53
fitStatmechPseudoRotors() (*in module rmgpy.data.statmechfit*), 53
fitStatmechToHeatCapacity() (*in module rmgpy.data.statmechfit*), 53
fitToData() (*rmgpy.kinetics.Arrhenius method*), 79
fitToData() (*rmgpy.kinetics.Chebyshev method*), 88
fitToData() (*rmgpy.kinetics.PDepArrhenius method*), 84
fitToData() (*rmgpy.thermo.Wilhoit method*), 249
fitToDataForConstantB() (*rmgpy.thermo.Wilhoit method*), 249
fixBarrierHeight() (*rmgpy.data.kinetics.DepositoryReaction method*), 20
fixBarrierHeight() (*rmgpy.data.kinetics.LibraryReaction method*), 44
fixBarrierHeight() (*rmgpy.data.kinetics.TemplateReaction method*), 63
fixBarrierHeight() (*rmgpy.reaction.Reaction method*), 182
fixBarrierHeight() (*rmgpy.rmg.pdep.PDepReaction method*), 197
fixDiffusionLimitedA() (*rmgpy.data.kinetics.DepositoryReaction method*), 20
fixDiffusionLimitedA() (*rmgpy.data.kinetics.LibraryReaction method*), 44
fixDiffusionLimitedA() (*rmgpy.data.kinetics.TemplateReaction method*), 63
fixDiffusionLimitedA() (*rmgpy.reaction.Reaction*

method), 182
fixDiffusionLimitedA() (*rmgpy.rmg.pdep.PDepReaction method*), 197
ForbiddenStructureException, 256
formBond (*rmgpy.molecule.AtomType attribute*), 106
fourier (*rmgpy.statmech.HinderedRotor attribute*), 240
frequencies (*rmgpy.statmech.HarmonicOscillator attribute*), 238
frequency (*rmgpy.kinetics.Eckart attribute*), 97
frequency (*rmgpy.kinetics.Wigner attribute*), 95
frequency (*rmgpy.species.TransitionState attribute*), 229
frequency (*rmgpy.statmech.HinderedRotor attribute*), 240
fromAdjacencyList() (*in module rmgpy.molecule.adjlist*), 145
fromAdjacencyList() (*rmgpy.molecule.Group method*), 130
fromAdjacencyList() (*rmgpy.molecule.Molecule method*), 117
fromAdjacencyList() (*rmgpy.rmg.model.Species method*), 202
fromAdjacencyList() (*rmgpy.species.Species method*), 226
fromAugmentedInChI (*in module rmgpy.molecule.translator*), 142
fromAugmentedInChI() (*rmgpy.molecule.Molecule method*), 117
fromInChI (*in module rmgpy.molecule.translator*), 142
fromInChI() (*rmgpy.molecule.Molecule method*), 117
fromOBMol (*in module rmgpy.molecule.converter*), 142
fromRDKitMol (*in module rmgpy.molecule.converter*), 142
fromSMARTS (*in module rmgpy.molecule.translator*), 142
fromSMARTS() (*rmgpy.molecule.Molecule method*), 117
fromSMILES (*in module rmgpy.molecule.translator*), 142
fromSMILES() (*rmgpy.molecule.Molecule method*), 117
fromSMILES() (*rmgpy.rmg.model.Species method*), 202
fromSMILES() (*rmgpy.species.Species method*), 226
fromXYZ() (*rmgpy.molecule.Molecule method*), 117

G

Gaussian (*class in rmgpy.qm.gaussian*), 162
GaussianLog (*class in arkane.gaussian*), 4
GaussianMol (*class in rmgpy.qm.gaussian*), 162
GaussianMolPM3 (*class in rmgpy.qm.gaussian*), 164
GaussianMolPM6 (*class in rmgpy.qm.gaussian*), 165
generate3dTS() (*rmgpy.data.kinetics.DepositoryReaction method*), 20
generate3dTS() (*rmgpy.data.kinetics.LibraryReaction method*), 45
generate3dTS() (*rmgpy.data.kinetics.TemplateReaction method*), 63
generate3dTS() (*rmgpy.reaction.Reaction method*), 182
generate3dTS() (*rmgpy.rmg.pdep.PDepReaction method*), 197
generate_adj_lone_pair_multiple_bond_resonance_structure
(*in module rmgpy.molecule.resonance*), 135
generate_adj_lone_pair_radical_multiple_bond_resonance_s
(*in module rmgpy.molecule.resonance*), 135
generate_adj_lone_pair_radical_resonance_structures (*in module rmgpy.molecule.resonance*), 136
generate_allyl_delocalization_resonance_structures (*in module rmgpy.molecule.resonance*), 136
generate_aromatic_resonance_structure (*in module rmgpy.molecule.resonance*), 136
generate_aryne_resonance_structures (*in module rmgpy.molecule.resonance*), 136
generate_aug_inchi() (*rmgpy.rmg.model.Species method*), 202
generate_aug_inchi() (*rmgpy.species.Species method*), 226
generate_clar_structures (*in module rmgpy.molecule.resonance*), 136
generate_H_bonded_structures() (*rmgpy.molecule.Molecule method*), 117
generate_high_p_limit_kinetics() (*rmgpy.data.kinetics.LibraryReaction method*), 45
generate_isomorphic_resonance_structures (*in module rmgpy.molecule.resonance*), 136
generate_kekule_structure (*in module rmgpy.molecule.resonance*), 136
generate_lone_pair_multiple_bond_resonance_structures (*in module rmgpy.molecule.resonance*), 137
generate_N5dc_radical_resonance_structures (*in module rmgpy.molecule.resonance*), 135
generate_optimal_aromatic_resonance_structures (*in module rmgpy.molecule.resonance*), 137
generate_rate_coefficients() (*rmgpy.solver.LiquidReactor method*), 219
generate_rate_coefficients() (*rmgpy.solver.SimpleReactor method*), 212
generate_reactant_product_indices() (*rmgpy.solver.LiquidReactor method*), 219
generate_reactant_product_indices() (*rmgpy.solver.ReactionSystem method*), 206
generate_reactant_product_indices() (*rmgpy.solver.SimpleReactor method*), 212
generate_reaction_indices() (*rmgpy.solver.LiquidReactor method*), 219
generate_reaction_indices() (*rmgpy.solver.ReactionSystem method*), 206
generate_reaction_indices() (*rmgpy.solver.SimpleReactor method*), 212
generate_reactions()

(*rmgpy.data.kinetics.KineticsDatabase method*), 24
`generate_reactions_from_families()` (*rmgpy.data.kinetics.KineticsDatabase method*), 24
`generate_reactions_from_libraries()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`generate_reactions_from_library()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`generate_resonance_structures` (*in module rmgpy.molecule.resonance*), 137
`generate_resonance_structures()` (*rmgpy.molecule.Molecule method*), 117
`generate_resonance_structures()` (*rmgpy.rmg.model.Species method*), 202
`generate_resonance_structures()` (*rmgpy.species.Species method*), 226
`generate_species_indices()` (*rmgpy.solver.LiquidReactor method*), 219
`generate_species_indices()` (*rmgpy.solver.ReactionSystem method*), 206
`generate_species_indices()` (*rmgpy.solver.SimpleReactor method*), 212
`generateCanteraFiles()` (*rmgpy.rmg.main.RMG method*), 191
`generateCollisionMatrix()` (*rmgpy.pdep.Configuration method*), 151
`generateCollisionMatrix()` (*rmgpy.pdep.SingleExponentialDown method*), 149
`generateEnergyTransferModel()` (*rmgpy.rmg.model.Species method*), 202
`generateEnergyTransferModel()` (*rmgpy.species.Species method*), 226
`generateFrequencies()` (*rmgpy.data.statmech.GroupFrequencies method*), 24
`generateFullMEMatrix()` (*in module rmgpy.pdep.me*), 155
`generateGroupAdditivityValues()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`generateKinetics()` (*arkane.KineticsJob method*), 7
`generateKinetics()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
`generateOldLibraryEntry()` (*rmgpy.data.statmech.StatmechGroups method*), 57
`generateOldLibraryEntry()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`generateOldLibraryEntry()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`generateOldLibraryEntry()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`generateOldTree()` (*rmgpy.data.base.Database method*), 16
`generateOldTree()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`generateOldTree()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`generateOldTree()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`generateOldTree()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`generateOldTree()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`generateOldTree()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`generateOldTree()` (*rmgpy.data.statmech.StatmechGroups method*), 57
`generateOldTree()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`generateOldTree()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`generateOldTree()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`generateOldTree()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`generatePairs()` (*rmgpy.data.kinetics.DepositoryReaction method*), 20
`generatePairs()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`generatePairs()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`generatePairs()` (*rmgpy.reaction.Reaction method*), 182
`generatePairs()` (*rmgpy.rmg.pdep.PDepReaction method*), 197
`generatePressureList()` (*arkane.PressureDependenceJob method*), 10
`generateProductTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`generateQMData()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`generateQMData()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`generateQMData()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`generateQMData()` (*rmgpy.qm.molecule.QMMolecule method*), 159
`generateQMData()` (*rmgpy.qm.mopac.MopacMol*

method), 168
`generateQMData()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`generateQMData()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
`generateQMData()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`generateRDKitGeometries()` (*rmgpy.qm.molecule.Geometry method*), 158
`generateReactions()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`generateReverseRateCoefficient()` (*rmgpy.data.kinetics.DepositoryReaction method*), 20
`generateReverseRateCoefficient()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`generateReverseRateCoefficient()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`generateReverseRateCoefficient()` (*rmgpy.reaction.Reaction method*), 182
`generateReverseRateCoefficient()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`generateStatMech()` (*rmgpy.rmg.model.Species method*), 202
`generateStatMech()` (*rmgpy.species.Species method*), 226
`generateTemperatureList()` (*arkane.PressureDependenceJob method*), 10
`generateThermo()` (*arkane.ThermoJob method*), 10
`generateThermoData()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`generateThermoData()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`generateThermoData()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`generateThermoData()` (*rmgpy.qm.molecule.QMMolecule method*), 159
`generateThermoData()` (*rmgpy.qm.mopac.MopacMol method*), 168
`generateThermoData()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`generateThermoData()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
`generateThermoData()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`generateTransportData()` (*rmgpy.rmg.model.Species method*), 202
`generateTransportData()` (*rmgpy.species.Species method*), 226
`generateTree()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`generic` (*rmgpy.molecule.AtomType attribute*), 106
`generic_visit()` (*arkane.output.PrettifyVisitor method*), 8
`Geometry` (*class in rmgpy.qm.molecule*), 158
`get_constSPCIndices()` (*rmgpy.solver.LiquidReactor method*), 219
`get_descriptor()` (*rmgpy.molecule.Atom method*), 111
`get_edges_in_cycle()` (*rmgpy.molecule.graph.Graph method*), 102
`get_edges_in_cycle()` (*rmgpy.molecule.Group method*), 131
`get_edges_in_cycle()` (*rmgpy.molecule.Molecule method*), 120
`get_element_count()` (*rmgpy.molecule.Group method*), 131
`get_element_count()` (*rmgpy.molecule.Molecule method*), 120
`get_energy_filtered_reactions()` (*rmgpy.rmg.pdep.PDepNetwork method*), 194
`get_mean_sigma_and_epsilon()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`get_mean_sigma_and_epsilon()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`get_mean_sigma_and_epsilon()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`get_mean_sigma_and_epsilon()` (*rmgpy.reaction.Reaction method*), 183
`get_mean_sigma_and_epsilon()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`get_rate_filtered_reactions()` (*rmgpy.rmg.pdep.PDepNetwork method*), 194
`get_reduced_mass()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`get_reduced_mass()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`get_reduced_mass()` (*rmgpy.data.kinetics.TemplateReaction*

method), 65
`get_reduced_mass()` (*rmgpy.reaction.Reaction method*), 183
`get_reduced_mass()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`get_species_index()` (*rmgpy.solver.LiquidReactor method*), 219
`get_species_index()` (*rmgpy.solver.ReactionSystem method*), 206
`get_species_index()` (*rmgpy.solver.SimpleReactor method*), 213
`get_thermo_data_from_ml()` (*rmgpy.data.thermo.ThermoDatabase method*), 68
`get_threshold_rate_constants()` (*rmgpy.solver.LiquidReactor method*), 220
`get_threshold_rate_constants()` (*rmgpy.solver.SimpleReactor method*), 213
`getActiveModes()` (*rmgpy.statmech.Conformer method*), 243
`getAllCycles()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllCycles()` (*rmgpy.molecule.Group method*), 130
`getAllCycles()` (*rmgpy.molecule.Molecule method*), 117
`getAllCyclesOfSize()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllCyclesOfSize()` (*rmgpy.molecule.Group method*), 130
`getAllCyclesOfSize()` (*rmgpy.molecule.Molecule method*), 117
`getAllCyclicVertices()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllCyclicVertices()` (*rmgpy.molecule.Group method*), 130
`getAllCyclicVertices()` (*rmgpy.molecule.Molecule method*), 117
`getAllEdges()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllEdges()` (*rmgpy.molecule.Group method*), 130
`getAllEdges()` (*rmgpy.molecule.Molecule method*), 117
`getAllPolycyclicVertices()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllPolycyclicVertices()` (*rmgpy.molecule.Group method*), 130
`getAllPolycyclicVertices()` (*rmgpy.molecule.Molecule method*), 118
`getAllRules()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`getAllSimpleCyclesOfSize()` (*rmgpy.molecule.graph.Graph method*), 101
`getAllSimpleCyclesOfSize()` (*rmgpy.molecule.Group method*), 130
`getAllSimpleCyclesOfSize()` (*rmgpy.molecule.Molecule method*), 118
`getAllSpecies()` (*rmgpy.pdep.Network method*), 154
`getAllSpecies()` (*rmgpy.rmg.pdep.PDepNetwork method*), 194
`getAllThermoData()` (*rmgpy.data.thermo.ThermoDatabase method*), 67
`getAlpha()` (*rmgpy.pdep.SingleExponentialDown method*), 149
`getAromaticRings()` (*rmgpy.molecule.Molecule method*), 118
`getAtomType()` (*in module rmgpy.molecule*), 107
`getAugmentedInChI()` (*rmgpy.rmg.model.Species method*), 202
`getAugmentedInChI()` (*rmgpy.species.Species method*), 226
`getBackboneRoots()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`getBond()` (*rmgpy.molecule.Group method*), 130
`getBond()` (*rmgpy.molecule.Molecule method*), 118
`getBondOrdersForAtom()` (*rmgpy.molecule.Atom method*), 111
`getBonds()` (*rmgpy.molecule.Group method*), 130
`getBonds()` (*rmgpy.molecule.Molecule method*), 118
`getCanteraEfficiencies()` (*rmgpy.kinetics.Chebyshev method*), 88
`getCanteraEfficiencies()` (*rmgpy.kinetics.Lindemann method*), 92
`getCanteraEfficiencies()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 85
`getCanteraEfficiencies()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`getCanteraEfficiencies()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`getCanteraEfficiencies()` (*rmgpy.kinetics.ThirdBody method*), 90
`getCanteraEfficiencies()` (*rmgpy.kinetics.Troe method*), 94
`getCenterOfMass()` (*rmgpy.statmech.Conformer method*), 243
`getChargeSpan()` (*rmgpy.molecule.Molecule method*), 118
`getConversionFactorFromSI()` (*rmgpy.quantity.ArrayQuantity method*), 177
`getConversionFactorFromSI()` (*rmgpy.quantity.ScalarQuantity method*), 176

`getConversionFactorToSI()` (*rmgpy.quantity.ArrayQuantity method*), 177
`getConversionFactorToSI()` (*rmgpy.quantity.ScalarQuantity method*), 176
`getCrudeMolFilePath()` (*rmgpy.qm.molecule.Geometry method*), 158
`getDensityOfStates()` (*in module rmgpy.statmech.schrodinger*), 242
`getDensityOfStates()` (*rmgpy.rmg.model.Species method*), 202
`getDensityOfStates()` (*rmgpy.species.Species method*), 226
`getDensityOfStates()` (*rmgpy.species.TransitionState method*), 229
`getDensityOfStates()` (*rmgpy.statmech.Conformer method*), 243
`getDensityOfStates()` (*rmgpy.statmech.HarmonicOscillator method*), 238
`getDensityOfStates()` (*rmgpy.statmech.HinderedRotor method*), 240
`getDensityOfStates()` (*rmgpy.statmech.IdealGasTranslation method*), 232
`getDensityOfStates()` (*rmgpy.statmech.KRotor method*), 235
`getDensityOfStates()` (*rmgpy.statmech.LinearRotor method*), 233
`getDensityOfStates()` (*rmgpy.statmech.NonlinearRotor method*), 234
`getDensityOfStates()` (*rmgpy.statmech.SphericalTopRotor method*), 237
`getDeterministicSmallestSetOfSmallestRings()` (*rmgpy.molecule.Molecule method*), 118
`getDisparateRings()` (*rmgpy.molecule.graph.Graph method*), 101
`getDisparateRings()` (*rmgpy.molecule.Group method*), 130
`getDisparateRings()` (*rmgpy.molecule.Molecule method*), 118
`getEdge()` (*rmgpy.molecule.graph.Graph method*), 101
`getEdge()` (*rmgpy.molecule.Group method*), 130
`getEdge()` (*rmgpy.molecule.Molecule method*), 118
`getEdges()` (*rmgpy.molecule.graph.Graph method*), 101
`getEdges()` (*rmgpy.molecule.Group method*), 130
`getEdges()` (*rmgpy.molecule.Molecule method*), 118
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.Chebyshev method*), 88
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.Lindemann method*), 92
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 85
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.ThirdBody method*), 90
`getEffectiveColliderEfficiencies()` (*rmgpy.kinetics.Troe method*), 94
`getEffectivePressure()` (*rmgpy.kinetics.Chebyshev method*), 88
`getEffectivePressure()` (*rmgpy.kinetics.Lindemann method*), 92
`getEffectivePressure()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`getEffectivePressure()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`getEffectivePressure()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`getEffectivePressure()` (*rmgpy.kinetics.ThirdBody method*), 90
`getEffectivePressure()` (*rmgpy.kinetics.Troe method*), 94
`getElement()` (*in module rmgpy.molecule*), 105
`getEndRoots()` (*rmgpy.data.kinetics.KineticsFamily method*), 31
`getEnthalpiesOfReaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 20
`getEnthalpiesOfReaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`getEnthalpiesOfReaction()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getEnthalpiesOfReaction()` (*rmgpy.reaction.Reaction method*), 182
`getEnthalpiesOfReaction()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getEnthalpy()` (*in module rmgpy.statmech.schrodinger*), 242
`getEnthalpy()` (*rmgpy.pdep.Configuration method*), 151
`getEnthalpy()` (*rmgpy.rmg.model.Species method*),

202
getEnthalpy() (*rmgpy.species.Species method*), 226
getEnthalpy() (*rmgpy.species.TransitionState method*), 229
getEnthalpy() (*rmgpy.statmech.Conformer method*), 243
getEnthalpy() (*rmgpy.statmech.HarmonicOscillator method*), 238
getEnthalpy() (*rmgpy.statmech.HinderedRotor method*), 240
getEnthalpy() (*rmgpy.statmech.IdealGasTranslation method*), 232
getEnthalpy() (*rmgpy.statmech.KRotor method*), 235
getEnthalpy() (*rmgpy.statmech.LinearRotor method*), 233
getEnthalpy() (*rmgpy.statmech.NonlinearRotor method*), 234
getEnthalpy() (*rmgpy.statmech.SphericalTopRotor method*), 237
getEnthalpy() (*rmgpy.thermo.NASA method*), 251
getEnthalpy() (*rmgpy.thermo.NASAPolynomial method*), 255
getEnthalpy() (*rmgpy.thermo.ThermoData method*), 246
getEnthalpy() (*rmgpy.thermo.Wilhoit method*), 249
getEnthalpyOfReaction() (*rmgpy.data.kinetics.DepositoryReaction method*), 20
getEnthalpyOfReaction() (*rmgpy.data.kinetics.LibraryReaction method*), 45
getEnthalpyOfReaction() (*rmgpy.data.kinetics.TemplateReaction method*), 64
getEnthalpyOfReaction() (*rmgpy.reaction.Reaction method*), 182
getEnthalpyOfReaction() (*rmgpy.rmg.pdep.PDepReaction method*), 198
getEntries() (*rmgpy.data.kinetics.KineticsRules method*), 41
getEntriesReactions() (*rmgpy.data.kinetics.KineticsFamily method*), 31
getEntriesToSave() (*rmgpy.data.base.Database method*), 17
getEntriesToSave() (*rmgpy.data.kinetics.KineticsDepository method*), 27
getEntriesToSave() (*rmgpy.data.kinetics.KineticsFamily method*), 31
getEntriesToSave() (*rmgpy.data.kinetics.KineticsGroups method*), 36
getEntriesToSave() (*rmgpy.data.kinetics.KineticsLibrary method*), 39
getEntriesToSave() (*rmgpy.data.kinetics.KineticsRules method*), 41
getEntriesToSave() (*rmgpy.data.statmech.StatmechDepository method*), 51
getEntriesToSave() (*rmgpy.data.statmech.StatmechGroups method*), 58
getEntriesToSave() (*rmgpy.data.statmech.StatmechLibrary method*), 60
getEntriesToSave() (*rmgpy.data.thermo.ThermoDepository method*), 70
getEntriesToSave() (*rmgpy.data.thermo.ThermoGroups method*), 72
getEntriesToSave() (*rmgpy.data.thermo.ThermoLibrary method*), 74
getEntropiesOfReaction() (*rmgpy.data.kinetics.DepositoryReaction method*), 20
getEntropiesOfReaction() (*rmgpy.data.kinetics.LibraryReaction method*), 45
getEntropiesOfReaction() (*rmgpy.data.kinetics.TemplateReaction method*), 64
getEntropiesOfReaction() (*rmgpy.reaction.Reaction method*), 182
getEntropiesOfReaction() (*rmgpy.rmg.pdep.PDepReaction method*), 198
getEntropy() (*in module rmgpy.statmech.schrodinger*), 242
getEntropy() (*rmgpy.pdep.Configuration method*), 151
getEntropy() (*rmgpy.rmg.model.Species method*), 202
getEntropy() (*rmgpy.species.Species method*), 226
getEntropy() (*rmgpy.species.TransitionState method*), 229
getEntropy() (*rmgpy.statmech.Conformer method*), 243
getEntropy() (*rmgpy.statmech.HarmonicOscillator method*), 239
getEntropy() (*rmgpy.statmech.HinderedRotor method*), 240
getEntropy() (*rmgpy.statmech.IdealGasTranslation method*), 232
getEntropy() (*rmgpy.statmech.KRotor method*), 236
getEntropy() (*rmgpy.statmech.LinearRotor method*), 233
getEntropy() (*rmgpy.statmech.NonlinearRotor method*), 234
getEntropy() (*rmgpy.statmech.SphericalTopRotor method*), 237
getEntropy() (*rmgpy.thermo.NASA method*), 251
getEntropy() (*rmgpy.thermo.NASAPolynomial method*), 255

`getEntropy()` (*rmgpy.thermo.ThermoData method*), 246
`getEntropy()` (*rmgpy.thermo.Wilhoit method*), 249
`getEntropyOfReaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getEntropyOfReaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`getEntropyOfReaction()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getEntropyOfReaction()` (*rmgpy.reaction.Reaction method*), 182
`getEntropyOfReaction()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getEquilibriumConstant()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getEquilibriumConstant()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`getEquilibriumConstant()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getEquilibriumConstant()` (*rmgpy.reaction.Reaction method*), 182
`getEquilibriumConstant()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getEquilibriumConstants()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getEquilibriumConstants()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`getEquilibriumConstants()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getEquilibriumConstants()` (*rmgpy.reaction.Reaction method*), 183
`getEquilibriumConstants()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getExtensionEdge()` (*rmgpy.data.kinetics.KineticsFamily method*), 32
`getExtensions()` (*rmgpy.molecule.Group method*), 130
`getFeatures()` (*rmgpy.molecule.AtomType method*), 106
`getFilePath()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`getFilePath()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
`getFilePath()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`getFilePath()` (*rmgpy.qm.molecule.Geometry method*), 158
`getFilePath()` (*rmgpy.qm.molecule.QMMolecule method*), 159
`getFilePath()` (*rmgpy.qm.mopac.MopacMol method*), 168
`getFilePath()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`getFilePath()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
`getFilePath()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`getFormula()` (*rmgpy.molecule.Molecule method*), 118
`getForwardReactionForFamilyEntry()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`getFreeEnergiesOfReaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getFreeEnergiesOfReaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 45
`getFreeEnergiesOfReaction()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getFreeEnergiesOfReaction()` (*rmgpy.reaction.Reaction method*), 183
`getFreeEnergiesOfReaction()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getFreeEnergy()` (*rmgpy.pdep.Configuration method*), 151
`getFreeEnergy()` (*rmgpy.rmg.model.Species method*), 202
`getFreeEnergy()` (*rmgpy.species.Species method*), 226
`getFreeEnergy()` (*rmgpy.species.TransitionState method*), 229
`getFreeEnergy()` (*rmgpy.statmech.Conformer method*), 243
`getFreeEnergy()` (*rmgpy.thermo.NASA method*), 251
`getFreeEnergy()` (*rmgpy.thermo.NASAPolynomial method*), 255
`getFreeEnergy()` (*rmgpy.thermo.ThermoData method*), 246
`getFreeEnergy()` (*rmgpy.thermo.Wilhoit method*), 249
`getFreeEnergyOfReaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getFreeEnergyOfReaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 46

getFreeEnergyOfReaction() (*rmgpy.data.kinetics.TemplateReaction method*), 64

getFreeEnergyOfReaction() (*rmgpy.reaction.Reaction method*), 183

getFreeEnergyOfReaction() (*rmgpy.rmg.pdep.PDepReaction method*), 198

getFrequency() (*rmgpy.species.TransitionState method*), 229

getFrequency() (*rmgpy.statmech.HinderedRotor method*), 240

getFrequencyGroups() (*rmgpy.data.statmech.StatmechGroups method*), 58

getHamiltonian() (*rmgpy.statmech.HinderedRotor method*), 240

getHeatCapacity() (*in module rmgpy.statmech.schrodinger*), 242

getHeatCapacity() (*rmgpy.pdep.Configuration method*), 151

getHeatCapacity() (*rmgpy.rmg.model.Species method*), 202

getHeatCapacity() (*rmgpy.species.Species method*), 226

getHeatCapacity() (*rmgpy.species.TransitionState method*), 229

getHeatCapacity() (*rmgpy.statmech.Conformer method*), 243

getHeatCapacity() (*rmgpy.statmech.HarmonicOscillator method*), 239

getHeatCapacity() (*rmgpy.statmech.HinderedRotor method*), 241

getHeatCapacity() (*rmgpy.statmech.IdealGasTranslation method*), 232

getHeatCapacity() (*rmgpy.statmech.KRotor method*), 236

getHeatCapacity() (*rmgpy.statmech.LinearRotor method*), 233

getHeatCapacity() (*rmgpy.statmech.NonlinearRotor method*), 234

getHeatCapacity() (*rmgpy.statmech.SphericalTopRotor method*), 237

getHeatCapacity() (*rmgpy.thermo.NASA method*), 251

getHeatCapacity() (*rmgpy.thermo.NASAPolynomial method*), 255

getHeatCapacity() (*rmgpy.thermo.ThermoData method*), 246

getHeatCapacity() (*rmgpy.thermo.Wilhoit method*), 249

getInChiKeyAug() (*rmgpy.qm.gaussian.GaussianMol method*), 163

getInChiKeyAug() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164

getInChiKeyAug() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166

getInChiKeyAug() (*rmgpy.qm.molecule.QMMolecule method*), 160

getInChiKeyAug() (*rmgpy.qm.mopac.MopacMol method*), 168

getInChiKeyAug() (*rmgpy.qm.mopac.MopacMolPM3 method*), 170

getInChiKeyAug() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171

getInChiKeyAug() (*rmgpy.qm.mopac.MopacMolPM7 method*), 173

getInternalReducedMomentOfInertia() (*rmgpy.statmech.Conformer method*), 243

getKinetics() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getKineticsForTemplate() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getKineticsFromDepository() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getLabeledAtom() (*rmgpy.molecule.Group method*), 130

getLabeledAtom() (*rmgpy.molecule.Molecule method*), 118

getLabeledAtoms() (*rmgpy.molecule.Group method*), 130

getLabeledAtoms() (*rmgpy.molecule.Molecule method*), 119

getLabeledReactantsAndProducts() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getLargestRing() (*rmgpy.molecule.graph.Graph method*), 101

getLargestRing() (*rmgpy.molecule.Group method*), 131

getLargestRing() (*rmgpy.molecule.Molecule method*), 119

getLayeringIndices() (*rmgpy.solver.LiquidReactor method*), 219

getLayeringIndices() (*rmgpy.solver.ReactionSystem method*), 206

getLayeringIndices() (*rmgpy.solver.SimpleReactor method*), 213

getLeakBranchingRatios() (*rmgpy.rmg.pdep.PDepNetwork method*), 194

getLeakCoefficient() (*rmgpy.rmg.pdep.PDepNetwork method*), 194

getLevelDegeneracy() (*rmgpy.statmech.HinderedRotor method*),

241
getLevelDegeneracy() (*rmgpy.statmech.KRotor method*), 236
getLevelDegeneracy() (*rmgpy.statmech.LinearRotor method*), 233
getLevelDegeneracy() (*rmgpy.statmech.SphericalTopRotor method*), 237
getLevelEnergy() (*rmgpy.statmech.HinderedRotor method*), 241
getLevelEnergy() (*rmgpy.statmech.KRotor method*), 236
getLevelEnergy() (*rmgpy.statmech.LinearRotor method*), 233
getLevelEnergy() (*rmgpy.statmech.SphericalTopRotor method*), 237
getLibraryReactions() (*rmgpy.data.kinetics.KineticsLibrary method*), 39
getLists() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
getMaxCycleOverlap() (*rmgpy.molecule.graph.Graph method*), 102
getMaxCycleOverlap() (*rmgpy.molecule.Group method*), 131
getMaxCycleOverlap() (*rmgpy.molecule.Molecule method*), 119
getMaximumLeakSpecies() (*rmgpy.rmg.pdep.PDepNetwork method*), 194
getModelSize() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
getMolecularWeight() (*rmgpy.molecule.Molecule method*), 119
getMolFilePathForCalculation() (*rmgpy.qm.gaussian.GaussianMol method*), 163
getMolFilePathForCalculation() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
getMolFilePathForCalculation() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
getMolFilePathForCalculation() (*rmgpy.qm.molecule.QMMolecule method*), 160
getMolFilePathForCalculation() (*rmgpy.qm.mopac.MopacMol method*), 168
getMolFilePathForCalculation() (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
getMolFilePathForCalculation() (*rmgpy.qm.mopac.MopacMolPM6 method*), 171
getMolFilePathForCalculation() (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
getMomentOfInertiaTensor() (*rmgpy.statmech.Conformer method*), 244
getMonocyclicRings() (*rmgpy.molecule.graph.Graph method*), 102
getMonocyclicRings() (*rmgpy.molecule.Group method*), 131
getMonocyclicRings() (*rmgpy.molecule.Molecule method*), 119
getNetCharge() (*rmgpy.molecule.Group method*), 131
getNetCharge() (*rmgpy.molecule.Molecule method*), 119
getNthNeighbor() (*rmgpy.molecule.Molecule method*), 119
getNumAtoms() (*rmgpy.molecule.Molecule method*), 119
getNumberDegreesOfFreedom() (*rmgpy.statmech.Conformer method*), 244
getNumberOfAtoms() (*arkane.gaussian.GaussianLog method*), 4
getNumberOfAtoms() (*arkane.molpro.MolproLog method*), 6
getNumberOfAtoms() (*arkane.qchem.QChemLog method*), 5
getOrderNum() (*rmgpy.molecule.Bond method*), 113
getOrderNum() (*rmgpy.molecule.GroupBond method*), 127
getOrderStr() (*rmgpy.molecule.Bond method*), 113
getOrderStr() (*rmgpy.molecule.GroupBond method*), 127
getOtherVertex() (*rmgpy.molecule.Bond method*), 113
getOtherVertex() (*rmgpy.molecule.graph.Edge method*), 100
getOtherVertex() (*rmgpy.molecule.GroupBond method*), 127
getParser() (*rmgpy.qm.gaussian.GaussianMol method*), 163
getParser() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 164
getParser() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
getParser() (*rmgpy.qm.mopac.Mopac method*), 167
getParser() (*rmgpy.qm.mopac.MopacMol method*), 168
getParser() (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
getParser() (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
getParser() (*rmgpy.qm.mopac.MopacMolPM7*

method), 173

getPartitionFunction() (*in module rmgpy.statmech.schrodinger*), 242

getPartitionFunction() (*rmgpy.rmg.model.Species method*), 202

getPartitionFunction() (*rmgpy.species.Species method*), 226

getPartitionFunction() (*rmgpy.species.TransitionState method*), 229

getPartitionFunction() (*rmgpy.statmech.Conformer method*), 244

getPartitionFunction() (*rmgpy.statmech.HarmonicOscillator method*), 239

getPartitionFunction() (*rmgpy.statmech.HinderedRotor method*), 241

getPartitionFunction() (*rmgpy.statmech.IdealGasTranslation method*), 232

getPartitionFunction() (*rmgpy.statmech.KRotor method*), 236

getPartitionFunction() (*rmgpy.statmech.LinearRotor method*), 233

getPartitionFunction() (*rmgpy.statmech.NonlinearRotor method*), 234

getPartitionFunction() (*rmgpy.statmech.SphericalTopRotor method*), 237

getPolycyclicRings() (*rmgpy.molecule.graph.Graph method*), 102

getPolycyclicRings() (*rmgpy.molecule.Group method*), 131

getPolycyclicRings() (*rmgpy.molecule.Molecule method*), 119

getPossibleStructures() (*rmgpy.data.base.LogicOr method*), 48

getPotential() (*rmgpy.statmech.HinderedRotor method*), 241

getPrincipalMomentsOfInertia() (*rmgpy.statmech.Conformer method*), 244

getRadicalAtoms() (*rmgpy.molecule.Molecule method*), 119

getRadicalCount() (*rmgpy.molecule.Molecule method*), 119

getRateCoefficient() (*rmgpy.data.kinetics.DepositoryReaction method*), 21

getRateCoefficient() (*rmgpy.data.kinetics.LibraryReaction method*), 46

getRateCoefficient() (*rmgpy.data.kinetics.TemplateReaction method*), 64

getRateCoefficient() (*rmgpy.kinetics.Arrhenius method*), 79

getRateCoefficient() (*rmgpy.kinetics.Chebyshev method*), 88

getRateCoefficient() (*rmgpy.kinetics.KineticsData method*), 78

getRateCoefficient() (*rmgpy.kinetics.Lindemann method*), 92

getRateCoefficient() (*rmgpy.kinetics.MultiArrhenius method*), 80

getRateCoefficient() (*rmgpy.kinetics.MultiPDepArrhenius method*), 86

getRateCoefficient() (*rmgpy.kinetics.PDepArrhenius method*), 84

getRateCoefficient() (*rmgpy.kinetics.PDepKineticsData method*), 82

getRateCoefficient() (*rmgpy.kinetics.ThirdBody method*), 90

getRateCoefficient() (*rmgpy.kinetics.Troe method*), 94

getRateCoefficient() (*rmgpy.reaction.Reaction method*), 183

getRateCoefficient() (*rmgpy.rmg.pdep.PDepReaction method*), 198

getRateRule() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getReactionPairs() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getReactionTemplate() (*rmgpy.data.kinetics.KineticsFamily method*), 32

getReactionTemplate() (*rmgpy.data.kinetics.KineticsGroups method*), 36

getReactionTemplateLabels() (*rmgpy.data.kinetics.KineticsFamily method*), 33

getRefinedMolFilePath() (*rmgpy.qm.molecule.Geometry method*), 158

getRelevantCycles() (*rmgpy.molecule.graph.Graph method*), 102

getRelevantCycles() (*rmgpy.molecule.Group method*), 131

getRelevantCycles() (*rmgpy.molecule.Molecule method*), 119

getResonanceHybrid() (*rmgpy.rmg.model.Species*

method), 203
`getResonanceHybrid()` (*rmgpy.species.Species method*), 226
`getReverse()` (*rmgpy.data.kinetics.ReactionRecipe method*), 49
`getRingGroupsFromComments()` (*rmgpy.data.thermo.ThermoDatabase method*), 68
`getRootTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`getRule()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`getSingletCarbeneCount()` (*rmgpy.molecule.Molecule method*), 119
`getSmallestSetOfSmallestRings()` (*rmgpy.molecule.graph.Graph method*), 102
`getSmallestSetOfSmallestRings()` (*rmgpy.molecule.Group method*), 131
`getSmallestSetOfSmallestRings()` (*rmgpy.molecule.Molecule method*), 119
`getSource()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getSource()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`getSource()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getSource()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getSourcesForTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`getSpecies()` (*rmgpy.data.base.Database method*), 17
`getSpecies()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`getSpecies()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`getSpecies()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`getSpecies()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`getSpecies()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`getSpecies()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`getSpecies()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`getSpecies()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`getSpecies()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`getSpecies()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`getSpecies()` (*rmgpy.data.thermo.ThermoLibrary method*), 74
`getSpecies()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
`getSpeciesIdentifier()` (*in module rmgpy.chemkin*), 13
`getStatmechData()` (*rmgpy.data.statmech.StatmechDatabase method*), 49
`getStatmechData()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`getStatmechDataFromDepository()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`getStatmechDataFromGroups()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`getStatmechDataFromLibrary()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`getStoichiometricCoefficient()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`getStoichiometricCoefficient()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`getStoichiometricCoefficient()` (*rmgpy.data.kinetics.TemplateReaction method*), 64
`getStoichiometricCoefficient()` (*rmgpy.reaction.Reaction method*), 183
`getStoichiometricCoefficient()` (*rmgpy.rmg.pdep.PDepReaction method*), 198
`getStoichiometryMatrix()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
`getSumOfStates()` (*in module rmgpy.statmech.schrodinger*), 242
`getSumOfStates()` (*rmgpy.rmg.model.Species method*), 203
`getSumOfStates()` (*rmgpy.species.Species method*), 226
`getSumOfStates()` (*rmgpy.species.TransitionState method*), 229
`getSumOfStates()` (*rmgpy.statmech.Conformer method*), 244
`getSumOfStates()` (*rmgpy.statmech.HarmonicOscillator method*), 239
`getSumOfStates()` (*rmgpy.statmech.HinderedRotor method*), 241
`getSumOfStates()` (*rmgpy.statmech.IdealGasTranslation method*), 232
`getSumOfStates()` (*rmgpy.statmech.KRotor method*), 236
`getSumOfStates()` (*rmgpy.statmech.LinearRotor*

method), 233
getSumOfStates() (rmgpy.statmech.NonlinearRotor method), 234
getSumOfStates() (rmgpy.statmech.SphericalTopRotor method), 237
getSymmetricTopRotors() (rmgpy.statmech.Conformer method), 244
getSymmetryNumber() (rmgpy.molecule.Molecule method), 120
getSymmetryNumber() (rmgpy.rmg.model.Species method), 203
getSymmetryNumber() (rmgpy.species.Species method), 226
getTemplateKinetics() (rmgpy.data.kinetics.KineticsFamily method), 33
getThermoData() (rmgpy.data.thermo.ThermoDatabase method), 68
getThermoData() (rmgpy.qm.main.QMCalculator method), 158
getThermoData() (rmgpy.rmg.model.Species method), 203
getThermoData() (rmgpy.species.Species method), 226
getThermoDataFromDepository() (rmgpy.data.thermo.ThermoDatabase method), 68
getThermoDataFromGroups() (rmgpy.data.thermo.ThermoDatabase method), 68
getThermoDataFromLibraries() (rmgpy.data.thermo.ThermoDatabase method), 68
getThermoDataFromLibrary() (rmgpy.data.thermo.ThermoDatabase method), 68
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMol method), 163
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMolPM3 method), 165
getThermoFilePath() (rmgpy.qm.gaussian.GaussianMolPM6 method), 166
getThermoFilePath() (rmgpy.qm.molecule.QMMolecule method), 160
getThermoFilePath() (rmgpy.qm.mopac.MopacMol method), 168
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM3 method), 170
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM6 method), 172
getThermoFilePath() (rmgpy.qm.mopac.MopacMolPM7 method), 173
getTopLevelGroups() (rmgpy.data.kinetics.KineticsFamily method), 33
getTotalMass() (rmgpy.statmech.Conformer method), 244
getTrainingDepository() (rmgpy.data.kinetics.KineticsFamily method), 33
getTransportData() (rmgpy.rmg.model.Species method), 203
getTransportData() (rmgpy.species.Species method), 227
getUncertainty() (rmgpy.quantity.ArrayQuantity method), 178
getUncertainty() (rmgpy.quantity.ScalarQuantity method), 176
getUncertaintyType() (rmgpy.quantity.ArrayQuantity method), 178
getUncertaintyType() (rmgpy.quantity.ScalarQuantity method), 176
getURL() (rmgpy.data.kinetics.DepositoryReaction method), 21
getURL() (rmgpy.data.kinetics.LibraryReaction method), 46
getURL() (rmgpy.data.kinetics.TemplateReaction method), 65
getURL() (rmgpy.molecule.Molecule method), 120
getURL() (rmgpy.reaction.Reaction method), 183
getURL() (rmgpy.rmg.pdep.PDepReaction method), 198
getValue() (rmgpy.quantity.ArrayQuantity method), 178
getValue() (rmgpy.quantity.ScalarQuantity method), 176
Graph (class in rmgpy.molecule.graph), 100
groundStateDegeneracy (rmgpy.qm.qmdata.QMData attribute), 160
Group (class in rmgpy.molecule), 128
GroupAtom (class in rmgpy.molecule), 124
GroupBond (class in rmgpy.molecule), 126
GroupFrequencies (class in rmgpy.data.statmech), 24

H

H0 (rmgpy.thermo.Wilhoit attribute), 248
H298 (rmgpy.thermo.ThermoData attribute), 246
HarmonicOscillator (class in rmgpy.statmech), 238
harmonicOscillator_d_heatCapacity_d_freq() (in module rmgpy.data.statmechfit), 53

`harmonicOscillator_heatCapacity()` (*in module rmgpy.data.statmechfit*), 53
`has_lone_pairs()` (*rmgpy.molecule.Molecule method*), 120
`has_reactive_molecule()` (*rmgpy.rmg.model.Species method*), 203
`has_reactive_molecule()` (*rmgpy.species.Species method*), 227
`hasAtom()` (*rmgpy.molecule.Group method*), 131
`hasAtom()` (*rmgpy.molecule.Molecule method*), 120
`hasBond()` (*rmgpy.molecule.Group method*), 131
`hasBond()` (*rmgpy.molecule.Molecule method*), 120
`hasEdge()` (*rmgpy.molecule.graph.Graph method*), 102
`hasEdge()` (*rmgpy.molecule.Group method*), 131
`hasEdge()` (*rmgpy.molecule.Molecule method*), 120
`hasRateRule()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`hasRule()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`hasStatMech()` (*rmgpy.pdep.Configuration method*), 151
`hasStatMech()` (*rmgpy.rmg.model.Species method*), 203
`hasStatMech()` (*rmgpy.species.Species method*), 227
`hasTemplate()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`hasTemplate()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`hasTemplate()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`hasTemplate()` (*rmgpy.reaction.Reaction method*), 183
`hasTemplate()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`hasThermo()` (*rmgpy.pdep.Configuration method*), 152
`hasThermo()` (*rmgpy.rmg.model.Species method*), 203
`hasThermo()` (*rmgpy.species.Species method*), 227
`hasVertex()` (*rmgpy.molecule.graph.Graph method*), 102
`hasVertex()` (*rmgpy.molecule.Group method*), 131
`hasVertex()` (*rmgpy.molecule.Molecule method*), 120
`hasWildcards()` (*rmgpy.molecule.GroupAtom method*), 125
`highPlimit` (*rmgpy.kinetics.Chebyshev attribute*), 88
`highPlimit` (*rmgpy.kinetics.Lindemann attribute*), 92
`highPlimit` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 86
`highPlimit` (*rmgpy.kinetics.PDepArrhenius attribute*), 84
`highPlimit` (*rmgpy.kinetics.PDepKineticsData attribute*), 82
`highPlimit` (*rmgpy.kinetics.ThirdBody attribute*), 90
`highPlimit` (*rmgpy.kinetics.Troe attribute*), 94
`HinderedRotor` (*class in rmgpy.statmech*), 239
`hinderedRotor_d_heatCapacity_d_barr()` (*in module rmgpy.data.statmechfit*), 53
`hinderedRotor_d_heatCapacity_d_freq()` (*in module rmgpy.data.statmechfit*), 53
`hinderedRotor_heatCapacity()` (*in module rmgpy.data.statmechfit*), 53

I

`id` (*rmgpy.molecule.Atom attribute*), 111
`IdealGasTranslation` (*class in rmgpy.statmech*), 231
`identifyRingMembership()` (*rmgpy.molecule.Molecule method*), 120
`ignore` (*rmgpy.molecule.Atom attribute*), 111
`ignore` (*rmgpy.molecule.graph.Vertex attribute*), 99
`ignore` (*rmgpy.molecule.GroupAtom attribute*), 125
`ILPSolutionError`, 256
`ImplicitBenzeneError`, 256
`implicitHydrogens` (*rmgpy.molecule.Molecule attribute*), 120
`InChI` (*rmgpy.molecule.Molecule attribute*), 115
`InchiException`, 256
`incrementBond` (*rmgpy.molecule.AtomType attribute*), 106
`incrementLonePair` (*rmgpy.molecule.AtomType attribute*), 106
`incrementLonePairs()` (*rmgpy.molecule.Atom method*), 111
`incrementOrder()` (*rmgpy.molecule.Bond method*), 113
`incrementRadical` (*rmgpy.molecule.AtomType attribute*), 106
`incrementRadical()` (*rmgpy.molecule.Atom method*), 111
`index` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 21
`index` (*rmgpy.data.kinetics.LibraryReaction attribute*), 46
`index` (*rmgpy.data.kinetics.TemplateReaction attribute*), 65
`index` (*rmgpy.reaction.Reaction attribute*), 183
`index` (*rmgpy.rmg.model.Species attribute*), 203
`index` (*rmgpy.rmg.pdep.PDepReaction attribute*), 199
`index` (*rmgpy.species.Species attribute*), 227
`inertia` (*rmgpy.statmech.HinderedRotor attribute*), 241
`inertia` (*rmgpy.statmech.KRotor attribute*), 236
`inertia` (*rmgpy.statmech.LinearRotor attribute*), 233
`inertia` (*rmgpy.statmech.NonlinearRotor attribute*), 234
`inertia` (*rmgpy.statmech.SphericalTopRotor attribute*), 237
`inflate()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
`initialConcentrations` (*rmgpy.solver.LiquidReactor attribute*), 220

`initialize()` (*rmgpy.data.statmechfit.DirectFit method*), 54
`initialize()` (*rmgpy.data.statmechfit.PseudoFit method*), 56
`initialize()` (*rmgpy.data.statmechfit.PseudoRotorFit method*), 55
`initialize()` (*rmgpy.pdep.Network method*), 154
`initialize()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`initialize()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
`initialize()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`initialize()` (*rmgpy.qm.main.QMCalculator method*), 158
`initialize()` (*rmgpy.qm.molecule.QMMolecule method*), 160
`initialize()` (*rmgpy.qm.mopac.MopacMol method*), 168
`initialize()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`initialize()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
`initialize()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`initialize()` (*rmgpy.rmg.main.RMG method*), 192
`initialize()` (*rmgpy.rmg.pdep.PDepNetwork method*), 194
`initialize()` (*rmgpy.solver.LiquidReactor method*), 220
`initialize()` (*rmgpy.solver.ReactionSystem method*), 206
`initialize()` (*rmgpy.solver.SimpleReactor method*), 213
`initialize_solver()` (*rmgpy.solver.LiquidReactor method*), 220
`initialize_solver()` (*rmgpy.solver.ReactionSystem method*), 207
`initialize_solver()` (*rmgpy.solver.SimpleReactor method*), 213
`initialize_surface()` (*rmgpy.solver.LiquidReactor method*), 220
`initialize_surface()` (*rmgpy.solver.ReactionSystem method*), 207
`initialize_surface()` (*rmgpy.solver.SimpleReactor method*), 213
`initializeIndexSpeciesDict()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
`initializeLog()` (*arkane.Arkane method*), 7
`initializeLog()` (*in module rmgpy.rmg.main*), 193
`initializeModel()` (*rmgpy.solver.LiquidReactor method*), 220
`initializeModel()` (*rmgpy.solver.ReactionSystem method*), 206
`initializeModel()` (*rmgpy.solver.SimpleReactor method*), 213
`initialMoleFractions` (*rmgpy.solver.SimpleReactor attribute*), 213
`initiate_tolerances()` (*rmgpy.solver.LiquidReactor method*), 220
`initiate_tolerances()` (*rmgpy.solver.ReactionSystem method*), 207
`initiate_tolerances()` (*rmgpy.solver.SimpleReactor method*), 213
`InputError`, 256
`inputFileKeywords()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`inputFileKeywords()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
`inputFileKeywords()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`inputFileKeywords()` (*rmgpy.qm.mopac.MopacMol method*), 168
`inputFileKeywords()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`inputFileKeywords()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
`inputFileKeywords()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`inputFilePath` (*rmgpy.qm.gaussian.GaussianMol attribute*), 163
`inputFilePath` (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 165
`inputFilePath` (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 166
`inputFilePath` (*rmgpy.qm.molecule.QMMolecule attribute*), 160
`inputFilePath` (*rmgpy.qm.mopac.MopacMol attribute*), 168
`inputFilePath` (*rmgpy.qm.mopac.MopacMolPM3 attribute*), 170
`inputFilePath` (*rmgpy.qm.mopac.MopacMolPM6 attribute*), 172
`inputFilePath` (*rmgpy.qm.mopac.MopacMolPM7 attribute*), 173
`inputFilePath` (*rmgpy.qm.symmetry.SymmetryJob attribute*), 161
`InvalidActionError`, 256
`InvalidAdjacencyListError`, 256
`invalidate()` (*rmgpy.pdep.Network method*), 154
`invalidate()` (*rmgpy.rmg.pdep.PDepNetwork*

method), 194
`InvalidMicrocanonicalRateError`, 256
`is_atom_able_to_gain_lone_pair` (*in module rmgpy.molecule.pathfinder*), 142
`is_atom_able_to_lose_lone_pair` (*in module rmgpy.molecule.pathfinder*), 142
`is_equal()` (*rmgpy.molecule.Molecule method*), 121
`is_forward` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22
`is_forward` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`is_forward` (*rmgpy.data.kinetics.TemplateReaction attribute*), 65
`is_forward` (*rmgpy.reaction.Reaction attribute*), 184
`is_forward` (*rmgpy.rmg.pdep.PDepReaction attribute*), 199
`is_structure_in_list()` (*rmgpy.rmg.model.Species method*), 203
`is_structure_in_list()` (*rmgpy.species.Species method*), 227
`isAromatic()` (*rmgpy.molecule.Molecule method*), 120
`isAromaticRing()` (*rmgpy.molecule.Group method*), 132
`isArylRadical()` (*rmgpy.molecule.Molecule method*), 120
`isAssociation()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`isAssociation()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`isAssociation()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isAssociation()` (*rmgpy.reaction.Reaction method*), 183
`isAssociation()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isAtomInCycle()` (*rmgpy.molecule.Molecule method*), 120
`isBalanced()` (*rmgpy.data.kinetics.DepositoryReaction method*), 21
`isBalanced()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`isBalanced()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isBalanced()` (*rmgpy.reaction.Reaction method*), 183
`isBalanced()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isBenzene()` (*rmgpy.molecule.Bond method*), 113
`isBenzene()` (*rmgpy.molecule.GroupBond method*), 127
`isBenzeneExplicit()` (*rmgpy.molecule.Group method*), 132
`isBimolecular()` (*rmgpy.pdep.Configuration method*), 152
`isBondInCycle()` (*rmgpy.molecule.Molecule method*), 120
`isCarbon()` (*rmgpy.molecule.Atom method*), 112
`isChlorine()` (*rmgpy.molecule.Atom method*), 112
`isCyclic()` (*rmgpy.molecule.graph.Graph method*), 102
`isCyclic()` (*rmgpy.molecule.Group method*), 132
`isCyclic()` (*rmgpy.molecule.Molecule method*), 121
`isDissociation()` (*rmgpy.data.kinetics.DepositoryReaction method*), 22
`isDissociation()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`isDissociation()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isDissociation()` (*rmgpy.reaction.Reaction method*), 183
`isDissociation()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isDouble()` (*rmgpy.molecule.Bond method*), 113
`isDouble()` (*rmgpy.molecule.GroupBond method*), 127
`isEdgeInCycle()` (*rmgpy.molecule.graph.Graph method*), 102
`isEdgeInCycle()` (*rmgpy.molecule.Group method*), 132
`isEdgeInCycle()` (*rmgpy.molecule.Molecule method*), 121
`isFluorine()` (*rmgpy.molecule.Atom method*), 112
`isHydrogen()` (*rmgpy.molecule.Atom method*), 112
`isHydrogenBond()` (*rmgpy.molecule.Bond method*), 114
`isHydrogenBond()` (*rmgpy.molecule.GroupBond method*), 127
`isIdentical()` (*rmgpy.molecule.Group method*), 132
`isIdentical()` (*rmgpy.molecule.Molecule method*), 121
`isIdentical()` (*rmgpy.rmg.model.Species method*), 203
`isIdentical()` (*rmgpy.species.Species method*), 227
`isIdenticalTo()` (*rmgpy.kinetics.Arrhenius method*), 79
`isIdenticalTo()` (*rmgpy.kinetics.Chebyshev method*), 88
`isIdenticalTo()` (*rmgpy.kinetics.KineticsData method*), 78
`isIdenticalTo()` (*rmgpy.kinetics.Lindemann method*), 92
`isIdenticalTo()` (*rmgpy.kinetics.MultiArrhenius method*), 81
`isIdenticalTo()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`isIdenticalTo()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`isIdenticalTo()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`isIdenticalTo()` (*rmgpy.kinetics.ThirdBody method*),

90
`isIdenticalTo()` (*rmgpy.kinetics.Troe method*), 94
`isIdenticalTo()` (*rmgpy.thermo.NASA method*), 251
`isIdenticalTo()` (*rmgpy.thermo.NASAPolynomial method*), 255
`isIdenticalTo()` (*rmgpy.thermo.ThermoData method*), 246
`isIdenticalTo()` (*rmgpy.thermo.Wilhoit method*), 249
`isIodine()` (*rmgpy.molecule.Atom method*), 112
`isIsomerization()` (*rmgpy.data.kinetics.DepositoryReaction method*), 22
`isIsomerization()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`isIsomerization()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isIsomerization()` (*rmgpy.reaction.Reaction method*), 183
`isIsomerization()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isIsomorphic()` (*rmgpy.data.kinetics.DepositoryReaction method*), 22
`isIsomorphic()` (*rmgpy.data.kinetics.LibraryReaction method*), 46
`isIsomorphic()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isIsomorphic()` (*rmgpy.molecule.graph.Graph method*), 102
`isIsomorphic()` (*rmgpy.molecule.Group method*), 132
`isIsomorphic()` (*rmgpy.molecule.Molecule method*), 121
`isIsomorphic()` (*rmgpy.molecule.vf2.VF2 method*), 104
`isIsomorphic()` (*rmgpy.reaction.Reaction method*), 183
`isIsomorphic()` (*rmgpy.rmg.model.Species method*), 203
`isIsomorphic()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isIsomorphic()` (*rmgpy.species.Species method*), 227
`isLinear()` (*rmgpy.molecule.Molecule method*), 121
`isMappingValid()` (*rmgpy.molecule.graph.Graph method*), 102
`isMappingValid()` (*rmgpy.molecule.Group method*), 132
`isMappingValid()` (*rmgpy.molecule.Molecule method*), 121
`isMoleculeForbidden()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`isNitrogen()` (*rmgpy.molecule.Atom method*), 112
`isNonHydrogen()` (*rmgpy.molecule.Atom method*), 112
`isNOS()` (*rmgpy.molecule.Atom method*), 112
`isOrder()` (*rmgpy.molecule.Bond method*), 114
`isotope` (*rmgpy.molecule.Element attribute*), 104
`isOxygen()` (*rmgpy.molecule.Atom method*), 112
`isOxygen()` (*rmgpy.molecule.GroupAtom method*), 125
`isPressureDependent()` (*rmgpy.kinetics.Arrhenius method*), 79
`isPressureDependent()` (*rmgpy.kinetics.Chebyshev method*), 88
`isPressureDependent()` (*rmgpy.kinetics.KineticsData method*), 78
`isPressureDependent()` (*rmgpy.kinetics.Lindemann method*), 92
`isPressureDependent()` (*rmgpy.kinetics.MultiArrhenius method*), 81
`isPressureDependent()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`isPressureDependent()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`isPressureDependent()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`isPressureDependent()` (*rmgpy.kinetics.ThirdBody method*), 90
`isPressureDependent()` (*rmgpy.kinetics.Troe method*), 94
`isPressureValid()` (*rmgpy.kinetics.Chebyshev method*), 88
`isPressureValid()` (*rmgpy.kinetics.Lindemann method*), 92
`isPressureValid()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`isPressureValid()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`isPressureValid()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`isPressureValid()` (*rmgpy.kinetics.ThirdBody method*), 90
`isPressureValid()` (*rmgpy.kinetics.Troe method*), 95
`isRadical()` (*rmgpy.molecule.Molecule method*), 121
`isSilicon()` (*rmgpy.molecule.Atom method*), 112
`isSimilarTo()` (*rmgpy.kinetics.Arrhenius method*), 79
`isSimilarTo()` (*rmgpy.kinetics.Chebyshev method*), 88
`isSimilarTo()` (*rmgpy.kinetics.KineticsData method*), 78
`isSimilarTo()` (*rmgpy.kinetics.Lindemann method*), 92
`isSimilarTo()` (*rmgpy.kinetics.MultiArrhenius method*), 81
`isSimilarTo()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`isSimilarTo()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`isSimilarTo()` (*rmgpy.kinetics.PDepKineticsData method*), 82

`isSimilarTo()` (*rmgpy.kinetics.ThirdBody method*), 90
`isSimilarTo()` (*rmgpy.kinetics.Troe method*), 95
`isSimilarTo()` (*rmgpy.thermo.NASA method*), 251
`isSimilarTo()` (*rmgpy.thermo.NASAPolynomial method*), 255
`isSimilarTo()` (*rmgpy.thermo.ThermoData method*), 246
`isSimilarTo()` (*rmgpy.thermo.Wilhoit method*), 249
`isSingle()` (*rmgpy.molecule.Bond method*), 114
`isSingle()` (*rmgpy.molecule.GroupBond method*), 127
`isSolvent` (*rmgpy.rmg.model.Species attribute*), 203
`isSolvent` (*rmgpy.species.Species attribute*), 227
`isSpecificCaseOf()` (*rmgpy.molecule.Atom method*), 112
`isSpecificCaseOf()` (*rmgpy.molecule.AtomType method*), 106
`isSpecificCaseOf()` (*rmgpy.molecule.Bond method*), 114
`isSpecificCaseOf()` (*rmgpy.molecule.graph.Edge method*), 100
`isSpecificCaseOf()` (*rmgpy.molecule.graph.Vertex method*), 99
`isSpecificCaseOf()` (*rmgpy.molecule.GroupAtom method*), 125
`isSpecificCaseOf()` (*rmgpy.molecule.GroupBond method*), 127
`isSubgraphIsomorphic()` (*rmgpy.molecule.graph.Graph method*), 102
`isSubgraphIsomorphic()` (*rmgpy.molecule.Group method*), 132
`isSubgraphIsomorphic()` (*rmgpy.molecule.Molecule method*), 121
`isSubgraphIsomorphic()` (*rmgpy.molecule.vf2.VF2 method*), 104
`isSulfur()` (*rmgpy.molecule.Atom method*), 112
`isSulfur()` (*rmgpy.molecule.GroupAtom method*), 125
`isTemperatureValid()` (*rmgpy.kinetics.Arrhenius method*), 79
`isTemperatureValid()` (*rmgpy.kinetics.Chebyshev method*), 88
`isTemperatureValid()` (*rmgpy.kinetics.KineticsData method*), 78
`isTemperatureValid()` (*rmgpy.kinetics.Lindemann method*), 92
`isTemperatureValid()` (*rmgpy.kinetics.MultiArrhenius method*), 81
`isTemperatureValid()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
`isTemperatureValid()` (*rmgpy.kinetics.PDepArrhenius method*), 84
`isTemperatureValid()` (*rmgpy.kinetics.PDepKineticsData method*), 82
`isTemperatureValid()` (*rmgpy.kinetics.ThirdBody method*), 90
`isTemperatureValid()` (*rmgpy.kinetics.Troe method*), 95
`isTemperatureValid()` (*rmgpy.thermo.NASA method*), 251
`isTemperatureValid()` (*rmgpy.thermo.NASAPolynomial method*), 255
`isTemperatureValid()` (*rmgpy.thermo.ThermoData method*), 246
`isTemperatureValid()` (*rmgpy.thermo.Wilhoit method*), 249
`isTransitionState()` (*rmgpy.pdep.Configuration method*), 152
`isTriple()` (*rmgpy.molecule.Bond method*), 114
`isTriple()` (*rmgpy.molecule.GroupBond method*), 127
`isUncertaintyAdditive()` (*rmgpy.quantity.ArrayQuantity method*), 178
`isUncertaintyAdditive()` (*rmgpy.quantity.ScalarQuantity method*), 176
`isUncertaintyMultiplicative()` (*rmgpy.quantity.ArrayQuantity method*), 178
`isUncertaintyMultiplicative()` (*rmgpy.quantity.ScalarQuantity method*), 176
`isUnimolecular()` (*rmgpy.data.kinetics.DepositoryReaction method*), 22
`isUnimolecular()` (*rmgpy.data.kinetics.LibraryReaction method*), 47
`isUnimolecular()` (*rmgpy.data.kinetics.TemplateReaction method*), 65
`isUnimolecular()` (*rmgpy.pdep.Configuration method*), 152
`isUnimolecular()` (*rmgpy.reaction.Reaction method*), 184
`isUnimolecular()` (*rmgpy.rmg.pdep.PDepReaction method*), 199
`isVertexInCycle()` (*rmgpy.molecule.graph.Graph method*), 103
`isVertexInCycle()` (*rmgpy.molecule.Group method*), 132
`isVertexInCycle()` (*rmgpy.molecule.Molecule method*), 121

J

`jacobian()` (*rmgpy.solver.LiquidReactor method*), 220
`jacobian()` (*rmgpy.solver.SimpleReactor method*), 213

`jacobianMatrix` (*rmgpy.solver.LiquidReactor attribute*), 220
`jacobianMatrix` (*rmgpy.solver.ReactionSystem attribute*), 207
`jacobianMatrix` (*rmgpy.solver.SimpleReactor attribute*), 213

K

`k_effective_cache` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22
`k_effective_cache` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`k_effective_cache` (*rmgpy.data.kinetics.TemplateReaction attribute*), 65
`k_effective_cache` (*rmgpy.reaction.Reaction attribute*), 184
`k_effective_cache` (*rmgpy.rmg.pdep.PDepReaction attribute*), 199
`kb` (*rmgpy.solver.LiquidReactor attribute*), 220
`kb` (*rmgpy.solver.ReactionSystem attribute*), 207
`kb` (*rmgpy.solver.SimpleReactor attribute*), 213
`kdata` (*rmgpy.kinetics.KineticsData attribute*), 78
`kdata` (*rmgpy.kinetics.PDepKineticsData attribute*), 82
`KekulizationError`, 256
`kekulize()` (*in module rmgpy.molecule.kekulize*), 139
`kekulize()` (*rmgpy.molecule.kekulize.AromaticRing method*), 139
`kekulize()` (*rmgpy.molecule.Molecule method*), 121
`Keq` (*rmgpy.solver.LiquidReactor attribute*), 217
`Keq` (*rmgpy.solver.ReactionSystem attribute*), 205
`Keq` (*rmgpy.solver.SimpleReactor attribute*), 210
`keywords` (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 165
`keywords` (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 166
`keywords` (*rmgpy.qm.mopac.MopacMol attribute*), 169
`kf` (*rmgpy.solver.LiquidReactor attribute*), 220
`kf` (*rmgpy.solver.ReactionSystem attribute*), 207
`kf` (*rmgpy.solver.SimpleReactor attribute*), 213
`kinetics` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22
`kinetics` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`kinetics` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`kinetics` (*rmgpy.reaction.Reaction attribute*), 184
`kinetics` (*rmgpy.rmg.pdep.PDepReaction attribute*), 199
`KineticsData` (*class in rmgpy.kinetics*), 77
`KineticsDatabase` (*class in rmgpy.data.kinetics*), 24
`KineticsDepository` (*class in rmgpy.data.kinetics*), 26
`KineticsError`, 256
`KineticsFamily` (*class in rmgpy.data.kinetics*), 28
`KineticsGroups` (*class in rmgpy.data.kinetics*), 36
`KineticsJob` (*class in arkane*), 6
`KineticsLibrary` (*class in rmgpy.data.kinetics*), 38
`KineticsRules` (*class in rmgpy.data.kinetics*), 40
`KRotor` (*class in rmgpy.statmech*), 235
`kunits` (*rmgpy.kinetics.Chebyshev attribute*), 88

L

`label` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22
`label` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`label` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`label` (*rmgpy.molecule.Atom attribute*), 112
`label` (*rmgpy.molecule.AtomType attribute*), 107
`label` (*rmgpy.molecule.GroupAtom attribute*), 125
`label` (*rmgpy.reaction.Reaction attribute*), 184
`label` (*rmgpy.rmg.model.Species attribute*), 203
`label` (*rmgpy.rmg.pdep.PDepNetwork attribute*), 194
`label` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
`label` (*rmgpy.species.Species attribute*), 227
`label` (*rmgpy.species.TransitionState attribute*), 229
`label` (*rmgpy.thermo.NASA attribute*), 251
`label` (*rmgpy.thermo.NASAPolynomial attribute*), 255
`label` (*rmgpy.thermo.ThermoData attribute*), 246
`label` (*rmgpy.thermo.Wilhoit attribute*), 249
`LibraryReaction` (*class in rmgpy.data.kinetics*), 43
`Lindemann` (*class in rmgpy.kinetics*), 90
`LinearRotor` (*class in rmgpy.statmech*), 232
`LiquidReactor` (*class in rmgpy.solver*), 217
`load()` (*arkane.StatMechJob method*), 10
`load()` (*rmgpy.data.base.Database method*), 17
`load()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`load()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`load()` (*rmgpy.data.kinetics.KineticsGroups method*), 36
`load()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`load()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`load()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`load()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`load()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`load()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`load()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`load()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`load()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`loadChemkinFile()` (*in module rmgpy.chemkin*), 12

`loadConformer()` (*arkane.gaussian.GaussianLog method*), 4
`loadConformer()` (*arkane.molpro.MolproLog method*), 6
`loadConformer()` (*arkane.qchem.QChemLog method*), 5
`loadDepository()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`loadDepository()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`loadEnergy()` (*arkane.gaussian.GaussianLog method*), 4
`loadEnergy()` (*arkane.molpro.MolproLog method*), 6
`loadEnergy()` (*arkane.qchem.QChemLog method*), 5
`loadEntry()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`loadFamilies()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`loadForbidden()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`loadForceConstantMatrix()` (*arkane.gaussian.GaussianLog method*), 4
`loadForceConstantMatrix()` (*arkane.molpro.MolproLog method*), 6
`loadForceConstantMatrix()` (*arkane.qchem.QChemLog method*), 5
`loadGeometry()` (*arkane.gaussian.GaussianLog method*), 5
`loadGeometry()` (*arkane.molpro.MolproLog method*), 6
`loadGeometry()` (*arkane.qchem.QChemLog method*), 5
`loadGroups()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`loadGroups()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`loadInput()` (*rmgpy.rmg.main.RMG method*), 192
`loadInputFile()` (*arkane.Arkane method*), 7
`loadInputFile()` (*in module arkane.input*), 6
`loadLibraries()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`loadLibraries()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`loadLibraries()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`loadNegativeFrequency()` (*arkane.gaussian.GaussianLog method*), 5
`loadNegativeFrequency()` (*arkane.molpro.MolproLog method*), 6
`loadNegativeFrequency()` (*arkane.qchem.QChemLog method*), 5
`loadOld()` (*rmgpy.data.base.Database method*), 17
`loadOld()` (*rmgpy.data.kinetics.KineticsDatabase method*), 25
`loadOld()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`loadOld()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`loadOld()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`loadOld()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`loadOld()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`loadOld()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`loadOld()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`loadOld()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`loadOld()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`loadOld()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`loadOld()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`loadOld()` (*rmgpy.data.thermo.ThermoGroups method*), 72
`loadOld()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`loadOldDictionary()` (*rmgpy.data.base.Database method*), 17
`loadOldDictionary()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`loadOldDictionary()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`loadOldDictionary()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`loadOldDictionary()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`loadOldDictionary()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`loadOldDictionary()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`loadOldDictionary()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`loadOldDictionary()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`loadOldDictionary()`

(*rmgpy.data.thermo.ThermoDepository method*), 70
`loadOldDictionary()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`loadOldDictionary()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`loadOldLibrary()` (*rmgpy.data.base.Database method*), 17
`loadOldLibrary()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`loadOldLibrary()` (*rmgpy.data.kinetics.KineticsFamily method*), 33
`loadOldLibrary()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`loadOldLibrary()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`loadOldLibrary()` (*rmgpy.data.kinetics.KineticsRules method*), 41
`loadOldLibrary()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`loadOldLibrary()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`loadOldLibrary()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`loadOldLibrary()` (*rmgpy.data.thermo.ThermoDepository method*), 70
`loadOldLibrary()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`loadOldLibrary()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`loadOldTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`loadOldTree()` (*rmgpy.data.base.Database method*), 17
`loadOldTree()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`loadOldTree()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`loadOldTree()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`loadOldTree()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`loadOldTree()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`loadOldTree()` (*rmgpy.data.statmech.StatmechDepository method*), 51
`loadOldTree()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`loadOldTree()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`loadOldTree()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`loadOldTree()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`loadOldTree()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`loadRecipe()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`loadRecommendedFamiliesList()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`loadRestartFile()` (*rmgpy.rmg.main.RMG method*), 192
`loadRMGJavaInput()` (*rmgpy.rmg.main.RMG method*), 192
`loadScanEnergies()` (*arkane.gaussian.GaussianLog method*), 5
`loadScanEnergies()` (*arkane.qchem.QChemLog method*), 5
`loadSpeciesDictionary()` (*in module rmgpy.chemkin*), 12
`loadTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`loadThermoData()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`loadThermoData()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
`loadThermoData()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 166
`loadThermoData()` (*rmgpy.qm.molecule.QMMolecule method*), 160
`loadThermoData()` (*rmgpy.qm.mopac.MopacMol method*), 169
`loadThermoData()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`loadThermoData()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
`loadThermoData()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 173
`loadThermoInput()` (*rmgpy.rmg.main.RMG method*), 192
`loadTransportFile()` (*in module rmgpy.chemkin*), 12
`loadZeroPointEnergy()` (*arkane.gaussian.GaussianLog method*), 5
`loadZeroPointEnergy()` (*arkane.molpro.MolproLog method*), 6
`loadZeroPointEnergy()` (*arkane.qchem.QChemLog method*), 5
`logConversions()` (*rmgpy.solver.LiquidReactor method*), 220
`logConversions()` (*rmgpy.solver.ReactionSystem method*), 207
`logConversions()` (*rmgpy.solver.SimpleReactor method*), 213
`logFooter()` (*arkane.Arkane method*), 7

logHeader() (*arkane.Arkane method*), 7
logHeader() (*rmgpy.rmg.main.RMG method*), 192
LogicAnd (*class in rmgpy.data.base*), 48
LogicNode (*class in rmgpy.data.base*), 48
LogicOr (*class in rmgpy.data.base*), 48
logRates() (*rmgpy.solver.LiquidReactor method*), 220
logRates() (*rmgpy.solver.ReactionSystem method*), 207
logRates() (*rmgpy.solver.SimpleReactor method*), 214
lonePairs (*rmgpy.molecule.Atom attribute*), 112
lonePairs (*rmgpy.molecule.AtomType attribute*), 107
lonePairs (*rmgpy.molecule.GroupAtom attribute*), 125

M

makeBond() (*rmgpy.molecule.GroupBond method*), 127
makeLogicNode() (*in module rmgpy.data.base*), 48
makeNewPDepReaction() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
makeNewReaction() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
makeNewSpecies() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 188
makeProfileGraph() (*in module rmgpy.rmg.main*), 193
makeSampleAtom() (*rmgpy.molecule.GroupAtom method*), 126
makeSampleMolecule() (*rmgpy.molecule.Group method*), 132
makeSeedMech() (*rmgpy.rmg.main.RMG method*), 192
makeSpeciesLabelsIndependent() (*rmgpy.rmg.main.RMG method*), 192
mapDensitiesOfStates() (*rmgpy.pdep.Network method*), 154
mapDensitiesOfStates() (*rmgpy.rmg.pdep.PDepNetwork method*), 195
mapDensityOfStates() (*rmgpy.pdep.Configuration method*), 152
mapping (*rmgpy.molecule.Atom attribute*), 112
mapping (*rmgpy.molecule.graph.Vertex attribute*), 99
mapping (*rmgpy.molecule.GroupAtom attribute*), 126
mapSumOfStates() (*rmgpy.pdep.Configuration method*), 152
markChemkinDuplicates() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
markDuplicateReactions() (*in module rmgpy.chemkin*), 13
markValidDuplicates() (*rmgpy.data.kinetics.KineticsLibrary method*), 39
mass (*rmgpy.molecule.Element attribute*), 104
mass (*rmgpy.statmech.Conformer attribute*), 244
mass (*rmgpy.statmech.IdealGasTranslation attribute*), 232
matchesSpecies() (*rmgpy.data.kinetics.DepositoryReaction method*), 22
matchesSpecies() (*rmgpy.data.kinetics.LibraryReaction method*), 47
matchesSpecies() (*rmgpy.data.kinetics.TemplateReaction method*), 66
matchesSpecies() (*rmgpy.reaction.Reaction method*), 184
matchesSpecies() (*rmgpy.rmg.pdep.PDepReaction method*), 200
matchLogicOr() (*rmgpy.data.base.LogicOr method*), 48
matchNodeToChild() (*rmgpy.data.base.Database method*), 17
matchNodeToChild() (*rmgpy.data.kinetics.KineticsDepository method*), 27
matchNodeToChild() (*rmgpy.data.kinetics.KineticsFamily method*), 34
matchNodeToChild() (*rmgpy.data.kinetics.KineticsGroups method*), 37
matchNodeToChild() (*rmgpy.data.kinetics.KineticsLibrary method*), 39
matchNodeToChild() (*rmgpy.data.kinetics.KineticsRules method*), 42
matchNodeToChild() (*rmgpy.data.statmech.StatmechDepository method*), 51
matchNodeToChild() (*rmgpy.data.statmech.StatmechGroups method*), 58
matchNodeToChild() (*rmgpy.data.statmech.StatmechLibrary method*), 60
matchNodeToChild() (*rmgpy.data.thermo.ThermoDepository method*), 71
matchNodeToChild() (*rmgpy.data.thermo.ThermoGroups method*), 73
matchNodeToChild() (*rmgpy.data.thermo.ThermoLibrary method*), 75
matchNodeToNode() (*rmgpy.data.base.Database method*), 17
matchNodeToNode() (*rmgpy.data.kinetics.KineticsDepository method*), 27
matchNodeToNode() (*rmgpy.data.kinetics.KineticsFamily method*), 34
matchNodeToNode() (*rmgpy.data.kinetics.KineticsGroups method*), 37
matchNodeToNode() (*rmgpy.data.kinetics.KineticsLibrary method*), 39
matchNodeToNode() (*rmgpy.data.kinetics.KineticsRules method*), 42
matchNodeToNode() (*rmgpy.data.statmech.StatmechDepository method*), 52
matchNodeToNode() (*rmgpy.data.statmech.StatmechGroups method*), 58

`matchNodeToNode()` (*rmgpy.data.statmech.StatmechLibrary method*), 60
`matchNodeToNode()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`matchNodeToNode()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`matchNodeToNode()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`matchNodeToStructure()` (*rmgpy.data.base.Database method*), 17
`matchNodeToStructure()` (*rmgpy.data.kinetics.KineticsDepository method*), 27
`matchNodeToStructure()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`matchNodeToStructure()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`matchNodeToStructure()` (*rmgpy.data.kinetics.KineticsLibrary method*), 39
`matchNodeToStructure()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`matchNodeToStructure()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`matchNodeToStructure()` (*rmgpy.data.statmech.StatmechGroups method*), 58
`matchNodeToStructure()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`matchNodeToStructure()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`matchNodeToStructure()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`matchNodeToStructure()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`matchToStructure()` (*rmgpy.data.base.LogicAnd method*), 48
`matchToStructure()` (*rmgpy.data.base.LogicOr method*), 48
`maxAttempts` (*rmgpy.qm.gaussian.GaussianMol attribute*), 163
`maxAttempts` (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 165
`maxAttempts` (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 166
`maxAttempts` (*rmgpy.qm.molecule.QMMolecule attribute*), 160
`maxAttempts` (*rmgpy.qm.mopac.MopacMol attribute*), 169
`maxAttempts` (*rmgpy.qm.mopac.MopacMolPM3 attribute*), 170
`maxAttempts` (*rmgpy.qm.mopac.MopacMolPM6 attribute*), 172
`maxAttempts` (*rmgpy.qm.mopac.MopacMolPM7 attribute*), 173
`maxEdgeSpeciesRateRatios` (*rmgpy.solver.LiquidReactor attribute*), 221
`maxEdgeSpeciesRateRatios` (*rmgpy.solver.ReactionSystem attribute*), 207
`maxEdgeSpeciesRateRatios` (*rmgpy.solver.SimpleReactor attribute*), 214
`maximumGrainSize` (*arkane.PressureDependenceJob attribute*), 10
`maxNetworkLeakRateRatios` (*rmgpy.solver.LiquidReactor attribute*), 221
`maxNetworkLeakRateRatios` (*rmgpy.solver.ReactionSystem attribute*), 207
`maxNetworkLeakRateRatios` (*rmgpy.solver.SimpleReactor attribute*), 214
`merge()` (*rmgpy.molecule.graph.Graph method*), 103
`merge()` (*rmgpy.molecule.Group method*), 132
`merge()` (*rmgpy.molecule.Molecule method*), 121
`merge()` (*rmgpy.rmg.model.ReactionModel method*), 190
`merge()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`mergeGroups()` (*rmgpy.molecule.Group method*), 132
`modes` (*rmgpy.statmech.Conformer attribute*), 244
`ModifiedStrongCollisionError`, 256
`molecularWeight` (*rmgpy.rmg.model.Species attribute*), 204
`molecularWeight` (*rmgpy.species.Species attribute*), 227
`Molecule` (*class in rmgpy.molecule*), 114
`molecule` (*rmgpy.rmg.model.Species attribute*), 204
`molecule` (*rmgpy.species.Species attribute*), 227
`MoleculeDrawer` (*class in rmgpy.molecule.draw*), 146
`MolproLog` (*class in arkane.molpro*), 6
`Mopac` (*class in rmgpy.qm.mopac*), 167
`MopacMol` (*class in rmgpy.qm.mopac*), 168
`MopacMolPM3` (*class in rmgpy.qm.mopac*), 169
`MopacMolPM6` (*class in rmgpy.qm.mopac*), 171
`MopacMolPM7` (*class in rmgpy.qm.mopac*), 172
`MultiArrhenius` (*class in rmgpy.kinetics*), 80
`MultiPDepArrhenius` (*class in rmgpy.kinetics*), 85
`multiplicity` (*rmgpy.molecule.Group attribute*), 132
`multiplicity` (*rmgpy.molecule.Molecule attribute*), 121

N

`n` (*rmgpy.kinetics.Arrhenius attribute*), 79

`n` (*rmgpy.pdep.SingleExponentialDown attribute*), 149

`name` (*rmgpy.molecule.Element attribute*), 104

`NASA` (*class in rmgpy.thermo*), 250

`NASAPolynomial` (*class in rmgpy.thermo*), 253

`NegativeBarrierException`, 256

`neq` (*rmgpy.solver.LiquidReactor attribute*), 221

`neq` (*rmgpy.solver.ReactionSystem attribute*), 207

`neq` (*rmgpy.solver.SimpleReactor attribute*), 214

`Network` (*class in rmgpy.pdep*), 152

`network_kinetics` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22

`network_kinetics` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47

`network_kinetics` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66

`network_kinetics` (*rmgpy.reaction.Reaction attribute*), 184

`network_kinetics` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200

`NetworkError`, 257

`networkIndices` (*rmgpy.solver.LiquidReactor attribute*), 221

`networkIndices` (*rmgpy.solver.ReactionSystem attribute*), 207

`networkIndices` (*rmgpy.solver.SimpleReactor attribute*), 214

`networkLeakCoefficients` (*rmgpy.solver.LiquidReactor attribute*), 221

`networkLeakCoefficients` (*rmgpy.solver.ReactionSystem attribute*), 207

`networkLeakCoefficients` (*rmgpy.solver.SimpleReactor attribute*), 214

`networkLeakRates` (*rmgpy.solver.LiquidReactor attribute*), 221

`networkLeakRates` (*rmgpy.solver.ReactionSystem attribute*), 207

`networkLeakRates` (*rmgpy.solver.SimpleReactor attribute*), 214

`NonlinearRotor` (*class in rmgpy.statmech*), 234

`nSims` (*rmgpy.solver.LiquidReactor attribute*), 221

`nSims` (*rmgpy.solver.SimpleReactor attribute*), 214

`number` (*rmgpy.molecule.Element attribute*), 104

`number` (*rmgpy.statmech.Conformer attribute*), 245

`numberOfAtoms` (*rmgpy.qm.qmdata.QMData attribute*), 160

`numCoreReactions` (*rmgpy.solver.LiquidReactor attribute*), 221

`numCoreReactions` (*rmgpy.solver.ReactionSystem attribute*), 208

`numCoreReactions` (*rmgpy.solver.SimpleReactor attribute*), 214

`numCoreSpecies` (*rmgpy.solver.LiquidReactor attribute*), 221

`numCoreSpecies` (*rmgpy.solver.ReactionSystem attribute*), 208

`numCoreSpecies` (*rmgpy.solver.SimpleReactor attribute*), 214

`numEdgeReactions` (*rmgpy.solver.LiquidReactor attribute*), 221

`numEdgeReactions` (*rmgpy.solver.ReactionSystem attribute*), 208

`numEdgeReactions` (*rmgpy.solver.SimpleReactor attribute*), 214

`numEdgeSpecies` (*rmgpy.solver.LiquidReactor attribute*), 221

`numEdgeSpecies` (*rmgpy.solver.ReactionSystem attribute*), 208

`numEdgeSpecies` (*rmgpy.solver.SimpleReactor attribute*), 214

`numPdepNetworks` (*rmgpy.solver.LiquidReactor attribute*), 221

`numPdepNetworks` (*rmgpy.solver.ReactionSystem attribute*), 208

`numPdepNetworks` (*rmgpy.solver.SimpleReactor attribute*), 214

O

`oDouble` (*rmgpy.molecule.AtomType attribute*), 107

`opticalIsomers` (*rmgpy.statmech.Conformer attribute*), 245

`order` (*rmgpy.molecule.Bond attribute*), 114

`order` (*rmgpy.molecule.GroupBond attribute*), 127

`ordered_vertices` (*rmgpy.molecule.graph.Graph attribute*), 103

`ordered_vertices` (*rmgpy.molecule.Group attribute*), 132

`ordered_vertices` (*rmgpy.molecule.Molecule attribute*), 121

`OutputError`, 257

`outputFilePath` (*rmgpy.qm.gaussian.GaussianMol attribute*), 163

`outputFilePath` (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 165

`outputFilePath` (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 167

`outputFilePath` (*rmgpy.qm.molecule.QMMolecule attribute*), 160

`outputFilePath` (*rmgpy.qm.mopac.MopacMol attribute*), 169

`outputFilePath` (*rmgpy.qm.mopac.MopacMolPM3 attribute*), 170

`outputFilePath` (*rmgpy.qm.mopac.MopacMolPM6 attribute*), 172

`outputFilePath` (*rmgpy.qm.mopac.MopacMolPM7 attribute*), 174

P

`P` (*rmgpy.solver.LiquidReactor attribute*), 217
`P` (*rmgpy.solver.SimpleReactor attribute*), 210
`pairs` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 22
`pairs` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`pairs` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`pairs` (*rmgpy.reaction.Reaction attribute*), 184
`pairs` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
`parse()` (*rmgpy.qm.gaussian.Gaussian method*), 162
`parse()` (*rmgpy.qm.gaussian.GaussianMol method*), 163
`parse()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
`parse()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 167
`parse()` (*rmgpy.qm.molecule.QMMolecule method*), 160
`parse()` (*rmgpy.qm.mopac.MopacMol method*), 169
`parse()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
`parse()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
`parse()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 174
`parse()` (*rmgpy.qm.symmetry.SymmetryJob method*), 161
`parseCommandLineArguments()` (*arkane.Arkane method*), 7
`parseOldLibrary()` (*rmgpy.data.base.Database method*), 18
`parseOldLibrary()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`parseOldLibrary()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`parseOldLibrary()` (*rmgpy.data.kinetics.KineticsGroups method*), 37
`parseOldLibrary()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`parseOldLibrary()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`parseOldLibrary()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`parseOldLibrary()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`parseOldLibrary()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`parseOldLibrary()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`parseOldLibrary()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`parseOldLibrary()` (*rmgpy.data.thermo.ThermoLibrary method*), 75
`Pdata` (*rmgpy.kinetics.PDepKineticsData attribute*), 81
`PDepArrhenius` (*class in rmgpy.kinetics*), 83
`pdepColliderKinetics` (*rmgpy.solver.SimpleReactor attribute*), 214
`pdepColliderReactionIndices` (*rmgpy.solver.SimpleReactor attribute*), 215
`PDepKineticsData` (*class in rmgpy.kinetics*), 81
`PDepNetwork` (*class in rmgpy.rmg.pdep*), 193
`PDepReaction` (*class in rmgpy.rmg.pdep*), 196
`pdepSpecificColliderKinetics` (*rmgpy.solver.SimpleReactor attribute*), 215
`pdepSpecificColliderReactionIndices` (*rmgpy.solver.SimpleReactor attribute*), 215
`pickWildcards()` (*rmgpy.molecule.Group method*), 133
`Plist` (*arkane.PressureDependenceJob attribute*), 9
`plot()` (*arkane.KineticsJob method*), 7
`plot()` (*arkane.ThermoJob method*), 11
`plotHinderedRotor()` (*arkane.StatMechJob method*), 10
`Pmax` (*arkane.PressureDependenceJob attribute*), 9
`Pmax` (*rmgpy.kinetics.Arrhenius attribute*), 79
`Pmax` (*rmgpy.kinetics.Chebyshev attribute*), 87
`Pmax` (*rmgpy.kinetics.KineticsData attribute*), 77
`Pmax` (*rmgpy.kinetics.Lindemann attribute*), 91
`Pmax` (*rmgpy.kinetics.MultiArrhenius attribute*), 80
`Pmax` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
`Pmax` (*rmgpy.kinetics.PDepArrhenius attribute*), 83
`Pmax` (*rmgpy.kinetics.PDepKineticsData attribute*), 81
`Pmax` (*rmgpy.kinetics.ThirdBody attribute*), 89
`Pmax` (*rmgpy.kinetics.Troe attribute*), 93
`Pmin` (*arkane.PressureDependenceJob attribute*), 9
`Pmin` (*rmgpy.kinetics.Arrhenius attribute*), 79
`Pmin` (*rmgpy.kinetics.Chebyshev attribute*), 87
`Pmin` (*rmgpy.kinetics.KineticsData attribute*), 77
`Pmin` (*rmgpy.kinetics.Lindemann attribute*), 91
`Pmin` (*rmgpy.kinetics.MultiArrhenius attribute*), 80
`Pmin` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
`Pmin` (*rmgpy.kinetics.PDepArrhenius attribute*), 83
`Pmin` (*rmgpy.kinetics.PDepKineticsData attribute*), 81
`Pmin` (*rmgpy.kinetics.ThirdBody attribute*), 89
`Pmin` (*rmgpy.kinetics.Troe attribute*), 93
`PointGroup` (*class in rmgpy.qm.symmetry*), 161
`PointGroupCalculator` (*class in rmgpy.qm.symmetry*), 161
`poly1` (*rmgpy.thermo.NASA attribute*), 252
`poly2` (*rmgpy.thermo.NASA attribute*), 252
`poly3` (*rmgpy.thermo.NASA attribute*), 252
`polynomials` (*rmgpy.thermo.NASA attribute*), 252
`populate_resonance_algorithms` (*in module rmgpy.molecule.resonance*), 137
`Prange` (*rmgpy.solver.SimpleReactor attribute*), 210
`prepareTreeForGeneration()`

(*rmgpy.data.kinetics.KineticsFamily method*), 34
PressureDependenceError, 257
PressureDependenceJob (*class in arkane*), 8
pressures (*rmgpy.kinetics.PDepArrhenius attribute*), 84
prettify() (*in module arkane.output*), 8
PrettifyVisitor (*class in arkane.output*), 8
printEnlargeSummary() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
printSummary() (*rmgpy.pdep.Network method*), 154
printSummary() (*rmgpy.rmg.pdep.PDepNetwork method*), 195
prioritizeThermo() (*rmgpy.data.thermo.ThermoDatabase method*), 69
process_bonds() (*rmgpy.molecule.kekulize.AromaticRing method*), 139
processNewReactions() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
processOldLibraryEntry() (*rmgpy.data.kinetics.KineticsRules method*), 42
processOldLibraryEntry() (*rmgpy.data.statmech.StatmechGroups method*), 59
processOldLibraryEntry() (*rmgpy.data.statmech.StatmechLibrary method*), 61
processOldLibraryEntry() (*rmgpy.data.thermo.ThermoGroups method*), 73
processOldLibraryEntry() (*rmgpy.data.thermo.ThermoLibrary method*), 76
processPdepNetworks() (*rmgpy.rmg.main.RMG method*), 192
processProfileStats() (*in module rmgpy.rmg.main*), 193
processReactionsToSpecies() (*rmgpy.rmg.main.RMG method*), 192
processToSpeciesNetworks() (*rmgpy.rmg.main.RMG method*), 192
productIndices (*rmgpy.solver.LiquidReactor attribute*), 221
productIndices (*rmgpy.solver.ReactionSystem attribute*), 208
productIndices (*rmgpy.solver.SimpleReactor attribute*), 215
products (*rmgpy.data.kinetics.DepositoryReaction attribute*), 23
products (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
products (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
products (*rmgpy.reaction.Reaction attribute*), 184
products (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
props (*rmgpy.molecule.Atom attribute*), 112
props (*rmgpy.molecule.Group attribute*), 133
props (*rmgpy.molecule.GroupAtom attribute*), 126
props (*rmgpy.molecule.Molecule attribute*), 121
props (*rmgpy.rmg.model.Species attribute*), 204
props (*rmgpy.species.Species attribute*), 227
prunableNetworkIndices (*rmgpy.solver.LiquidReactor attribute*), 221
prunableNetworkIndices (*rmgpy.solver.ReactionSystem attribute*), 208
prunableNetworkIndices (*rmgpy.solver.SimpleReactor attribute*), 215
prunableNetworks (*rmgpy.solver.LiquidReactor attribute*), 222
prunableNetworks (*rmgpy.solver.ReactionSystem attribute*), 208
prunableNetworks (*rmgpy.solver.SimpleReactor attribute*), 215
prunableSpecies (*rmgpy.solver.LiquidReactor attribute*), 222
prunableSpecies (*rmgpy.solver.ReactionSystem attribute*), 208
prunableSpecies (*rmgpy.solver.SimpleReactor attribute*), 215
prunableSpeciesIndices (*rmgpy.solver.LiquidReactor attribute*), 222
prunableSpeciesIndices (*rmgpy.solver.ReactionSystem attribute*), 208
prunableSpeciesIndices (*rmgpy.solver.SimpleReactor attribute*), 215
prune() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
pruneHeteroatoms() (*rmgpy.data.thermo.ThermoDatabase method*), 69
PseudoFit (*class in rmgpy.data.statmechfit*), 56
PseudoRotorFit (*class in rmgpy.data.statmechfit*), 55

Q

QChemLog (*class in arkane.qchem*), 5
QMCalculator (*class in rmgpy.qm.main*), 157
QMData (*class in rmgpy.qm.qmdata*), 160
QMMolecule (*class in rmgpy.qm.molecule*), 159
QMSettings (*class in rmgpy.qm.main*), 157
QMVerifier (*class in rmgpy.qm.qmverifier*), 160
Quantity() (*in module rmgpy.quantity*), 178
QuantityError, 257
quantum (*rmgpy.statmech.HarmonicOscillator attribute*), 239

`quantum` (*rmgpy.statmech.HinderedRotor attribute*), 241
`quantum` (*rmgpy.statmech.IdealGasTranslation attribute*), 232
`quantum` (*rmgpy.statmech.KRotor attribute*), 236
`quantum` (*rmgpy.statmech.LinearRotor attribute*), 233
`quantum` (*rmgpy.statmech.NonlinearRotor attribute*), 235
`quantum` (*rmgpy.statmech.SphericalTopRotor attribute*), 237

R

`radicalCount` (*rmgpy.molecule.Group attribute*), 133
`radicalElectrons` (*rmgpy.molecule.Atom attribute*), 112
`radicalElectrons` (*rmgpy.molecule.GroupAtom attribute*), 126
`rd_build()` (*rmgpy.qm.molecule.Geometry method*), 158
`rd_embed()` (*rmgpy.qm.molecule.Geometry method*), 158
`rdMol` (*rmgpy.molecule.Molecule attribute*), 122
`rdMolConfId` (*rmgpy.molecule.Molecule attribute*), 122
`rDouble` (*rmgpy.molecule.AtomType attribute*), 107
`react_molecules()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`reactantIndices` (*rmgpy.solver.LiquidReactor attribute*), 222
`reactantIndices` (*rmgpy.solver.ReactionSystem attribute*), 208
`reactantIndices` (*rmgpy.solver.SimpleReactor attribute*), 215
`reactants` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 23
`reactants` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`reactants` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`reactants` (*rmgpy.reaction.Reaction attribute*), 184
`reactants` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
`Reaction` (*class in rmgpy.reaction*), 179
`ReactionDrawer` (*class in rmgpy.molecule.draw*), 147
`ReactionError`, 257
`reactionIndex` (*rmgpy.solver.LiquidReactor attribute*), 222
`reactionIndex` (*rmgpy.solver.ReactionSystem attribute*), 208
`reactionIndex` (*rmgpy.solver.SimpleReactor attribute*), 215
`ReactionModel` (*class in rmgpy.rmg.model*), 190
`ReactionPairsError`, 257
`ReactionRecipe` (*class in rmgpy.data.kinetics*), 49
`ReactionSystem` (*class in rmgpy.solver*), 205
`reactive` (*rmgpy.molecule.Molecule attribute*), 122
`reactive` (*rmgpy.rmg.model.Species attribute*), 204
`reactive` (*rmgpy.species.Species attribute*), 227
`readInputFile()` (*in module rmgpy.rmg.input*), 190
`readKineticsEntry()` (*in module rmgpy.chemkin*), 12
`readMeaningfulLineJava()` (*rmgpy.rmg.main.RMG method*), 192
`readReactionComments()` (*in module rmgpy.chemkin*), 12
`readReactionsBlock()` (*in module rmgpy.chemkin*), 12
`readThermoEntry()` (*in module rmgpy.chemkin*), 12
`reconstructKineticsFromSource()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`recordPolycyclicGenericNodes()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`recordRingGenericNodes()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`reg_dim` (*rmgpy.molecule.GroupBond attribute*), 127
`reg_dim_atm` (*rmgpy.molecule.GroupAtom attribute*), 126
`reg_dim_u` (*rmgpy.molecule.GroupAtom attribute*), 126
`register_listeners()` (*rmgpy.rmg.main.RMG method*), 192
`registerReaction()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
`regularize()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`remove_disconnected_reactions()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`remove_H_bonds()` (*rmgpy.molecule.Molecule method*), 122
`remove_reactions()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`removeAtom()` (*rmgpy.molecule.Group method*), 133
`removeAtom()` (*rmgpy.molecule.Molecule method*), 122
`removeBond()` (*rmgpy.molecule.Group method*), 133
`removeBond()` (*rmgpy.molecule.Molecule method*), 122
`removeCommentFromLine()` (*in module rmgpy.chemkin*), 12
`removeEdge()` (*rmgpy.molecule.graph.Graph method*), 103
`removeEdge()` (*rmgpy.molecule.Group method*), 133
`removeEdge()` (*rmgpy.molecule.Molecule method*), 122
`removeEmptyPdepNetworks()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
`removeGroup()` (*rmgpy.data.base.Database method*), 18
`removeGroup()` (*rmgpy.data.kinetics.KineticsDepository method*), 28

`removeGroup()` (*rmgpy.data.kinetics.KineticsFamily method*), 34
`removeGroup()` (*rmgpy.data.kinetics.KineticsGroups method*), 38
`removeGroup()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`removeGroup()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`removeGroup()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`removeGroup()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`removeGroup()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`removeGroup()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`removeGroup()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`removeGroup()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`removeSpeciesFromEdge()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
`removeVertex()` (*rmgpy.molecule.graph.Graph method*), 103
`removeVertex()` (*rmgpy.molecule.Group method*), 133
`removeVertex()` (*rmgpy.molecule.Molecule method*), 122
`render()` (*rmgpy.molecule.draw.MoleculeDrawer method*), 146
`ReservoirStateError`, 257
`reset_max_edge_species_rate_ratios()` (*rmgpy.solver.LiquidReactor method*), 222
`reset_max_edge_species_rate_ratios()` (*rmgpy.solver.ReactionSystem method*), 208
`reset_max_edge_species_rate_ratios()` (*rmgpy.solver.SimpleReactor method*), 215
`resetConnectivityValues()` (*rmgpy.molecule.Atom method*), 113
`resetConnectivityValues()` (*rmgpy.molecule.graph.Graph method*), 103
`resetConnectivityValues()` (*rmgpy.molecule.graph.Vertex method*), 100
`resetConnectivityValues()` (*rmgpy.molecule.Group method*), 133
`resetConnectivityValues()` (*rmgpy.molecule.GroupAtom method*), 126
`resetConnectivityValues()` (*rmgpy.molecule.Molecule method*), 122
`resetRingMembership()` (*rmgpy.molecule.Group method*), 133
`residual()` (*rmgpy.solver.LiquidReactor method*), 222
`residual()` (*rmgpy.solver.ReactionSystem method*), 208
`residual()` (*rmgpy.solver.SimpleReactor method*), 215
`resolved` (*rmgpy.molecule.kekulize.AromaticRing attribute*), 139
`ResonanceError`, 257
`restore_vertex_order()` (*rmgpy.molecule.graph.Graph method*), 103
`restore_vertex_order()` (*rmgpy.molecule.Group method*), 133
`restore_vertex_order()` (*rmgpy.molecule.Molecule method*), 122
`retrieve()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
`retrieveNewSpecies()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
`retrieveOriginalEntry()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`retrieveTemplate()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`reverseThisArrheniusRate()` (*rmgpy.data.kinetics.DepositoryReaction method*), 23
`reverseThisArrheniusRate()` (*rmgpy.data.kinetics.LibraryReaction method*), 47
`reverseThisArrheniusRate()` (*rmgpy.data.kinetics.TemplateReaction method*), 66
`reverseThisArrheniusRate()` (*rmgpy.reaction.Reaction method*), 185
`reverseThisArrheniusRate()` (*rmgpy.rmg.pdep.PDepReaction method*), 200
`reversible` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 23
`reversible` (*rmgpy.data.kinetics.LibraryReaction attribute*), 47
`reversible` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`reversible` (*rmgpy.reaction.Reaction attribute*), 185
`reversible` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
`ring_bonds` (*rmgpy.molecule.kekulize.AromaticBond attribute*), 138
`RMG` (*class in rmgpy.rmg.main*), 190
`rmgpy.chemkin` (*module*), 11
`rmgpy.constants` (*module*), 14
`rmgpy.data` (*module*), 14
`rmgpy.exceptions` (*module*), 255
`rmgpy.kinetics` (*module*), 76
`rmgpy.molecule` (*module*), 97

`rmgpy.molecule.adjlist` (*module*), 143
`rmgpy.molecule.converter` (*module*), 142
`rmgpy.molecule.kekulize` (*module*), 137
`rmgpy.molecule.pathfinder` (*module*), 139
`rmgpy.molecule.resonance` (*module*), 134
`rmgpy.molecule.translator` (*module*), 142
`rmgpy.pdep` (*module*), 147
`rmgpy.qm` (*module*), 156
`rmgpy.quantity` (*module*), 174
`rmgpy.reaction` (*module*), 179
`rmgpy.rmg` (*module*), 185
`rmgpy.solver` (*module*), 204
`rmgpy.species` (*module*), 224
`rmgpy.statmech` (*module*), 229
`rmgpy.statmech.schrodinger` (*module*), 241
`rmgpy.thermo` (*module*), 245
`rotationalConstant` (*rmgpy.statmech.HinderedRotor attribute*), 241
`rotationalConstant` (*rmgpy.statmech.KRotor attribute*), 236
`rotationalConstant` (*rmgpy.statmech.LinearRotor attribute*), 234
`rotationalConstant` (*rmgpy.statmech.NonlinearRotor attribute*), 235
`rotationalConstant` (*rmgpy.statmech.SphericalTopRotor attribute*), 237
`rtol_array` (*rmgpy.solver.LiquidReactor attribute*), 222
`rtol_array` (*rmgpy.solver.ReactionSystem attribute*), 209
`rtol_array` (*rmgpy.solver.SimpleReactor attribute*), 215
`run()` (*rmgpy.qm.symmetry.SymmetryJob method*), 161

S

`S0` (*rmgpy.thermo.Wilhoit attribute*), 248
`S298` (*rmgpy.thermo.ThermoData attribute*), 246
`saturate_radicals()` (*rmgpy.molecule.Molecule method*), 122
`saturate_unfilled_valence()` (*rmgpy.molecule.Molecule method*), 122
`save()` (*arkane.KineticsJob method*), 7
`save()` (*arkane.StatMechJob method*), 10
`save()` (*arkane.ThermoJob method*), 11
`save()` (*rmgpy.data.base.Database method*), 18
`save()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`save()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`save()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`save()` (*rmgpy.data.kinetics.KineticsGroups method*), 38
`save()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`save()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`save()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`save()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`save()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`save()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`save()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`save()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`save()` (*rmgpy.data.thermo.ThermoGroups method*), 73
`save()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`saveChemkinFile()` (*in module rmgpy.chemkin*), 12
`saveCoordinatesFromQMData()` (*rmgpy.qm.molecule.Geometry method*), 158
`saveDepository()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveDepository()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`saveDepository()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`saveDictionary()` (*rmgpy.data.base.Database method*), 18
`saveDictionary()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`saveDictionary()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveDictionary()` (*rmgpy.data.kinetics.KineticsGroups method*), 38
`saveDictionary()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`saveDictionary()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`saveDictionary()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`saveDictionary()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`saveDictionary()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`saveDictionary()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`saveDictionary()` (*rmgpy.data.thermo.ThermoGroups method*), 74
`saveDictionary()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`saveDiffHTML()` (*in module rmgpy.rmg.output*), 193
`saveEntry()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`saveEntry()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveEntry()` (*rmgpy.data.kinetics.KineticsLibrary*

method), 40
`saveEntry()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`saveEntry()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`saveEntry()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`saveEntry()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`saveEntry()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`saveEntry()` (*rmgpy.data.thermo.ThermoGroups method*), 74
`saveEntry()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`saveEverything()` (*rmgpy.rmg.main.RMG method*), 192
`saveFamilies()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`saveGeneratedTree()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveGroups()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveGroups()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`saveGroups()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`saveHTMLFile()` (*in module rmgpy.chemkin*), 13
`saveInput()` (*rmgpy.rmg.main.RMG method*), 192
`saveInputFile()` (*arkane.PressureDependenceJob method*), 10
`saveInputFile()` (*in module rmgpy.rmg.input*), 190
`saveJavaKineticsLibrary()` (*in module rmgpy.chemkin*), 13
`saveLibraries()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`saveLibraries()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`saveLibraries()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`saveOld()` (*rmgpy.data.base.Database method*), 18
`saveOld()` (*rmgpy.data.kinetics.KineticsDatabase method*), 26
`saveOld()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`saveOld()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveOld()` (*rmgpy.data.kinetics.KineticsGroups method*), 38
`saveOld()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`saveOld()` (*rmgpy.data.kinetics.KineticsRules method*), 42
`saveOld()` (*rmgpy.data.statmech.StatmechDatabase method*), 50
`saveOld()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`saveOld()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`saveOld()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`saveOld()` (*rmgpy.data.thermo.ThermoDatabase method*), 69
`saveOld()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`saveOld()` (*rmgpy.data.thermo.ThermoGroups method*), 74
`saveOld()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`saveOldDictionary()` (*rmgpy.data.base.Database method*), 18
`saveOldDictionary()` (*rmgpy.data.kinetics.KineticsDepository method*), 28
`saveOldDictionary()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`saveOldDictionary()` (*rmgpy.data.kinetics.KineticsGroups method*), 38
`saveOldDictionary()` (*rmgpy.data.kinetics.KineticsLibrary method*), 40
`saveOldDictionary()` (*rmgpy.data.kinetics.KineticsRules method*), 43
`saveOldDictionary()` (*rmgpy.data.statmech.StatmechDepository method*), 52
`saveOldDictionary()` (*rmgpy.data.statmech.StatmechGroups method*), 59
`saveOldDictionary()` (*rmgpy.data.statmech.StatmechLibrary method*), 61
`saveOldDictionary()` (*rmgpy.data.thermo.ThermoDepository method*), 71
`saveOldDictionary()` (*rmgpy.data.thermo.ThermoGroups method*), 74
`saveOldDictionary()` (*rmgpy.data.thermo.ThermoLibrary method*), 76
`saveOldLibrary()` (*rmgpy.data.base.Database method*), 18
`saveOldLibrary()` (*rmgpy.data.kinetics.KineticsDepository method*), 28

saveOldLibrary() (*rmgpy.data.kinetics.KineticsFamily method*), 35
saveOldLibrary() (*rmgpy.data.kinetics.KineticsGroups method*), 38
saveOldLibrary() (*rmgpy.data.kinetics.KineticsLibrary method*), 40
saveOldLibrary() (*rmgpy.data.kinetics.KineticsRules method*), 43
saveOldLibrary() (*rmgpy.data.statmech.StatmechDepository method*), 52
saveOldLibrary() (*rmgpy.data.statmech.StatmechGroups method*), 59
saveOldLibrary() (*rmgpy.data.statmech.StatmechLibrary method*), 61
saveOldLibrary() (*rmgpy.data.thermo.ThermoDepository method*), 71
saveOldLibrary() (*rmgpy.data.thermo.ThermoGroups method*), 74
saveOldLibrary() (*rmgpy.data.thermo.ThermoLibrary method*), 76
saveOldTemplate() (*rmgpy.data.kinetics.KineticsFamily method*), 35
saveOldTree() (*rmgpy.data.base.Database method*), 18
saveOldTree() (*rmgpy.data.kinetics.KineticsDepository method*), 28
saveOldTree() (*rmgpy.data.kinetics.KineticsFamily method*), 35
saveOldTree() (*rmgpy.data.kinetics.KineticsGroups method*), 38
saveOldTree() (*rmgpy.data.kinetics.KineticsLibrary method*), 40
saveOldTree() (*rmgpy.data.kinetics.KineticsRules method*), 43
saveOldTree() (*rmgpy.data.statmech.StatmechDepository method*), 52
saveOldTree() (*rmgpy.data.statmech.StatmechGroups method*), 59
saveOldTree() (*rmgpy.data.statmech.StatmechLibrary method*), 61
saveOldTree() (*rmgpy.data.thermo.ThermoDepository method*), 72
saveOldTree() (*rmgpy.data.thermo.ThermoGroups method*), 74
saveOldTree() (*rmgpy.data.thermo.ThermoLibrary method*), 76
saveOutputHTML() (*in module rmgpy.rmg.output*), 193
saveRecommendedFamilies() (*rmgpy.data.kinetics.KineticsDatabase method*), 26
saveSpeciesDictionary() (*in module rmgpy.chemkin*), 12
saveThermoData() (*rmgpy.qm.gaussian.GaussianMol method*), 163
saveThermoData() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
saveThermoData() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 167
saveThermoData() (*rmgpy.qm.molecule.QMMolecule method*), 160
saveThermoData() (*rmgpy.qm.mopac.MopacMol method*), 169
saveThermoData() (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
saveThermoData() (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
saveThermoData() (*rmgpy.qm.mopac.MopacMolPM7 method*), 174
saveTrainingReactions() (*rmgpy.data.kinetics.KineticsFamily method*), 35
saveTransportFile() (*in module rmgpy.chemkin*), 13
ScalarQuantity (*class in rmgpy.quantity*), 176
scriptAttempts (*rmgpy.qm.gaussian.GaussianMol attribute*), 163
scriptAttempts (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 165
scriptAttempts (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 167
scriptAttempts (*rmgpy.qm.molecule.QMMolecule attribute*), 160
scriptAttempts (*rmgpy.qm.mopac.MopacMol attribute*), 169
scriptAttempts (*rmgpy.qm.mopac.MopacMolPM3 attribute*), 170
scriptAttempts (*rmgpy.qm.mopac.MopacMolPM6 attribute*), 172
scriptAttempts (*rmgpy.qm.mopac.MopacMolPM7 attribute*), 174
sDouble (*rmgpy.molecule.AtomType attribute*), 107
searchRetrieveReactions() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
selectEnergyGrains() (*rmgpy.pdep.Network method*), 154
selectEnergyGrains() (*rmgpy.rmg.pdep.PDepNetwork method*), 195
selectPolynomial() (*rmgpy.thermo.NASA method*), 252
semiclassical (*rmgpy.statmech.HinderedRotor attribute*), 241
sensConditions (*rmgpy.solver.LiquidReactor attribute*), 222
sensConditions (*rmgpy.solver.SimpleReactor attribute*), 215
sensitiveSpecies (*rmgpy.solver.LiquidReactor attribute*), 222

sensitiveSpecies (*rmgpy.solver.ReactionSystem attribute*), 209
sensitiveSpecies (*rmgpy.solver.SimpleReactor attribute*), 216
sensitivityCoefficients (*rmgpy.solver.LiquidReactor attribute*), 222
sensitivityCoefficients (*rmgpy.solver.ReactionSystem attribute*), 209
sensitivityCoefficients (*rmgpy.solver.SimpleReactor attribute*), 216
sensitivityThreshold (*rmgpy.solver.LiquidReactor attribute*), 222
sensitivityThreshold (*rmgpy.solver.ReactionSystem attribute*), 209
sensitivityThreshold (*rmgpy.solver.SimpleReactor attribute*), 216
set_colliders() (*rmgpy.solver.SimpleReactor method*), 216
set_initial_conditions() (*rmgpy.solver.LiquidReactor method*), 222
set_initial_conditions() (*rmgpy.solver.ReactionSystem method*), 209
set_initial_conditions() (*rmgpy.solver.SimpleReactor method*), 216
set_initial_derivative() (*rmgpy.solver.LiquidReactor method*), 222
set_initial_derivative() (*rmgpy.solver.ReactionSystem method*), 209
set_initial_derivative() (*rmgpy.solver.SimpleReactor method*), 216
set_initial_reaction_thresholds() (*rmgpy.solver.LiquidReactor method*), 223
set_initial_reaction_thresholds() (*rmgpy.solver.ReactionSystem method*), 209
set_initial_reaction_thresholds() (*rmgpy.solver.SimpleReactor method*), 216
set_prunable_indices() (*rmgpy.solver.LiquidReactor method*), 223
set_prunable_indices() (*rmgpy.solver.ReactionSystem method*), 209
set_prunable_indices() (*rmgpy.solver.SimpleReactor method*), 216
setActions() (*rmgpy.molecule.AtomType method*), 107
setCanteraKinetics() (*rmgpy.kinetics.Arrhenius method*), 79
setCanteraKinetics() (*rmgpy.kinetics.Chebyshev method*), 88
setCanteraKinetics() (*rmgpy.kinetics.KineticsData method*), 78
setCanteraKinetics() (*rmgpy.kinetics.Lindemann method*), 92
setCanteraKinetics() (*rmgpy.kinetics.MultiArrhenius method*), 81
setCanteraKinetics() (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
setCanteraKinetics() (*rmgpy.kinetics.PDepArrhenius method*), 84
setCanteraKinetics() (*rmgpy.kinetics.PDepKineticsData method*), 83
setCanteraKinetics() (*rmgpy.kinetics.ThirdBody method*), 90
setCanteraKinetics() (*rmgpy.kinetics.Troe method*), 95
setConditions() (*rmgpy.pdep.Network method*), 154
setConditions() (*rmgpy.rmg.pdep.PDepNetwork method*), 195
setDefaultOutputDirectory() (*rmgpy.qm.main.QMCalculator method*), 158
setE0WithThermo() (*rmgpy.rmg.model.Species method*), 204
setE0WithThermo() (*rmgpy.species.Species method*), 228
setFrequency() (*rmgpy.species.TransitionState method*), 229
setLonePairs() (*rmgpy.molecule.Atom method*), 113
setOrderNum() (*rmgpy.molecule.Bond method*), 114
setOrderNum() (*rmgpy.molecule.GroupBond method*), 128
setOrderStr() (*rmgpy.molecule.Bond method*), 114
setOrderStr() (*rmgpy.molecule.GroupBond method*), 128
setThermodynamicFilteringParameters() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 189
SettingsError, 257
setUncertainty() (*rmgpy.quantity.ArrayQuantity method*), 178
setUncertainty() (*rmgpy.quantity.ScalarQuantity method*), 176
setUncertaintyType() (*rmgpy.quantity.ArrayQuantity method*), 178
setUncertaintyType() (*rmgpy.quantity.ScalarQuantity method*), 176
setValue() (*rmgpy.quantity.ArrayQuantity method*), 178
setValue() (*rmgpy.quantity.ScalarQuantity method*), 176
SimpleReactor (*class in rmgpy.solver*), 210
simpleRegularization()

(*rmgpy.data.kinetics.KineticsFamily method*), 35
`simulate()` (*rmgpy.solver.LiquidReactor method*), 223
`simulate()` (*rmgpy.solver.ReactionSystem method*), 209
`simulate()` (*rmgpy.solver.SimpleReactor method*), 216
`single` (*rmgpy.molecule.AtomType attribute*), 107
`SingleExponentialDown` (*class in rmgpy.pdep*), 148
`snapshots` (*rmgpy.solver.LiquidReactor attribute*), 223
`snapshots` (*rmgpy.solver.ReactionSystem attribute*), 209
`snapshots` (*rmgpy.solver.SimpleReactor attribute*), 216
`solve()` (*rmgpy.data.statmechfit.DirectFit method*), 54
`solve()` (*rmgpy.data.statmechfit.PseudoFit method*), 57
`solve()` (*rmgpy.data.statmechfit.PseudoRotorFit method*), 55
`solve_SS_network()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`solveFullME()` (*rmgpy.pdep.Network method*), 154
`solveFullME()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`solveReducedME()` (*rmgpy.pdep.Network method*), 154
`solveReducedME()` (*rmgpy.rmg.pdep.PDepNetwork method*), 195
`solveSchrodingerEquation()` (*rmgpy.statmech.HinderedRotor method*), 241
`sortAtoms()` (*rmgpy.molecule.Group method*), 133
`sortAtoms()` (*rmgpy.molecule.Molecule method*), 122
`sortByConnectivity()` (*rmgpy.molecule.Group method*), 133
`sortingLabel` (*rmgpy.molecule.Atom attribute*), 113
`sortingLabel` (*rmgpy.molecule.graph.Vertex attribute*), 100
`sortingLabel` (*rmgpy.molecule.GroupAtom attribute*), 126
`sortVertices()` (*rmgpy.molecule.graph.Graph method*), 103
`sortVertices()` (*rmgpy.molecule.Group method*), 133
`sortVertices()` (*rmgpy.molecule.Molecule method*), 122
`Species` (*class in rmgpy.rmg.model*), 201
`Species` (*class in rmgpy.species*), 224
`species` (*rmgpy.pdep.Configuration attribute*), 152
`SpeciesError`, 257
`speciesIndex` (*rmgpy.solver.LiquidReactor attribute*), 223
`speciesIndex` (*rmgpy.solver.ReactionSystem attribute*), 209
`speciesIndex` (*rmgpy.solver.SimpleReactor attribute*), 216
`specific` (*rmgpy.molecule.AtomType attribute*), 107
`specificCollider` (*rmgpy.data.kinetics.DepositoryReaction attribute*), 23
`specificCollider` (*rmgpy.data.kinetics.LibraryReaction attribute*), 48
`specificCollider` (*rmgpy.data.kinetics.TemplateReaction attribute*), 66
`specificCollider` (*rmgpy.reaction.Reaction attribute*), 185
`specificCollider` (*rmgpy.rmg.pdep.PDepReaction attribute*), 200
`specificColliderSpecies` (*rmgpy.solver.SimpleReactor attribute*), 216
`specifyAtomExtensions()` (*rmgpy.molecule.Group method*), 133
`specifyBondExtensions()` (*rmgpy.molecule.Group method*), 133
`specifyExternalNewBondExtensions()` (*rmgpy.molecule.Group method*), 133
`specifyInternalNewBondExtensions()` (*rmgpy.molecule.Group method*), 134
`specifyUnpairedExtensions()` (*rmgpy.molecule.Group method*), 134
`SphericalTopRotor` (*class in rmgpy.statmech*), 236
`spinMultiplicity` (*rmgpy.statmech.Conformer attribute*), 245
`split()` (*rmgpy.molecule.graph.Graph method*), 103
`split()` (*rmgpy.molecule.Group method*), 134
`split()` (*rmgpy.molecule.Molecule method*), 122
`splitReactions()` (*rmgpy.data.kinetics.KineticsFamily method*), 35
`standardizeAtomType()` (*rmgpy.molecule.Group method*), 134
`standardizeGroup()` (*rmgpy.molecule.Group method*), 134
`StatmechDatabase` (*class in rmgpy.data.statmech*), 49
`StatmechDepository` (*class in rmgpy.data.statmech*), 51
`StatmechError`, 257
`StatmechFitError`, 257
`StatmechGroups` (*class in rmgpy.data.statmech*), 57
`StatMechJob` (*class in arkane*), 10
`StatmechLibrary` (*class in rmgpy.data.statmech*), 59
`step()` (*rmgpy.solver.LiquidReactor method*), 223
`step()` (*rmgpy.solver.ReactionSystem method*), 209
`step()` (*rmgpy.solver.SimpleReactor method*), 216
`succesfulJobExists()` (*rmgpy.qm.qmverifier.QMVerifier method*), 161
`successKeys` (*rmgpy.qm.gaussian.Gaussian attribute*), 162
`successKeys` (*rmgpy.qm.mopac.Mopac attribute*), 167
`sumStates` (*rmgpy.pdep.Configuration attribute*), 152
`surfaceReactionIndices` (*rmgpy.solver.LiquidReactor attribute*), 223
`surfaceReactionIndices` (*rmgpy.solver.ReactionSystem attribute*), 209

`surfaceReactionIndices` (*rmgpy.solver.SimpleReactor attribute*), 217
`surfaceSpeciesIndices` (*rmgpy.solver.LiquidReactor attribute*), 223
`surfaceSpeciesIndices` (*rmgpy.solver.ReactionSystem attribute*), 210
`surfaceSpeciesIndices` (*rmgpy.solver.SimpleReactor attribute*), 217
`symbol` (*rmgpy.molecule.Element attribute*), 105
`symmetry` (*rmgpy.statmech.HinderedRotor attribute*), 241
`symmetry` (*rmgpy.statmech.KRotor attribute*), 236
`symmetry` (*rmgpy.statmech.LinearRotor attribute*), 234
`symmetry` (*rmgpy.statmech.NonlinearRotor attribute*), 235
`symmetry` (*rmgpy.statmech.SphericalTopRotor attribute*), 237
`SymmetryJob` (*class in rmgpy.qm.symmetry*), 161
`symmetryNumber` (*rmgpy.molecule.Molecule attribute*), 123
`symmetryNumber` (*rmgpy.rmg.model.Species attribute*), 204
`symmetryNumber` (*rmgpy.species.Species attribute*), 228

T

`T` (*rmgpy.solver.LiquidReactor attribute*), 218
`T` (*rmgpy.solver.SimpleReactor attribute*), 211
`T0` (*rmgpy.kinetics.Arrhenius attribute*), 79
`T0` (*rmgpy.pdep.SingleExponentialDown attribute*), 149
`t0` (*rmgpy.solver.LiquidReactor attribute*), 223
`t0` (*rmgpy.solver.ReactionSystem attribute*), 210
`t0` (*rmgpy.solver.SimpleReactor attribute*), 217
`T1` (*rmgpy.kinetics.Troe attribute*), 93
`T2` (*rmgpy.kinetics.Troe attribute*), 93
`T3` (*rmgpy.kinetics.Troe attribute*), 93
`Tdata` (*rmgpy.kinetics.KineticsData attribute*), 77
`Tdata` (*rmgpy.kinetics.PDepKineticsData attribute*), 82
`Tdata` (*rmgpy.thermo.ThermoData attribute*), 246
`TemplateReaction` (*class in rmgpy.data.kinetics*), 62
`terminal` (*rmgpy.molecule.Atom attribute*), 113
`terminal` (*rmgpy.molecule.graph.Vertex attribute*), 100
`terminal` (*rmgpy.molecule.GroupAtom attribute*), 126
`termination` (*rmgpy.solver.LiquidReactor attribute*), 223
`termination` (*rmgpy.solver.ReactionSystem attribute*), 210
`termination` (*rmgpy.solver.SimpleReactor attribute*), 217
`TerminationConversion` (*class in rmgpy.solver*), 224
`TerminationTime` (*class in rmgpy.solver*), 224
`thermo` (*rmgpy.rmg.model.Species attribute*), 204
`thermo` (*rmgpy.species.Species attribute*), 228
`ThermoData` (*class in rmgpy.thermo*), 245
`ThermoDatabase` (*class in rmgpy.data.thermo*), 67
`ThermoDepository` (*class in rmgpy.data.thermo*), 70
`thermoFilterDown()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 190
`thermoFilterSpecies()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 190
`ThermoGroups` (*class in rmgpy.data.thermo*), 72
`ThermoJob` (*class in arkane*), 10
`ThermoLibrary` (*class in rmgpy.data.thermo*), 74
`ThirdBody` (*class in rmgpy.kinetics*), 89
`Tlist` (*arkane.KineticsJob attribute*), 6
`Tlist` (*arkane.PressureDependenceJob attribute*), 9
`Tmax` (*arkane.KineticsJob attribute*), 6
`Tmax` (*arkane.PressureDependenceJob attribute*), 9
`Tmax` (*rmgpy.kinetics.Arrhenius attribute*), 79
`Tmax` (*rmgpy.kinetics.Chebyshev attribute*), 87
`Tmax` (*rmgpy.kinetics.KineticsData attribute*), 77
`Tmax` (*rmgpy.kinetics.Lindemann attribute*), 91
`Tmax` (*rmgpy.kinetics.MultiArrhenius attribute*), 80
`Tmax` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
`Tmax` (*rmgpy.kinetics.PDepArrhenius attribute*), 83
`Tmax` (*rmgpy.kinetics.PDepKineticsData attribute*), 82
`Tmax` (*rmgpy.kinetics.ThirdBody attribute*), 89
`Tmax` (*rmgpy.kinetics.Troe attribute*), 93
`Tmax` (*rmgpy.thermo.NASA attribute*), 251
`Tmax` (*rmgpy.thermo.NASAPolynomial attribute*), 253
`Tmax` (*rmgpy.thermo.ThermoData attribute*), 246
`Tmax` (*rmgpy.thermo.Wilhoit attribute*), 248
`Tmin` (*arkane.KineticsJob attribute*), 6
`Tmin` (*arkane.PressureDependenceJob attribute*), 9
`Tmin` (*rmgpy.kinetics.Arrhenius attribute*), 79
`Tmin` (*rmgpy.kinetics.Chebyshev attribute*), 87
`Tmin` (*rmgpy.kinetics.KineticsData attribute*), 77
`Tmin` (*rmgpy.kinetics.Lindemann attribute*), 91
`Tmin` (*rmgpy.kinetics.MultiArrhenius attribute*), 80
`Tmin` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 85
`Tmin` (*rmgpy.kinetics.PDepArrhenius attribute*), 83
`Tmin` (*rmgpy.kinetics.PDepKineticsData attribute*), 82
`Tmin` (*rmgpy.kinetics.ThirdBody attribute*), 89
`Tmin` (*rmgpy.kinetics.Troe attribute*), 94
`Tmin` (*rmgpy.thermo.NASA attribute*), 251
`Tmin` (*rmgpy.thermo.NASAPolynomial attribute*), 253
`Tmin` (*rmgpy.thermo.ThermoData attribute*), 246
`Tmin` (*rmgpy.thermo.Wilhoit attribute*), 248
`toAdjacencyList()` (*in module rmgpy.molecule.adjlist*), 145
`toAdjacencyList()` (*rmgpy.molecule.Group method*), 134
`toAdjacencyList()` (*rmgpy.molecule.Molecule method*), 123
`toAdjacencyList()` (*rmgpy.rmg.model.Species method*), 204

toAdjacencyList() (*rmgpy.species.Species method*), 228
toArrhenius() (*rmgpy.kinetics.MultiArrhenius method*), 81
toArrheniusEP() (*rmgpy.kinetics.Arrhenius method*), 80
toAugmentedInChI() (*rmgpy.molecule.Molecule method*), 123
toAugmentedInChIKey() (*rmgpy.molecule.Molecule method*), 123
toCantera() (*rmgpy.data.kinetics.DepositoryReaction method*), 23
toCantera() (*rmgpy.data.kinetics.LibraryReaction method*), 48
toCantera() (*rmgpy.data.kinetics.TemplateReaction method*), 66
toCantera() (*rmgpy.reaction.Reaction method*), 185
toCantera() (*rmgpy.rmg.model.Species method*), 204
toCantera() (*rmgpy.rmg.pdep.PDepReaction method*), 200
toCantera() (*rmgpy.species.Species method*), 228
toCantera() (*rmgpy.thermo.NASA method*), 252
toCanteraKinetics() (*rmgpy.kinetics.Arrhenius method*), 80
toChemkin() (*rmgpy.data.kinetics.DepositoryReaction method*), 23
toChemkin() (*rmgpy.data.kinetics.LibraryReaction method*), 48
toChemkin() (*rmgpy.data.kinetics.TemplateReaction method*), 66
toChemkin() (*rmgpy.reaction.Reaction method*), 185
toChemkin() (*rmgpy.rmg.model.Species method*), 204
toChemkin() (*rmgpy.rmg.pdep.PDepReaction method*), 200
toChemkin() (*rmgpy.species.Species method*), 228
toGroup() (*rmgpy.molecule.Molecule method*), 123
toHTML() (*rmgpy.kinetics.Arrhenius method*), 80
toHTML() (*rmgpy.kinetics.Chebyshev method*), 89
toHTML() (*rmgpy.kinetics.KineticsData method*), 78
toHTML() (*rmgpy.kinetics.Lindemann method*), 92
toHTML() (*rmgpy.kinetics.MultiArrhenius method*), 81
toHTML() (*rmgpy.kinetics.MultiPDepArrhenius method*), 86
toHTML() (*rmgpy.kinetics.PDepArrhenius method*), 85
toHTML() (*rmgpy.kinetics.PDepKineticsData method*), 83
toHTML() (*rmgpy.kinetics.ThirdBody method*), 90
toHTML() (*rmgpy.kinetics.Troe method*), 95
toInChI (*in module rmgpy.molecule.translator*), 143
toInChI() (*rmgpy.molecule.Molecule method*), 123
toInChIKey (*in module rmgpy.molecule.translator*), 143
toInChIKey() (*rmgpy.molecule.Molecule method*), 123
toLabeledStr() (*rmgpy.data.kinetics.DepositoryReaction method*), 23
toLabeledStr() (*rmgpy.data.kinetics.LibraryReaction method*), 48
toLabeledStr() (*rmgpy.data.kinetics.TemplateReaction method*), 67
toLabeledStr() (*rmgpy.reaction.Reaction method*), 185
toLabeledStr() (*rmgpy.rmg.pdep.PDepReaction method*), 200
toNASA() (*rmgpy.thermo.ThermoData method*), 246
toNASA() (*rmgpy.thermo.Wilhoit method*), 250
toOBMol (*in module rmgpy.molecule.converter*), 142
toRDKitMol (*in module rmgpy.molecule.converter*), 142
toRDKitMol() (*rmgpy.molecule.Molecule method*), 123
toSingleBonds() (*rmgpy.molecule.Molecule method*), 123
toSMARTS (*in module rmgpy.molecule.translator*), 143
toSMARTS() (*rmgpy.molecule.Molecule method*), 123
toSMILES (*in module rmgpy.molecule.translator*), 143
toSMILES() (*rmgpy.molecule.Molecule method*), 123
toThermoData() (*rmgpy.thermo.NASA method*), 252
toThermoData() (*rmgpy.thermo.Wilhoit method*), 250
toWilhoit() (*rmgpy.thermo.NASA method*), 252
toWilhoit() (*rmgpy.thermo.ThermoData method*), 247
Trange (*rmgpy.solver.LiquidReactor attribute*), 218
Trange (*rmgpy.solver.SimpleReactor attribute*), 211
TransitionState (*class in rmgpy.species*), 228
transitionState (*rmgpy.data.kinetics.DepositoryReaction attribute*), 23
transitionState (*rmgpy.data.kinetics.LibraryReaction attribute*), 48
transitionState (*rmgpy.data.kinetics.TemplateReaction attribute*), 67
transitionState (*rmgpy.reaction.Reaction attribute*), 185
transitionState (*rmgpy.rmg.pdep.PDepReaction attribute*), 201
transportData (*rmgpy.rmg.model.Species attribute*), 204
transportData (*rmgpy.species.Species attribute*), 228
trimolecular (*rmgpy.solver.LiquidReactor attribute*), 223
trimolecular (*rmgpy.solver.ReactionSystem attribute*), 210
trimolecular (*rmgpy.solver.SimpleReactor attribute*), 217
trimolecularThreshold (*rmgpy.solver.LiquidReactor attribute*), 223
trimolecularThreshold (*rmgpy.solver.ReactionSystem attribute*), 210
trimolecularThreshold (*rmgpy.solver.SimpleReactor attribute*), 217
triple (*rmgpy.molecule.AtomType attribute*), 107
Troe (*class in rmgpy.kinetics*), 92

tunneling (*rmgpy.species.TransitionState attribute*), 229

U

uncertainty (*rmgpy.quantity.ArrayQuantity attribute*), 178
uncertainty (*rmgpy.quantity.ScalarQuantity attribute*), 176
uncertainty_si (*rmgpy.quantity.ArrayQuantity attribute*), 178
uncertainty_si (*rmgpy.quantity.ScalarQuantity attribute*), 177
uncertaintyType (*rmgpy.quantity.ArrayQuantity attribute*), 178
uncertaintyType (*rmgpy.quantity.ScalarQuantity attribute*), 177
UndeterminableKineticsError, 257
UnexpectedChargeError, 257
unimolecularThreshold (*rmgpy.solver.LiquidReactor attribute*), 223
unimolecularThreshold (*rmgpy.solver.ReactionSystem attribute*), 210
unimolecularThreshold (*rmgpy.solver.SimpleReactor attribute*), 217
uniqueID (*rmgpy.qm.molecule.Geometry attribute*), 159
uniqueID (*rmgpy.qm.symmetry.SymmetryJob attribute*), 161
uniqueIDlong (*rmgpy.qm.molecule.Geometry attribute*), 159
unitDegeneracy() (*in module rmgpy.statmech.schrodinger*), 243
units (*rmgpy.quantity.ArrayQuantity attribute*), 178
units (*rmgpy.quantity.ScalarQuantity attribute*), 177
unresolved (*rmgpy.molecule.kekulize.AromaticRing attribute*), 139
update() (*rmgpy.molecule.Group method*), 134
update() (*rmgpy.molecule.kekulize.AromaticBond method*), 138
update() (*rmgpy.molecule.kekulize.AromaticRing method*), 139
update() (*rmgpy.molecule.Molecule method*), 123
update() (*rmgpy.rmg.pdep.PDepNetwork method*), 195
update_charge() (*rmgpy.molecule.Group method*), 134
updateAtomTypes() (*rmgpy.molecule.Molecule method*), 124
updateCharge() (*rmgpy.molecule.Atom method*), 113
updateConfigurations() (*rmgpy.rmg.pdep.PDepNetwork method*), 196
updateConnectivityValues() (*rmgpy.molecule.graph.Graph method*), 103
updateConnectivityValues() (*rmgpy.molecule.Group method*), 134
updateConnectivityValues() (*rmgpy.molecule.Molecule method*), 124
updateFingerprint() (*rmgpy.molecule.Group method*), 134
updateLonePairs() (*rmgpy.molecule.Molecule method*), 124
updateMultiplicity() (*rmgpy.molecule.Molecule method*), 124
updateReactionThresholdAndReactFlags() (*rmgpy.rmg.main.RMG method*), 192
updateUnimolecularReactionNetworks() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 190
usePolar (*rmgpy.qm.mopac.Mopac attribute*), 167

V

V (*rmgpy.solver.LiquidReactor attribute*), 218
V (*rmgpy.solver.SimpleReactor attribute*), 211
validLayeringIndices (*rmgpy.solver.LiquidReactor attribute*), 224
validLayeringIndices (*rmgpy.solver.ReactionSystem attribute*), 210
validLayeringIndices (*rmgpy.solver.SimpleReactor attribute*), 217
value (*rmgpy.quantity.ArrayQuantity attribute*), 178
value (*rmgpy.quantity.ScalarQuantity attribute*), 177
value_si (*rmgpy.quantity.ArrayQuantity attribute*), 178
value_si (*rmgpy.quantity.ScalarQuantity attribute*), 177
verifyOutputFile() (*rmgpy.qm.gaussian.Gaussian method*), 162
verifyOutputFile() (*rmgpy.qm.gaussian.GaussianMol method*), 163
verifyOutputFile() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
verifyOutputFile() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 167
verifyOutputFile() (*rmgpy.qm.mopac.Mopac method*), 167
verifyOutputFile() (*rmgpy.qm.mopac.MopacMol method*), 169
verifyOutputFile() (*rmgpy.qm.mopac.MopacMolPM3 method*), 170
verifyOutputFile() (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
verifyOutputFile() (*rmgpy.qm.mopac.MopacMolPM7 method*), 174
Vertex (*class in rmgpy.molecule.graph*), 99
vertex1 (*rmgpy.molecule.Bond attribute*), 114
vertex1 (*rmgpy.molecule.graph.Edge attribute*), 100
vertex1 (*rmgpy.molecule.GroupBond attribute*), 128

`vertex2` (*rmgpy.molecule.Bond attribute*), 114
`vertex2` (*rmgpy.molecule.graph.Edge attribute*), 100
`vertex2` (*rmgpy.molecule.GroupBond attribute*), 128
`vertices` (*rmgpy.molecule.graph.Graph attribute*), 103
`vertices` (*rmgpy.molecule.Group attribute*), 134
`vertices` (*rmgpy.molecule.Molecule attribute*), 124
`VF2` (*class in rmgpy.molecule.vf2*), 103
`VF2Error`, 257
`viscosity` (*rmgpy.solver.LiquidReactor attribute*), 224
`visit()` (*arkane.output.PrettifyVisitor method*), 8
`visit_Call()` (*arkane.output.PrettifyVisitor method*), 8
`visit_Dict()` (*arkane.output.PrettifyVisitor method*), 8
`visit_List()` (*arkane.output.PrettifyVisitor method*), 8
`visit_Num()` (*arkane.output.PrettifyVisitor method*), 8
`visit_Str()` (*arkane.output.PrettifyVisitor method*), 8
`visit_Tuple()` (*arkane.output.PrettifyVisitor method*), 8

W

`Wigner` (*class in rmgpy.kinetics*), 95
`Wilhoit` (*class in rmgpy.thermo*), 247
`writeInputFile()` (*rmgpy.qm.gaussian.GaussianMol method*), 164
`writeInputFile()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 165
`writeInputFile()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 167
`writeInputFile()` (*rmgpy.qm.mopac.MopacMol method*), 169
`writeInputFile()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 171
`writeInputFile()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 172
`writeInputFile()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 174
`writeInputFile()` (*rmgpy.qm.symmetry.SymmetryJob method*), 162
`writeKineticsEntry()` (*in module rmgpy.chemkin*), 13
`writeThermoEntry()` (*in module rmgpy.chemkin*), 13

Y

`y0` (*rmgpy.solver.LiquidReactor attribute*), 224
`y0` (*rmgpy.solver.ReactionSystem attribute*), 210
`y0` (*rmgpy.solver.SimpleReactor attribute*), 217