



RMG-Py API Reference

Release 3.3.0

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

Jul 10, 2025

CONTENTS

1	RMG API Reference	3
1.1	Arkane (arkane)	3
1.2	Chemkin files (rmgpy.chemkin)	23
1.3	Physical constants (rmgpy.constants)	26
1.4	Database (rmgpy.data)	27
1.5	Kinetics (rmgpy.kinetics)	95
1.6	Molecular representations (rmgpy.molecule)	125
1.7	Pressure dependence (rmgpy.pdep)	179
1.8	QMTP (rmgpy.qm)	189
1.9	Physical quantities (rmgpy.quantity)	210
1.10	Reactions (rmgpy.reaction)	215
1.11	Reaction mechanism generation (rmgpy.rmg)	221
1.12	Reaction system simulation (rmgpy.solver)	238
1.13	Species (rmgpy.species)	250
1.14	Statistical mechanics (rmgpy.statmech)	255
1.15	Thermodynamics (rmgpy.thermo)	274
1.16	RMG Exceptions (rmgpy.exceptions)	287
	Bibliography	293
	Python Module Index	295
	Index	297

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 <https://reactionmechanismgenerator.github.io/RMG-Py/>

RMG API REFERENCE

This document provides the complete details of the application programming interface (API) for the Python version of the Reaction Mechanism Generator. The functionality of RMG-Py is divided into many modules and subpackages. An overview of these components is given in the table below. Click on the name of a component to learn more and view its API.

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

Class	Description
<i>ESSAdapter</i>	Abstract Base Class for generic log files
<i>GaussianLog</i>	Extract chemical parameters from Gaussian log files
<i>MolproLog</i>	Extract chemical parameters from Molpro log files
<i>OrcaLog</i>	Extract chemical parameters from Orca log files
<i>QChemLog</i>	Extract chemical parameters from Q-Chem log files
<i>TeraChemLog</i>	Extract chemical parameters from TeraChem log files

Function	Description
<code>register_ess_adapter()</code>	Registers the ESS classes with the factory
<code>ess_factory()</code>	Generates the corresponding ESS adapter

1.1.2 Input

Function	Description
<code>load_input_file()</code>	Load an Arkane job input file

1.1.3 Output

Function	Description
<i>PrettifyVisitor</i>	Custom Abstract Syntax Tree (AST) visitor class
<code>prettify()</code>	Pretty formatting for a Python syntax string
<code>get_str_xyz()</code>	Pretty formatting for XYZ coordinates
<code>save_thermo_lib()</code>	Save an RMG thermo library
<code>save_kinetics_lib()</code>	Save an RMG kinetics library

1.1.4 Job classes

Class	Description
<i>Arkane</i>	Main class for Arkane jobs
<i>StatMechJob</i>	Compute the molecular degrees of freedom for a molecular conformation
<i>ThermoJob</i>	Compute the thermodynamic properties of a species
<i>KineticsJob</i>	Compute the high pressure-limit rate coefficient for a reaction using transition state theory
<i>PressureDependenceJob</i>	Compute the phenomenological pressure-dependent rate coefficients $k(T, P)$ for a uni-molecular reaction network
<i>ExplorerJob</i>	Explore a potential energy surface starting from a source

1.1.5 Sensitivity analysis

Class	Description
<i>KineticsSensitivity</i>	Perform sensitivity analysis for a kinetics job
<i>PDepSensitivity</i>	Perform sensitivity analysis for a pressure dependence job

1.1.6 Utility modules

Class	Description
<i>arkane.common</i>	Contains <i>common.ArkanesSpecies</i> and other commonly used functions

arkane.ess.ESSAdapter

class `arkane.ess.ESSAdapter`(*path*, *check_for_errors=True*, *scratch_directory=None*)

An abstract ESS Adapter class

abstract `check_for_errors()`

Checks the log file for common errors. Optionally runs when the class is initialized to catch errors before parsing relevant information.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

abstract `get_number_of_atoms()`

Return the number of atoms in the molecular configuration.

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

abstract `load_conformer`(*symmetry=None*, *spin_multiplicity=0*, *optical_isomers=None*, *label=""*)

Return the optimum geometry of the molecular configuration.

abstract `load_energy`(*zpe_scale_factor=1.0*)

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

abstract `load_force_constant_matrix()`

Return the force constant matrix (in Cartesian coordinates). If multiple such matrices are identified, only the last is returned. The units of the returned force constants are J/m². If no force constant matrix can be found in the log file, None is returned.

abstract load_geometry()

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

abstract load_negative_frequency()

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

abstract load_scan_energies()

Extract the optimized energies in J/mol from a torsional scan log file.

abstract load_scan_frozen_atoms()

Extract the atom numbers which were frozen during the scan. Return a list of list of atom numbers starting with the first atom as 1. Each element of the outer lists represents a frozen bond. Inner lists with length 2 represent frozen bond lengths. Inner lists with length 3 represent frozen bond angles. Inner lists with length 4 represent frozen dihedral angles.

abstract load_scan_pivot_atoms()

Extract the atom numbers which the rotor scan pivots around. Return a list of atom numbers starting with the first atom as 1.

abstract load_zero_point_energy()

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

Arkane ESS factory

`arkane.ess.factory.register_ess_adapter(ess: str, ess_class: Type[ESSAdapter]) → None`

A register for the ESS adapters.

Parameters

- **ess** – A string representation for an ESS adapter.
- **ess_class** – The ESS adapter class.

Raises

TypeError – If `ess_class` is not an `ESSAdapter` instance.

`arkane.ess.factory.ess_factory(fullpath: str, check_for_errors: bool = True, scratch_directory: Optional[str] = None) → Type[ESSAdapter]`

A factory generating the ESS adapter corresponding to `ess_adapter`. Given a path to the log file of a QM software, determine whether it is Gaussian, Molpro, Orca, Psi4, QChem, or TeraChem.

Parameters

- **fullpath** (*str*) – The disk location of the output file of interest.
- **check_for_errors** (*bool*) – Boolean indicating whether to check the QM log for common errors before parsing relevant information.

Returns

The requested `ESSAdapter` child, initialized with the respective arguments.

Return type

`Type[ESSAdapter]`

arkane.ess.GaussianLog

class arkane.ess.GaussianLog(*path*, *check_for_errors=True*, *scratch_directory=None*)

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. GaussianLog is an adapter for the abstract class ESSAdapter.

check_for_errors()

Checks for common errors in a Gaussian log file. If any are found, this method will raise an error and crash.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_number_of_atoms()

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

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

load_conformer(*symmetry=None*, *spin_multiplicity=0*, *optical_isomers=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.

load_energy(*zpe_scale_factor=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.

load_force_constant_matrix()

Return the force constant matrix from the Gaussian log file. The job that generated the log file must have the option `io(7/33=1)` in order for the proper force constant matrix (in Cartesian coordinates) to be printed in the log file. If multiple such matrices are identified, only the last is returned. The units of the returned force constants are J/m². If no force constant matrix can be found in the log file, `None` is returned.

load_geometry()

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

load_negative_frequency()

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

load_scan_energies()

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

load_scan_frozen_atoms()

Extract the atom numbers which were frozen during the scan Return a list of list of atom numbers starting with the first atom as 1 Each element of the outer lists represents a frozen bond Inner lists with length 2 represent frozen bond lengths Inner lists with length 3 represent frozen bond angles Inner lists with length 4 represent frozen dihedral angles

load_scan_pivot_atoms()

Extract the atom numbers which the rotor scan pivots around Return a list of atom numbers starting with the first atom as 1

load_zero_point_energy()

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

arkane.ess.MolproLog

class arkane.ess.MolproLog(*path*, *check_for_errors=True*, *scratch_directory=None*)

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. MolproLog is an adapter for the abstract class ESSAdapter.

check_for_errors()

Checks for common errors in a Molpro log file. If any are found, this method will raise an error and crash.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence

get_number_of_atoms()

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

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

load_conformer(symmetry=None, spin_multiplicity=0, optical_isomers=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.

load_energy(zpe_scale_factor=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.

load_force_constant_matrix()

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

load_geometry()

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

load_negative_frequency()

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

load_scan_energies()

Rotor scans are not implemented in Molpro

load_scan_frozen_atoms()

Not implemented for Molpro

load_scan_pivot_atoms()

Not implemented for Molpro

load_zero_point_energy()

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

arkane.ess.OrcaLog

class arkane.ess.OrcaLog(*path*, *check_for_errors=True*, *scratch_directory=None*)

Represent an output file from Orca. The attribute *path* refers to the location on disk of the Orca output file of interest. Methods are provided to extract a variety of information into Arkane classes and/or NumPy arrays. OrcaLog is an adapter for the abstract class ESSAdapter.

check_for_errors()

Checks for common errors in an Orca log file. If any are found, this method will raise an error and crash.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. T1 diagnostic only available in Coupled Cluster calculations.

get_number_of_atoms()

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

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

load_conformer(symmetry=None, spin_multiplicity=0, optical_isomers=None, label="")

Load the molecular degree of freedom data from a log file created as the result of an Orca "Freq" quantum chemistry calculation. As Orca'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 Orca log file will be adopted.

load_energy(zpe_scale_factor=1.0)

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

load_force_constant_matrix()

Return the force constant matrix from the Orca log file. Orca prints the Hessian matrix as a '.hess' file which must be located in the same folder as the log file for this method to parse it. The units of the returned force constants are J/m². If no force constant matrix can be found, None is returned.

load_geometry()

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

load_negative_frequency()

Return the imaginary frequency from a transition state frequency calculation in cm⁻¹. Since there can be many imaginary frequencies, only the first one is returned.

load_scan_energies()

not implemented in Orca

load_scan_frozen_atoms()

Not implemented for Orca

load_scan_pivot_atoms()

Not implemented for Orca

load_zero_point_energy()

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

arkane.ess.QchemLog

class arkane.ess.QChemLog(*path*, *check_for_errors=True*, *scratch_directory=None*)

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. QChemLog is an adapter for the abstract class ESSAdapter.

check_for_errors()

Checks for common errors in a QChem log file. If any are found, this method will raise an error and crash.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_number_of_atoms()

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

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

load_conformer(*symmetry=None, spin_multiplicity=0, optical_isomers=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.

load_energy(*zpe_scale_factor=1.0*)

Load the energy in J/mol from a QChem log file. Prioritize the energy from a converged geometry optimization. If the file does not contain an optimization job or if the optimization hit the maximum cycles, return the next equivalent source, such as from a frequency job. The zero-point energy is *not* included in the returned value.

load_force_constant_matrix()

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/m². If no force constant matrix can be found in the log file, *None* is returned.

load_geometry()

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

load_negative_frequency()

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

load_scan_energies()

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

load_scan_frozen_atoms()

Not implemented for QChem

load_scan_pivot_atoms()

Not implemented for QChem

load_zero_point_energy()

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

arkane.ess.TeraChemLog

class arkane.ess.TeraChemLog(*path, check_for_errors=True, scratch_directory=None*)

Represent a log file from TeraChem. The attribute *path* refers to the location on disk of the TeraChem log file of interest. Methods are provided to extract a variety of information into Arkane classes and/or NumPy arrays. TeraChemLog is an adapter for the abstract class ESSAdapter.

check_for_errors()

Checks for common errors in a TeraChem log file. If any are found, this method will raise an error and crash.

get_D1_diagnostic()

Returns the D1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_T1_diagnostic()

Returns the T1 diagnostic from output log. If multiple occurrences exist, returns the last occurrence. Should be implemented by the relevant subclass.

get_number_of_atoms()

Return the number of atoms in the molecular configuration used in the TeraChem output file. Accepted output files: TeraChem's log file, xyz format file, TeraChem's output.geometry file.

get_software()

Return the name of the software. Should correspond to the class name without 'Log'.

get_symmetry_properties()

This method uses the symmetry package from RMG's QM module and returns a tuple where the first element is the number of optical isomers, the second element is the symmetry number, and the third element is the point group identified.

load_conformer(*symmetry=None, spin_multiplicity=0, optical_isomers=None, label=""*)

Load the molecular degree of freedom data from an output file created as the result of a TeraChem "Freq" calculation. As TeraChem's guess of the external symmetry number might not always be correct, you can use the *symmetry* parameter to substitute your own value; if not provided, the value in the TeraChem output file will be adopted.

load_energy(*zpe_scale_factor=1.0*)

Load the energy in J/mol from a TeraChem log file. Only the last energy in the file is returned, unless the log file represents a frequencies calculation, in which case the first energy is returned. The zero-point energy is *not* included in the returned value.

load_force_constant_matrix()

Return the force constant matrix (in Cartesian coordinates) from the TeraChem log file. If multiple such matrices are identified, only the last is returned. The units of the returned force constants are J/m². If no force constant matrix can be found in the log file, *None* is returned.

load_geometry()

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

load_negative_frequency()

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

load_scan_energies()

Extract the optimized energies in J/mol from a TeraChem torsional scan log file.

load_scan_frozen_atoms()

Not implemented for TeraChem

load_scan_pivot_atoms()

Not implemented for TeraChem

load_zero_point_energy()

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

Arkane input files

`arkane.input.load_input_file(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, *three_params*=True)

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.

three_params - a boolean representing if the modified three-parameter Arrhenius equation is used to calculate

high pressure kinetic rate coefficients. If it is False, the classical two-parameter Arrhenius equation is used.

property Tlist

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

property Tmax

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

property Tmin

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

draw(*output_directory*, *file_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(*output_directory*=None, *plot*=True)

Execute the kinetics job, saving the results within the *output_directory*.

If *plot* is True, then plots of the raw and fitted values for the kinetics will be saved.

generate_kinetics()

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

plot(*output_directory*)

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_yaml(*output_directory*)

Save a YAML file for TSs if structures of the respective reactant/s and product/s are known

write_chemkin(*output_directory*)

Appends the kinetics rates to *chem.inp* in *output_directory*

write_output(*output_directory*)

Save the results of the kinetics job to the *output.py* file located in *output_directory*.

arkane.Arkane

class arkane.Arkane(*input_file=None, output_directory=None, verbose=20, save_rmg_libraries=True*)

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

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

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

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

execute()

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

get_libraries()

Get RMG kinetics and thermo libraries

load_input_file(input_file)

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

parse_command_line_arguments()

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

This module contains helper functionality for writing Arkane output files.

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*.

visit_UnaryOp(*node*)

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

`arkane.output.get_str_xyz(spc)`

Get a string representation of the 3D coordinates from the conformer.

Parameters

spc (*Species*) – A Species instance.

Returns

A string representation of the coordinates

Return type

str

`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.

`arkane.output.save_kinetics_lib(rxn_list, path, name, lib_long_desc)`

Save an RMG kinetics library.

Parameters

- **rxn_list** (*list*) – Entries are Reaction object instances for which kinetics will be saved.
- **path** (*str*) – The base folder in which the kinetic library will be saved.
- **name** (*str*) – The library name.
- **lib_long_desc** (*str*) – A multiline string with relevant description.

`arkane.output.save_thermo_lib(species_list, path, name, lib_long_desc)`

Save an RMG thermo library.

Parameters

- **species_list** (*list*) – Entries are Species object instances for which thermo will be saved.
- **path** (*str*) – The base folder in which the thermo library will be saved.
- **name** (*str*) – The library name.
- **lib_long_desc** (*str*) – A multiline string with relevant description.

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, sensitivity_perturbation=0)
```

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

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

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

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

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

property Plist

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

property Pmax

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

property Pmin

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

property Tlist

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

property Tmax

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

property 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(output_directory, file_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*.

execute(output_file, plot, file_format='pdf', print_summary=True)

Execute a PressureDependenceJob

fit_interpolation_model(Tdata, Pdata, kdata, k_units)

Fit an interpolation model to a pressure dependent rate

fit_interpolation_models()

Fit all pressure dependent rates with interpolation models

generate_P_list()

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.

generate_T_list()

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.

initialize()

Initialize a PressureDependenceJob

property maximum_grain_size

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

plot(output_directory)

Plot pressure dependent rates

save(output_file)

Save the output of a pressure dependent job

save_input_file(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.

create_hindered_rotor_figure(*angle, v_list, cosine_rotor, fourier_rotor, rotor, rotor_index*)

Plot the potential for the rotor, along with its cosine and Fourier series potential fits, and save it in the *hindered_rotor_plots* attribute.

execute(*output_directory=None, plot=False, pdep=False*)

Execute the statmech job, saving the results within the *output_directory*.

If *plot* is True, then plots of the hindered rotor fits will be saved.

load(*pdep=False, plot=False*)

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.

save_hindered_rotor_figures(*directory*)

Save hindered rotor plots as set of files of the form *rotor_[species_label]_0.pdf* in the specified directory

write_output(*output_directory*)

Save the results of the statmech job to the *output.py* file located in *output_directory*.

arkane.ThermoJob

class arkane.ThermoJob(*species, thermo_class*)

A representation of an Arkane thermodynamics job. This job is used to compute and save the thermodynamics information for a single species.

element_count_from_conformer()

Get an element count in a dictionary form (e.g., {'C': 3, 'H': 8}) from the species.conformer attribute.

Returns

Element count, keys are element symbols,

values are number of occurrences of the element in the molecule.

Return type

dict

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

Execute the thermodynamics job, saving the results within the *output_directory*.

If *plot* is true, then plots of the raw and fitted values for heat capacity, entropy, enthalpy, gibbs free energy, and hindered rotors will be saved.

generate_thermo()

Generate the thermodynamic data for the species and fit it to the desired heat capacity model (as specified in the *thermo_class* attribute).

plot(*output_directory*)

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

write_chemkin(*output_directory*)

Appends the thermo block to *chem.inp* and species name to *species_dictionary.txt* within the *output_directory* specified

write_output(*output_directory*)

Save the results of the thermodynamics job to the *output.py* file located in *output_directory*.

arkane.ExplorerJob

class arkane.ExplorerJob(*source*, *pdepjob*, *explore_tol*, *energy_tol=inf*, *flux_tol=0.0*, *bath_gas=None*, *maximum_radical_electrons=inf*)

A representation of an Arkane explorer job. This job is used to explore a potential energy surface (PES).

copy()

Return a copy of the explorer job.

execute(*output_file*, *plot*, *file_format='pdf'*, *print_summary=True*, *species_list=None*, *thermo_library=None*, *kinetics_library=None*)

Execute an ExplorerJob

Sensitivity Analysis

This module contains classes for sensitivity analysis of kinetics and pressure-dependent jobs.

class arkane.sensitivity.KineticsSensitivity(*job*, *output_directory*)

The *KineticsSensitivity* class represents an instance of a sensitivity analysis job performed for a KineticsJob. The attributes are:

Attribute	Description
<i>conditions</i>	A list of the conditions at which the sensitivity coefficients are calculated
<i>job</i>	The KineticsJob object
<i>f_rates</i>	A list of forward rates from <i>job</i> at the respective <i>conditions</i> in the appropriate units
<i>r_rates</i>	A list of reverse rates from <i>job</i> at the respective <i>conditions</i> in the appropriate units
<i>f_sa_rates</i>	A dictionary with Species as keys and each value is a list of forward rates from <i>job</i> at the respective <i>conditions</i> in the appropriate units after perturbing the corresponding Species' E0
<i>r_sa_rates</i>	Same as <i>f_sa_rates</i> , only for the reverse direction
<i>f_sa_coefficients</i>	A dictionary with Species keys and sensitivity coefficients in the forward direction as values
<i>r_sa_coefficients</i>	A dictionary with Species keys and sensitivity coefficients in the reverse direction as values

execute()

Execute the sensitivity analysis for a :class:KineticsJob: object

perturb(*species*)

Perturb a species' E0

plot()

Plot the SA results as horizontal bars

save()

Save the SA results as tabulated data as well as in YAML format

unperturb(*species*)

Return the species' E0 to its original value

class arkane.sensitivity.PDepSensitivity(*job, output_directory, perturbation, max_iters=5*)

The Sensitivity class represents an instance of a sensitivity analysis job performed for a PressureDependenceJob. The attributes are:

At-tribute	Description
<i>con- di- tions</i>	A list of the conditions (each entry is a list of one T and one P quantities) at which the sensitivity coefficients are calculated
<i>job</i>	The PressureDependenceJob object
<i>rates</i>	A dictionary with net_reactions as keys. Values are lists of forward rates from <i>job</i> for the respective path reaction at the respective <i>conditions</i> in the appropriate units
<i>sa_rates</i>	A dictionary with string representations of net_reactions as keys. Values are dictionaries with Wells or TransitionStates as keys and each value is a list of forward rates from <i>job</i> at the respective <i>conditions</i> after perturbing the corresponding well or TS's E0
<i>sa_coefficients</i>	A dictionary with similar structure as <i>sa_rates</i> , containing the sensitivity coefficients in the forward direction

execute()

Execute the sensitivity analysis for a :class:PressureDependenceJob: object

perturb(*entry, unperturb=False*)

Perturb E0 of *entry* which could be either a :class:TransitionState or a :class:Configuration In the latter case, only the first species in the Configuration.species list is perturbed. The perturbation is done by addition of the energy amount in self.perturbation. If unperturb is *False*, the perturbation is addition of the energy amount in self.perturbation. If unperturb is *False*, this is done by subtracting.

plot(*wells, transition_states*)

Draw the SA results as horizontal bars

save(*wells, transition_states*)

Save the SA output as tabulated data as well as in YAML format

unperturb(*entry*)

A helper function for calling self.perturb cleanly when unperturbing

arkane.common

Arkane common module

class arkane.common.ArkaneSpecies(*species=None, conformer=None, author="", level_of_theory="", model_chemistry="", frequency_scale_factor=None, use_hindered_rotors=None, use_bond_corrections=None, atom_energies="", chemkin_thermo_string="", smiles=None, adjacency_list=None, inchi=None, inchi_key=None, xyz=None, molecular_weight=None, symmetry_number=None, transport_data=None, energy_transfer_model=None, thermo=None, thermo_data=None, label=None, datetime=None, RMG_version=None, reactants=None, products=None, reaction_label=None, is_ts=None, charge=None, formula=None, multiplicity=None*)

A class for archiving an Arkane species including its statmech data into .yaml files

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

load_yaml(path, label=None, pdep=False)

Load the all statMech data from the .yaml file in *path* into *species* *pdep* is a boolean specifying whether or not *job_list* includes a pressureDependentJob.

make_object(data, class_dict)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

save_yaml(path)

Save the species with all statMech data to a .yaml file

update_species_attributes(species=None)

Update the object with a new species/TS (while keeping non-species-dependent attributes unchanged)

update_xyz_string()

Generate an xyz string built from self.conformer, and standardize the result

Returns

3D coordinates in an XYZ format.

Return type

str

arkane.common.check_conformer_energy(energies, path)

Check to see that the starting energy of the species in the potential energy scan calculation is not 0.5 kcal/mol (or more) higher than any other energies in the scan. If so, print and log a warning message.

arkane.common.clean_dir(base_dir_path: str = "", files_to_delete: Optional[List[str]] = None, file_extensions_to_delete: Optional[List[str]] = None, files_to_keep: Optional[List[str]] = None, sub_dir_to_keep: Optional[List[str]] = None) → None

Clean up a directory. Commonly used for removing unwanted files after unit tests.

Parameters

- **base_dir_path** (*str*) – absolute path of the directory to clean up.
- **files_to_delete** (*list[str]*) – full name of the file (includes extension) to delete.
- **file_extensions_to_delete** – extensions of files to delete.
- **files_to_keep** – full name of the file (includes extension) to keep, files specified here will NOT be deleted even if its extension is also in *file_extensions_to_delete*.

- **sub_dir_to_keep** – name of the subdirectories in the base directory to keep.

`arkane.common.convert_imaginary_freq_to_negative_float(freq: Union[str, float, int])`

Convert a string representation of an imaginary frequency into a negative float representation, e.g.:

'635.0i' -> -635.0 '500.0' -> 500.0

Parameters

freq (*str*) – The imaginary frequency representation.

Returns

A float representation of the frequency value.

Return type

float

`arkane.common.get_center_of_mass(coords, numbers=None, symbols=None)`

Calculate and return the 3D position of the center of mass of the current geometry. Either numbers or symbols must be given.

Parameters

- **coords** (*np.array*) – Entries are 3-length lists of xyz coordinates for an atom.
- **numbers** (*np.array, list*) – Entries are atomic numbers corresponding to coords.
- **symbols** (*list*) – Entries are atom symbols corresponding to coords.

Returns

The center of mass coordinates.

Return type

np.array

`arkane.common.get_element_mass(input_element, isotope=None)`

Returns the mass and z number of the requested isotope for a given element. 'input_element' can be wither the atomic number (integer) or an element symbol. 'isotope' is an integer of the atomic z number. If 'isotope' is None, returns the most common isotope. Data taken from NIST, https://physics.nist.gov/cgi-bin/Compositions/stand_alone.pl (accessed October 2018)

`arkane.common.get_moment_of_inertia_tensor(coords, numbers=None, symbols=None)`

Calculate and return the moment of inertia tensor for the current geometry in amu*angstrom^2. If the coordinates are not at the center of mass, they are temporarily shifted there for the purposes of this calculation. Adapted from J.W. Allen: <https://github.com/jwallen/ChemPy/blob/master/chempy/geometry.py>

Parameters

- **coords** (*np.array*) – Entries are 3-length lists of xyz coordinates for an atom.
- **numbers** (*np.array, list*) – Entries are atomic numbers corresponding to coords.
- **symbols** (*list*) – Entries are atom symbols corresponding to coords.

Returns

The 3x3 moment of inertia tensor.

Return type

np.array

Raises

InputError – If neither symbols nor numbers are given, or if they have a different length than coords

`arkane.common.get_principal_moments_of_inertia(coords, numbers=None, symbols=None)`

Calculate and return the principal moments of inertia in $\text{amu} \cdot \text{\AA}^2$ in decending order and the corresponding principal axes for the current geometry. The moments of inertia are in translated to the center of mass. The principal axes have unit lengths. Adapted from J.W. Allen: <https://github.com/jwallen/ChemPy/blob/master/chempy/geometry.py>

Parameters

- **coords** (*np.array*) – Entries are 3-length lists of xyz coordinates for an atom.
- **numbers** (*np.array*, *list*) – Entries are atomic numbers corresponding to coords.
- **symbols** (*list*) – Entries are atom symbols corresponding to coords.

Returns

The principal moments of inertia. tuple: The corresponding principal axes.

Return type

tuple

`arkane.common.is_pdep(job_list)`

A helper function to determine whether a job is PressureDependenceJob or not

`arkane.common.replace_yaml_syntax(content, label=None)`

PEP8 compliant changes to RMG objects could be backward incompatible with Arkane's YAML files. Search for knows phrases which were replace, and fix the format on the fly.

Parameters

content (*str*) – The content of an Arkane YAML file.

Returns

The modified content to be processed via `yaml.safe_load()`.

Return type

str

`arkane.common.str_repr(dumper, data)`

Repair YAML string representation

1.2 Chemkin files (`rmgpy.chemkin`)

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

1.2.1 Reading Chemkin files

Function	Description
<code>load_chemkin_file()</code>	Load a reaction mechanism from a Chemkin file
<code>load_species_dictionary()</code>	Load a species dictionary from a file
<code>load_transport_file()</code>	Load a Chemkin transport properties file
<code>read_kinetics_entry()</code>	Read a single reaction entry from a Chemkin file
<code>read_reaction_comments()</code>	Read the comments associated with a reaction entry
<code>read_reactions_block()</code>	Read the reactions block of a Chemkin file
<code>read_thermo_entry()</code>	Read a single thermodynamics entry from a Chemkin file
<code>remove_comment_from_line()</code>	Remove comment text from a line of a Chemkin file or species dictionary

1.2.2 Writing Chemkin files

Function	Description
<code>save_chemkin_file()</code>	Save a reaction mechanism to a Chemkin file
<code>save_species_dictionary()</code>	Save a species dictionary to a file
<code>save_transport_file()</code>	Save a Chemkin transport properties file
<code>get_species_identifier()</code>	Return the Chemkin-valid identifier for a given species
<code>mark_duplicate_reactions()</code>	Find and mark all duplicate reactions in a mechanism
<code>write_kinetics_entry()</code>	Write a single reaction entry to a Chemkin file
<code>write_thermo_entry()</code>	Write a single thermodynamics entry to a Chemkin file

Reading Chemkin files

Main functions

`rmgpy.chemkin.load_chemkin_file(path, dictionary_path=None, transport_path=None, read_comments=True, thermo_path=None, use_chemkin_names=False, check_duplicates=True, generate_resonance_structures=True, surface_path=False)`

Load a Chemkin input file located at *path* on disk to *path*, returning lists of the species and reactions in the Chemkin file. The ‘thermo_path’ point to a separate thermo file, or, if ‘None’ is specified, the function will look for the thermo database within the chemkin mechanism file. If *generate_resonance_structures* is True (default if omitted) then resonance isomers for each species are generated. If *surface path* is specified, the gas and surface species and reactions will be combined

`rmgpy.chemkin.load_species_dictionary(path, generate_resonance_structures=True)`

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. If *generate_resonance_structures* is True (default if omitted) then resonance isomers for each species are generated.

`rmgpy.chemkin.load_transport_file(path, species_dict, skip_missing_species=False)`

Load a Chemkin transport properties file located at *path* and store the properties on the species in *species_dict*. If *skip_missing_species*=True then species not defined in the *species_dict* are just skipped over, with a warning.

Helper functions

`rmgpy.chemkin.read_kinetics_entry(entry, species_dict, Aunits, Aunits_surf, Eunits)`

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

`rmgpy.chemkin.read_reaction_comments(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.read_reactions_block(f, species_dict, read_comments=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.read_thermo_entry(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 <https://shepherd.caltech.edu/EDL/PublicResources/sdt/formats/chemkin.html>

`rmgpy.chemkin.remove_comment_from_line(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.save_chemkin_file(path, species, reactions, verbose=True, check_for_duplicates=True)`

Save a Chemkin input file to *path* on disk containing the provided lists of *species* and *reactions*. If *check_for_duplicates* 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.save_species_dictionary(path, species, old_style=False)`

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

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

`rmgpy.chemkin.save_transport_file(path, species)`

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

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

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

Helper functions

`rmgpy.chemkin.get_species_identifier(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.write_kinetics_entry(reaction, species_list, verbose=True, java_library=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 *java_library = True* in order to generate a kinetics entry suitable for an RMG-Java kinetics library.

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

1.4 Database (rmgpy.data)

1.4.1 General classes

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

1.4.2 Thermodynamics database

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

1.4.3 Kinetics database

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

1.4.4 Statistical mechanics database

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

1.4.5 Statistical mechanics fitting

Class/Function	Description
<i>DirectFit</i>	DQED class for fitting a small number of vibrational frequencies and hindered rotors
<i>PseudoFit</i>	DQED class for fitting a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both
<i>PseudoRotorFit</i>	DQED class for fitting a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only
<i>fit_statmech_direct()</i>	Directly fit a small number of vibrational frequencies and hindered rotors
<i>fit_statmech_pseudo()</i>	Fit a large number of vibrational frequencies and hindered rotors by assuming degeneracies for both
<i>fit_statmech_pseudo_rotor()</i>	Fit a moderate number of vibrational frequencies and hindered rotors by assuming degeneracies for hindered rotors only
<i>fit_statmech_to_heat_capacity()</i>	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, short_desc="", long_desc="", metal=None, site=None, facet=None*)

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 `load_entry()`, `save_entry()`, `process_old_library_entry()`, and `generate_old_library_entry()` 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.

are_siblings(*node, node_other*)

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

descend_tree(*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.

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels=1*)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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,
                                             specific_collider=None, kinetics=None,
                                             reversible=True, transition_state=None,
                                             duplicate=False, degeneracy=1, pairs=None,
                                             depository=None, family=None, entry=None,
                                             electrons=0)
```

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.

apply_solvent_correction(*solvent*)

apply kinetic solvent correction

calculate_coll_limit(*temp*, *reverse=False*)

Calculate the collision limit rate in m³/mol-s for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

calculate_microcanonical_rate_coefficient(*e_list*, *j_list*, *reac_dens_states*,
prod_dens_states=None, *T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.

calculate_tst_rate_coefficient(*T*)

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

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

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

can_tst()

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(*t_min*, *t_max*, *p_min*, *p_max*)

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.

copy()

Create a deep copy of the current reaction.

degeneracy

The reaction path degeneracy for this reaction.

If the reaction has kinetics, changing the degeneracy will adjust the reaction rate by a ratio of the new degeneracy to the old degeneracy.

draw(*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.

ensure_species(*reactant_resonance=False, product_resonance=False, save_order=False*)

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. If *save_order* is True the atom order is reset after performing atom isomorphism.

fix_barrier_height(*force_positive=False, solvent="", apply_solvation_correction=True*)

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 *force_positive* is True, then all reactions are forced to have a non-negative barrier.

fix_diffusion_limited_a_factor(*T*)

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

generate_3d_ts(*reactants, products*)

Generate the 3D structure of the transition state. Called from `model.generate_kinetics()`.

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

generate_high_p_limit_kinetics()

Used for incorporating library reactions with pressure-dependent kinetics in PDep networks. Only implemented for LibraryReaction

generate_pairs()

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. This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generate_reverse_rate_coefficient(*network_kinetics=False, Tmin=None, Tmax=None, surface_site_density=0.0*)

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.

If the reaction kinetics model is Sticking Coefficient, please provide a nonzero surface site density in mol/m^2 which is required to evaluate the rate coefficient.

get_enthalpies_of_reaction(*Tlist*)

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

get_enthalpy_of_reaction(*T*)

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

get_entropies_of_reaction(*Tlist*)

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

get_entropy_of_reaction(*T*)

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

get_equilibrium_constant(*T*, *potential*=0.0, *type*='Kc', *surface_site_density*=2.5e-05)

Return the equilibrium constant for the reaction at the specified temperature *T* in K and reference *surface_site_density* in mol/m² (2.5e-05 default) The *type* parameter lets you specify the quantities used in the equilibrium constant: Ka for activities, Kc for concentrations (default), or Kp for pressures. This function assumes a reference pressure of 1e5 Pa for gas phases species and uses the ideal gas law to determine reference concentrations. For surface species, the *surface_site_density* is the assumed reference. For protons (H⁺), a reference concentration of 1000 mol/m³ (1 mol/L) is assumed

get_equilibrium_constants(*Tlist*, *potential*=0.0, *type*='Kc')

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.

get_free_energies_of_reaction(*Tlist*, *potential*=0.0)

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

get_free_energy_of_reaction(*T*, *potential*=0.0)

Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K and potential in Volts (if applicable)

get_mean_sigma_and_epsilon(*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_rate_coefficient(*T*, *P*=0.0, *surface_site_density*=0.0, *potential*=0.0)

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 *diffusion_limiter* 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.

If the reaction has sticking coefficient kinetics, a nonzero surface site density in mol/m² must be provided

get_reduced_mass(*reverse*=False)

Returns the reduced mass of the reactants if *reverse* is False Returns the reduced mass of the products if *reverse* is True

get_reversible_potential(*T*)

Get the Potential in V at *T* in 'K' at which the charge transfer reaction is at equilibrium

get_source()

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

get_stoichiometric_coefficient(*spec*)

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.

get_surface_rate_coefficient(*T*, *surface_site_density*, *potential*=0.0)

Return the overall surface rate coefficient for the forward reaction at temperature *T* in K with surface site density *surface_site_density* in mol/m². Value is returned in combination of [m,mol,s]

get_url()

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

has_template(*reactants*, *products*)

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

is_association()

Return True if the reaction represents an association reaction $A + B \rightleftharpoons C$ or False if not.

is_balanced()

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

is_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_dissociation()

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

is_isomerization()

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

is_isomorphic(*other*, *either_direction=True*, *check_identical=False*, *check_only_label=False*, *check_template_rxn_products=False*, *generate_initial_map=False*, *strict=True*, *save_order=False*)

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.

Parameters

- **either_direction** (*bool*, *optional*) – if False, then the reaction direction must match.
- **check_identical** (*bool*, *optional*) – if True, check that atom ID's match (used for checking degeneracy)
- **check_only_label** (*bool*, *optional*) – if True, only check the string representation, ignoring molecular structure comparisons
- **check_template_rxn_products** (*bool*, *optional*) – if True, only check isomorphism of reaction products (used when we know the reactants are identical, i.e. in generating reactions)
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons
- **save_order** (*bool*, *optional*) – if True, perform isomorphism saving atom order

is_surface_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_surface_reaction()

Return True if one or more reactants or products are surface species (or surface sites)

is_unimolecular()

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

matches_species(*reactants*, *products*=None)

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

protons

The stoichiometric coeff for protons in charge transfer reactions

reverse_arrhenius_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin*=None, *Tmax*=None)

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

reverse_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin*=None, *Tmax*=None)

Reverses the given *k_forward*, 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).

reverse_sticking_coeff_rate(*k_forward*, *reverse_units*, *surface_site_density*, *Tmin*=None, *Tmax*=None)

Reverses the given *k_forward*, which must be a *StickingCoefficient* type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self). The *surface_site_density* in *mol/m^2* is used to evaluate the forward rate constant.

reverse_surface_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin*=None, *Tmax*=None)

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

reverse_surface_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin*=None, *Tmax*=None)

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

set_reference_potential(*T*)

Set the reference Potential of the *SurfaceChargeTransfer* kinetics model to the reversible potential of the reaction

to_cantera(*species_list*=None, *use_chemkin_identifier*=False)

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

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

to_chemkin(*species_list*=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 *species_list* to figure out third body colliders.) Otherwise, only the reaction string will be returned.

to_labeled_str(*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

rmgpy.data.base.Entry

```
class rmgpy.data.base.Entry(index=-1, label='', item=None, parent=None, children=None, data=None,
                             data_count=None, reference=None, reference_type='', short_desc='',
                             long_desc='', rank=None, nodal_distance=None, metal=None, facet=None,
                             site=None, binding_energies=None, surface_site_density=None)
```

A class for representing individual records in an RMG database. Each entry in the database associates a chemical item (generally a species, functional group, or reaction) with a piece of data corresponding to that item. A significant amount of metadata can also be stored with each entry.

The attributes are:

Attribute	Description
<i>index</i>	A unique nonnegative integer index for the entry
<i>label</i>	A unique string identifier for the entry (or '' if not used)
<i>item</i>	The item that this entry represents
<i>parent</i>	The parent of the entry in the hierarchy (or None if not used)
<i>children</i>	A list of the children of the entry in the hierarchy (or None if not used)
<i>data</i>	The data to associate with the item
<i>data_count</i>	The number of data used to fit the group values in the group additivity method
<i>reference</i>	A Reference object containing bibliographic reference information to the source of the data
<i>reference_type</i>	The way the data was determined: 'theoretical', 'experimental', or 'review'
<i>short_desc</i>	A brief (one-line) description of the data
<i>long_desc</i>	A long, verbose description of the data
<i>rank</i>	An integer indicating the degree of confidence in the entry data, or None if not used
<i>nodal_distance</i>	A float representing the distance of a given entry from it's parent entry
<i>-</i>	For surface species thermo calculations:
<i>metal</i>	Which metal the thermo calculation was done on (None if not used)
<i>facet</i>	Which facet the thermo calculation was done on (None if not used)
<i>site</i>	Which surface site the molecule prefers (None if not used)
<i>binding_energies</i>	The surface binding energies for C,H,O, and N
<i>sur-face_site_density</i>	The surface site density

get_all_descendants()

retrieve all the descendants of 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.

generate_frequencies(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.

extract_source_from_comments(*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.load_chemkin_file 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.

get_forward_reaction_for_family_entry(*entry*, *family*, *thermo_database*)

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 *thermo_database* 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.

load_families(*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.

load_libraries(*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.

load_old(*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.

load_recommended_families(*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.

reconstruct_kinetics_from_source(*reaction*, *source*, *fix_barrier_height*=False, *force_positive_barrier*=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.extract_source_from_comments, self-constructed. fix_barrier_height and force_positive_barrier will change the kinetics based on the Reaction.fix_barrier_height 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.

save_families(*path*)

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

save_libraries(*path*)

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

save_old(*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.

save_recommended_families(*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="", short_desc="", long_desc="",  
                                             metal=None, site=None, facet=None)
```

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.

are_siblings(*node*, *node_other*)

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

descend_tree(*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.

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index*, *reactant1*=None, *reactant2*=None, *reactant3*=None, *product1*=None, *product2*=None, *product3*=None, *specificCollider*=None, *kinetics*=None, *degeneracy*=1, *label*="", *duplicate*=False, *reversible*=True, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="", *rank*=None, *metal*=None, *site*=None, *facet*=None)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

remove_group(*group_to_remove*)

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*, *reindex*=True)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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, short_desc="", long_desc="",
                                         forward_template=None, forward_recipe=None,
                                         reverse_template=None, reverse_recipe=None,
                                         forbidden=None, boundary_atoms=None,
                                         tree_distances=None, save_order=False)
```

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

Attribute	Type	Description
<i>reverse</i>	string	The name of the reverse reaction family
<i>reversible</i>	Boolean	Is family reversible? (True by default)
<i>forward_template</i>	Reaction	The forward reaction template
<i>forward_recipe</i>	ReactionRecipe	The steps to take when applying the forward reaction to a set of reactants
<i>reverse_template</i>	Reaction	The reverse reaction template
<i>reverse_recipe</i>	ReactionRecipe	The steps to take when applying the reverse reaction to a set of reactants
<i>forbidden</i>	ForbiddenStructure	(Optional) Forbidden product structures in either direction
<i>own_reverse</i>	Boolean	It's its own reverse?
'boundary_atoms'	list	Labels which define the boundaries of end groups in backbone/end families
<i>tree_distances</i>	dict	The default distance from parent along each tree, if not set default is 1 for every tree
'save_order'	Boolean	Whether to preserve atom order when manipulating structures.
<i>groups</i>	KineticsGroups	The set of kinetics group additivity values
<i>rules</i>	KineticsRules	The set of kinetics rate rules from RMG-Java
<i>depositories</i>	list	A set of additional depositories used to store kinetics data from various sources

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

add_atom_labels_for_reaction(*reaction*, *output_with_resonance=True*, *save_order=False*, *relabel_atoms=False*)

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 successfully able to map to the reaction is used. *None* is returned. If *save_order* is *True* the atom order is reset after performing atom isomorphism. If *relabel_atoms* is *True*, product atom labels of reversible families will be reversed to assist in identifying forbidden structures.

add_entry(*parent*, *grp*, *name*)

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

add_reverse_attribute(*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.

add_rules_from_training(*thermo_database=None*, *train_indices=None*)

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

ancestors(*node*)

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

apply_recipe(*reactant_structures*, *forward=True*, *unique=True*, *relabel_atoms=True*)

Apply the recipe for this reaction family to the list of *Molecule* or *Group* objects *reactant_structures*. 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. If `relabel_atoms` is `True`, product atom labels of reversible families will be reversed to assist in identifying forbidden structures.

are_siblings(*node*, *node_other*)

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

calculate_degeneracy(*reaction*, *resonance=True*)

For a *reaction* with *Molecule* or *Species* objects given in the direction in which the kinetics are defined, compute the reaction-path degeneracy. Can specify whether to consider resonance.

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.

clean_tree_groups()

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

cross_validate(*folds=5*, *template_rxn_map=None*, *test_rxn_inds=None*, *T=1000.0*, *iters=0*, *random_state=1*)

Perform K-fold cross validation on an automatically generated tree at temperature T after finding an appropriate node for kinetics estimation it will move up the tree *iters* times. Returns a dictionary mapping {rxn:Ln(k_Est/k_Train)}

cross_validate_old(*folds=5*, *T=1000.0*, *random_state=1*, *estimator='rate rules'*, *thermo_database=None*, *get_reverse=False*, *uncertainties=True*)

Perform K-fold cross validation on an automatically generated tree at temperature T Returns a dictionary mapping {rxn:Ln(k_Est/k_Train)}

descend_tree(*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.

distribute_tree_distances()

Fills in *nodal_distance* (the distance between an entry and its parent) if not already entered with the value from *tree_distances* associated with the tree the entry comes from

estimate_kinetics_using_rate_rules(*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.

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

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

extend_node(*parent, template_rxn_map, obj=None, T=1000.0, iter_max=inf, iter_item_cap=inf*)

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

extract_source_from_comments(*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?,
                '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.

fill_rules_by_averaging_up(*verbose=False*)

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

generate_old_tree(*entries, level*)

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

generate_product_template(*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.

generate_reactions(*reactants, products=None, prod_resonance=True, delete_labels=True, relabel_atoms=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.
- **delete_labels** (*bool, optional*) – Delete the labeled atoms from each generated reaction (optional). Default is True, atom labels are deleted.
- **relabel_atoms** (*bool, optional*) – Default is True, atoms are re-labeled.

Returns

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

generate_tree(*rxns=None, obj=None, thermo_database=None, T=1000.0, nprocs=1, min_splitable_entry_num=2, min_rxns_to_spawn=20, max_batch_size=800, outlier_fraction=0.02, stratum_num=8, new_fraction_threshold_to_reopt_node=0.25, extension_iter_max=inf, extension_iter_item_cap=inf*)

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)

Parameters

- **rxns** – List of reactions to generate tree from (if None pull the whole training set)
- **obj** – Object to expand tree from (if None uses top node)
- **thermo_database** – Thermodynamic database used for reversing training reactions
- **T** – Temperature the tree is optimized for
- **nprocs** – Number of process for parallel tree generation
- **min_splitable_entry_num** – the minimum number of splitable reactions at a node in order to spawn a new process solving that node
- **min_rxns_to_spawn** – the minimum number of reactions at a node to spawn a new process solving that node
- **max_batch_size** – the maximum number of reactions allowed in a batch, most batches will be this size the last will be smaller, if the # of reactions < max_batch_size the cascade algorithm is not used
- **outlier_fraction** – Fraction of reactions that are fastest/slowest and will be automatically included in the first batch
- **stratum_num** – Number of strata used in stratified sampling scheme
- **max_rxns_to_reopt_node** – Nodes with more matching reactions than this will not be pruned

get_backbone_roots()

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

get_end_roots()

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

get_entries_to_save()

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.

get_extension_edge(*parent, template_rxn_map, obj, T, iter_max=inf, iter_item_cap=inf*)

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.

get_kinetics(*reaction*, *template_labels*, *degeneracy*=1, *estimator*="", *return_all_kinetics*=True)

Return the kinetics for the given *reaction* by searching the various depositories as well as generating a result using the user-specified *estimator*. Currently, only 'rate rules' is a supported estimator. Unlike the regular *get_kinetics()* 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. This can be *True* or *False* if using a depository

If *return_all_kinetics*==False, only the first (best?) matching kinetics is returned.

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

Return an estimate of the kinetics for a reaction with the given *template* and reaction-path *degeneracy*. There is currently only one method to use: 'rate rules' (old RMG-Java behavior, and default RMG-Py behavior). Group additivity was removed in August 2023.

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, then the tuple returned is (kinetics, None).

get_kinetics_from_depository(*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.

get_labeled_reactants_and_products(*reactants*, *products*, *relabel_atoms*=True)

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. If *relabel_atoms* is *True*, product atom labels of reversible families will be reversed to assist in identifying forbidden structures.

get_rate_rule(*template*)

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

get_reaction_matches(*rxns*=None, *thermo_database*=None, *remove_degeneracy*=False, *estimate_thermo*=True, *fix_labels*=False, *exact_matches_only*=False, *get_reverse*=False, *rxns_with_kinetics_only*=False)

returns a dictionary mapping for each entry in the tree: (entry.label,entry.item) : list of all training reactions (or the list given) that match that entry

get_reaction_pairs(*reaction*)

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

get_reaction_template(*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.

get_reaction_template_labels(*reaction*)

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

get_root_template()

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`).

get_rxn_batches(*rxns*, *T=1000.0*, *max_batch_size=800*, *outlier_fraction=0.02*, *stratum_num=8*)

Breaks reactions into batches based on a modified stratified sampling scheme Effectively: The top and bottom outlier_fraction of all reactions are always included in the first batch The remaining reactions are ordered by the rate coefficients at T The list of reactions is then split into stratum_num similarly sized intervals batches sample equally from each interval, but randomly within each interval until they reach max_batch_size reactions A list of lists of reactions containing the batches is returned

get_sources_for_template(*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),...]

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

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

get_top_level_groups(*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.

get_training_depository()

Returns the *training* depository from `self.depositories`

get_training_set(*thermo_database=None*, *remove_degeneracy=False*, *estimate_thermo=True*, *fix_labels=False*, *get_reverse=False*, *rxns_with_kinetics_only=False*)

retrieves all reactions in the training set, assigns thermo to the species objects reverses reactions as necessary so that all reactions are in the forward direction and returns the resulting list of reactions in the forward direction with thermo assigned

has_rate_rule(*template*)

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

is_entry_match(*mol*, *entry*, *resonance=True*)

determines if the labeled molecule object of reactants matches the entry entry

is_molecule_forbidden(*molecule*)

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

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

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

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

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

load_forbidden(*label*, *group*, *shortDesc*="", *longDesc*="")

Load information about a forbidden structure. Note that argument names are retained for backward compatibility with loading database files.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

load_recipe(*actions*)

Load information about the reaction recipe.

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

Load information about the reaction template. Note that argument names are retained for backward compatibility with loading database files.

make_tree(*obj*=None, *regularization*=<function KineticsFamily.simple_regularization>, *thermo_database*=None, *T*=1000.0)

generates tree structure and then generates rules for the tree

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

prune_tree(*rxns*, *newrxns*, *thermo_database*=None, *new_fraction_threshold_to_reopt_node*=0.25, *fix_labels*=True, *exact_matches_only*=True, *get_reverse*=True)

Remove nodes that have less than maxRxnToReoptNode reactions that match and clear the regularization dimensions of their parent This is used to remove smaller easier to optimize and more likely to change nodes before adding a new batch in cascade model generation

regularize(*regularization*=<function KineticsFamily.simple_regularization>, *keep_root*=True, *thermo_database*=None, *template_rxn_map*=None, *rxns*=None)

Regularizes the tree according to the regularization function regularization

remove_group(*group_to_remove*)

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

retrieve_original_entry(*template_label*)

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

retrieve_template(*template_labels*)

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.

save_depository(*depository*, *path*)

Save the given kinetics family *depository* to the location *path* on disk.

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_generated_tree(*path=None*)

clears the rules and saves the family to its current location in database

save_groups(*path*)

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

save_old(*dictstr, treestr, libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*path*)

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

save_training_reactions(*reactions, reference=None, reference_type="", short_desc="", long_desc="", 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.

simple_regularization(*node, template_rxn_map, test=True*)

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

The test option to this function determines whether or not the reactions under a node match the extended group before adding an extension. If the test fails the extension is skipped.

In general test=True is needed if the cascade algorithm was used to generate the tree and test=False is ok if the cascade algorithm wasn't used.

rmgpy.data.kinetics.KineticsGroups

class rmgpy.data.kinetics.**KineticsGroups**(*entries=None, top=None, label="", name="", short_desc="", long_desc="", 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.

are_siblings(*node, node_other*)

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

descend_tree(*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.

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_reaction_template(*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.

get_species(*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.

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

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

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

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels=1*)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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, short_desc="",  
                                          long_desc="", auto_generated=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.

are_siblings(*node, node_other*)

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

check_for_duplicates(*mark_duplicates=False*)

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

convert_duplicates_to_multi()

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

descend_tree(*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.

generate_old_tree(*entries, level*)

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

get_entries_to_save()

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.

get_library_reactions()

makes library and template reactions as appropriate from the library comments and returns a list of all of these LibraryReaction and TemplateReaction objects

get_species(*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.

load_entry(*index*, *label*, *kinetics*, *degeneracy*=1, *duplicate*=False, *reversible*=True, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="", *allow_pdep_route*=False, *elementary_high_p*=False, *allow_max_rate_violation*=False, *metal*=None, *site*=None, *facet*=None)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*path*)

Load an old-style RMG kinetics library from the location *path*.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

mark_valid_duplicates(*reactions1*, *reactions2*)

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

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

remove_group(*group_to_remove*)

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.

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

Write the given *entry* in the kinetics library to the file object *f*.

save_old(*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.

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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="", short_desc="", long_desc="",
                                         auto_generated=False)
```

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.

are_siblings(*node*, *node_other*)

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

descend_tree(*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.

estimate_kinetics(*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.

fill_rules_by_averaging_up(*root_template*, *already_done*, *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)

generate_old_tree(*entries*, *level*)

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

get_all_rules(*template*)

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

get_entries()

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

get_entries_to_save()

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

get_rule(*template*)

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

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

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

has_rule(*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.

load_entry(*index*, *kinetics*=None, *degeneracy*=1, *label*="", *duplicate*=False, *reversible*=True, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="", *rank*=None, *nodalDistance*=None, *treeDistances*=None)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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,  
                                          specific_collider=None, kinetics=None,  
                                          network_kinetics=None, reversible=True,  
                                          transition_state=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.

apply_solvent_correction(*solvent*)

apply kinetic solvent correction

calculate_coll_limit(*temp*, *reverse=False*)

Calculate the collision limit rate in m³/mol-s for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

calculate_microcanonical_rate_coefficient(*e_list*, *j_list*, *reac_dens_states*,
 prod_dens_states=None, *T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.

calculate_tst_rate_coefficient(*T*)

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

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

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

can_tst()

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(*t_min*, *t_max*, *p_min*, *p_max*)

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.

copy()

Create a deep copy of the current reaction.

degeneracy

The reaction path degeneracy for this reaction.

If the reaction has kinetics, changing the degeneracy will adjust the reaction rate by a ratio of the new degeneracy to the old degeneracy.

draw(*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.

ensure_species(*reactant_resonance=False*, *product_resonance=False*, *save_order=False*)

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. If *save_order* is True the atom order is reset after performing atom isomorphism.

fix_barrier_height(*force_positive=False*, *solvent=""*, *apply_solvation_correction=True*)

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 *force_positive* is True, then all reactions are forced to have a non-negative barrier.

fix_diffusion_limited_a_factor(*T*)

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

generate_3d_ts(*reactants*, *products*)

Generate the 3D structure of the transition state. Called from `model.generate_kinetics()`.

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

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 $P_{\max} \geq 90$ 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`.

generate_pairs()

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. This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generate_reverse_rate_coefficient(*network_kinetics=False*, *Tmin=None*, *Tmax=None*, *surface_site_density=0.0*)

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.

If the reaction kinetics model is Sticking Coefficient, please provide a nonzero surface site density in mol/m^2 which is required to evaluate the rate coefficient.

get_enthalpies_of_reaction(*Tlist*)

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

get_enthalpy_of_reaction(*T*)

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

get_entropies_of_reaction(*Tlist*)

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

get_entropy_of_reaction(*T*)

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

get_equilibrium_constant(*T*, *potential=0.0*, *type='Kc'*, *surface_site_density=2.5e-05*)

Return the equilibrium constant for the reaction at the specified temperature *T* in K and reference *surface_site_density* in mol/m^2 (2.5e-05 default) The *type* parameter lets you specify the quantities used in

the equilibrium constant: K_a for activities, K_c for concentrations (default), or K_p for pressures. This function assumes a reference pressure of $1e5$ Pa for gas phases species and uses the ideal gas law to determine reference concentrations. For surface species, the *surface_site_density* is the assumed reference. For protons (H^+), a reference concentration of 1000 mol/m^3 (1 mol/L) is assumed

get_equilibrium_constants(*Tlist*, *potential*=0.0, *type*='Kc')

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

get_free_energies_of_reaction(*Tlist*, *potential*=0.0)

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

get_free_energy_of_reaction(*T*, *potential*=0.0)

Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K and potential in Volts (if applicable)

get_mean_sigma_and_epsilon(*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_rate_coefficient(*T*, *P*=0.0, *surface_site_density*=0.0, *potential*=0.0)

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 *diffusion_limiter* 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.

If the reaction has sticking coefficient kinetics, a nonzero surface site density in mol/m^2 must be provided

get_reduced_mass(*reverse*=False)

Returns the reduced mass of the reactants if *reverse* is False Returns the reduced mass of the products if *reverse* is True

get_reversible_potential(*T*)

Get the Potential in V at *T* in 'K' at which the charge transfer reaction is at equilibrium

get_source()

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

get_sticking_coefficient(*T*)

Helper function to get sticking coefficient

get_stoichiometric_coefficient(*spec*)

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.

get_surface_rate_coefficient(*T*, *surface_site_density*, *potential*=0.0)

Return the overall surface rate coefficient for the forward reaction at temperature *T* in K with surface site density *surface_site_density* in mol/m^2 . Value is returned in combination of [m,mol,s]

get_url()

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

has_template(*reactants*, *products*)

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

is_association()

Return True if the reaction represents an association reaction $A + B \rightleftharpoons C$ or False if not.

is_balanced()

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

is_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_dissociation()

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

is_isomerization()

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

is_isomorphic(*other*, *either_direction=True*, *check_identical=False*, *check_only_label=False*, *check_template_rxn_products=False*, *generate_initial_map=False*, *strict=True*, *save_order=False*)

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.

Parameters

- **either_direction** (*bool*, *optional*) – if False, then the reaction direction must match.
- **check_identical** (*bool*, *optional*) – if True, check that atom ID's match (used for checking degeneracy)
- **check_only_label** (*bool*, *optional*) – if True, only check the string representation, ignoring molecular structure comparisons
- **check_template_rxn_products** (*bool*, *optional*) – if True, only check isomorphism of reaction products (used when we know the reactants are identical, i.e. in generating reactions)
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons
- **save_order** (*bool*, *optional*) – if True, perform isomorphism saving atom order

is_surface_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_surface_reaction()

Return True if one or more reactants or products are surface species (or surface sites)

is_unimolecular()

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

matches_species(*reactants*, *products=None*)

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

protons

The stoichiometric coeff for protons in charge transfer reactions

reverse_arrhenius_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceChargeTransfer` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

reverse_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, 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).

reverse_sticking_coeff_rate(*k_forward*, *reverse_units*, *surface_site_density*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `StickingCoefficient` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self). The *surface_site_density* in mol/m^2 is used to evaluate the forward rate constant.

reverse_surface_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceArrhenius` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

reverse_surface_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceChargeTransfer` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

set_reference_potential(*T*)

Set the reference Potential of the `SurfaceChargeTransfer` kinetics model to the reversible potential of the reaction

to_cantera(*species_list=None*, *use_chemkin_identifier=False*)

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

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

to_chemkin(*species_list=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 *species_list* to figure out third body colliders.) Otherwise, only the reaction string will be returned.

to_labeled_str(*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

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.

match_to_structure(*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.

get_possible_structures(*entries*)

Return a list of the possible structures below this node.

match_logic_or(*other*)

Is other the same LogicOr group as self?

match_to_structure(*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.make_logic_node(*string*)

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

String should be of the form:

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

And the returned object will be of class LogicOr or LogicAnd

rmgpy.data.kinetics.ReactionRecipe

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

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

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

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

add_action(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.

apply_forward(struct, unique=True)

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

apply_reverse(struct, unique=True)

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

get_reverse()

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.

get_statmech_data(molecule, thermo_model=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.

get_statmech_data_from_depository(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).

get_statmech_data_from_groups(molecule, thermo_model)

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 *thermo_model*. This always returns valid degrees of freedom data.

get_statmech_data_from_library(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.

load_depository(*path*)

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

load_groups(*path*)

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

load_libraries(*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.

load_old(*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.

save_depository(*path*)

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

save_groups(*path*)

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

save_libraries(*path*)

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

save_old(*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="", short_desc="", long_desc=""*)

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.

are_siblings(*node*, *node_other*)

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

descend_tree(*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.

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index*, *label*, *molecule*, *statmech*, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="")

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

remove_group(*group_to_remove*)

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*, *reindex*=True)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*path*)

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

rmgpy.data.statmechfit

Fitting functions

`rmgpy.data.statmechfit.fit_statmech_to_heat_capacity(Tlist, Cvlist, n_vib, n_rot, molecule=None)`

For a given set of dimensionless heat capacity data *Cvlist* corresponding to temperature list *Tlist* in K, fit *n_vib* harmonic oscillator and *n_rot* 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.fit_statmech_direct(Tlist, Cvlist, n_vib, n_rot, molecule=None)`

Fit *n_vib* harmonic oscillator and *n_rot* 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.fit_statmech_pseudo_rotors(Tlist, Cvlist, n_vib, n_rot, molecule=None)`

Fit *n_vib* harmonic oscillator and *n_rot* 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.fit_statmech_pseudo(Tlist, Cvlist, n_vib, n_rot, molecule=None)`

Fit *n_vib* harmonic oscillator and *n_rot* 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.harmonic_oscillator_heat_capacity(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.harmonic_oscillator_d_heat_capacity_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.hindered_rotor_heat_capacity(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.hindered_rotor_d_heat_capacity_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.hindered_rotor_d_heat_capacity_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, n_vib, n_rot)`

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) constraints: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

initialize(*Neq, Nvars, Ncons, bounds=None, tolf=1e-05, told=1e-05, tolx=1e-05, maxIter=100, verbose=False*)

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(*x0*)

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, n_vib, n_rot)`

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) constraints: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

initialize(*Neq*, *Nvars*, *Ncons*, *bounds=None*, *tolf=1e-05*, *told=1e-05*, *tolx=1e-05*, *maxIter=100*, *verbose=False*)

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(*x0*)

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*, *n_vib*, *n_rot*)

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) constraints: $J'_{ij} = \frac{\partial g_i}{\partial x_j}$.

initialize(*Neq*, *Nvars*, *Ncons*, *bounds=None*, *tolf=1e-05*, *told=1e-05*, *tolx=1e-05*, *maxIter=100*, *verbose=False*)

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(*x0*)

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=""*, *short_desc=""*, *long_desc=""*)

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.

are_siblings(*node*, *node_other*)

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

descend_tree(*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.

generate_old_library_entry(*data*)

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

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_frequency_groups(*molecule*)

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

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

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

get_statmech_data(*molecule*, *thermo_model*)

Use the previously-loaded frequency database to generate a set of characteristic group frequencies corresponding to the specified *molecule*. The provided thermo data in *thermo_model* 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.

load_entry(*index*, *label*, *group*, *statmech*, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="")

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

process_old_library_entry(*data*)

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

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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="", short_desc="", long_desc=""*)

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.

are_siblings(*node, node_other*)

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

descend_tree(*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.

generate_old_library_entry(*data*)

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

generate_old_tree(*entries, level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index, label, molecule, statmech, reference=None, referenceType="", shortDesc="", longDesc=""*)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels=1*)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

process_old_library_entry(*data*)

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

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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,
                                           specific_collider=None, kinetics=None, reversible=True,
                                           transition_state=None, duplicate=False, degeneracy=1,
                                           pairs=None, family=None, template=None,
                                           estimator=None, reverse=None, is_forward=None,
                                           electrons=0)
```

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

At-tribute	Type	Description
<i>family</i>	str	The kinetics family that the reaction was created from.
<i>estimator</i>	str	The name of the kinetic estimator; currently only rate rules is supported.
<i>reverse</i>	bool	Whether the reaction is the reverse reaction, for families that are their own reverse.
<i>is_forward</i>	bool	Whether the reaction was generated in the forward direction of the family.
<i>labeled_atoms</i>	dict	Keys are 'reactants' or 'products', values are dictionaries. Keys in the second level dictionary are template labels (e.g., *1), values are the respective Atom object instance in the reactants.

apply_solvent_correction(*solvent*)

apply kinetic solvent correction in this case the parameters are dGTSSite instead of GTS

calculate_coll_limit(*temp*, *reverse=False*)

Calculate the collision limit rate in m³/mol-s for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

calculate_microcanonical_rate_coefficient(*e_list*, *j_list*, *reac_dens_states*,
prod_dens_states=None, *T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.

calculate_tst_rate_coefficient(*T*)

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

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

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

can_tst()

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(*t_min*, *t_max*, *p_min*, *p_max*)

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.

copy()

creates a new instance of `TemplateReaction`

degeneracy

The reaction path degeneracy for this reaction.

If the reaction has kinetics, changing the degeneracy will adjust the reaction rate by a ratio of the new degeneracy to the old degeneracy.

draw(*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.

ensure_species(*reactant_resonance=False, product_resonance=False, save_order=False*)

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. If *save_order* is True the atom order is reset after performing atom isomorphism.

fix_barrier_height(*force_positive=False, solvent="", apply_solvation_correction=True*)

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 *force_positive* is True, then all reactions are forced to have a non-negative barrier.

fix_diffusion_limited_a_factor(*T*)

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

generate_3d_ts(*reactants, products*)

Generate the 3D structure of the transition state. Called from `model.generate_kinetics()`.

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

generate_high_p_limit_kinetics()

Used for incorporating library reactions with pressure-dependent kinetics in PDep networks. Only implemented for LibraryReaction

generate_pairs()

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. This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generate_reverse_rate_coefficient(*network_kinetics=False, Tmin=None, Tmax=None, surface_site_density=0.0*)

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.

If the reaction kinetics model is Sticking Coefficient, please provide a nonzero surface site density in mol/m^2 which is required to evaluate the rate coefficient.

get_enthalpies_of_reaction(*Tlist*)

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

get_enthalpy_of_reaction(*T*)

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

get_entropies_of_reaction(*Tlist*)

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

get_entropy_of_reaction(*T*)

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

get_equilibrium_constant(*T*, *potential*=0.0, *type*='Kc', *surface_site_density*=2.5e-05)

Return the equilibrium constant for the reaction at the specified temperature *T* in K and reference *surface_site_density* in mol/m² (2.5e-05 default) The *type* parameter lets you specify the quantities used in the equilibrium constant: Ka for activities, Kc for concentrations (default), or Kp for pressures. This function assumes a reference pressure of 1e5 Pa for gas phases species and uses the ideal gas law to determine reference concentrations. For surface species, the *surface_site_density* is the assumed reference. For protons (H⁺), a reference concentration of 1000 mol/m³ (1 mol/L) is assumed

get_equilibrium_constants(*Tlist*, *potential*=0.0, *type*='Kc')

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.

get_free_energies_of_reaction(*Tlist*, *potential*=0.0)

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

get_free_energy_of_reaction(*T*, *potential*=0.0)

Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K and potential in Volts (if applicable)

get_mean_sigma_and_epsilon(*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_rate_coefficient(*T*, *P*=0.0, *surface_site_density*=0.0, *potential*=0.0)

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 *diffusion_limiter* 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.

If the reaction has sticking coefficient kinetics, a nonzero surface site density in mol/m² must be provided

get_reduced_mass(*reverse*=False)

Returns the reduced mass of the reactants if *reverse* is False Returns the reduced mass of the products if *reverse* is True

get_reversible_potential(*T*)

Get the Potential in V at *T* in 'K' at which the charge transfer reaction is at equilibrium

get_source()

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

get_stoichiometric_coefficient(*spec*)

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.

get_surface_rate_coefficient(*T*, *surface_site_density*, *potential*=0.0)

Return the overall surface rate coefficient for the forward reaction at temperature *T* in K with surface site density *surface_site_density* in mol/m². Value is returned in combination of [m,mol,s]

get_url()

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

has_template(*reactants*, *products*)

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

is_association()

Return True if the reaction represents an association reaction $A + B \rightleftharpoons C$ or False if not.

is_balanced()

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

is_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_dissociation()

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

is_isomerization()

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

is_isomorphic(*other*, *either_direction=True*, *check_identical=False*, *check_only_label=False*, *check_template_rxn_products=False*, *generate_initial_map=False*, *strict=True*, *save_order=False*)

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.

Parameters

- **either_direction** (*bool*, *optional*) – if False, then the reaction direction must match.
- **check_identical** (*bool*, *optional*) – if True, check that atom ID's match (used for checking degeneracy)
- **check_only_label** (*bool*, *optional*) – if True, only check the string representation, ignoring molecular structure comparisons
- **check_template_rxn_products** (*bool*, *optional*) – if True, only check isomorphism of reaction products (used when we know the reactants are identical, i.e. in generating reactions)
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons
- **save_order** (*bool*, *optional*) – if True, perform isomorphism saving atom order

is_surface_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_surface_reaction()

Return True if one or more reactants or products are surface species (or surface sites)

is_unimolecular()

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

matches_species(*reactants*, *products=None*)

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

protons

The stoichiometric coeff for protons in charge transfer reactions

reverse_arrhenius_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

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

reverse_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, 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).

reverse_sticking_coeff_rate(*k_forward*, *reverse_units*, *surface_site_density*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a *StickingCoefficient* type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self). The *surface_site_density* in *mol/m^2* is used to evaluate the forward rate constant.

reverse_surface_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

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

reverse_surface_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

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

set_reference_potential(*T*)

Set the reference Potential of the *SurfaceChargeTransfer* kinetics model to the reversible potential of the reaction

to_cantera(*species_list=None*, *use_chemkin_identifier=False*)

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

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

to_chemkin(*species_list=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 *species_list* to figure out third body colliders.) Otherwise, only the reaction string will be returned.

to_labeled_str(*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

rmgpy.data.thermo.ThermoDatabase**class** rmgpy.data.thermo.ThermoDatabase

A class for working with the RMG thermodynamics database.

compute_group_additivity_thermo(*molecule*)

Return the set of thermodynamic parameters corresponding to a given `Molecule` object *molecule* using the group additivity values method. 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.

correct_binding_energy(*thermo*, *species*, *metal_to_scale_from*=None, *metal_to_scale_to*=None)

Changes the provided thermo, by applying a linear scaling relation to correct the adsorption energy.

Parameters

- **thermo** – starting thermo data
- **species** – the species (which is an adsorbate)
- **metal_to_scale_from** – the metal you want to scale from (string eg. ‘Pt111’ or None)
- **metal_to_scale_to** – the metal you want to scale to (string e.g ‘Pt111’ or None)

Returns

corrected thermo

estimate_radical_thermo_via_hbi(*molecule*, *stable_thermo_estimator*)

Estimate the thermodynamics of a radical by saturating it, applying the provided *stable_thermo_estimator* 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.

estimate_thermo_via_group_additivity(*molecule*)

Return the set of thermodynamic parameters corresponding to a given `Molecule` object *molecule* using the group additivity values method. 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.

extract_source_from_comments(*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)]}

get_all_thermo_data(*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)

get_ring_groups_from_comments(*thermo_data*)

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

get_thermo_data(*species*, *metal_to_scale_to*=None, *training_set*=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.

If either metal to scale to or from is not specified, assume the binding energies given in the input file

Returns: ThermoData

get_thermo_data_for_surface_species(*species*)

Get the thermo data for an adsorbed species, by desorbing it, finding the thermo of the gas-phase species, then adding an adsorption correction that is found from the groups/adsorption tree. Does not apply linear scaling relationship.

Returns a ThermoData object, with no Cp0 or CpInf

get_thermo_data_from_depository(*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 (thermo_data, depository, entry) without any Cp0 or CpInf data.

get_thermo_data_from_groups(*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 should correct for it.

Returns: ThermoData

get_thermo_data_from_libraries(*species*, *training_set*=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. *training_set* 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

get_thermo_data_from_library(*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*, *surface=False*)

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

load_depository(*path*)

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

load_groups(*path*)

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

load_libraries(*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.

load_old(*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.

load_surface()

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

prioritize_thermo(*species*, *thermo_data_list*)

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.

prune_heteroatoms(*allowed=None*)

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

record_polycyclic_generic_nodes()

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

Necessary for polycyclic heuristic.

record_ring_generic_nodes()

Identify generic nodes in tree for ring groups. Saves them as a list in the *generic_nodes* 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.

save_depository(path)

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

save_groups(path)

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

save_libraries(path)

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

save_old(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.

save_surface(path)

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

set_binding_energies(binding_energies='Pt111')

Sets and stores the atomic binding energies specified in the input file.

All adsorbates will be scaled to use these elemental binding energies.

Parameters

binding_energies (*dict*, *optional*) – the desired binding energies with elements as keys and binding energy/unit tuples (or Energy quantities) as values

Returns

None, stores result in self.binding_energies

rmgpy.data.thermo.ThermoDepository

```
class rmgpy.data.thermo.ThermoDepository(label="", name="", short_desc="", long_desc="", metal=None,
                                         site=None, facet=None)
```

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.

are_siblings(node, node_other)

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

descend_tree(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.

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index*, *label*, *molecule*, *thermo*, *reference=None*, *referenceType=""*, *shortDesc=""*, *longDesc=""*, *rank=None*, *metal=None*, *site=None*, *facet=None*)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels=1*)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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="", short_desc="", long_desc="", metal=None,  
                                     site=None, facet=None)
```

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.

are_siblings(*node*, *node_other*)

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

copy_data(*source*, *destination*)

This method copies 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

descend_tree(*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.

generate_old_library_entry(*data*)

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

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index*, *label*, *group*, *thermo*, *reference=None*, *referenceType=""*, *shortDesc=""*, *longDesc=""*, *rank=None*, *metal=None*, *facet=None*, *site=None*)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels*=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.

process_old_library_entry(*data*)

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

remove_group(*group_to_remove*)

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

This is not (as of 2024/03) used within RMG-Py, but could be useful for people making ancillary scripts to manipulate or edit the database.

Returns the removed group.

save(*path*, *reindex*=True)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*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, short_desc="", long_desc="",  
                                     metal=None, site=None, facet=None)
```

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.

are_siblings(*node*, *node_other*)

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

descend_tree(*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.

generate_old_library_entry(*data*)

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

generate_old_tree(*entries*, *level*)

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

get_entries_to_save()

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.

get_species(*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.

load_entry(*index*, *label*, *molecule*, *thermo*, *reference*=None, *referenceType*="", *shortDesc*="", *longDesc*="", *rank*=None, *metal*=None, *facet*=None, *site*=None)

Method for parsing entries in database files. Note that these argument names are retained for backward compatibility.

load_old(*dictstr*, *treestr*, *libstr*, *num_parameters*, *num_labels*=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.

load_old_dictionary(*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*.

load_old_library(*path*, *num_parameters*, *num_labels*=1)

Parse an RMG database library located at *path*.

load_old_tree(*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.

match_node_to_child(*parent_node*, *child_node*)

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

match_node_to_node(*node*, *node_other*)

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

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

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

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

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

parse_old_library(*path*, *num_parameters*, *num_labels=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.

process_old_library_entry(*data*)

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

remove_group(*group_to_remove*)

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*, *reindex=True*)

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

save_dictionary(*path*)

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

save_entry(*f*, *entry*)

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

save_old(*dictstr*, *treestr*, *libstr*)

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

save_old_dictionary(*path*)

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

save_old_library(*path*)

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

save_old_tree(*path*)

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

1.5 Kinetics (`rmgpy.kinetics`)

The `rmgpy.kinetics` subpackage contains classes that represent various kinetics models of chemical reaction rates and models of quantum mechanical tunneling through an activation barrier.

1.5.1 Pressure-independent kinetics models

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

1.5.2 Pressure-dependent kinetics models

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

1.5.3 Tunneling models

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

1.5.4 Heterogenous catalysis kinetics models

Class	Description
<i>StickingCoefficient</i>	A kinetics model to give Sticking Coefficients for surface adsorption, following Arrhenius form.
<i>SurfaceArrhenius</i>	A kinetics model based on (modified) Arrhenius for surface reactions.

rmgpy.kinetics.KineticsData

class rmgpy.kinetics.**KineticsData**(*Tdata=None, kdata=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None, comment=""*)

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

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

Pmax

The maximum pressure at which the model is valid, or **None** if not defined.

Pmin

The minimum pressure at which the model is valid, or **None** if not defined.

Tdata

An array of temperatures at which rate coefficient values are known.

Tmax

The maximum temperature at which the model is valid, or **None** if not defined.

Tmin

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

comment

unicode

Type

comment

discrepancy(*self, KineticsModel other_kinetics*) → double

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

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K.

is_identical_to(*self, KineticsModel other_kinetics*) → bool

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

is_pressure_dependent(*self*) → bool

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

is_similar_to(*self, KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the kinetics for a cantera reaction object.

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

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*, *uncertainty=None*, *solute=None*, *comment=""*)

A kinetics model based on the (modified) Arrhenius equation. The attributes are:

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

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

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

Above, *A* is the preexponential factor, *T*₀ is the reference temperature, *n* is the temperature exponent, and *E*_a is the activation energy.

A

The preexponential factor.

Ea

The activation energy.

Pmax

The maximum pressure at which the model is valid, or None if not defined.

Pmin

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

T0

The reference temperature.

Tmax

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

Tmin

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

change_rate(*self*, *double factor*)

Changes A factor in Arrhenius expression by multiplying it by a *factor*.

change_t0(*self*, *double T0*)

Changes the reference temperature used in the exponent to *T0* in K, and adjusts the preexponential factor accordingly.

comment

unicode

Type

comment

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

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

fit_to_data(*self*, *ndarray Tlist*, *ndarray klist*, *unicode kunits*, *double T0=1*, *ndarray weights=None*, *bool three_params=True*)

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

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K.

is_identical_to(*self*, *KineticsModel other_kinetics*) → 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.

is_pressure_dependent(*self*) → bool

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

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Passes in a cantera Reaction() object and sets its rate to a Cantera ArrheniusRate object.

solute

object

Type

solute

to_arrhenius_ep(*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).

to_cantera_kinetics(*self*, *arrhenius_class=False*)

Converts the RMG Arrhenius object to a cantera ArrheniusRate or the auxiliary cantera Arrhenius class (used by falloff reactions). Inputs for both are (A,b,E) where A is in units of m³/kmol/s, b is dimensionless, and E is in J/kmol

arrhenius_class: If True, uses cantera.Arrhenius (for falloff reactions). If False, uses Cantera.ArrheniusRate

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.MultiArrhenius

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

A kinetics model based on a set of (modified) Arrhenius equations, which are summed to obtain the overall rate. The attributes are:

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

Pmax

The maximum pressure at which the model is valid, or None if not defined.

Pmin

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

Tmax

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

Tmin

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

arrhenius

list

Type

arrhenius

change_rate(*self*, *double factor*)

Change kinetics rate by a multiple factor.

comment

unicode

Type

comment

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

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

get_rate_coefficient(*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.

is_identical_to(*self*, *KineticsModel other_kinetics*) → 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`

is_pressure_dependent(*self*) → bool

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

is_similar_to(*self*, *KineticsModel other_kinetics*) → bool

Returns `True` if rates of reaction at temperatures 500,1000,1500,2000 K and 1 and 10 bar are within ± 0.5 for $\log(k)$, in other words, within a factor of 3.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

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

solute

object

Type

solute

to_arrhenius(*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 T_{\min} to T_{\max} , in Kelvin. If T_{\min} or T_{\max} are unspecified (or -1) then the MultiArrhenius's T_{\min} and T_{\max} are used. A linear least-squares fit is used, which guarantees that the resulting parameters provide the best possible approximation to the data.

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.PDepKineticsData

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

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

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

Pdata

An array of pressures at which rate coefficient values are known.

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

Tdata

An array of temperatures at which rate coefficient values are known.

Tmax

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

Tmin

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

comment

unicode

Type

comment

discrepancy(*self, KineticsModel other_kinetics*) → double

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

efficiencies

dict

Type

efficiencies

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K and pressure *P* in Pa.

highPlimit

rmgpy.kinetics.model.KineticsModel

Type

highPlimit

is_identical_to(*self*, *KineticsModel other_kinetics*) → bool

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

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the kinetics for a cantera reaction object.

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.PDepArrhenius

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

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

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

The pressure-dependent Arrhenius formulation is sometimes used to extend the Arrhenius expression to handle pressure-dependent kinetics. The formulation simply parameterizes A , n , and E_a to be dependent on pressure:

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

Although this suggests some physical insight, the $k(T, P)$ data is often highly complex and non-Arrhenius, limiting the usefulness of this formulation to simple systems.

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

Tmax

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

Tmin

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

arrhenius

list

Type

arrhenius

change_rate(*self, double factor*)

Changes kinetics rate by a multiple factor.

comment

unicode

Type

comment

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

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

efficiencies

dict

Type

efficiencies

fit_to_data(*self*, *ndarray* *Tlist*, *ndarray* *Plist*, *ndarray* *K*, *unicode* *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 change_rate(self, double factor)at` each pressure.

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K and pressure *P* in Pa.

highPlimit

`rmgpy.kinetics.model.KineticsModel`

Type

highPlimit

is_identical_to(*self*, *KineticsModel* *other_kinetics*) → 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

is_pressure_dependent(*self*) → bool

Return True since all objects derived from `PDepKineticsModel` represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel* *other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

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

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.MultiPDepArrhenius

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

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

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

Pmax

The maximum pressure at which the model is valid, or None if not defined.

Pmin

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

Tmax

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

Tmin

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

arrhenius

list

Type

arrhenius

change_rate(*self*, *double factor*)

Change kinetic rate by a multiple factor.

comment

unicode

Type

comment

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

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

efficiencies

dict

Type

efficiencies

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K and pressure *P* in Pa.

highPlimit

rmgpy.kinetics.model.KineticsModel

Type

highPlimit

is_identical_to(*self*, *KineticsModel other_kinetics*) → 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

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the PLOG kinetics for multiple cantera *Reaction* objects, provided in a list. *ct_reaction* is a list of cantera reaction objects.

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.Chebyshev

class rmgpy.kinetics.**Chebyshev**(*coeffs=None*, *kunits=""*, *highPlimit=None*, *Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *comment=""*)

A model of a phenomenological rate coefficient $k(T, P)$ using a set of Chebyshev polynomials in temperature and pressure. The attributes are:

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

The Chebyshev polynomial formulation is a means of fitting a wide range of complex $k(T, P)$ behavior. However, there is no meaningful physical interpretation of the polynomial-based fit, and one must take care to minimize the magnitude of Runge's phenomenon. The formulation is as follows:

$$\log k(T, P) = \sum_{t=1}^{N_T} \sum_{p=1}^{N_P} \alpha_{tp} \phi_t(\tilde{T}) \phi_p(\tilde{P})$$

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

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

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

are reduced temperature and reduced pressure designed to map the ranges (T_{\min}, T_{\max}) and (P_{\min}, P_{\max}) to $(-1, 1)$.

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

Tmax

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

Tmin

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

change_rate(*self*, *double factor*)

Changes kinetics rates by a multiple factor.

chebyshev(*self*, *int n*, *double x*) → *double*

Return the value of the *n*th-order Chebyshev polynomial at the given value of *x*.

coeffs

The Chebyshev coefficients.

comment

unicode

Type

comment

degreeP

'int'

Type

degreeP

degreeT

'int'

Type

degreeT

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

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

efficiencies

dict

Type

efficiencies

fit_to_data(*self*, *ndarray Tlist*, *ndarray Plist*, *ndarray K*, *unicode 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.

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K and pressure *P* in Pa by evaluating the Chebyshev expression.

get_reduced_pressure(*self*, *double P*) → double

Return the reduced pressure corresponding to the given pressure *P* in Pa. This maps the logarithm of the pressure onto the domain [-1, 1] using the *Pmin* and *Pmax* attributes as the limits.

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

Return the reduced temperature corresponding to the given temperature *T* in K. This maps the inverse of the temperature onto the domain [-1, 1] using the *Tmin* and *Tmax* attributes as the limits.

highPlimit

rmgpy.kinetics.model.KineticsModel

Type

highPlimit

is_identical_to(*self*, *KineticsModel other_kinetics*) → bool

Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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

unicode

Type

kunits

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

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).

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.ThirdBody

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

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

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

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

$$k(T, P) = k_0(T)[M]$$

Above, $[M] \approx P/RT$ is the concentration of the bath gas. This formulation is equivalent to stating that the kinetics are always in the low-pressure limit.

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

Tmax

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

Tmin

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

arrheniusLow

rmgpy.kinetics.arrhenius.Arrhenius

Type

arrheniusLow

change_rate(*self*, *double factor*)

Changes kinetics rate by a multiple *factor*.

comment

unicode

Type

comment

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

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

efficiencies

dict

Type

efficiencies

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

get_rate_coefficient(*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 [get_effective_pressure\(\)](#) to compute the effective pressure, and pass that value as the pressure to this method.

highPlimit

rmgpy.kinetics.model.KineticsModel

Type

highPlimit

is_identical_to(*self*, *KineticsModel other_kinetics*) → bool

Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the kinetics and efficiencies for a cantera *ThreeBodyReaction* object

solute

object

Type

solute

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.Lindemann

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

A kinetic model of a phenomenological rate coefficient $k(T, P)$ using the Lindemann formulation. The attributes are:

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

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

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

where

$$P_r = \frac{k_0(T)}{k_{\infty}(T)} [M]$$

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

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

and $[M] \approx P/RT$ is the concentration of the bath gas. The Arrhenius expressions $k_0(T)$ and $k_{\infty}(T)$ represent the low-pressure and high-pressure limit kinetics, respectively.

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

Tmax

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

Tmin

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

arrheniusHigh

`rmgpy.kinetics.arrhenius.Arrhenius`

Type

`arrheniusHigh`

arrheniusLow

`rmgpy.kinetics.arrhenius.Arrhenius`

Type

`arrheniusLow`

change_rate(*self*, double factor)

Changes kinetics rate by a multiple factor.

comment

unicode

Type

`comment`

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

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

efficiencies

dict

Type

`efficiencies`

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

get_rate_coefficient(*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 [get_effective_pressure\(\)](#) to compute the effective pressure, and pass that value as the pressure to this method.

highPLimit

rmgpy.kinetics.model.KineticsModel

Type

highPLimit

is_identical_to(*self*, *KineticsModel other_kinetics*) → bool

Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the efficiencies and kinetics for a cantera reaction.

solute

object

Type

solute

to_cantera_kinetics(*self*)

Converts the Lindemann object to a cantera LindemannRate object

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.Troe

```
class rmgpy.kinetics.Troe(arrheniusHigh=None, arrheniusLow=None, alpha=0.0, T3=None, T1=None,
                          T2=None, Tmin=None, Tmax=None, Pmin=None, Pmax=None,
                          efficiencies=None, comment="")
```

A kinetic model of a phenomenological rate coefficient $k(T, P)$ using the Troe formulation. The attributes are:

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

The Troe model attempts to make the Lindemann model quantitative by introducing a broadening factor F . The formulation is as follows:

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

where

$$P_r = \frac{k_0(T)}{k_{\infty}(T)} [M]$$

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

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

and $[M] \approx P/RT$ is the concentration of the bath gas. The Arrhenius expressions $k_0(T)$ and $k_{\infty}(T)$ represent the low-pressure and high-pressure limit kinetics, respectively. The broadening factor F is computed via

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

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

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

$$d = 0.14$$

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

Pmax

The maximum pressure at which the model is valid, or `None` if not defined.

Pmin

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

T1

The Troe T_1 parameter.

T2

The Troe T_2 parameter.

T3

The Troe T_3 parameter.

Tmax

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

Tmin

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

alpha

'double'

Type

alpha

arrheniusHigh

rmgpy.kinetics.arrhenius.Arrhenius

Type

arrheniusHigh

arrheniusLow

rmgpy.kinetics.arrhenius.Arrhenius

Type

arrheniusLow

change_rate(*self*, *double factor*)

Changes kinetics rate by a multiple factor.

comment

unicode

Type

comment

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

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

efficiencies

dict

Type

efficiencies

get_cantera_efficiencies(*self*, *species_list*)

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*

get_effective_collider_efficiencies(*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.

get_effective_pressure(*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*.

get_rate_coefficient(*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 [get_effective_pressure\(\)](#) to compute the effective pressure, and pass that value as the pressure to this method.

highPLimit

rmgpy.kinetics.model.KineticsModel

Type

highPLimit

is_identical_to(*self*, KineticsModel *other_kinetics*) → bool

Checks to see if kinetics matches that of other kinetics and returns True if coeffs, kunits, Tmin,

is_pressure_dependent(*self*) → bool

Return True since all objects derived from PDepKineticsModel represent pressure-dependent kinetics.

is_pressure_valid(*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.

is_similar_to(*self*, KineticsModel *other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Sets the efficiencies, kinetics, and troe falloff parameters for a cantera FalloffReaction.

solute

object

Type

solute

to_cantera_kinetics(*self*)

Converts the Troe object to a cantera Troe object

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.Wigner**class** rmgpy.kinetics.**Wigner**(*frequency*)

A tunneling model based on the Wigner formula. The attributes are:

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

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

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

where h is the Planck constant, ν_{TS} is the negative frequency, k_{B} is the Boltzmann constant, and T is the absolute temperature.

The Wigner formula represents the first correction term in a perturbative expansion for a parabolic barrier [1959Bell], and is therefore only accurate in the limit of a small tunneling correction. There are many cases for which the tunneling correction is very large; for these cases the Wigner model is inappropriate.

calculate_tunneling_factor(*self*, *double T*) → *double*Calculate and return the value of the Wigner tunneling correction for the reaction at the temperature T in K.**calculate_tunneling_function**(*self*, *ndarray Elist*) → *ndarray*Raises `NotImplementedError`, as the Wigner tunneling model does not have a well-defined energy-dependent tunneling function.**frequency**

The negative frequency along the reaction coordinate.

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

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

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

If *E0_prod* is not given, it is assumed to be the same as the reactants; this results in the so-called “symmetric” Eckart model. Providing *E0_prod*, and thereby using the “asymmetric” Eckart model, is the recommended approach.

The Eckart tunneling model is based around a potential of the form

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

where x represents the reaction coordinate and A and B are parameters. The potential is symmetric if $A = 0$ and asymmetric if $A \neq 0$. If we add the constraint $|B| > |A|$ then the potential has a maximum at

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

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

The one-dimensional Schrodinger equation with the Eckart potential is analytically solvable. The resulting microcanonical tunneling factor $\kappa(E)$ is a function of the total energy of the molecular system:

$$\kappa(E) = 1 - \frac{\cosh(2\pi a - 2\pi b) + \cosh(2\pi d)}{\cosh(2\pi a + 2\pi b) + \cosh(2\pi d)}$$

where

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

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

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

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

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

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

ΔV_1 and ΔV_2 are the thermal energy difference between the transition state and the reactants and products, respectively; ν_{TS} is the negative frequency, h is the Planck constant.

Applying a Laplace transform gives the canonical tunneling factor as a function of temperature T (expressed as $\beta \equiv 1/k_{\text{B}}T$):

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

If product data is not available, then it is assumed that $\alpha_2 \approx \alpha_1$.

The Eckart correction requires information about the reactants as well as the transition state. For best results, information about the products should also be given. (The former is called the symmetric Eckart correction, the latter the asymmetric Eckart correction.) This extra information allows the Eckart correction to generally give a better result than the Wigner correction.

E0_TS

The ground-state energy of the transition state.

E0_prod

The ground-state energy of the products.

E0_reac

The ground-state energy of the reactants.

calculate_tunneling_factor(*self*, *double* T) \rightarrow *double*

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

calculate_tunneling_function(*self*, *ndarray* E_{list}) \rightarrow *ndarray*

Calculate and return the value of the Eckart tunneling function for the reaction at the energies e_{list} in J/mol.

frequency

The negative frequency along the reaction coordinate.

rmgpy.kinetics.SurfaceArrhenius

class rmgpy.kinetics.**SurfaceArrhenius** (*A=None, n=0.0, Ea=None, T0=(1.0, 'K'), Tmin=None, Tmax=None, Pmin=None, Pmax=None, coverage_dependence=None, uncertainty=None, comment=""*)

A kinetics model based on (modified) Arrhenius for surface reactions.

It is very similar to the gas phase [Arrhenius](#)

The attributes are:

Attribute	Description
<i>A</i>	The preexponential factor
<i>T0</i>	The reference temperature
<i>n</i>	The temperature exponent
<i>Ea</i>	The activation energy
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>coverage_dependence</i>	A dictionary of coverage dependent parameters to a certain surface species with: <i>a</i> , the coefficient for exponential dependence on the coverage, <i>m</i> , the power-law exponent of coverage dependence, and <i>E</i> , the activation energy dependence on coverage.
<i>uncertainty</i>	Uncertainty information
<i>comment</i>	Information about the model (e.g. its source)

The SurfaceArrhenius class uses a rate expression that is identical to the Arrhenius class:

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

Above, *A* is the preexponential factor, *T*₀ is the reference temperature, *n* is the temperature exponent, and *E*_a is the activation energy.

The only difference is the units for *A* are in terms of surface concentration (mol/m²) instead of volume concentration (mol/m³).

A

The preexponential factor.

This (and the coverage dependence) is the only thing different from a normal Arrhenius class.

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.

change_rate(*self*, *double factor*)

Changes *A* factor in Arrhenius expression by multiplying it by a *factor*.

change_t0(*self*, *double T0*)

Changes the reference temperature used in the exponent to *T0* in K, and adjusts the preexponential factor accordingly.

comment

unicode

Type

comment

coverage_dependence

The coverage dependence parameters.

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

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

fit_to_data(*self*, *ndarray Tlist*, *ndarray klist*, *unicode kunits*, *double T0=1*, *ndarray weights=None*, *bool three_params=True*)

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

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

Return the rate coefficient in the appropriate combination of m³, mol, and s at temperature *T* in K.

is_identical_to(*self*, *KineticsModel other_kinetics*) → 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`.

is_pressure_dependent(*self*) → bool

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

is_similar_to(*self*, *KineticsModel other_kinetics*) → 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.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Takes in a cantera Reaction object and sets its rate to a cantera InterfaceArrheniusRate object.

solute

object

Type

solute

to_arrhenius_ep(*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 vice versa).

to_cantera_kinetics(*self*)

Converts the RMG SurfaceArrhenius object to a cantera InterfaceArrheniusRate

InterfaceArrheniusRate(A,b,E) where A is in units like m²/kmol/s (depending on dimensionality) b is dimensionless, and E is in J/kmol

to_html(*self*)

Return an HTML rendering.

to_surface_charge_transfer(*self*, *double V0*, *double electrons=-1*) → SurfaceChargeTransfer

Return an SurfaceChargeTransfer instance of the kinetics model with reversible potential *V0* in Volts and electron stoichiometric coeff`electrons`

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

rmgpy.kinetics.StickingCoefficient

```
class rmgpy.kinetics.StickingCoefficient(A=None, n=0.0, Ea=None, T0=(1.0, 'K'), Tmin=None,  
                                         Tmax=None, Pmin=None, Pmax=None,  
                                         coverage_dependence=None, comment="")
```

A kinetics model to give Sticking Coefficients for surface adsorption, following Arrhenius form. Similar to [Arrhenius](#) but with different units for A. The attributes are:

Attribute	Description
<i>A</i>	The preexponential factor. Unitless (sticking probability)
<i>T0</i>	The reference temperature
<i>n</i>	The temperature exponent
<i>Ea</i>	The activation energy
<i>Tmin</i>	The minimum temperature at which the model is valid, or zero if unknown or undefined
<i>Tmax</i>	The maximum temperature at which the model is valid, or zero if unknown or undefined
<i>Pmin</i>	The minimum pressure at which the model is valid, or zero if unknown or undefined
<i>Pmax</i>	The maximum pressure at which the model is valid, or zero if unknown or undefined
<i>coverage_dependence</i>	A dictionary of coverage dependent parameters to a certain surface species with: <i>a</i> , the coefficient for exponential dependence on the coverage, <i>m</i> , the power-law exponent of coverage dependence, and <i>E</i> , the activation energy dependence on coverage.
<i>comment</i>	Information about the model (e.g. its source)

The Sticking coefficient class uses an Arrhenius-like expression to determine the sticking coefficient γ of a species on a surface.

$$\gamma = A(T/T_0)^n \exp\left(-\frac{E_a}{RT}\right)$$

Above, *A* is the preexponential factor, *T*₀ is the reference temperature, *n* is the temperature exponent, and *E*_a is the activation energy.

The sticking coefficient is then used to calculate the rate coefficient *k* using the following expression:

$$k = \frac{\gamma}{\Gamma} \sqrt{\frac{RT}{2\pi W}}$$

where Γ is the surface site density and *W* is the molecular weight of the gas phase species.

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.

change_rate(*self*, double *factor*)

Changes A factor in Arrhenius expression by multiplying it by a *factor*.

change_t0(*self*, double *T0*)

Changes the reference temperature used in the exponent to *T0* in K, and adjusts the preexponential factor accordingly.

comment

unicode

Type

comment

coverage_dependence

The coverage dependence parameters.

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

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

fit_to_data(*self*, ndarray *Tlist*, ndarray *klist*, unicode *kunits*, double *T0=1*, ndarray *weights=None*, bool *three_params=True*)

Fit Arrhenius parameters to a set of sticking 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.

get_rate_coefficient(*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. This method must be overloaded in the derived class.

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

Return the sticking coefficient (dimensionless) at temperature *T* in K.

is_identical_to(*self*, *KineticsModel* *other_kinetics*) → 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`.

is_pressure_dependent(*self*) → bool

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

is_similar_to(*self*, *KineticsModel* *other_kinetics*) → bool

Returns `True` if the sticking coefficient at temperatures 500,1000,1500,2000 K are within +/- .5 for log(k), in other words, within a factor of 3.

is_temperature_valid(*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.

set_cantera_kinetics(*self*, *ct_reaction*, *species_list*)

Passes in a cantera Reaction() object and sets its rate to a Cantera ArrheniusRate object.

solute

object

Type

solute

to_cantera_kinetics(*self*)

Converts the RMG StickingCoefficient object to a cantera StickingArrheniusRate

StickingArrheniusRate(A,b,E) where A is dimensionless, b is dimensionless, and E is in J/kmol

to_html(*self*)

Return an HTML rendering.

uncertainty

rmgpy.kinetics.uncertainties.RateUncertainty

Type

uncertainty

1.6 Molecular representations (rmgpy.molecule)

The *rmgpy.molecule* subpackage contains classes and functions for working with molecular representations, particularly using chemical graph theory.

1.6.1 Graphs

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

1.6.2 Graph isomorphism

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

1.6.3 Elements and atom types

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

1.6.4 Molecules

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

1.6.5 Functional groups

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

1.6.6 Molecule Utilities

Class	Description
<i>rmgpy.molecule.resonance</i>	Resonance structure generation methods
<i>rmgpy.molecule.kekulize</i>	Kekule structure generation
<i>rmgpy.molecule.filtration</i>	Resonance structure filtration methods
<i>rmgpy.molecule.pathfinder</i>	Resonance path enumeration
<i>rmgpy.molecule.converter</i>	Molecule object converter (RDKit/OpenBabel)
<i>rmgpy.molecule.translator</i>	Molecule string representation translator

1.6.7 Adjacency lists

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

1.6.8 Symmetry numbers

Class	Description
<i>calculate_atom_symmetry_number()</i>	Calculate the atom-centered symmetry number for an atom in a molecule
<i>calculate_bond_symmetry_number()</i>	Calculate the bond-centered symmetry number for a bond in a molecule
<i>calculate_axis_symmetry_number()</i>	Calculate the axis-centered symmetry number for a double bond axis in a molecule
<i>calculate_cyclic_symmetry_number()</i>	Calculate the ring-centered symmetry number for a ring in a molecule
<i>calculate_symmetry_number()</i>	Calculate the total internal + external symmetry number for a molecule

1.6.9 Molecule and reaction drawing

Class	Description
<i>MoleculeDrawer</i>	Draw the skeletal formula of a molecule
<i>ReactionDrawer</i>	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
<i>connectivity1</i>	int	The number of nearest neighbors
<i>connectivity2</i>	int	The sum of the neighbors' <i>connectivity1</i> values
<i>connectivity3</i>	int	The sum of the neighbors' <i>connectivity2</i> values
<i>edges</i>	dict	Dictionary of edges with keys being neighboring vertices
<i>sorting_label</i>	int	An integer label used to sort the vertices

copy()

Return a copy of the vertex. The default implementation assumes that no semantic information is associated with each vertex, and therefore simply returns a new [*Vertex*](#) object.

equivalent(*other*, *strict=True*)

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.

is_specific_case_of(*other*)

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.

reset_connectivity_values()

Reset the cached structure information for this vertex.

rmgpy.molecule.graph.Edge

class rmgpy.molecule.graph.Edge

A base class for edges in a graph. The vertices which comprise the edge can be accessed using the *vertex1* and *vertex2* attributes.

copy()

Return a copy of the edge. The default implementation assumes that no semantic information is associated with each edge, and therefore simply returns a new [*Edge*](#) object. Note that the vertices are not copied in this implementation.

equivalent(*other*)

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.

get_other_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.

is_specific_case_of(*other*)

Return `True` if *self* is semantically more specific than *other*, or `False` if not. You should reimplement this function in a derived class if your edges have semantic information.

rmgpy.molecule.graph.Graph

class `rmgpy.molecule.graph.Graph`

A graph data type. The vertices of the graph are stored in a list *vertices*; this provides a consistent traversal order. A single edge can be accessed using the `get_edge()` 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 `get_edges()` method or `vertex.edges`.

add_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.

add_vertex(*vertex*)

Add a *vertex* to the graph. The vertex is initialized with no edges.

copy(*deep=False*)

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.

copy_and_map()

Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from `Graph.copy()` method

find_isomorphism(*other*, *initial_map=None*, *save_order=False*, *strict=True*)

Returns `True` if *other* is subgraph isomorphic and `False` otherwise, and the matching mapping. Uses the VF2 algorithm of Vento and Foggia.

Parameters

- **initial_map** (*dict*, *optional*) – initial atom mapping to use
- **save_order** (*bool*, *optional*) – if `True`, reset atom order after performing atom isomorphism
- **strict** (*bool*, *optional*) – if `False`, perform isomorphism ignoring electrons

find_subgraph_isomorphisms(*other*, *initial_map=None*, *save_order=False*)

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.

get_all_cycles(*starting_vertex*)

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]`

get_all_cycles_of_size(*size*)

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).

get_all_cyclic_vertices()

Returns all vertices belonging to one or more cycles.

get_all_edges()

Returns a list of all edges in the graph.

get_all_polycyclic_vertices()

Return all vertices belonging to two or more cycles, fused or spirocyclic.

get_all_simple_cycles_of_size(*size*)

Return a list of all non-duplicate monocyclic rings with length '*size*'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

get_disparate_cycles()

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`

get_edge(*vertex1*, *vertex2*)

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

get_edges(*vertex*)

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

get_edges_in_cycle(*vertices*, *sort=False*)

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 `get_relevant_cycles` or `get_smallest_set_of_smallest_rings`.

get_largest_ring(*vertex*)

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 `get_polycycles` are not necessarily in order in the ring structure.

get_max_cycle_overlap()

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.

get_monocycles()

Return a list of cycles that are monocyclic.

get_polycycles()

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.

get_relevant_cycles()

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

get_smallest_set_of_smallest_rings()

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

has_edge(vertex1, vertex2)

Returns True if vertices *vertex1* and *vertex2* are connected by an edge, or False if not.

has_vertex(vertex)

Returns True if *vertex* is a vertex in the graph, or False if not.

is_cyclic()

Return True if one or more cycles are present in the graph or False otherwise.

is_edge_in_cycle(edge)

Return True if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or False if not.

is_isomorphic(other, initial_map=None, generate_initial_map=False, save_order=False, strict=True)

Returns True if two graphs are isomorphic and False otherwise. Uses the VF2 algorithm of Vento and Foggia.

Parameters

- **initial_map** (*dict*, *optional*) – initial atom mapping to use
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **save_order** (*bool*, *optional*) – if True, reset atom order after performing atom isomorphism
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons

is_mapping_valid(other, mapping, equivalent=True, strict=True)

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. If *strict* is True, electrons and bond orders are considered, and ignored if False.

is_subgraph_isomorphic(other, initial_map=None, save_order=False)

Returns True if *other* is subgraph isomorphic and False otherwise. Uses the VF2 algorithm of Vento and Foggia.

is_vertex_in_cycle(vertex)

Return True if the given *vertex* is contained in one or more cycles in the graph, or False if not.

merge(*other*)

Merge two graphs so as to store them in a single Graph object.

remove_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.

remove_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.

reset_connectivity_values()

Reset any cached connectivity information. Call this method when you have modified the graph.

restore_vertex_order()

reorder the vertices to what they were before sorting if you saved the order

sort_cyclic_vertices(*vertices*)

Given a list of vertices comprising a cycle, sort them such that adjacent entries in the list are connected to each other. Warning: Assumes that the cycle is elementary, ie. no bridges.

sort_vertices(*save_order=False*)

Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

split()

Convert a single Graph object containing two or more unconnected graphs into separate graphs.

update_connectivity_values()

Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

rmgpy.molecule.vf2.VF2**class rmgpy.molecule.vf2.VF2**

An implementation of the second version of the Vento-Foggia (VF2) algorithm for graph and subgraph isomorphism.

feasible(*vertex1*, *vertex2*)

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.

find_isomorphism(*graph1*, *graph2*, *initial_mapping*, *save_order=False*, *strict=True*)

Return a list of dicts of all valid isomorphism mappings from graph *graph1* to graph *graph2* with the optional initial mapping *initial_mapping*. If no valid isomorphisms are found, an empty list is returned.

find_subgraph_isomorphisms(*graph1*, *graph2*, *initial_mapping*, *save_order=False*)

Return a list of dicts of all valid subgraph isomorphism mappings from graph *graph1* to subgraph *graph2* with the optional initial mapping *initial_mapping*. If no valid subgraph isomorphisms are found, an empty list is returned.

is_isomorphic(*graph1*, *graph2*, *initial_mapping*, *save_order=False*, *strict=True*)

Return True if graph *graph1* is isomorphic to graph *graph2* with the optional initial mapping *initial_mapping*, or False otherwise.

is_subgraph_isomorphic(*graph1*, *graph2*, *initial_mapping*, *save_order=False*)

Return True if graph *graph1* is subgraph isomorphic to subgraph *graph2* with the optional initial mapping *initial_mapping*, or False otherwise.

rmgpy.molecule.Element

class rmgpy.molecule.Element

A chemical element. The attributes are:

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

rmgpy.molecule.**get_element**(*value*, *isotope=-1*)

Return the *Element* object corresponding to the given parameter *value*. If an integer is provided, the value is treated as the atomic number. If a string is provided, the value is treated as the symbol. An *ElementError* is raised if no matching element is found.

rmgpy.molecule.AtomType

class rmgpy.molecule.AtomType

A class for internal representation of atom types. Using unique objects rather than strings allows us to use fast pointer comparisons instead of slow string comparisons, as well as store extra metadata. In particular, we store metadata describing the atom type's hierarchy with regard to other atom types, and the atom types that can result when various actions involving this atom type are taken. The attributes are:

Attribute	Type	Description
<i>label</i>	str	A unique label for the atom type
<i>generic</i>	list	The atom types that are more generic than this one
<i>specific</i>	list	The atom types that are more specific than this one
<i>increment_bond</i>	list	The atom type(s) that result when an adjacent bond's order is incremented
<i>decrement_bond</i>	list	The atom type(s) that result when an adjacent bond's order is decremented
<i>form_bond</i>	list	The atom type(s) that result when a new single bond is formed to this atom type
<i>break_bond</i>	list	The atom type(s) that result when an existing single bond to this atom type is broken
<i>increment_radical</i>	list	The atom type(s) that result when the number of radical electrons is incremented
<i>decrement_radical</i>	list	The atom type(s) that result when the number of radical electrons is decremented
<i>increment_charge</i>	list	The atom type(s) that result when the number of radical electrons is decremented and charge is incremented
<i>decrement_charge</i>	list	The atom type(s) that result when the number of radical electrons is incremented and charge is decremented
<i>increment_lone_pair</i>	list	The atom type(s) that result when the number of lone electron pairs is incremented
<i>decrement_lone_pair</i>	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		
<i>single</i>	list	The total number of single bonds on the atom
<i>all_double</i>	list	The total number of double bonds on the atom
<i>r_double</i>	list	The number of double bonds to any non-oxygen, nonsulfur
<i>o_double</i>	list	The number of double bonds to oxygen
<i>s_double</i>	list	The number of double bonds to sulfur
<i>triple</i>	list	The total number of triple bonds on the atom
<i>quadruple</i>	list	The total number of quadruple bonds on the atom
<i>benzene</i>	list	The total number of benzene bonds on the atom
<i>lone_pairs</i>	list	The number of lone pairs on the atom
<i>charge</i>	list	The partial charge of the atom

equivalent(*other*)

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.

get_features()

Returns a list of the features that are checked to determine atomtype

is_specific_case_of(*other*)

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

rmgpy.molecule.get_atomtype(*atom*, *bonds*)

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

The atom type of an atom describes the atom itself and (often) something about the local bond structure around that atom. This is a useful semantic tool for accelerating graph isomorphism queries, and a useful shorthand when specifying molecular substructure patterns via an RMG-style adjacency list.

We define the following basic atom types:

Atom type	Description
<i>General atom types</i>	
R	any atom with any local bond structure
R!H	any non-hydrogen atom with any local bond structure
R!H!Val7	any non-hydrogen and non-halogen atom (a non-terminal atom)
<i>Hydrogen atom types</i>	
H	hydrogen atom with up to one single bond
<i>Carbon atom types</i>	
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 bonds
Cdc	charged carbon atom with one double bond and up to one single bond
C0	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 bonds
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 up to one single bond
C2tc	charged carbon atom with one lone pair (valance 2), one triple bond
<i>Nitrogen atom types</i>	
N	nitrogen atom with any local bond structure
N0sc	charged nitrogen atom with three lone pairs (valance 0) with up to one single bond
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 bonds
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 one single bond
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 bonds
N5dc	charged nitrogen atom with no lone pairs (valance 5), one double bond and up to one single bond
N5ddc	charged nitrogen atom with with no lone pairs (valance 5) and two double bonds
N5dddc	charged nitrogen atom with with no lone pairs (valance 5) and three double bonds
N5tc	charged nitrogen atom with with no lone pairs (valance 5), one triple bond and up to one single bond
N5b	nitrogen atom with with no lone pairs (valance 5) and two benzene bonds (one double bond and one triple bond)
N5bd	nitrogen atom with with no lone pairs (valance 5), two benzene bonds, and up to one single bond
<i>Oxygen atom types</i>	
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 bond
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 bond

Table 2 – continued from previous page

Atom type	Description
02d	oxygen atom with two lone pairs (valance 2) and one double bond
04sc	charged oxygen atom with one lone pair (valance 4) and up to three single bonds
04dc	charged oxygen atom with one lone pair (valance 4), one double bond and up to two single bonds
04tc	charged oxygen atom with one lone pair (valance 4) and one triple bond
04b	oxygen atom with one lone pair (valance 4) and two benzene bonds
<i>Silicon atom types</i>	
Si	silicon atom with any local bond structure
Sis	silicon atom with four single bonds
Sid	silicon atom with one double bond (to carbon) and two single bonds
Si0	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
<i>Phosphorus atom types</i>	
P	phosphorus atom with any local bond structure
P0sc	charged phosphorus atom with three lone pairs (valence 0) and up to 1 single bond
P1s	phosphorus atom with two lone pairs (valence 1) and up to 1 single bond
P1sc	charged phosphorus atom with two lone pairs (valence 1) and up to 2 single bonds
P1dc	charged phosphorus atom with two lone pairs (valence 1) and 1 double bond
P3s	phosphorus atom with one lone pair (valence 3) and up to 3 single bonds
P3d	phosphorus atom with one lone pair (valence 3), 1 double bond and up to 1 single bond
P3t	phosphorus atom with one lone pair (valence 3) and 1 triple bond
P3b	phosphorus atom with one lone pair (valence 3) and 2 benzene bonds
P5s	phosphorus atom with no lone pairs (valence 5) and up to 5 single bonds
P5sc	charged phosphorus atom with no lone pairs (valence 5) and up to 6 single bonds
P5d	phosphorus atom with no lone pairs (valence 5), 1 double bond and up to 3 single bonds
P5dd	phosphorus atom with no lone pairs (valence 5), 2 double bonds and up to 1 single bond
P5dc	charged phosphorus atom with no lone pairs (valence 5), 1 double bond and up to 2 single bonds
P5ddc	charged phosphorus atom with no lone pairs (valence 5) and 2 double bonds
P5t	phosphorus atom with no lone pairs (valence 5), 1 triple bond and up to 2 single bonds
P5td	phosphorus atom with no lone pairs (valence 5), 1 triple bond and 1 double bond
P5tc	charged phosphorus atom with no lone pairs (valence 5), 1 triple bond and up to 1 single bond
P5b	phosphorus atom with no lone pairs (valence 5), 2 benzene bonds and up to 1 single bond
P5bd	phosphorus atom with no lone pairs (valence 5), 2 benzene bonds and 1 double bond
<i>Sulfur atom types</i>	
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 bond
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 bonds
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 up to one single bond
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 bonds
S4d	sulfur atom with one lone pair (valance 4), one double bond and up to two single bonds
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 bonds

Table 2 – continued from previous page

Atom type	Description
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 bo
S6d	sulfur atom with no lone pairs (valance 6), one double bond and up to four s
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,
<i>Chlorine atom types</i>	
Cl	chlorine atom with any local bond structure
Cl1s	chlorine atom with three lone pairs and zero to one single bonds
<i>Bromine atom types</i>	
Br	bromine atom with any local bond structure
Br1s	bromine atom with three lone pairs and zero to one single bonds
<i>Iodine atom types</i>	
I	iodine atom with any local bond structure
I1s	iodine atom with three lone pairs and zero to one single bonds
<i>Fluorine atom types</i>	
F	fluorine atom with any local bond structure
F1s	fluorine 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	<i>center1</i> , <i>order</i> , <i>center2</i>	change the bond order of the bond between <i>center1</i> and <i>center2</i> by <i>order</i> ; do not break or form bonds
FORM_BOND	<i>center1</i> , <i>order</i> , <i>center2</i>	form a new bond between <i>center1</i> and <i>center2</i> of type <i>order</i>
BREAK_BOND	<i>center1</i> , <i>order</i> , <i>center2</i>	break the bond between <i>center1</i> and <i>center2</i> , which should be of type <i>order</i>
GAIN_RADICAL	<i>center</i> , <i>radical</i>	increase the number of free electrons on <i>center</i> by <i>radical</i>
LOSE_RADICAL	<i>center</i> , <i>radical</i>	decrease the number of free electrons on <i>center</i> by <i>radical</i>

rmgpy.molecule.Atom**class** rmgpy.molecule.Atom

An atom. The attributes are:

Attribute	Type	Description
<i>atomtype</i>	AtomType	The <i>atom type</i>
<i>element</i>	Element	The chemical element the atom represents
<i>radical_electrons</i>	short	The number of radical electrons
<i>charge</i>	short	The formal charge of the atom
<i>label</i>	str	A string label that can be used to tag individual atoms
<i>coords</i>	numpy array	The (x,y,z) coordinates in Angstrom
<i>lone_pairs</i>	short	The number of lone electron pairs
<i>id</i>	int	Number assignment for atom tracking purposes
<i>bonds</i>	dict	Dictionary of bond objects with keys being neighboring atoms
<i>props</i>	dict	Dictionary for storing additional atom properties
<i>mass</i>	int	atomic mass of element (read only)
<i>number</i>	int	atomic number of element (read only)
<i>symbol</i>	str	atomic symbol of element (read only)
<i>site</i>	str	type of adsorption site
<i>morphology</i>	str	morphology of the adsorption site

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`.

apply_action(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](#).

copy()

Generate a deep copy of the current atom. Modifying the attributes of the copy will not affect the original.

decrement_charge()

Update the atom pattern as a result of applying a LOSE_CHARGE action

decrement_lone_pairs()

Update the lone electron pairs pattern as a result of applying a LOSE_PAIR action.

decrement_radical()

Update the atom pattern as a result of applying a LOSE_RADICAL action, where *radical* specifies the number of radical electrons to remove.

equivalent(other, strict=True)

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. If *strict* is False, then only the element is compared and electrons are ignored.

get_total_bond_order()

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

increment_charge()

Update the atom pattern as a result of applying a GAIN_CHARGE action

increment_lone_pairs()

Update the lone electron pairs pattern as a result of applying a GAIN_PAIR action.

increment_radical()

Update the atom pattern as a result of applying a GAIN_RADICAL action, where *radical* specifies the number of radical electrons to add.

is_bonded_to_halogen()

Return True if the atom is bonded to at least one halogen (F, Cl, Br, or I) False if it is not

is_bonded_to_surface()

Return True if the atom is bonded to a surface atom X False if it is not

is_bromine()

Return True if the atom represents a bromine atom or False if not.

is_carbon()

Return True if the atom represents a carbon atom or False if not.

is_chlorine()

Return True if the atom represents a chlorine atom or False if not.

is_electron()

Return True if the atom represents an electron or False if not.

is_fluorine()

Return True if the atom represents a fluorine atom or False if not.

is_halogen()

Return True if the atom represents a halogen atom (F, Cl, Br, I) False if it does.

is_hydrogen()

Return True if the atom represents a hydrogen atom or False if not.

is_iodine()

Return True if the atom represents an iodine atom or False if not.

is_lithium()

Return True if the atom represents a hydrogen atom or False if not.

is_nitrogen()

Return True if the atom represents a nitrogen atom or False if not.

is_non_hydrogen()

Return True if the atom does not represent a hydrogen atom or False if it does.

is_nos()

Return True if the atom represent either nitrogen, sulfur, or oxygen False if it does not.

is_oxygen()

Return True if the atom represents an oxygen atom or False if not.

is_phosphorus()

Return True if the atom represents a phosphorus atom or False if not.

is_proton()

Return True if the atom represents a proton or False if not.

is_silicon()

Return True if the atom represents a silicon atom or False if not.

is_specific_case_of(*other*)

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.

is_sulfur()

Return True if the atom represents a sulfur atom or False if not.

is_surface_site()

Return True if the atom represents a surface site or False if not.

reset_connectivity_values()

Reset the cached structure information for this vertex.

set_lone_pairs(*lone_pairs*)

Set the number of lone electron pairs.

sorting_key

Returns a sorting key for comparing Atom objects. Read-only

update_charge()

Update self.charge, according to the valence, and the number and types of bonds, radicals, and lone pairs.

rmgpy.molecule.Bond**class rmgpy.molecule.Bond**

A chemical bond. The attributes are:

Attribute	Type	Description
<i>order</i>	float	The <i>bond type</i>
<i>atom1</i>	Atom	An Atom object connecting to the bond
<i>atom2</i>	Atom	An Atom object connecting to the bond

apply_action(*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()

Generate a deep copy of the current bond. Modifying the attributes of the copy will not affect the original.

decrement_order()

Update the bond as a result of applying a CHANGE_BOND action to decrease the order by one.

equivalent(*other*)

Return True if *other* is indistinguishable from this bond, or False otherwise. *other* can be either a [Bond](#) or a [GroupBond](#) object.

get_bde()

estimate the bond dissociation energy in J/mol of the bond based on the order of the bond and the atoms involved in the bond

get_bond_string()

Represent the bond object as a string (eg. 'C#N'). The returned string is independent of the atom ordering, with the atom labels in alphabetical order (i.e. 'C-H' is possible but not 'H-C') :return: str

get_order_num()

returns the bond order as a number

get_order_str()

returns a string representing the bond order

get_other_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.

increment_order()

Update the bond as a result of applying a `CHANGE_BOND` action to increase the order by one.

is_benzene()

Return `True` if the bond represents a benzene bond or `False` if not.

is_double()

Return `True` if the bond represents a double bond or `False` if not.

is_double_or_triple()

Return `True` if the bond represents a double or triple bond or `False` if not.

is_hydrogen_bond()

Return `True` if the bond represents a hydrogen bond or `False` if not.

is_order(other_order)

Return `True` if the bond is of order `other_order` 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 switch to python 3.5+

is_quadruple()

Return `True` if the bond represents a quadruple bond or `False` if not.

is_reaction_bond()

Return `True` if the bond represents a reaction bond or `False` if not.

is_single()

Return `True` if the bond represents a single bond or `False` if not.

is_specific_case_of(other)

Return `True` if *self* is a specific case of *other*, or `False` otherwise. *other* can be either a [Bond](#) or a [GroupBond](#) object.

is_triple()

Return `True` if the bond represents a triple bond or `False` if not.

is_van_der_waals()

Return `True` if the bond represents a van der Waals bond or `False` if not.

set_order_num(*new_order*)

change the bond order with a number

set_order_str(*new_order*)

set the bond order using a valid bond-order character

sorting_key

Returns a sorting key for comparing Bond objects. Read-only

Bond types

The bond type simply indicates the order of a chemical bond. We define the following bond types:

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

rmgpy.molecule.Molecule

class rmgpy.molecule.Molecule

A representation of a molecular structure using a graph data type, extending the Graph class. Attributes are:

Attribute	Type	Description
<i>atoms</i>	list	A list of Atom objects in the molecule
<i>symmetry_number</i>	float	The (estimated) external + internal symmetry number of the molecule, modified for chirality
<i>multiplicity</i>	int	The multiplicity of this species, $\text{multiplicity} = 2 * \text{total_spin} + 1$
<i>reactive</i>	bool	True (by default) if the molecule participates in reaction families. It is set to False by the filtration functions if a non representative resonance structure was generated by a template reaction
<i>props</i>	dict	A list of properties describing the state of the molecule.
<i>inchi</i>	str	A string representation of the molecule in InChI
<i>smiles</i>	str	A string representation of the molecule in SMILES
<i>fingerprint</i>	str	A representation for fast comparison, set as molecular formula
<i>metal</i>	str	The metal of the metal surface the molecule is associated with
<i>facet</i>	str	The facet of the metal surface the molecule is associated with

A new molecule object can be easily instantiated by passing the *smiles* or *inchi* string representing the molecular structure.

add_atom(*atom*)

Add an *atom* to the graph. The atom is initialized with no bonds.

add_bond(*bond*)

Add a *bond* to the graph as an edge connecting the two atoms *atom1* and *atom2*.

add_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.

add_vertex(*vertex*)

Add a *vertex* to the graph. The vertex is initialized with no edges.

assign_atom_ids()

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

atom_ids_valid()

Checks to see if the atom IDs are valid in this structure

atoms

List of atoms contained in the current molecule.

Renames the inherited vertices attribute of Graph.

calculate_cp0()

Return the value of the heat capacity at zero temperature in J/mol*K.

calculate_cpinf()

Return the value of the heat capacity at infinite temperature in J/mol*K.

calculate_symmetry_number()

Return the symmetry number for the structure. The symmetry number includes both external and internal modes.

clear_labeled_atoms()

Remove the labels from all atoms in the molecule.

connect_the_dots(*critical_distance_factor=0.45, raise_atomtype_exception=True*)

Delete all bonds, and set them again based on the Atoms' coords. Does not detect bond type.

contains_labeled_atom(*label*)

Return True if the molecule contains an atom with the label *label* and False otherwise.

contains_surface_site()

Returns True iff the molecule contains an 'X' surface site.

copy(*deep=False*)

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.

copy_and_map()

Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from Graph.copy() method

count_aromatic_rings()

Count the number of aromatic rings in the current molecule, as determined by the benzene bond type. This is purely dependent on representation and is unrelated to the actual aromaticity of the molecule.

Returns an integer corresponding to the number of aromatic rings.

count_internal_rotors()

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 is considered to be a pivot of an internal rotor.

delete_hydrogens()

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(*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.

enumerate_bonds()

Count the number of each type of bond (e.g. 'C-H', 'C=C') present in the molecule :return: dictionary, with bond strings as keys and counts as values

find_h_bonds()

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

find_isomorphism(*other*, *initial_map*=None, *save_order*=False, *strict*=True)

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.

Parameters

- **initial_map** (*dict*, *optional*) – initial atom mapping to use
- **save_order** (*bool*, *optional*) – if True, reset atom order after performing atom isomorphism
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons

find_subgraph_isomorphisms(*other*, *initial_map*=None, *save_order*=False)

Returns True if *other* is subgraph isomorphic and False otherwise. Also returns the lists all of valid mappings. The *initial_map* 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.

fingerprint

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.

Use an expanded molecular formula to also enable sorting.

from_adjacency_list(*adjlist*, *saturate_h*=False, *raise_atomtype_exception*=True, *raise_charge_exception*=False, *check_consistency*=True)

Convert a string adjacency list *adjlist* to a molecular structure. Skips the first line (assuming it's a label) unless *withLabel* is False.

from_augmented_inchi(*aug_inchi*, *raise_atomtype_exception*=True)

Convert an Augmented InChI string *aug_inchi* to a molecular structure.

from_inchi(*inchistr*, *backend*='openbabel-first', *raise_atomtype_exception*=True)

Convert an InChI string *inchistr* to a molecular structure.

RDKit and Open Babel are the two backends used in RMG. It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'openbabel-first'(default), 'rdkit-first', 'rdkit', or 'openbabel'.

from_smarts(*smartsstr*, *raise_atomtype_exception=True*)

Convert a SMARTS string *smartsstr* to a molecular structure. Uses [RDKit](#) to perform the conversion. This Kekulizes everything, removing all aromatic atom types.

from_smiles(*smilesstr*, *backend='openbabel-first'*, *raise_atomtype_exception=True*)

Convert a SMILES string *smilesstr* to a molecular structure.

RDKit and Open Babel are the two backends used in RMG. It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'openbabel-first' (default), 'rdkit-first', 'rdkit', or 'openbabel'.

from_xyz(*atomic_nums*, *coordinates*, *critical_distance_factor=0.45*, *raise_atomtype_exception=True*)

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()

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(*keep_isomorphic=False*, *filter_structures=True*, *save_order=False*)

Returns a list of resonance structures of the molecule.

get_adatoms()

Get a list of adatoms in the molecule. :returns: A list containing the adatoms in the molecule :rtype: List(Atom)

get_all_cycles(*starting_vertex*)

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]

get_all_cycles_of_size(*size*)

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).

get_all_cyclic_vertices()

Returns all vertices belonging to one or more cycles.

get_all_edges()

Returns a list of all edges in the graph.

get_all_labeled_atoms()

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.

get_all_polycyclic_vertices()

Return all vertices belonging to two or more cycles, fused or spirocyclic.

get_all_simple_cycles_of_size(*size*)

Return a list of all non-duplicate monocyclic rings with length '*size*'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

get_aromatic_rings(*rings=None, save_order=False*)

Returns all aromatic rings as a list of atoms and a list of bonds.

Identifies rings using `Graph.get_smallest_set_of_smallest_rings()`, 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.

By default, the atom order will be sorted to get consistent results from different runs. The atom order can be saved when dealing with problems that are sensitive to the atom map.

get_bond(*atom1, atom2*)

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

get_bonds(*atom*)

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

get_charge_span()

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.

get_desorbed_molecules()

Get a list of desorbed molecules by desorbing the molecule from the surface.

Returns a list of Molecules. Each molecule's atoms will be labeled corresponding to the bond order with the surface: *1 - single bond *2 - double bond *3 - triple bond *4 - quadruple bond

get_deterministic_sssr()

Modified `Graph` method `get_smallest_set_of_smallest_rings` by sorting calculated cycles by short length and then high atomic number instead of just short length (for cases where multiple cycles with same length are found, `get_smallest_set_of_smallest_rings` outputs non-deterministically).

For instance, molecule with this smiles: C1CC2C3CSC(CO3)C2C1, will have non-deterministic output from `get_smallest_set_of_smallest_rings`, which leads to non-deterministic bicyclic 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.get_smallest_set_of_smallest_rings` method in new applications. Alternatively, consider using `Graph.get_relevant_cycles` 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.

get_disparate_cycles()

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

get_edge(*vertex1, vertex2*)

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

get_edges (*vertex*)

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

get_edges_in_cycle (*vertices*, *sort=False*)

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 `get_relevant_cycles` or `get_smallest_set_of_smallest_rings`.

get_element_count ()

Returns the element count for the molecule as a dictionary.

get_formula ()

Return the molecular formula for the molecule.

get_labeled_atoms (*label*)

Return the atoms in the molecule that are labeled.

get_largest_ring (*vertex*)

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 `get_polycycles` are not necessarily in order in the ring structure.

get_max_cycle_overlap ()

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.

get_molecular_weight ()

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

get_monocycles ()

Return a list of cycles that are monocyclic.

get_net_charge ()

Iterate through the atoms in the structure and calculate the net charge on the overall molecule.

get_nth_neighbor (*starting_atoms*, *distance_list*, *ignore_list=None*, *n=1*)

Recursively get the Nth nonHydrogen neighbors of the *starting_atoms*, and return them in a list. *starting_atoms* is a list of :class:Atom for which we will get the nth neighbor. *distance_list* is a list of integers, corresponding to the desired neighbor distances. *ignore_list* is a list of :class:Atom that have been counted in (n-1)th neighbor, and will not be returned. *n* is an integer, corresponding to the distance to be calculated in the current iteration.

get_num_atoms (*element=None*)

Return the number of atoms in molecule. If element is given, ie. “H” or “C”, the number of atoms of that element is returned.

get_polycycles ()

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.

get_radical_atoms ()

Return the atoms in the molecule that have unpaired electrons.

get_radical_count()

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.

get_relevant_cycles()

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

get_singlet_carbene_count()

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.

get_smallest_set_of_smallest_rings()

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_surface_sites()

Get a list of surface site atoms in the molecule. :returns: A list containing the surface site atoms in the molecule :rtype: List(Atom)

get_symmetry_number()

Returns the symmetry number of Molecule. First checks whether the value is stored as an attribute of Molecule. If not, it calls the calculate_symmetry_number method.

get_url()

Get a URL to the molecule's info page on the RMG website.

has_atom(atom)

Returns True if *atom* is an atom in the graph, or False if not.

has_bond(atom1, atom2)

Returns True if atoms *atom1* and *atom2* are connected by an bond, or False if not.

has_edge(vertex1, vertex2)

Returns True if vertices *vertex1* and *vertex2* are connected by an edge, or False if not.

has_halogen()

Return True if the molecule contains at least one halogen (F, Cl, Br, or I), or False otherwise.

has_lone_pairs()

Return True if the molecule contains at least one lone electron pair, or False otherwise.

has_vertex(vertex)

Returns True if *vertex* is a vertex in the graph, or False if not.

identify_ring_membership()

Performs ring perception and saves ring membership information to the Atom.props attribute.

inchi

InChI string for this molecule. Read-only.

is_aromatic()

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.

is_aryl_radical(*aromatic_rings=None, save_order=False*)

Return `True` if the molecule only contains aryl radicals, i.e., radical on an aromatic ring, or `False` otherwise. If no *aromatic_rings* provided, aromatic rings will be searched in-place, and this process may involve atom order change by default. Set *save_order* to `True` to force the atom order unchanged.

is_atom_in_cycle(*atom*)

Return `:data:True` if *atom* is in one or more cycles in the structure, and `:data:False` if not.

is_bond_in_cycle(*bond*)

Return `:data:True` if the bond between atoms *atom1* and *atom2* is in one or more cycles in the graph, or `:data:False` if not.

is_cyclic()

Return `True` if one or more cycles are present in the graph or `False` otherwise.

is_edge_in_cycle(*edge*)

Return `True` if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or `False` if not.

is_electron()

Returns `True` iff the molecule is nothing but an electron 'e'.

is_heterocyclic()

Returns `True` if the molecule is heterocyclic, or `False` if not.

is_identical(*other, strict=True*)

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.

If *strict=False*, performs the check ignoring electrons and resonance structures.

is_isomorphic(*other, initial_map=None, generate_initial_map=False, save_order=False, strict=True*)

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.

Parameters

- **initial_map** (*dict, optional*) – initial atom mapping to use
- **generate_initial_map** (*bool, optional*) – if `True`, initialize map by pairing atoms with same labels
- **save_order** (*bool, optional*) – if `True`, reset atom order after performing atom isomorphism
- **strict** (*bool, optional*) – if `False`, perform isomorphism ignoring electrons

is_linear()

Return True if the structure is linear and False otherwise.

is_mapping_valid(*other*, *mapping*, *equivalent*=True, *strict*=True)

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. If *strict* is True, electrons and bond orders are considered, and ignored if False.

is_multidentate()

Return True if the adsorbate contains at least two binding sites, or False otherwise.

is_proton()

Returns True iff the molecule is nothing but a proton 'H'.

is_radical()

Return True if the molecule contains at least one radical electron, or False otherwise.

is_subgraph_isomorphic(*other*, *initial_map*=None, *generate_initial_map*=False, *save_order*=False)

Returns True if *other* is subgraph isomorphic and False otherwise. The *initial_map* 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.

is_surface_site()

Returns True iff the molecule is nothing but a surface site 'X'.

is_vertex_in_cycle(*vertex*)

Return True if the given *vertex* is contained in one or more cycles in the graph, or False if not.

kekulize()

Kekulizes an aromatic molecule.

merge(*other*)

Merge two molecules so as to store them in a single [Molecule](#) object. The merged [Molecule](#) object is returned.

number_of_surface_sites()

Returns the number of surface sites in the molecule. e.g. 2 for a bidentate adsorbate

remove_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.

remove_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.

remove_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.

remove_h_bonds()

removes any present hydrogen bonds from the molecule

remove_van_der_waals_bonds()

Remove all van der Waals bonds.

remove_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.

replace_halogen_with_hydrogen(*raise_atomtype_exception=True*)

Replace all halogens in a molecule with hydrogen atoms. Changes self molecule object.

reset_connectivity_values()

Reset any cached connectivity information. Call this method when you have modified the graph.

restore_vertex_order()

reorder the vertices to what they were before sorting if you saved the order

saturate_radicals(*raise_atomtype_exception=True*)

Saturate the molecule by replacing all radicals with bonds to hydrogen atoms. Changes self molecule object.

saturate_unfilled_valence(*update=True*)

Saturate the molecule by adding H atoms to any unfilled valence

smiles

SMILES string for this molecule. Read-only.

sort_atoms()

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.

sort_cyclic_vertices(*vertices*)

Given a list of vertices comprising a cycle, sort them such that adjacent entries in the list are connected to each other. Warning: Assumes that the cycle is elementary, ie. no bridges.

sort_vertices(*save_order=False*)

Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

sorting_key

Returns a sorting key for comparing Molecule objects. Read-only

split()

Convert a single *Molecule* object containing two or more unconnected molecules into separate *class:Molecule* objects.

to_adjacency_list(*label=", remove_h=False, remove_lone_pairs=False, old_style=False*)

Convert the molecular structure to a string adjacency list.

to_augmented_inchi(*backend='rdkit-first'*)

Adds an extra layer to the InChI denoting the multiplicity of the molecule.

Separate layer with a forward slash character.

RDKit and Open Babel are the two backends used in RMG. It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'rdkit-first'(default), 'openbabel-first', 'rdkit', or 'openbabel'.

to_augmented_inchi_key(*backend='rdkit-first'*)

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.

RDKit and Open Babel are the two backends used in RMG. It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'rdkit-first'(default), 'openbabel-first', 'rdkit', or 'openbabel'.

to_group()

This method converts a list of atoms in a Molecule to a Group object.

to_inchi(*backend='rdkit-first'*)

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.

It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'rdkit-first'(default), 'openbabel-first', 'rdkit', or 'openbabel'.

to_inchi_key(*backend='rdkit-first'*)

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.

It is possible to use a single backend or try different backends in sequence. The available options for the *backend* argument: 'rdkit-first'(default), 'openbabel-first', 'rdkit', or 'openbabel'.

to_rdkit_mol(**args, **kwargs*)

Convert a molecular structure to a RDKit rdmol object.

to_single_bonds(*raise_atomtype_exception=True*)

Returns a copy of the current molecule, consisting of only single bonds.

This is useful for isomorphism comparison against something that was made via `from_xyz`, which does not attempt to perceive bond orders

to_smarts()

Convert a molecular structure to an SMARTS string. Uses [RDKit](#) to perform the conversion. Perceives aromaticity and removes Hydrogen atoms.

to_smiles()

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.

update(*log_species=True, raise_atomtype_exception=True, sort_atoms=True*)

Update the lone_pairs, charge, and atom types of atoms. Update multiplicity, and sort atoms (if *sort_atoms* is True) Does not necessarily update the connectivity values (which are used in isomorphism checks) If you need that, call `update_connectivity_values()`

update_atomtypes (*log_species=True, raise_exception=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 *raise_exception* is *False*, then the generic atomtype 'R' will be prescribed to any atom when *get_atomtype* fails. Currently used for resonance hybrid atom types.

update_connectivity_values ()

Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

update_lone_pairs ()

Iterate through the atoms in the structure and calculate the number of lone electron pairs, assuming a neutral molecule.

update_multiplicity ()

Update the multiplicity of a newly formed molecule.

rmgpy.molecule.GroupAtom**class** rmgpy.molecule.GroupAtom

An atom group. This class is based on the [Atom](#) class, except that it uses *atom types* instead of elements, and all attributes are lists rather than individual values. The attributes are:

Attribute	Type	Description
<i>atomtype</i>	list	The allowed atom types (as AtomType objects)
<i>radical_electrons</i>	list	The allowed numbers of radical electrons (as short integers)
<i>charge</i>	list	The allowed formal charges (as short integers)
<i>label</i>	str	A string label that can be used to tag individual atoms
<i>lone_pairs</i>	list	The number of lone electron pairs
<i>charge</i>	list	The partial charge of the atom
<i>site</i>	list	The allowed adsorption sites
<i>morphology</i>	list	The allowed morphologies
<i>props</i>	dict	Dictionary for storing additional atom properties
<i>reg_dim_atm</i>	list	List of atom types that are free dimensions in tree optimization
<i>reg_dim_u</i>	list	List of unpaired electron numbers that are free dimensions in tree optimization
<i>reg_dim_r</i>	list	List of inRing values that are free dimensions in tree optimization
<i>reg_dim_site</i>	list	List of sites that are free dimensions in tree optimization
<i>reg_dim_morphology</i>	list	List of morphologies 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 *radical_electrons*, and *charge* attributes are linked such that an atom must match values from the same index in each of these in order to match.

apply_action (*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](#).

copy ()

Return a deep copy of the [GroupAtom](#) object. Modifying the attributes of the copy will not affect the original.

count_bonds(*wildcards=False*)

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

equivalent(*other, strict=True*)

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.

has_wildcards()

Return True if the atom has wildcards in any of the attributes: atomtype, radical electrons, lone pairs, charge, and bond order. Returns "False" if no attribute has wildcards.

is_bonded_to_surface()

Return True if the atom is bonded to a surface GroupAtom X False if it is not

is_bromine()

Return True if the atom represents a bromine atom or False if not.

is_carbon()

Return True if the atom represents a carbon atom or False if not.

is_chlorine()

Return True if the atom represents a chlorine atom or False if not.

is_electron()

Return True if the atom represents a surface site or False if not.

is_fluorine()

Return True if the atom represents a fluorine atom or False if not.

is_lithium()

Return True if the atom represents a bromine atom or False if not.

is_nitrogen()

Return True if the atom represents a nitrogen atom or False if not.

is_oxygen()

Return True if the atom represents an oxygen atom or False if not.

is_proton()

Return True if the atom represents a surface site or False if not.

is_specific_case_of(*other*)

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.

is_sulfur()

Return True if the atom represents an sulfur atom or False if not.

is_surface_site()

Return True if the atom represents a surface site or False if not.

make_sample_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.radical_electrons, self.lone_pairs, and self.charge

reset_connectivity_values()

Reset the cached structure information for this vertex.

rmgpy.molecule.GroupBond**class rmgpy.molecule.GroupBond**

A bond group. This class is based on the [Bond](#) class, except that all attributes are lists rather than individual values. The allowed bond types are given [here](#). The attributes are:

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

apply_action(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()

Return a deep copy of the [GroupBond](#) object. Modifying the attributes of the copy will not affect the original.

equivalent(other)

Returns True if *other* is equivalent to *self* or False if not, where *other* can be either an [Bond](#) or an [GroupBond](#) object.

get_order_num()

returns the bond order as a list of numbers

get_order_str()

returns a list of strings representing the bond order

get_other_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.

is_benzene(wildcards=False)

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

is_double(wildcards=False)

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.

is_hydrogen_bond(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.

is_quadruple(*wildcards=False*)

Return True if the bond represents a quadruple 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 quadruple.

is_reaction_bond(*wildcards=False*)

Return True if the bond represents a reaction 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 reaction bonds.

is_single(*wildcards=False*)

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 switch to python 3.5+

is_specific_case_of(*other*)

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.

is_triple(*wildcards=False*)

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.

is_van_der_waals(*wildcards=False*)

Return True if the bond represents a van der Waals 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 van der Waals.

make_bond(*molecule, atom1, 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

set_order_num(*new_order*)

change the bond order with a list of numbers

set_order_str(*new_order*)

set the bond order using a valid bond-order character list

rmgpy.molecule.Group**class** rmgpy.molecule.Group

A representation of a molecular substructure group using a graph data type, extending the Graph class. The attributes are:

Attribute	Type	Description
<i>atoms</i>	list	Aliases for the <i>vertices</i> storing <i>GroupAtom</i>
<i>multiplicity</i>	list	Range of multiplicities accepted for the group
<i>props</i>	dict	Dictionary of arbitrary properties/flags classifying state of Group object
<i>metal</i>	list	List of metals accepted for the group
<i>facet</i>	list	List of facets accepted for the group

Corresponding alias methods to Molecule have also been provided.

add_atom(*atom*)

Add an *atom* to the graph. The atom is initialized with no bonds.

add_bond(*bond*)

Add a *bond* to the graph as an edge connecting the two atoms *atom1* and *atom2*.

add_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.

add_explicit_ligands()

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'

add_implicit_atoms_from_atomtype()

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

add_implicit_benzene()

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.

Note that it also works on other aromatic atomtypes like N5bd etc.

add_vertex(*vertex*)

Add a *vertex* to the graph. The vertex is initialized with no edges.

atoms

List of atoms contained in the current molecule.

Renames the inherited vertices attribute of Graph.

classify_benzene_carbons(*partners=None*)**Parameters**

- **group** – :class:Group with atoms to classify
- **partners** – dictionary of partnered up atoms, which must be a cbf atom

Some non-carbon 'benzene' type atoms (eg. N3b) are included and classified.

Returns: tuple with lists of each atom classification: cb_atom_list, cbf_atom_list, cbf_atom_list1, cbf_atom_list2, connected_cbfs

clear_labeled_atoms()

Remove the labels from all atoms in the molecular group.

clear_reg_dims()

clear regularization dimensions

contains_labeled_atom(*label*)

Return True if the group contains an atom with the label *label* and False otherwise.

contains_surface_site()

Returns True iff the group contains an 'X' surface site.

copy(*deep=False*)

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.

copy_and_map()

Create a deep copy of the current graph, and return the dict 'mapping'. Method was modified from Graph.copy() method

create_and_connect_atom(*atomtypes*, *connecting_atom*, *bond_orders*)

This method creates an non-radical, uncharged, :class:GroupAtom with specified list of atomtypes and connects it to one atom of the group, 'connecting_atom'. This is useful for making sample atoms.

Parameters

- **atomtypes** – list of atomtype labels (strs)
- **connecting_atom** – :class:GroupAtom that is connected to the new benzene atom
- **bond_orders** – list of bond Orders connecting new_atom and connecting_atom

Returns: the newly created atom

draw(*file_format*)

Use pydot to draw a basic graph of the group.

Use format to specify the desired output file_format, eg. 'png', 'svg', 'ps', 'pdf', 'plain', etc.

find_isomorphism(*other*, *initial_map=None*, *save_order=False*, *strict=True*)

Returns True if *other* is isomorphic and False otherwise, and the matching mapping. The *initial_map* 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.

find_subgraph_isomorphisms(*other*, *initial_map=None*, *save_order=False*)

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 *initial_map* 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.

from_adjacency_list(*adjlist*, *check_consistency=True*)

Convert a string adjacency list *adjlist* to a molecular structure. Skips the first line (assuming it's a label) unless *withLabel* is False.

get_all_cycles(*starting_vertex*)

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]

get_all_cycles_of_size(*size*)

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).

get_all_cyclic_vertices()

Returns all vertices belonging to one or more cycles.

get_all_edges()

Returns a list of all edges in the graph.

get_all_labeled_atoms()

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.

get_all_polycyclic_vertices()

Return all vertices belonging to two or more cycles, fused or spirocyclic.

get_all_simple_cycles_of_size(size)

Return a list of all non-duplicate monocyclic rings with length 'size'.

Naive approach by eliminating polycyclic rings that are returned by `getAllCyclicsOfSize`.

get_bond(atom1, atom2)

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

get_bonds(atom)

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

get_disparate_cycles()

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

get_edge(vertex1, vertex2)

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

get_edges(vertex)

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

get_edges_in_cycle(vertices, sort=False)

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 `get_relevant_cycles` or `get_smallest_set_of_smallest_rings`.

get_element_count()

Returns the element count for the molecule as a dictionary. Wildcards are not counted as any particular element.

get_extensions(r=None, r_bonds=None, r_un=None, basename="", atm_ind=None, atm_ind2=None, n_splits=None)

generate all allowed group extensions and their complements note all atomtypes except for elements and r/r!H's must be removed

get_labeled_atoms(label)

Return the atom in the group that is labeled with the given *label*. Raises `ValueError` if no atom in the group has that label.

get_largest_ring(*vertex*)

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 `get_polycycles` are not necessarily in order in the ring structure.

get_max_cycle_overlap()

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.

get_monocycles()

Return a list of cycles that are monocyclic.

get_net_charge()

Iterate through the atoms in the group and calculate the net charge

get_polycycles()

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.

get_relevant_cycles()

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

get_smallest_set_of_smallest_rings()

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_surface_sites()

Get a list of surface site GroupAtoms in the group. :returns: A list containing the surface site GroupAtoms in the molecule :rtype: List(GroupAtom)

has_atom(*atom*)

Returns True if *atom* is an atom in the graph, or False if not.

has_bond(*atom1*, *atom2*)

Returns True if atoms *atom1* and *atom2* are connected by an bond, or False if not.

has_edge(*vertex1*, *vertex2*)

Returns True if vertices *vertex1* and *vertex2* are connected by an edge, or False if not.

has_vertex(*vertex*)

Returns True if *vertex* is a vertex in the graph, or False if not.

has_wildcards()

This function is a Group level wildcards checker.

Returns a ‘True’ if any of the atoms in this group has wildcards.

is_aromatic_ring()

This method returns a boolean telling if the group has a 5 or 6 cyclic with benzene bonds exclusively

is_benzene_explicit()

Returns: 'True' if all Cb, Cbf (and other 'b' atoms) are in completely explicitly stated benzene rings.

Otherwise return 'False'

is_cyclic()

Return True if one or more cycles are present in the graph or False otherwise.

is_edge_in_cycle(*edge*)

Return True if the edge between vertices *vertex1* and *vertex2* is in one or more cycles in the graph, or False if not.

is_electron()

Returns True iff the group is an electron

is_identical(*other*, *save_order=False*)

Returns True if *other* is identical and False otherwise. The function *is_isomorphic* respects wildcards, while this function does not, make it more useful for checking groups to groups (as opposed to molecules to groups)

is_isomorphic(*other*, *initial_map=None*, *generate_initial_map=False*, *save_order=False*, *strict=True*)

Returns True if two graphs are isomorphic and False otherwise. The *initial_map* 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.

is_mapping_valid(*other*, *mapping*, *equivalent=True*, *strict=True*)

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. If *strict* is True, electrons and bond orders are considered, and ignored if False.

is_proton()

Returns True iff the group is a proton

is_subgraph_isomorphic(*other*, *initial_map=None*, *generate_initial_map=False*, *save_order=False*)

Returns True if *other* is subgraph isomorphic and False otherwise. In other words, return True if *self* is more specific than *other*. The *initial_map* 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.

is_surface_site()

Returns True iff the group is nothing but a surface site 'X'.

is_vertex_in_cycle(*vertex*)

Return True if the given *vertex* is contained in one or more cycles in the graph, or False if not.

make_sample_molecule()

Returns: A sample class :Molecule: from the group

merge(*other*)

Merge two groups so as to store them in a single *Group* object. The merged *Group* object is returned.

merge_groups(*other*, *keep_identical_labels=False*)

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 if *keep_identical_labels=True* *merge_groups* will not try to merge atoms with the same labels

pick_wildcards()

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

remove_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.

remove_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.

remove_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.

remove_van_der_waals_bonds()

Remove all bonds that are definitely only van der Waals bonds.

remove_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.

reset_connectivity_values()

Reset any cached connectivity information. Call this method when you have modified the graph.

reset_ring_membership()

Resets ring membership information in the GroupAtom.props attribute.

restore_vertex_order()

reorder the vertices to what they were before sorting if you saved the order

sort_atoms()

Sort the atoms in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

sort_by_connectivity(*atom_list*)**Parameters**

atom_list – input list of atoms

Returns: a sorted list of atoms where each atom is connected to a previous atom in the list if possible

sort_cyclic_vertices(*vertices*)

Given a list of vertices comprising a cycle, sort them such that adjacent entries in the list are connected to each other. Warning: Assumes that the cycle is elementary, ie. no bridges.

sort_vertices(*save_order=False*)

Sort the vertices in the graph. This can make certain operations, e.g. the isomorphism functions, much more efficient.

specify_atom_extensions(*i*, *basename*, *r*)

generates extensions for specification of the type of atom defined by a given atomtype or set of atomtypes

specify_bond_extensions(*i*, *j*, *basename*, *r_bonds*)

generates extensions for the specification of bond order for a given bond

specify_external_new_bond_extensions(*i*, *basename*, *r_bonds*)

generates extensions for the creation of a bond (of undefined order) between an atom and a new atom that is not H

specify_internal_new_bond_extensions(*i*, *j*, *n_splits*, *basename*, *r_bonds*)

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

specify_ring_extensions(*i*, *basename*)

generates extensions for specifying if an atom is in a ring

specify_unpaired_extensions(*i*, *basename*, *r_un*)

generates extensions for specification of the number of electrons on a given atom

split()

Convert a single *Group* object containing two or more unconnected groups into separate class:*Group* objects.

standardize_atomtype()

This function changes the atomtypes in a group if the atom must be a specific atomtype based on its bonds and valency.

Currently only standardizes oxygen, carbon and sulfur ATOMTYPES

We also only check when there is exactly one atomtype, one bondType, one radical setting. For any group where there are wildcards or multiple attributes, we cannot apply this check.

In the case where the atomtype is ambiguous based on bonds and valency, this function will not change the type.

Returns a 'True' if the group was modified otherwise returns 'False'

standardize_group()

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'

to_adjacency_list(*label*="")

Convert the molecular structure to a string adjacency list.

update_charge()

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.

update_connectivity_values()

Update the connectivity values for each vertex in the graph. These are used to accelerate the isomorphism checking.

update_fingerprint()

Update the molecular fingerprint used to accelerate the subgraph isomorphism checks.

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

- **Multidentate adsorbates only**

- `generate_adsorbate_shift_down_resonance_structures`: shift 2 electrons from a C=#C bond to the X-C bond
- `generate_adsorbate_shift_up_resonance_structures`: shift 2 electrons from a X=#C bond to a C-C bond
- `generate_adsorbate_conjugate_resonance_structures`: shift 2 electrons in a conjugate pi system for bridged X-C-C-C-X adsorbates

`rmgpy.molecule.resonance.analyze_molecule(mol, save_order=False)`

Identify key features of molecule important for resonance structure generation. `save_order` is used to maintain the atom order, when analyzing the molecule, defaults to False.

Returns a dictionary of features.

`rmgpy.molecule.resonance.generate_N5dc_radical_resonance_structures(mol)`

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(mol)`

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 lone_pairs 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(mol)`

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 lone_pairs 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 loses a radical. In direction 2 atom1 <gains> a lone pair, loses a radical, and atom2 gains a radical)

`rmgpy.molecule.resonance.generate_adj_lone_pair_radical_resonance_structures(mol)`

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_adsorbate_conjugate_resonance_structures(mol)`

Generate all of the resonance structures formed by the shift of two electrons in a conjugated pi bond system of a bidentate adsorbate with a bridging atom in between.

Example XCHCHXC: [X]#CC=C[X] <=> [X]=C=CC=[X] (where '#' denotes a triple bond, '=' denotes a double bond)

`rmgpy.molecule.resonance.generate_adsorbate_shift_down_resonance_structures(mol)`

Generate all of the resonance structures formed by the shift a pi bond between two C-C atoms to both X-C bonds. Example XCHXCH: [X]C=C[X] <=> [X]=CC=[X] (where '=' denotes a double bond)

`rmgpy.molecule.resonance.generate_adsorbate_shift_up_resonance_structures(mol)`

Generate all of the resonance structures formed by the shift of two electrons from X-C bonds to increase the bond order between two C-C atoms by 1. Example XCHXCH: [X]=CC=[X] <=> [X]C=C[X] (where '=' denotes a double bond, '#' denotes a triple bond)

`rmgpy.molecule.resonance.generate_allyl_delocalization_resonance_structures(mol)`

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(mol, aromatic_bonds=None, copy=True, save_order=False)`

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(mol)`

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(mol, save_order=False)`

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(mol, saturate_h=False)`

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 lone_pairs and partial charges in a molecule wrong.

WIP: do not generate aromatic resonance isomers.

`rmgpy.molecule.resonance.generate_kekule_structure(mol)`

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(mol)`

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(mol,`
`features=None,`
`save_order=False)`

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(mol, clar_structures=True,`
`keep_isomorphic=False,`
`filter_structures=True, save_order=False)`

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(features=None)`

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`

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.

update()

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`

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.

kekulize()

Attempts to kekulize a single aromatic ring in a molecule.

Returns True if successful, and False otherwise.

process_bonds()

Create AromaticBond objects for each endocyclic bond.

update()

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(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.filtration`

This module contains functions for filtering a list of `Molecules` representing a single `Species`, keeping only the representative structures. Relevant for filtration of negligible mesomerism contributing structures.

The rules this module follows are (by order of importance):

1. Minimum overall deviation from the Octet Rule (elaborated for Dectet for sulfur as a third row element)
2. Additional charge separation is only allowed for radicals if it makes a new radical site in the species
3. If a structure must have charge separation, negative charges will be assigned to more electronegative atoms, whereas positive charges will be assigned to less electronegative atoms (charge stabilization)
4. Opposite charges will be as close as possible to one another, and vice versa (charge stabilization)

(inspired by http://web.archive.org/web/20140310074727/http://www.chem.ucla.edu/~harding/tutorials/resonance/imp_res_str.html which is quite like http://www.chem.ucla.edu/~harding/IGOC/R/resonance_contributor_preference_rules.html)

`rmgpy.molecule.filtration.aromaticity_filtration(mol_list, features)`

Returns a filtered list of molecules based on heuristics for determining representative aromatic resonance structures.

For monocyclic aromatics, Kekule structures are removed, with the assumption that an equivalent aromatic structure exists. Non-aromatic structures are maintained if they present new radical sites. Instead of explicitly checking the radical sites, we only check for the SDSDS bond motif since radical delocalization will disrupt that pattern.

For polycyclic aromatics, structures without any benzene bonds are removed. The idea is that radical delocalization into the aromatic pi system is unfavorable because it disrupts aromaticity. Therefore, structures where the radical is delocalized so far into the molecule such that none of the rings are aromatic anymore are not representative. While this isn't strictly true, it helps reduce the number of representative structures by focusing on the most important ones.

`rmgpy.molecule.filtration.charge_filtration(filtered_list, charge_span_list)`

Returns a new `filtered_list`, filtered based on `charge_span_list`, electronegativity and proximity considerations. If structures with an additional charge layer introduce reactive sites (i.e., radicals or multiple bonds) they will also be considered. For example:

- Both of NO₂'s resonance structures will be kept: $[O]N=O \rightleftharpoons O=[N+].[O-]$
- NCO will only have two resonance structures $[N.]#C=O \rightleftharpoons N\#C[O.]$, and will lose the third structure which has the same octet deviation, has a charge separation, but the radical site has already been considered: $[N+.]#C[O-]$
- CH₂NO keeps all three structures, since a new radical site is introduced: $[CH_2.]N=O \rightleftharpoons C=N[O.] \rightleftharpoons C=[N+].[O-]$
- NH₂CHO has two structures, one of which is charged since it introduces a multiple bond: $NC=O \rightleftharpoons [NH_2+]=C[O-]$

However, if the species is not a radical, or multiple bonds do not alter, we only keep the structures with the minimal charge span. For example:

- NSH will only keep the N#S form and not $[N-]=[SH+]$
- The following species will lose two thirds of its resonance structures, which are charged: $CS(=O)SC \rightleftharpoons CS(=O)\#SC \rightleftharpoons C[S+](O-)[SC] \rightleftharpoons CS([O-])=[S+]C \rightleftharpoons C[S+](O-)\#SC \rightleftharpoons C[S+](=O)=[S-]C$
- Azide is known to have three resonance structures: $[NH-][N+]\#N \rightleftharpoons N=[N+]=[N-] \rightleftharpoons [NH+]\#[N+][N-]$; here we filter the third one out due to the higher charge span, which does not contribute to reactivity in RMG

`rmgpy.molecule.filtration.check_reactive(filtered_list)`

Check that there's at least one reactive structure in the returned list. If not, raise an error (does not return anything)

`rmgpy.molecule.filtration.filter_structures(mol_list, mark_unreactive=True, allow_expanded_octet=True, features=None, save_order=False)`

We often get too many resonance structures from the combination of all rules, particularly for species containing lone pairs. This function filters them out by minimizing the number of C/N/O/S atoms without a full octet. If `save_order` is `True` the atom order is reset after performing atom isomorphism.

`rmgpy.molecule.filtration.find_unique_sites_in_charged_list(mol, rad_sorting_list, mul_bond_sorting_list)`

A helper function for reactive site discovery in charged species

`rmgpy.molecule.filtration.get_charge_span_list(mol_list)`

Returns the a list of charge spans for a respective list of :class:Molecule objects This is also calculated in the `octet_filtration()` function along with the octet filtration process

`rmgpy.molecule.filtration.get_octet_deviation(mol, allow_expanded_octet=True)`

Returns the octet deviation for a :class:Molecule object if `allow_expanded_octet` is `True` (by default), then the function also considers dectet for third row elements (currently sulfur is the only hypervalence third row element in RMG)

`rmgpy.molecule.filtration.get_octet_deviation_list(mol_list, allow_expanded_octet=True)`

Returns the a list of octet deviations for a respective list of :class:Molecule objects

`rmgpy.molecule.filtration.ionic_bond_filtration(mol_list)`

Returns a filtered list removing structures that put a negative charge on lithium which is ionically bonded and thus cannot donate/recieve electrons covalently

`rmgpy.molecule.filtration.mark_unreactive_structures(filtered_list, mol_list, save_order=False)`

Mark selected structures in `filtered_list` with the `Molecule.reactive` flag set to *False* (it is *True* by default) Changes the `filtered_list` object, and does not return anything. If `save_order` is *True* the atom order is reset after performing atom isomorphism.

`rmgpy.molecule.filtration.octet_filtration(mol_list, octet_deviation_list)`

Returns a filtered list based on the `octet_deviation_list`. Also computes and returns a `charge_span_list`. Filtering using the octet deviation criterion rules out most unrepresentative structures. However, since some charge-strained species are still kept (e.g., [NH]N=S=O <-> [NH+][N+][S-][O-]), we also generate during the same loop a `charge_span_list` to keep track of the charge spans. This is used for further filtering.

`rmgpy.molecule.filtration.stabilize_charges_by_electronegativity(mol_list, allow_empty_list=False)`

Only keep structures that obey the electronegativity rule. If a structure must have charge separation, negative charges will be assigned to more electronegative atoms, and vice versa. If `allow_empty_list` is set to *False* (default), this function will not return an empty list. If it is set to *True* and all structures in `mol_list` violate the electronegativity heuristic, the original `mol_list` is returned (examples: [C-]#[O+], CS, [NH+][C-], [OH+]=[N-], [C-][S+]=C violate this heuristic).

`rmgpy.molecule.filtration.stabilize_charges_by_proximity(mol_list)`

Only keep structures that obey the charge proximity rule. Opposite charges will be as close as possible to one another, and vice versa.

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(path)`

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(path)`

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(path)`

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(atom_indices, mol)`

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(atom1)`

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:

- $N=[N+](O)(O-)\rightleftharpoons N=[N+](O-)(O)$, 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(atom1)`

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:

- $[:NH-]-[CH2+]\rightleftharpoons[:NH]=[CH2]$
- $[:N]\#[CH]\rightleftharpoons[:N-]=[CH+]$
- other examples: $S\#N$, $N\#[S]$, $O=S([O])=O$

Direction “1” is the direction <increasing> the bond order as in $[:NH-]-[CH2+]\rightleftharpoons[:NH]=[CH2]$ Direction “2” is the direction <decreasing> the bond order as in $[:NH]=[CH2]\rightleftharpoons[:NH-]-[CH2+]$ (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(atom1)`

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(atom1)`

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 <loses> a lone pair, gains a radical, and atom2 loses a radical. In direction 2 atom1 <gains> a lone pair, loses a radical, and atom2 gains a radical)

`rmgpy.molecule.pathfinder.find_adsorbate_conjugate_delocalization_paths(atom1)`

Find all multidentate adsorbates which have a bonding configuration X-C-C-C-X. Examples:

- XCHCHXCH/XCHCHXC, where X is the surface site. The adsorption site X is always placed on the left-hand side of the adatom and every adatom is bonded to only one surface site X.

In this transition atom1 and atom5 are surface sites while atom2, atom3, and atom4 are carbon or nitrogen atoms.

`rmgpy.molecule.pathfinder.find_adsorbate_delocalization_paths(atom1)`

Find all multidentate adsorbates which have a bonding configuration X-C-C-X. Examples:

- XCXC, XCHXCH, XCXCH, where X is the surface site. The adsorption site X is always placed on the left-hand side of the adatom and every adatom is bonded to only one surface site X.

In this transition atom1 and atom4 are surface sites while atom2 and atom3 are carbon or nitrogen atoms.

`rmgpy.molecule.pathfinder.find_allyl_delocalization_paths(atom1)`

Find all the delocalization paths allyl to the radical center indicated by atom1.

`rmgpy.molecule.pathfinder.find_allyl_end_with_charge(start)`

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(start, end)`

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(start)`

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(atom1)`

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 ($\text{N}\#[\text{N}^+][\text{O}^-] \leftrightarrow [\text{N}^-]=[\text{N}^+]=\text{O}$)
- Azide ($\text{N}\#[\text{N}^+][\text{NH}^-] \leftrightarrow [\text{N}^-]=[\text{N}^+]=\text{N} \leftrightarrow [\text{N}^-][\text{N}^+]\#[\text{NH}^+]$)
- N#N group on sulfur ($\text{O}[\text{S}^-](\text{O})[\text{N}^+]\#\text{N} \leftrightarrow \text{OS}(\text{O})=[\text{N}^+]=[\text{N}^-] \leftrightarrow \text{O}[\text{S}^+](\text{O})\#[\text{N}^+][\text{N}^-]$)
- $\text{N}[\text{N}^+](\text{O}^-)=\text{O} \Leftrightarrow \text{N}[\text{N}^+](=\text{O})[\text{O}^-]$, these structures are isomorphic but not identical, this transition is important for correct degeneracy calculations

`rmgpy.molecule.pathfinder.is_atom_able_to_gain_lone_pair(atom)`

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(atom)`

Helper function Returns True if atom is N/O/S and is able to <lose> 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.debug_rdkit_mol(rdmol, level=20)`

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.from_ob_mol(mol, obmol, raise_atomtype_exception=True)`

Convert a OpenBabel Mol object *obmol* to a molecular structure. Uses [OpenBabel](#) to perform the conversion.

It estimates radical placement based on undervalence of atoms, and assumes overall spin multiplicity is radical count + 1

`rmgpy.molecule.converter.from_rdkit_mol(mol, rdkitmol, raise_atomtype_exception=True)`

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.to_ob_mol(mol, return_mapping=False, save_order=False)`

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

`rmgpy.molecule.converter.to_rdkit_mol(mol, remove_h=True, return_mapping=False, sanitize=True, save_order=False)`

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

If `return_mapping==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.from_augmented_inchi(mol, aug_inchi, raise_atomtype_exception=True)`

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.from_inchi(mol, inchi_str, backend='openbabel-first',
                                       raise_atomtype_exception=True)
```

Convert an InChI string *inchi_str* to a molecular structure. Uses a user-specified backend for conversion, currently supporting 'openbabel-first' (default), rdkit-first, rdkit, and openbabel.

```
rmgpy.molecule.translator.from_smarts(mol, smarts_str, backend='rdkit',
                                       raise_atomtype_exception=True)
```

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

```
rmgpy.molecule.translator.from_smiles(mol, smiles_str, backend='openbabel-first',
                                       raise_atomtype_exception=True)
```

Convert a SMILES string *smiles_str* to a molecular structure. Uses a user-specified backend for conversion, currently supporting openbabel-first (default), rdkit-first, rdkit and openbabel.

```
rmgpy.molecule.translator.to_inchi(mol, backend='rdkit-first', aug_level=0)
```

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

- **backend** (*backend choice of*) –
- **'rdkit-first'** (*default*) –
- **augmentation** (*aug_level level of*) –
- **0** –
- **1** –
- **2** (*or*) –

```
rmgpy.molecule.translator.to_inchi_key(mol, backend='rdkit-first', aug_level=0)
```

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

- **backend** (*backend choice of*) –
- **'rdkit-first'** (*default*) –
- **augmentation** (*aug_level level of*) –
- **0** –
- **1** –
- **2** (*or*) –

`rmgpy.molecule.translator.to_smarts(mol, backend='rdkit')`

Convert a molecular structure to an SMARTS string. Uses [RDKit](#) to perform the conversion. Perceives aromaticity and removes Hydrogen atoms.

`rmgpy.molecule.translator.to_smiles(mol, backend='default')`

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: https://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 can be: `multiplicity`, `metal` and `facet`.

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.

`metal` and `facet` work similarly and will correspond to lines like `metal Fe`, `metal [Fe,Cu,Ag]`, `facet 111`, `facet [111,211,110]`.

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>] [s<site>] [m
↪<morphology>] <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)
- s the site type a site atom is e.g. s "fcc"
- m the morphology of a site atom e.g. m "terrace"

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], c[0,+1,+2,+3,+4], or s["hcp","fcc"] or may be a wildcard ``x which matches any valid value, eg. px is the same as p[0,1,2,3,4,...] and cx is the same as c[...,-4,-3,-2,-1,0,+1,+2,+3,+4,...]. Lists must be enclosed in square brackets, and separated by commas, without spaces. If lone pairs or formal charges are omitted from a group definition, the wildcard is assumed.

The last set of tokens is the list of bonds. To indicate a bond, place the number of the atom at the other end of the bond and the bond type within curly braces and separated by a comma, e.g. {2,S}. Multiple bonds from the same atom should be separated by whitespace.

Note: You must take care to make sure each bond is listed on the lines of *both* atoms in the bond, and that these entries have the same bond type. RMG will raise an exception if it encounters such an invalid adjacency list.

When writing a molecular substructure pattern, you may specify multiple elements, radical counts, and bond types as a comma-separated list inside square brackets. For example, to specify any carbon or oxygen atom, use the syntax [C,O]. For a single or double bond to atom 2, write {2,[S,D]}.

Atom types such as R!H or Cdd may also be used as a shorthand. (Atom types like Cdd can also be used in full molecules, but this use is discouraged, as RMG can compute them automatically for full molecules.)

Below is an example adjacency list, for 1,3-hexadiene, with the weakest bond in the molecule labeled with *1 and *2. Note that hydrogen atoms can be omitted if desired, as their presence is inferred, provided that unpaired electrons, lone pairs, and charges are all correctly defined:

```
HXD13
multiplicity 1
```

(continues on next page)

(continued from previous page)

```

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	
	vdW Van der Waals bond	
	H Hydrogen bond	
	R Reaction bond	

`rmgpy.molecule.adjlist.from_adjacency_list(adjlist, group=False, saturate_h=False, check_consistency=True)`

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

`rmgpy.molecule.adjlist.to_adjacency_list(atoms, multiplicity, metal="", facet="", label=None, group=False, remove_h=False, remove_lone_pairs=False, old_style=False)`

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

rmgpy.molecule.symmetry

`rmgpy.molecule.symmetry.calculate_atom_symmetry_number(molecule, atom)`

Return the symmetry number centered at *atom* in the structure. The *atom* of interest must not be in a cycle.

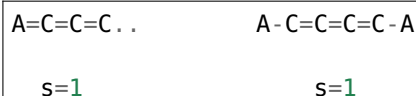
`rmgpy.molecule.symmetry.calculate_bond_symmetry_number(molecule, atom1, atom2)`

Return the symmetry number centered at *bond* in the structure.

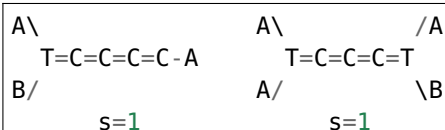
`rmgpy.molecule.symmetry.calculate_axis_symmetry_number(molecule)`

Get the axis symmetry number correction. The “axis” refers to a series of two or more cumulated double bonds (e.g. C=C=C, etc.). Corrections for single C=C bonds are handled in `getBondSymmetryNumber()`.

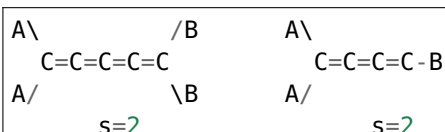
Each axis (C=C=C) has the potential to double the symmetry number. If an end has 0 or 1 groups (eg. =C=CJJ or =C=C-R) then it cannot alter the axis symmetry and is disregarded:



If an end has 2 groups that are different then it breaks the symmetry and the symmetry for that axis is 1, no matter what's at the other end:



If you have one or more ends with 2 groups, and neither end breaks the symmetry, then you have an axis symmetry number of 2:



`rmgpy.molecule.symmetry.calculate_cyclic_symmetry_number(molecule)`

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.calculate_symmetry_number(molecule)`

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, file_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*, *file_format*, *target=None*)

Draw the given *molecule* using the given image *file_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, file_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, file_format, path=None*)

Draw the given *reaction* using the given image *file_format* - pdf, svg, ps, or png. If *path* is given, the drawing is saved to that location on disk.

This function returns the Cairo surface and context used to create the drawing, as well as a bounding box for the molecule being drawn as the tuple (*left, top, width, height*).

1.7 Pressure dependence (rmgpy.pdep)

The *rmgpy.pdep* subpackage provides functionality for calculating the pressure-dependent rate coefficients $k(T, P)$ for unimolecular reaction networks.

A unimolecular reaction network is defined by a set of chemically reactive molecular configurations - local minima on a potential energy surface - divided into unimolecular isomers and bimolecular reactants or products. In our vernacular, reactants can associate to form an isomer, while such association is neglected for products. These configurations are connected by chemical reactions to form a network; these are referred to as *path* reactions. The system also consists of an excess of inert gas M, representing a thermal bath; this allows for neglecting all collisions other than those between an isomer and the bath gas.

An isomer molecule at sufficiently high internal energy can be transformed by a number of possible events:

- The isomer molecule can collide with any other molecule, resulting in an increase or decrease in energy
- The isomer molecule can isomerize to an adjacent isomer at the same energy
- The isomer molecule can dissociate into any directly connected bimolecular reactant or product channel

It is this competition between collision and reaction events that gives rise to pressure-dependent kinetics.

1.7.1 Collision events

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

1.7.2 Reaction events

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

1.7.3 Pressure-dependent reaction networks

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

1.7.4 The master equation

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

1.7.5 Master equation reduction methods

Function	Description
<code>msc.apply_modified_strong_collision_method()</code>	Reduce the master equation to phenomenological rate coefficients $k(T, P)$ using the modified strong collision method
<code>rs.apply_reservoir_state_method()</code>	Reduce the master equation to phenomenological rate coefficients $k(T, P)$ using the reservoir state method
<code>cse.apply_chemically_significant_eigenvalues_method()</code>	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
<i>alpha0</i>	The average energy transferred in a deactivating collision at the reference temperature
<i>T0</i>	The reference temperature
<i>n</i>	The temperature exponent

Based around the collisional energy transfer probability function

$$P(E, E') = C(E') \exp\left(-\frac{E' - E}{\alpha}\right) \quad E < E'$$

where the parameter $\alpha = \langle \Delta E_d \rangle$ represents the average energy transferred in a deactivating collision. This is the most commonly-used collision model, simply because it only has one parameter to determine. The parameter α is specified using the equation

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

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

T0

The reference temperature.

alpha0

The average energy transferred in a deactivating collision at the reference temperature.

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

calculate_collision_efficiency(*self*, *double T*, *ndarray e_list*, *ndarray j_list*, *ndarray dens_states*, *double E0*, *double e_reac*)

Calculate an efficiency factor for collisions, particularly useful for the modified strong collision method. The collisions involve the given *species* with density of states *dens_states* corresponding to energies *e_list* in J/mol, ground-state energy *E0* in kJ/mol, and first reactive energy *e_reac* in kJ/mol. The collisions occur at temperature *T* in K and are described by the average energy transferred in a deactivating collision *d_e_down* in kJ/mol. The algorithm here is implemented as described by A. Y. Chang, J. W. Bozzelli, and A. M. Dean. *Z. Phys. Chem.* **214**, p. 1533-1568 (2000). doi: [10.1524/zpch.2000.214.11.1533](https://doi.org/10.1524/zpch.2000.214.11.1533)

generate_collision_matrix(*self*, *double T*, *ndarray dens_states*, *ndarray e_list*, *ndarray j_list=None*)

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

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

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

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

n

'double'

Type

n

Reaction events

Microcanonical rate coefficients

`rmgpy.pdep.reaction.calculate_microcanonical_rate_coefficient`(*reaction*, *ndarray e_list*, *ndarray j_list*, *ndarray reac_dens_states*, *ndarray prod_dens_states=None*, *double T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.reaction.apply_rrkm_theory`(*transition_state*, *ndarray e_list*, *ndarray j_list*, *ndarray dens_states*)

Calculate the microcanonical rate coefficient for a reaction using RRKM theory, where *transition_state* is the transition state of the reaction, *e_list* is the array of energies in J/mol at which to evaluate the microcanonical rate, and *dens_states* 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.reaction.apply_inverse_laplace_transform_method(transition_state, Arrhenius kinetics, ndarray e_list, ndarray j_list, ndarray dens_states, 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, *e_list* is the array of energies in kJ/mol at which to evaluate the microcanonical rate, and *dens_states* is the density of states of the reactant. The temperature *T* in K is not required, and is only used when the temperature exponent of the Arrhenius expression is negative (for which the inverse transform is undefined).

The inverse Laplace transform method exploits the following relationship to determine the microcanonical rate coefficient:

$$\mathcal{L}[k(E)\rho(E)] = \int_0^\infty k(E)\rho(E)e^{-E/k_B T} dE = k_\infty(T)Q(T)$$

Given a high-pressure limit rate coefficient $k_\infty(T)$ represented as an Arrhenius expression with positive n and E_a , the microcanonical rate coefficient $k(E)$ can be determined via an inverse Laplace transform. For $n = 0$ the transform can be defined analytically:

$$k(E) = A \frac{\rho(E - E_a)}{\rho(E)} \quad (n = 0)$$

For $n > 0$ the transform is defined numerically. For $n < 0$ or $E_a < 0$ the transform is not defined; in this case we approximate by simply lumping the T^n or $e^{-E_a/RT}$ terms into the preexponential factor, and use a different $k(E)$ at each temperature.

The ILT method does not require detailed transition state information, but only the high-pressure limit kinetics. However, it assumes that (1) $k_\infty(T)$ is valid over the temperature range from zero to infinity and (2) the activation energy E_a is physically identical to the reaction barrier $E_0^\ddagger - E_0$.

rmgpy.pdep.Configuration

class `rmgpy.pdep.Configuration`

A representation of a molecular configuration on a potential energy surface.

E0

The ground-state energy of the configuration in J/mol.

calculate_collision_frequency(*T*, *P*, *bath_gas*)

Return the value of the collision frequency in Hz at the given temperature *T* in K and pressure *P* in Pa. If a dictionary *bath_gas* of bath gas species and corresponding mole fractions is given, the collision parameters of the bath gas species will be averaged with those of the species before computing the collision frequency.

Only the Lennard-Jones collision model is currently supported.

calculate_density_of_states(*e_list*, *active_j_rotor*=True, *active_k_rotor*=True, *rmgmode*=False)

Calculate the density (and sum) of states for the configuration at the given energies above the ground state *e_list* in J/mol. The *active_j_rotor* and *active_k_rotor* flags control whether the J-rotor and/or K-rotor are treated as active (and therefore included in the density and sum of states). The computed density and sum of states arrays are stored on the object for future use.

cleanup()

Delete intermediate arrays used in computing k(T,P) values.

generate_collision_matrix(*T*, *dens_states*, *e_list*, *j_list=None*)

Return the collisional energy transfer probabilities matrix for the configuration at the given temperature *T* in K using the given energies *e_list* in kJ/mol and total angular momentum quantum numbers *j_list*. The density of states of the configuration *dens_states* in mol/kJ is also required.

get_enthalpy(*T*)

Return the enthalpy in kJ/mol at the specified temperature *T* in K.

get_entropy(*T*)

Return the entropy in J/mol*K at the specified temperature *T* in K.

get_free_energy(*T*)

Return the Gibbs free energy in kJ/mol at the specified temperature *T* in K.

get_heat_capacity(*T*)

Return the constant-pressure heat capacity in J/mol*K at the specified temperature *T* in K.

has_statmech()

Return True if all species in the configuration have statistical mechanics parameters, or False otherwise.

has_thermo()

Return True if all species in the configuration have thermodynamics parameters, or False otherwise.

is_bimolecular()

Return True if the configuration represents a bimolecular reactant or product channel, or False otherwise.

is_termolecular()

Return True if the configuration represents a termolecular reactant or product channel, or False otherwise.

is_transition_state()

Return True if the configuration represents a transition state, or False otherwise.

is_unimolecular()

Return True if the configuration represents a unimolecular isomer, or False otherwise.

map_density_of_states(*e_list*, *j_list=None*)

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

map_sum_of_states(*e_list*, *j_list=None*)

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

rmgpy.pdep.Network

```
class rmgpy.pdep.Network(label="", isomers=None, reactants=None, products=None, path_reactions=None,  
                        bath_gas=None, net_reactions=None, T=0.0, P=0.0, e_list=None, j_list=None,  
                        n_grains=0, n_j=0, active_k_rotor=True, active_j_rotor=True, grain_size=0.0,  
                        grain_count=0, E0=None, Emax=None, Emin=None)
```

A representation of a unimolecular reaction network. The attributes are:

Attribute	Description
<i>isomers</i>	A list of the unimolecular isomers (Configuration objects) in the network
<i>reactants</i>	A list of the bimolecular reactant channels (Configuration objects) in the network

continues o

Table 3 – continued from previous page

Attribute	Description
<i>products</i>	A list of the bimolecular product channels (Configuration objects) in the network
<i>path_reactions</i>	A list of Reaction objects that connect isomers to their unimolecular and bimolecular products (the high-pressure-limit)
<i>bath_gas</i>	A dictionary of the bath gas species (keys) and their mole fractions (values)
<i>net_reactions</i>	A list of Reaction objects that connect any pair of isomers (pressure dependent reactions)
<i>T</i>	The current temperature in K
<i>P</i>	The current pressure in Pa
<i>e_list</i>	The current array of energy grains in J/mol
<i>j_list</i>	The current array of total angular momentum quantum numbers
<i>n_isom</i>	The number of unimolecular isomers in the network
<i>n_reac</i>	The number of bimolecular reactant channels in the network
<i>n_prod</i>	The number of bimolecular product channels in the network
<i>n_grains</i>	The number of energy grains
<i>n_j</i>	The number of angular momentum grains
<i>grain_size</i>	Maximum size of separation between energies
<i>grain_count</i>	Minimum number of discrete energies separated
<i>E0</i>	A list of ground state energies of isomers, reactants, and products (J/mol)
<i>E_{max}</i>	Highest energy level considered in graining before padding
<i>E_{min}</i>	Minimum energy level considered in graining
<i>active_k_rotor</i>	True if the K-rotor is treated as active, False if treated as adiabatic
<i>active_j_rotor</i>	True if the J-rotor is treated as active, False if treated as adiabatic
<i>rmgmode</i>	True if in RMG mode, False otherwise
<i>eq_ratios</i>	An array containing concentration of each isomer and reactant channel present at equilibrium
<i>coll_freq</i>	An array of the frequency of collision between isomers and the bath gas
<i>Mcoll</i>	Matrix of first-order rate coefficients for collisional population transfer between grains for each isomer
<i>dens_states</i>	3D np array of stable configurations, number of grains, and number of J
<i>K_{ij}</i>	The microcanonical rates to go from isomer j to isomer i . 4D array with indexes: i, j, energies, rotational energies
<i>G_{nj}</i>	The microcanonical rates to go from isomer j to reactant/product n . 4D array with indexes: n, j, energies, rotational energies
<i>F_{im}</i>	The microcanonical rates to go from reactant m to isomer i . 4D array with indexes: n, j, energies, rotational energies
<i>K</i>	2D Array of phenomenological rates at the specified T and P
<i>p0</i>	Pseudo-steady state population distributions
<i>U</i>	2D Array of estimated uncorrelated uncertainties in K matrix

apply_chemically_significant_eigenvalues_method(*lumping_order*=None, *method*='allen',
neglect_high_energy_collisions=False,
high_energy_rate_tol=0.01)

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the chemically-significant eigenvalues method. If a *lumping_order* 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.

apply_modified_strong_collision_method(*efficiency_model*='default')

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the modified strong collision method.

apply_reservoir_state_method()

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the reservoir state method.

calculate_collision_model()

Calculate the matrix of first-order rate coefficients for collisional population transfer between grains for each isomer, including the corresponding collision frequencies.

calculate_densities_of_states()

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.

calculate_equilibrium_ratios()

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.

calculate_microcanonical_rates()

Calculate and return arrays containing the microcanonical rate coefficients $k(E)$ for the isomerization, dissociation, and association path reactions in the network.

get_all_species()

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*, *maximum_grain_size*=0.0, *minimum_grain_count*=0, *active_j_rotor*=True, *active_k_rotor*=True, *rmgmode*=False)

Initialize a pressure dependence calculation by computing several quantities that are independent of the conditions. You must specify the temperature and pressure ranges of interest using *Tmin* and *Tmax* in K and *Pmin* and *Pmax* in Pa. You must also specify the maximum energy grain size *grain_size* in J/mol and/or the minimum number of grains *grain_count*.

invalidate()

Mark the network as in need of a new calculation to determine the pressure-dependent rate coefficients

log_summary(*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.

map_densities_of_states()

Map the overall densities of states to the current energy grains. Semi-logarithmic interpolation will be used if the grain sizes of *Elist0* and *e_list* do not match; this should not be a significant source of error as long as the grain sizes are sufficiently small.

select_energy_grains(*T*, *grain_size*=0.0, *grain_count*=0)

Select a suitable list of energies to use for subsequent calculations. This is done by finding the minimum and maximum energies on the potential energy surface, then adding a multiple of $k_B T$ onto the maximum energy.

You must specify either the desired grain spacing *grain_size* in J/mol or the desired number of grains *n_grains*, as well as a temperature *T* in K to use for the equilibrium calculation. You can specify both *grain_size* and *grain_count*, 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.

set_conditions(*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.

solve_full_me(*tlist*, *x0*)

Directly solve the full master equation using a stiff ODE solver. Pass the reaction *network* to solve, the temperature *T* in K and pressure *P* in Pa to solve at, the energies *e_list* 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 *dens_states* in mol/J of each isomer. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

solve_reduced_me(*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.generate_full_me_matrix(network, products=True, exclude_association=False,
                                     neglect_high_energy_collisions=False,
                                     high_energy_rate_tol=0.01)
```

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.apply_modified_strong_collision_method(network, efficiency_model='default')
```

A method for applying the Modified Strong Collision approach for solving the master equation.

The modified strong collision method utilizes a greatly simplified collision model that allows for a decoupling of the energy grains. In the simplified collision model, collisional stabilization of a reactive isomer is treated as a single-step process, ignoring the effects of collisional energy redistribution within the reactive energy space. An attempt to correct for the effect of collisional energy redistribution is made by modifying the collision frequency $\omega_i(T, P)$ with a collision efficiency $\beta_i(T)$ estimated from the low-pressure limit fall-off of a single isomer.

By approximating the reactive populations as existing in pseudo-steady state, the master equation is converted to a matrix equation at each energy. Solving these small matrix equations gives the pseudo-steady state populations of each isomer as a function of the total population of each isomer and reactant channel, which are then applied to determine the $k(T, P)$ values.

In practice, the modified strong collision method is the fastest and most robust of the methods, and is reasonably accurate over a wide range of temperatures and pressures.

The reservoir state method

`rmgpy.pdep.rs.apply_reservoir_state_method(network)`

A method for applying the Reservoir State approach for solving the master equation.

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.apply_chemically_significant_eigenvalues_method(network, lumping_order=None,
ne-
glect_high_energy_collisions=False,
high_energy_rate_tol=0.01)`

A method for applying the Chemically Significant Eigenvalues approach for solving the master equation.

In the chemically-significant eigenvalues method, the master equation matrix is diagonalized to determine its eigenmodes. Only the slowest of these modes are relevant to the chemistry; the rest involve internal energy relaxation due to collisions. Keeping only these “chemically-significant” eigenmodes allows for reduction to $k(T, P)$ values.

The chemically-significant eigenvalues method is the most accurate method, and is considered to be exact as long as the chemically-significant eigenmodes are separable and distinct from the internal energy relaxation eigenmodes. However, this is often only the case near the high-pressure limit, even for networks of only modest size. The chemically-significant eigenvalues method is also substantially more expensive to apply than the other methods.

1.8 QMTP (rmgpy.qm)

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

1.8.1 Main

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

1.8.2 Molecule

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

1.8.3 QM Data

Class/Function	Description
<i>QMData</i>	General class for data extracted from a QM calculation

1.8.4 QM Verifier

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

1.8.5 Symmetry

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

1.8.6 Gaussian

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

1.8.7 Mopac

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

rmgpy.qm.main

class rmgpy.qm.main.QMSettings(*software=None, method='pm3', fileStore=None, scratchDirectory=None, onlyCyclics=True, maxRadicalNumber=0*)

A minimal class to store settings related to quantum mechanics calculations.

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

check_all_set()

Check that all the required settings are set.

class rmgpy.qm.main.QMCalculator(*software=None, method='pm3', fileStore=None, scratchDirectory=None, onlyCyclics=True, maxRadicalNumber=0*)

A Quantum Mechanics calculator object, to store settings.

The attributes are:

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

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

get_thermo_data(*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.

run_jobs(*spc_list*, *procnum*=1)

Run QM jobs for the provided species list (in parallel if requested).

set_default_output_directory(*output_directory*)

If the `fileStore` or `scratchDirectory` are not already set, put them in here.

rmgpy.qm.molecule**class rmgpy.qm.molecule.Geometry(*settings*, *unique_id*, *molecule*, *unique_id_long*=None)**

A geometry, used for quantum calculations.

Created from a molecule. Geometry estimated by RDKit.

The attributes are:

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

generate_rdkit_geometries()

Use RDKit to guess geometry.

Save mol files of both crude and refined. Saves coordinates on atoms.

get_crude_mol_file_path()

Returns the path of the crude mol file.

get_file_path(*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.

get_refined_mol_file_path()

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*, *num_conf_attempts*)

Embed the RDKit molecule and create the crude molecule file.

save_coordinates_from_qm_data(*qmdata*)

Save geometry info from QMData (eg CCLibData)

unique_id

A short unique ID such as an augmented InChI Key.

unique_id_long

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:

- `output_file_extension`
- `input_file_extension`
- `generate_qm_data()` ... and whatever else is needed to make this method work.

The attributes are:

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

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as `self.thermo`. `self.qm_data` and `self.point_group` need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates `self.geometry` with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in `self.point_group`

generate_qm_data()

Calculate the QM data somehow and return a CCLibData object, or None if it fails.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(*backend='rdkit-first'*)

Returns the augmented InChI from `self.molecule`

get_file_path(*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.

get_mol_file_path_for_calculation(*attempt*)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If *attempt* <= *self.script_attempts* then we use the refined coordinates, then we start to use the crude coordinates.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

property input_file_path

Get the input file name.

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

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

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

numberOfAtoms

Number of atoms.

rmgpy.qm.qmverifier

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

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

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

check_for_inchi_key_collision(*log_file_inchi*)

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.

successful_job_exists()

checks whether one of the flags is true. If so, it returns true.

rmgpy.qm.symmetry

class rmgpy.qm.symmetry.PointGroup(*point_group, symmetry_number, chiral*)

A symmetry Point Group.

Attributes are:

- point_group
- symmetry_number
- chiral
- linear

class rmgpy.qm.symmetry.PointGroupCalculator(*settings, unique_id, qm_data*)

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, unique_id, qm_data*)

Determine the point group using the SYMMETRY program

(Originally <http://www.cobalt.chem.ucalgary.ca/ps/symmetry/>
now mirrored at <https://github.com/nquesada/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.

property input_file_path

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.

unique_id

The object that holds information from a previous QM Job on 3D coords, molecule etc...

write_input_file()

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.

failure_keys = ['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.

success_keys = ['Normal termination of Gaussian']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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*.

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['ERROR TERMINATION', 'IMAGINARY FREQUENCIES']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['Normal termination of Gaussian']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(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.

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['ERROR TERMINATION', 'IMAGINARY FREQUENCIES']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

```

keywords = ['# pm3 opt=(verytight,gdiis) freq IOP(2/16=3)', '# pm3
opt=(verytight,gdiis) freq IOP(2/16=3) IOP(4/21=2)', '# pm3
opt=(verytight,calcf, maxcyc=200) freq IOP(2/16=3) nosymm', '# pm3
opt=(verytight,calcf, maxcyc=200) freq=numerical IOP(2/16=3) nosymm', '# pm3
opt=(verytight,gdiis,small) freq IOP(2/16=3)', '# pm3
opt=(verytight,nolinear,calcf,small) freq IOP(2/16=3)', '# pm3
opt=(verytight,gdiis, maxcyc=200) freq=numerical IOP(2/16=3)', '# pm3 opt=tight
freq IOP(2/16=3)', '# pm3 opt=tight freq=numerical IOP(2/16=3)', '# pm3
opt=(tight,nolinear,calcf,small, maxcyc=200) freq IOP(2/16=3)', '# pm3 opt freq
IOP(2/16=3)', '# pm3 opt=(verytight,gdiis) freq=numerical IOP(2/16=3)
IOP(4/21=200)', '# pm3
opt=(calcf, verytight, newton, notrustupdate, small, maxcyc=100, maxstep=100)
freq=(numerical, step=10) IOP(2/16=3) nosymm', '# pm3
opt=(tight, gdiis, small, maxcyc=200, maxstep=100) freq=numerical IOP(2/16=3) nosymm',
'# pm3 opt=(verytight, gdiis, calcf) IOP(2/16=3)', '# pm3
opt=(verytight, gdiis, calcf, small, maxcyc=200) IOP(2/16=3) IOP(4/21=2) nosymm', '#
pm3 opt=(verytight, gdiis, calcf, small) IOP(2/16=3) nosymm', '# pm3
opt=(calcf, small, maxcyc=100) IOP(2/16=3)']

```

Keywords that will be added at the top of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['Normal termination of Gaussian']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(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.

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as `self.thermo`. `self.qm_data` and `self.point_group` need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates `self.geometry` with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting `PointGroup` in `self.point_group`

failure_keys = ['ERROR TERMINATION', 'IMAGINARY FREQUENCIES']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a `QMData` object.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(*backend='rdkit-first'*)

Returns the augmented InChI from `self.molecule`

get_file_path(*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.

get_mol_file_path_for_calculation(*attempt*)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If `attempt <= self.script_attempts` then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(*output_file*)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(*attempt*)

Return the top keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

```
keywords = ['# pm6 opt=(verytight,gdiis) freq IOP(2/16=3)', '# pm6
opt=(verytight,gdiis) freq IOP(2/16=3) IOP(4/21=2)', '# pm6
opt=(verytight,calcfc,maxcyc=200) freq IOP(2/16=3) nosymm', '# pm6
opt=(verytight,calcfc,maxcyc=200) freq=numerical IOP(2/16=3) nosymm', '# pm6
opt=(verytight,gdiis,small) freq IOP(2/16=3)', '# pm6
opt=(verytight,nolinear,calcfc,small) freq IOP(2/16=3)', '# pm6
opt=(verytight,gdiis,maxcyc=200) freq=numerical IOP(2/16=3)', '# pm6 opt=tight
freq IOP(2/16=3)', '# pm6 opt=tight freq=numerical IOP(2/16=3)', '# pm6
opt=(tight,nolinear,calcfc,small,maxcyc=200) freq IOP(2/16=3)', '# pm6 opt freq
IOP(2/16=3)', '# pm6 opt=(verytight,gdiis) freq=numerical IOP(2/16=3)
IOP(4/21=200)', '# pm6
opt=(calcfc,verytight,newton,notrustupdate,small,maxcyc=100,maxstep=100)
freq=(numerical,step=10) IOP(2/16=3) nosymm', '# pm6
opt=(tight,gdiis,small,maxcyc=200,maxstep=100) freq=numerical IOP(2/16=3) nosymm',
'# pm6 opt=(verytight,gdiis,calcall) IOP(2/16=3)', '# pm6
opt=(verytight,gdiis,calcall,small,maxcyc=200) IOP(2/16=3) IOP(4/21=2) nosymm', '#
pm6 opt=(verytight,gdiis,calcall,small) IOP(2/16=3) nosymm', '# pm6
opt=(calcall,small,maxcyc=100) IOP(2/16=3)']
```

Keywords that will be added at the top of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['Normal termination of Gaussian']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

Check's that an output file exists and was successful.

Returns a boolean flag that states whether a successful GAUSSIAN simulation already exists for the molecule with the given (augmented) InChI Key.

The definition of finding a successful simulation is based on these criteria: 1) finding an output file with the file name equal to the InChI Key 2) NOT finding any of the keywords that denote a calculation failure 3) finding all the keywords that denote a calculation success. 4) finding a match between the InChI of the given molecule and the InChI found in the calculation files 5) checking that the optimized geometry, when connected by single bonds, is isomorphic with self.molecule (converted to single bonds)

If any of the above criteria is not matched, False will be returned. If all are satisfied, it will return True.

write_input_file(*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.

failure_keys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']

List of phrases that indicate failure NONE of these must be present in a succesful job.

get_parser(*output_file*)

Returns the appropriate cclib parser.

success_keys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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](#).

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object, or None if it fails.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top, bottom, and polar keywords.

property input_file_path

Get the input file name.

```
keywords = [{'top': 'precise nosym THREADS=1', 'bottom': 'oldgeo thermo nosym
precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0 nonr THREADS=1', 'bottom':
'oldgeo thermo nosym precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}, {'top':
'precise nosym gnorm=0.0 bfgs THREADS=1', 'bottom': 'oldgeo thermo nosym precise
THREADS=1 '}, {'top': 'precise nosym recalc=10 dmax=0.10 nonr cycles=2000 t=2000
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}]
```

Keywords that will be added at the top and bottom of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(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

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object, or None if it fails.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

```
keywords = [{'top': 'precise nosym THREADS=1', 'bottom': 'oldgeo thermo nosym
precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0 nonr THREADS=1', 'bottom':
'oldgeo thermo nosym precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}, {'top':
'precise nosym gnorm=0.0 bfgs THREADS=1', 'bottom': 'oldgeo thermo nosym precise
THREADS=1 '}, {'top': 'precise nosym recalc=10 dmax=0.10 nonr cycles=2000 t=2000
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}]
```

Keywords that will be added at the top and bottom of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(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

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object, or None if it fails.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

```
keywords = [{'top': 'precise nosym THREADS=1', 'bottom': 'oldgeo thermo nosym
precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0 nonr THREADS=1', 'bottom':
'oldgeo thermo nosym precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}, {'top':
'precise nosym gnorm=0.0 bfgs THREADS=1', 'bottom': 'oldgeo thermo nosym precise
THREADS=1 '}, {'top': 'precise nosym recalc=10 dmax=0.10 nonr cycles=2000 t=2000
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}]
```

Keywords that will be added at the top and bottom of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(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

calculate_chirality_correction()

Returns the chirality correction to entropy ($R \cdot \ln(2)$ if chiral) in J/mol/K.

calculate_thermo_data()

Calculate the thermodynamic properties.

Stores and returns a ThermoData object as self.thermo. self.qm_data and self.point_group need to be generated before this method is called.

check_paths()

Check the paths in the settings are OK. Make folders as necessary.

check_ready()

Check that it's ready to run calculations.

create_geometry()

Creates self.geometry with RDKit geometries

determine_point_group()

Determine point group using the SYMMETRY Program

Stores the resulting PointGroup in self.point_group

failure_keys = ['IMAGINARY FREQUENCIES', 'EXCESS NUMBER OF OPTIMIZATION CYCLES', 'NOT ENOUGH TIME FOR ANOTHER CYCLE']

List of phrases that indicate failure NONE of these must be present in a succesful job.

generate_qm_data()

Calculate the QM data and return a QMData object, or None if it fails.

generate_thermo_data()

Generate Thermo Data via a QM calc.

Returns None if it fails.

get_augmented_inchi_key(backend='rdkit-first')

Returns the augmented InChI from self.molecule

get_file_path(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.

get_mol_file_path_for_calculation(attempt)

Get the path to the MOL file of the geometry to use for calculation *attempt*.

If attempt <= self.script_attempts then we use the refined coordinates, then we start to use the crude coordinates.

get_parser(output_file)

Returns the appropriate cclib parser.

get_thermo_file_path()

Returns the path the thermo data file.

initialize()

Do any startup tasks.

input_file_keywords(attempt)

Return the top, bottom, and polar keywords for attempt number *attempt*.

NB. *attempt* begins at 1, not 0.

property input_file_path

Get the input file name.

```
keywords = [{'top': 'precise nosym THREADS=1', 'bottom': 'oldgeo thermo nosym
precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0 nonr THREADS=1', 'bottom':
'oldgeo thermo nosym precise THREADS=1 '}, {'top': 'precise nosym gnorm=0.0
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}, {'top':
'precise nosym gnorm=0.0 bfgs THREADS=1', 'bottom': 'oldgeo thermo nosym precise
THREADS=1 '}, {'top': 'precise nosym recalc=10 dmax=0.10 nonr cycles=2000 t=2000
THREADS=1', 'bottom': 'oldgeo thermo nosym precise THREADS=1 '}]
```

Keywords that will be added at the top and bottom of the qm input file

load_thermo_data()

Try loading a thermo data from a previous run.

property max_attempts

The total number of attempts to try

property output_file_path

Get the output file name.

parse()

Parses the results of the Mopac calculation, and returns a QMData object.

save_thermo_data()

Save the generated thermo data.

property script_attempts

The number of attempts with different script keywords

success_keys = ['DESCRIPTION OF VIBRATIONS', 'MOPAC DONE']

List of phrases to indicate success. ALL of these must be present in a successful job.

verify_output_file()

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.

write_input_file(attempt)

Using the Geometry object, write the input file for the *attempt*.

1.9 Physical quantities (rmgpy.quantity)

A physical quantity is defined by a numerical value and a unit of measurement.

The *rmgpy.quantity* module contains classes and methods for working with physical quantities. Physical quantities are represented by either the *ScalarQuantity* or *ArrayQuantity* class depending on whether a scalar or vector (or tensor) value is used. The *Quantity* function automatically chooses the appropriate class based on the input value. In both cases, the value of a physical quantity is available from the *value* attribute, and the units from the *units* attribute.

For efficient computation, the value is stored internally in the SI equivalent units. The SI value can be accessed directly using the *value_si* attribute. Usually it is good practice to read the *value_si* attribute into a local variable and then use it for computations, especially if it is referred to multiple times in the calculation.

Physical quantities also allow for storing of uncertainty values for both scalars and arrays. The *uncertaintyType* attribute indicates whether the given uncertainties are additive (" + | - ") or multiplicative (" * | / "), and the *uncertainty*

attribute contains the stored uncertainties. For additive uncertainties these are stored in the given units (not the SI equivalent), since they are generally not needed for efficient computations. For multiplicative uncertainties, the uncertainty values are by definition dimensionless.

1.9.1 Quantity objects

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

1.9.2 Unit types

Units can be classified into categories based on the associated dimensionality. For example, miles and kilometers are both units of length; seconds and hours are both units of time, etc. Clearly, quantities of different unit types are fundamentally different.

RMG provides functions that create physical quantities (scalar or array) and validate the units for a variety of unit types. This prevents the user from inadvertently mixing up their units - e.g. by setting an enthalpy with entropy units - which should reduce errors. RMG recognizes the following unit types:

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

In RMG, all energies, heat capacities, concentrations, fluxes, and rate coefficients are treated as intensive; this means that these quantities are always expressed “per mole” or “per molecule”. All other unit types are extensive. A special exception is added for mass so as to allow for coercion of g/mol to amu .

RMG also handles rate coefficient units as a special case, as there are multiple allowed dimensionalities based on the reaction order. Note that RMG generally does not attempt to verify that the rate coefficient units match the reaction order, but only that it matches one of the possibilities.

The table above gives the SI unit that RMG uses internally to work with physical quantities. This does not necessarily correspond with the units used when outputting values. For example, pressures are often output in units of bar instead of Pa, and moments of inertia in $\text{amu} \cdot \text{\AA}^2$ instead of $\text{kg} \cdot \text{m}^2$. The recommended rule of thumb is to use prefixed SI units (or aliases thereof) in the output; for example, use kJ/mol instead of kcal/mol for energy values.

rmgpy.quantity.ScalarQuantity

class rmgpy.quantity.ScalarQuantity

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

Attribute	Description
<i>value</i>	The numeric value of the quantity in the given units
<i>units</i>	The units the value was specified in
<i>uncertainty</i>	The numeric uncertainty in the value in the given units (unitless if multiplicative)
<i>uncertainty_type</i>	The type of uncertainty: '+' '-' for additive, '*' '/' for multiplicative
<i>value_si</i>	The numeric value of the quantity in the corresponding SI units
<i>uncertainty_si</i>	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.

as_dict()

A helper function for YAML dumping

copy()

Return a copy of the quantity.

equals(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.)

get_conversion_factor_from_si()

Return the conversion factor for converting a quantity to a given set of *units* from the SI equivalent units.

get_conversion_factor_from_si_to_cm_mol_s()

Return the conversion factor for converting into SI units only with all lengths in cm, instead of m. This is useful for outputting chemkin file kinetics. Depending on the stoichiometry of the reaction the reaction rate coefficient could be /s, $\text{cm}^3/\text{mol}/\text{s}$, $\text{cm}^6/\text{mol}^2/\text{s}$, and for heterogeneous reactions even more possibilities. Only lengths are changed. Everything else is in SI, i.e. moles (not molecules) and seconds (not minutes).

get_conversion_factor_to_si()

Return the conversion factor for converting a quantity in a given set of *units* to the SI equivalent units.

is_uncertainty_additive()

Return True if the uncertainty is specified in additive format and False otherwise.

is_uncertainty_multiplicative()

Return True if the uncertainty is specified in multiplicative format and False otherwise.

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

uncertainty

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

uncertainty_type

'+|-' for additive, '*|/' for multiplicative

Type

The type of uncertainty

value

The numeric value of the quantity, in the given units

rmgpy.quantity.ArrayQuantity**class** rmgpy.quantity.ArrayQuantity

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

Attribute	Description
<i>value</i>	The numeric value of the quantity in the given units
<i>units</i>	The units the value was specified in
<i>uncertainty</i>	The numeric uncertainty in the value (unitless if multiplicative)
<i>uncertainty_type</i>	The type of uncertainty: '+ -' for additive, '* /' for multiplicative
<i>value_si</i>	The numeric value of the quantity in the corresponding SI units
<i>uncertainty_si</i>	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.

as_dict()

A helper function for YAML dumping

copy()

Return a copy of the quantity.

equals(*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.)

get_conversion_factor_from_si()

Return the conversion factor for converting a quantity to a given set of *units* from the SI equivalent units.

get_conversion_factor_from_si_to_cm_mol_s()

Return the conversion factor for converting into SI units only with all lengths in cm, instead of m. This is useful for outputting chemkin file kinetics. Depending on the stoichiometry of the reaction the reaction rate coefficient could be /s, cm³/mol/s, cm⁶/mol²/s, and for heterogeneous reactions even more possibilities. Only lengths are changed. Everything else is in SI, i.e. moles (not molecules) and seconds (not minutes).

get_conversion_factor_to_si()

Return the conversion factor for converting a quantity in a given set of units to the SI equivalent units.

is_uncertainty_additive()

Return True if the uncertainty is specified in additive format and False otherwise.

is_uncertainty_multiplicative()

Return True if the uncertainty is specified in multiplicative format and False otherwise.

make_object(data, class_dict)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

uncertainty

The numeric value of the uncertainty, in the given units if additive, or no units if multiplicative.

uncertainty_type

'+| -' for additive, '*|/' for multiplicative

Type

The type of uncertainty

value

The numeric value of the array quantity, in the given units.

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 *uncertainty_type*.
- A tuple of the form (value,), (value, units), (value, units, uncertainty), or (value, units, uncertainty_type, 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
<code>Reaction</code>	A chemical reaction

`rmgpy.reaction.Reaction`

class `rmgpy.reaction.Reaction`

A chemical reaction. The attributes are:

Attribute	Type	Description
<i>index</i>	int	A unique nonnegative integer index
<i>label</i>	str	A descriptive string label
<i>reactants</i>	list	The reactant species (as <code>Species</code> objects)
<i>products</i>	list	The product species (as <code>Species</code> objects)
<i>'specific_collider'</i>	<code>Species</code>	The collider species (as a <code>Species</code> object)
<i>kinetics</i>	<code>KineticsModel</code>	The kinetics model to use for the reaction
<i>network_kinetics</i>	<code>Arrhenius</code>	The kinetics model to use for PDep network exploration if the <i>kinetics</i> attribute is <code>:class:PDepKineticsModel:</code>
<i>reversible</i>	bool	True if the reaction is reversible, False if not
<i>transition_state</i>	<code>TransitionState</code>	The transition state
<i>duplicate</i>	bool	True if the reaction is known to be a duplicate, False if not
<i>degeneracy</i>	double	The reaction path degeneracy for the reaction
<i>pairs</i>	list	Reactant-product pairings to use in converting reaction flux to species flux
<i>allow_pdep_route</i>	bool	True if the reaction has an additional PDep pathway, False if not (by default), used for <code>LibraryReactions</code>
<i>elementary_high_p</i>	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)
<i>comment</i>	str	A description of the reaction source (optional)
<i>is_forward</i>	bool	Indicates if the reaction was generated in the forward (true) or reverse (false)
<i>rank</i>	int	Integer indicating the accuracy of the kinetics for this reaction

apply_solvent_correction(*solvent*)

apply kinetic solvent correction

calculate_coll_limit(*temp*, *reverse=False*)

Calculate the collision limit rate in m³/mol-s for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

calculate_microcanonical_rate_coefficient(*e_list*, *j_list*, *reac_dens_states*,
prod_dens_states=None, *T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.

calculate_tst_rate_coefficient(*T*)

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

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

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

can_tst()

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(*t_min*, *t_max*, *p_min*, *p_max*)

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.

copy()

Create a deep copy of the current reaction.

degeneracy

The reaction path degeneracy for this reaction.

If the reaction has kinetics, changing the degeneracy will adjust the reaction rate by a ratio of the new degeneracy to the old degeneracy.

draw(*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.

ensure_species(*reactant_resonance=False*, *product_resonance=False*, *save_order=False*)

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. If *save_order* is True the atom order is reset after performing atom isomorphism.

fix_barrier_height(*force_positive=False, solvent="", apply_solvation_correction=True*)

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 *force_positive* is True, then all reactions are forced to have a non-negative barrier.

fix_diffusion_limited_a_factor(*T*)

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

generate_3d_ts(*reactants, products*)

Generate the 3D structure of the transition state. Called from `model.generate_kinetics()`.

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

generate_high_p_limit_kinetics()

Used for incorporating library reactions with pressure-dependent kinetics in PDep networks. Only implemented for `LibraryReaction`

generate_pairs()

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. This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generate_reverse_rate_coefficient(*network_kinetics=False, Tmin=None, Tmax=None, surface_site_density=0.0*)

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.

If the reaction kinetics model is Sticking Coefficient, please provide a nonzero surface site density in mol/m^2 which is required to evaluate the rate coefficient.

get_enthalpies_of_reaction(*Tlist*)

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

get_enthalpy_of_reaction(*T*)

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

get_entropies_of_reaction(*Tlist*)

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

get_entropy_of_reaction(*T*)

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

get_equilibrium_constant(*T, potential=0.0, type='Kc', surface_site_density=2.5e-05*)

Return the equilibrium constant for the reaction at the specified temperature *T* in K and reference *surface_site_density* in mol/m^2 (2.5e-05 default) The *type* parameter lets you specify the quantities used in

the equilibrium constant: K_a for activities, K_c for concentrations (default), or K_p for pressures. This function assumes a reference pressure of $1e5$ Pa for gas phases species and uses the ideal gas law to determine reference concentrations. For surface species, the *surface_site_density* is the assumed reference. For protons (H^+), a reference concentration of 1000 mol/m^3 (1 mol/L) is assumed

get_equilibrium_constants(*Tlist*, *potential*=0.0, *type*='Kc')

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

get_free_energies_of_reaction(*Tlist*, *potential*=0.0)

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

get_free_energy_of_reaction(*T*, *potential*=0.0)

Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K and potential in Volts (if applicable)

get_mean_sigma_and_epsilon(*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_rate_coefficient(*T*, *P*=0.0, *surface_site_density*=0.0, *potential*=0.0)

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 *diffusion_limiter* 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.

If the reaction has sticking coefficient kinetics, a nonzero surface site density in mol/m^2 must be provided

get_reduced_mass(*reverse*=False)

Returns the reduced mass of the reactants if *reverse* is False Returns the reduced mass of the products if *reverse* is True

get_reversible_potential(*T*)

Get the Potential in V at *T* in 'K' at which the charge transfer reaction is at equilibrium

get_stoichiometric_coefficient(*spec*)

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.

get_surface_rate_coefficient(*T*, *surface_site_density*, *potential*=0.0)

Return the overall surface rate coefficient for the forward reaction at temperature *T* in K with surface site density *surface_site_density* in mol/m^2 . Value is returned in combination of [m,mol,s]

get_url()

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

has_template(*reactants*, *products*)

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

is_association()

Return True if the reaction represents an association reaction $A + B \rightleftharpoons C$ or False if not.

is_balanced()

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

is_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_dissociation()

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

is_isomerization()

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

is_isomorphic(*other*, *either_direction=True*, *check_identical=False*, *check_only_label=False*, *check_template_rxn_products=False*, *generate_initial_map=False*, *strict=True*, *save_order=False*)

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.

Parameters

- **either_direction** (*bool*, *optional*) – if False, then the reaction direction must match.
- **check_identical** (*bool*, *optional*) – if True, check that atom ID's match (used for checking degeneracy)
- **check_only_label** (*bool*, *optional*) – if True, only check the string representation, ignoring molecular structure comparisons
- **check_template_rxn_products** (*bool*, *optional*) – if True, only check isomorphism of reaction products (used when we know the reactants are identical, i.e. in generating reactions)
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons
- **save_order** (*bool*, *optional*) – if True, perform isomorphism saving atom order

is_surface_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_surface_reaction()

Return True if one or more reactants or products are surface species (or surface sites)

is_unimolecular()

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

matches_species(*reactants*, *products=None*)

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

protons

The stoichiometric coeff for protons in charge transfer reactions

reverse_arrhenius_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceChargeTransfer` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

reverse_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, 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).

reverse_sticking_coeff_rate(*k_forward*, *reverse_units*, *surface_site_density*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `StickingCoefficient` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self). The *surface_site_density* in mol/m^2 is used to evaluate the forward rate constant.

reverse_surface_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceArrhenius` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

reverse_surface_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceChargeTransfer` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

set_reference_potential(*T*)

Set the reference Potential of the `SurfaceChargeTransfer` kinetics model to the reversible potential of the reaction

to_cantera(*species_list=None*, *use_chemkin_identifier=False*)

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

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

to_chemkin(*species_list=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 *species_list* to figure out third body colliders.) Otherwise, only the reaction string will be returned.

to_labeled_str(*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

1.11 Reaction mechanism generation (`rmgpy.rmg`)

The `rmgpy.rmg` subpackage contains the main functionality for using RMG-Py to automatically generate detailed reaction mechanisms.

1.11.1 Reaction models

Class	Description
<i>CoreEdgeReactionModel</i>	A reaction model comprised of core and edge species and reactions

1.11.2 Input

Function	Description
<i>read_input_file()</i>	Load an RMG job input file
<i>save_input_file()</i>	Save an RMG job input file

1.11.3 Output

Function	Description
<i>save_output_html()</i>	Save the results of an RMG job to an HTML file
<i>save_diff_html()</i>	Save a comparison of two reaction mechanisms to an HTML file

1.11.4 Job classes

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

1.11.5 Pressure dependence

Class	Description
<i>PDepReaction</i>	A pressure-dependent “net” reaction
<i>PDepNetwork</i>	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
<i>core</i>	The species and reactions of the current model core
<i>edge</i>	The species and reactions of the current model edge
<i>network_dict</i>	A dictionary of pressure-dependent reaction networks (Network objects) indexed by source.
<i>network_list</i>	A list of pressure-dependent reaction networks (Network objects)
<i>network_count</i>	A counter for the number of pressure-dependent networks created
<i>in-dex_species_dict</i>	A dictionary with a unique index pointing to the species objects
<i>solvent_name</i>	String describing solvent name for liquid reactions. Empty for non-liquid estimation
<i>surface_site_density</i>	The surface site density (a SurfaceConcentration quantity) or None if no heterogeneous catalyst.

add_new_surface_objects(*obj, new_surface_species, new_surface_reactions, reaction_system*)

obj is the list of objects for enlargement coming from `simulate` *new_surface_species* and *new_surface_reactions* are the current lists of surface species and surface reactions following simulation *reaction_system* 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

add_reaction_library_to_edge(*reaction_library, requires_rms=False*)

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

add_reaction_library_to_output(*reaction_library*)

Add all species and reactions from *reaction_library*, a **KineticsPrimaryDatabase** object, to the output. This does not bring any of the reactions or species into the core itself.

add_reaction_to_core(*rxn, requires_rms=False*)

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).

add_reaction_to_edge(*rxn, requires_rms=False*)

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).

add_reaction_to_unimolecular_networks(*newReaction, new_species, 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.

add_seed_mechanism_to_core(*seed_mechanism, react=False, requires_rms=False*)

Add all species and reactions from *seed_mechanism*, 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.

add_species_to_core(*spec, requires_rms=False*)

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.

add_species_to_edge(*spec*, *requires_rms=False*)

Add a species *spec* to the reaction model edge and optionally the RMS phase.

adjust_surface()

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)

apply_kinetics_to_reaction(*reaction*)

retrieve the best kinetics for the reaction and apply it towards the forward or reverse direction (if reverse, flip the direction).

apply_thermo_to_species(*procnum*)

Generate thermo for species. QM calculations are parallelized if requested.

check_for_existing_reaction(*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).

check_for_existing_species(*molecule*)

Check to see if an existing species contains the same `molecule.Molecule` as *molecule*. Comparison is done using isomorphism without consideration of electrons. Therefore, resonance structures of a species will all match each other.

Returns the matched species if found and `None` otherwise.

clear_surface_adjustments()

empties surface tracking variables

enlarge(*new_object=None*, *react_edge=False*, *unimolecular_react=None*, *bimolecular_react=None*, *trimolecular_react=None*, *requires_rms=False*)

Enlarge a reaction model by processing the objects in the list *new_object*. If *new_object* 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 *new_object* is a `rmg.unirxn.network.Network` object, then reactions are generated for the species in the network with the largest leak flux.

If the *react_edge* flag is `True`, then no *new_object* is needed, and instead the algorithm proceeds to react the core species together to form edge reactions.

generate_kinetics(*reaction*)

Generate best possible kinetics for the given *reaction* using the kinetics database.

generate_thermo(*spc*, *rename=False*)

Generate thermo for species.

get_model_size()

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.

get_species_reaction_lists()

Return lists of all of the species and reactions in the core and the edge.

get_stoichiometry_matrix()

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.

initialize_index_species_dict()

Populates the core species dictionary

integer -> core Species

with the species that are currently in the core.

log_enlarge_summary(*new_core_species*, *new_core_reactions*, *new_edge_species*, *new_edge_reactions*,
reactions_moved_from_edge=None, *react_edge*=False)

Output a summary of a model enlargement step to the log. The details of the enlargement are passed in the *new_core_species*, *new_core_reactions*, *new_edge_species*, and *new_edge_reactions* objects.

make_new_pdep_reaction(*forward*)

Make a new pressure-dependent reaction based on a list of *reactants* and a list of *products*. The reaction belongs to the specified *network* and has pressure-dependent kinetics given by *kinetics*.

No checking for existing reactions is made here. The returned PDepReaction object is not added to the global list of reactions, as that is intended to represent only the high-pressure-limit set. The reaction_counter is incremented, however, since the returned reaction can and will exist in the model edge and/or core.

make_new_reaction(*forward*, *check_existing*=True, *generate_thermo*=True, *generate_kinetics*=True,
perform_cut=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.new_reaction_list if it is new.

make_new_species(*object*, *label*="", *reactive*=True, *check_existing*=True, *generate_thermo*=True,
check_decay=False, *check_cut*=False)

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.

mark_chemkin_duplicates()

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.

process_coverage_dependence(*kinetics*)

Process the coverage dependence kinetics.

This ensures that species that are used in coverage-dependent kinetic expressions exist in the model. (Before this is called, they may have only existed in a reaction library instance).

If <CoreEdgeReactionModel>self.coverage_dependence is False then it instead removes any coverage_dependence from the kinetics.

process_new_reactions(*new_reactions*, *new_species*, *pdep_network=None*, *generate_thermo=True*, *generate_kinetics=True*, *requires_rms=False*)

Process a list of newly-generated reactions involving the new core species or explored isomer *new_species* in network *pdep_network*.

Makes a reaction and decides where to put it: core, edge, or PDepNetwork.

prune(*reaction_systems*, *tol_keep_in_edge*, *tol_move_to_core*, *maximum_edge_species*, *min_species_exist_iterations_for_prune*, *requires_rms=False*)

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

register_reaction(*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.

remove_empty_pdep_networks()

searches for and deletes any empty pdep networks

remove_species_from_edge(*reaction_systems*, *spec*, *requires_rms=False*)

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.

search_retrieve_reactions(*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.

set_thermodynamic_filtering_parameters(*Tmax*, *thermo_tol_keep_spc_in_edge*, *min_core_size_for_prune*, *maximum_edge_species*, *reaction_systems*)

sets parameters for thermodynamic filtering based on the current core *Tmax* is the maximum reactor temperature in K *thermo_tol_keep_spc_in_edge* is the Gibbs number above which species will be filtered *min_core_size_for_prune* is the core size at which thermodynamic filtering will start *maximum_edge_species* is the maximum allowed number of edge species *reaction_systems* is a list of reaction_system objects

thermo_filter_down(*maximum_edge_species*, *min_species_exist_iterations_for_prune=0*, *requires_rms=False*)

removes species from the edge based on their Gibbs energy until *maximum_edge_species* is reached under the constraint that all removed species are older than *min_species_exist_iterations_for_prune* iterations *maximum_edge_species* is the maximum allowed number of edge species *min_species_exist_iterations_for_prune* is the number of iterations a species must be in the edge before it is eligible for thermo filtering

thermo_filter_species(*spcs*, *requires_rms=False*)

checks Gibbs energy of the species in species against the maximum allowed Gibbs energy

update_unimolecular_reaction_networks()

Iterate through all of the currently-existing unimolecular reaction networks, updating those that have been marked as invalid. In each update, the phenomenological rate coefficients $k(T, P)$ are computed for each net reaction in the network, and the resulting reactions added or updated.

class `rmgpy.rmg.model.ReactionModel`(*species=None*, *reactions=None*, *phases=None*, *interfaces={}*)

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.read_input_file`(*path*, *rmg0*)

Read an RMG input file at *path* on disk into the RMG object *rmg*.

`rmgpy.rmg.input.save_input_file`(*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`(*input_file=None*, *output_directory=None*, *profiler=None*, *stats_file=None*)

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

Attribute	Description
<i>input_file</i>	The path to the input file
<i>profiler</i>	A <code>cProfile.Profile</code> object for time profiling RMG
<i>database_directory</i>	The directory containing the RMG database
<i>thermo_libraries</i>	The thermodynamics libraries to load
<i>reaction_libraries</i>	The kinetics libraries to load
<i>statmech_libraries</i>	The statistical mechanics libraries to load
<i>seed_mechanisms</i>	The seed mechanisms included in the model
<i>kinetics_families</i>	The kinetics families to use for reaction generation
<i>kinetics_depositories</i>	The kinetics depositories to use for looking up kinetics in each family
<i>kinetics_estimator</i>	The method to use to estimate kinetics: currently, only 'rate rules' is supported
<i>solvent</i>	If solvation estimates are required, the name of the solvent.
<i>solvent_data</i>	The parameters associated with the solvent
<i>liquid_volumetric_mass_transfer_coefficient_power_law</i>	If kLA estimates are required, the coefficients for kLA power law
<i>reaction_model</i>	The core-edge reaction model generated by this job
<i>reaction_systems</i>	A list of the reaction systems used in this job
<i>database</i>	The RMG database used in this job
<i>model_settings_list</i>	List of <code>ModelSettings</code> objects containing information related to how to make the model
<i>simulator_settings_list</i>	List of <code>SimulatorSettings</code> objects containing information on how to run simulations
<i>init_react_tuples</i>	List of name tuples of species to react at beginning of run
<i>trimolecular</i>	True to consider reactions between three species (i.e., if trimolecular reactions are allowed)
<i>unimolecular_threshold</i>	Array of flags indicating whether a species is above the unimolecular reaction threshold
<i>bimolecular_threshold</i>	Array of flags indicating whether two species are above the bimolecular reaction threshold

Table 4 – continued from previous page

Attribute	Description
<i>trimolecular_threshold</i>	Array of flags indicating whether three species are above the trimolecular threshold
<i>unimolecular_react</i>	Array of flags indicating whether a species should react unimolecularly in the enlarged network
<i>bimolecular_react</i>	Array of flags indicating whether two species should react in the enlarged network
<i>trimolecular_react</i>	Array of flags indicating whether three species should react in the enlarged network
<i>termination</i>	A list of termination targets (i.e TerminationTime and TerminationConcentration)
<i>species_constraints</i>	Dictates the maximum number of atoms, carbons, electrons, etc. generated for each species
<i>output_directory</i>	The directory used to save output files
<i>verbosity</i>	The level of logging verbosity for console output
<i>units</i>	The unit system to use to save output files (currently must be 'si')
<i>generate_output_html</i>	True to draw pictures of the species and reactions, saving a visualized mechanism
<i>generate_plots</i>	True to generate plots of the job execution statistics after each iteration, False otherwise
<i>verbose_comments</i>	True to keep the verbose comments for database estimates, False otherwise
<i>save_edge_species</i>	True to save chemkin and HTML files of the edge species, False otherwise
<i>keep_irreversible</i>	True to keep irreversibility of library reactions as is ('<=>' or '=>'). False otherwise
<i>trimolecular_product_reversible</i>	True (default) to allow families with trimolecular products to react in the enlarged network
<i>pressure_dependence</i>	Whether to process unimolecular (pressure-dependent) reaction networks
<i>quantum_mechanics</i>	Whether to apply quantum mechanical calculations instead of group additivity
<i>ml_estimator</i>	To use thermo estimation with machine learning
<i>ml_settings</i>	Settings for ML estimation
<i>walltime</i>	The maximum amount of CPU time in the form DD:HH:MM:SS to expend on the job
<i>max_iterations</i>	The maximum number of RMG iterations allowed, after which the job will stop
<i>kinetics_datastore</i>	True if storing details of each kinetic database entry in text file, False otherwise
<i>initialization_time</i>	The time at which the job was initiated, in seconds since the epoch (i.e. from 00:00:00 on 1 Jan 1970)
<i>done</i>	Whether the job has completed (there is nothing new to add)

check_input()

Check for a few common mistakes in the input file.

check_libraries()

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(initialize=True, **kwargs)

Execute an RMG job using the command-line arguments *args* as returned by the *argparse* package. *initialize* is a bool type flag used to determine whether to call *self.initialize()*

finish()

Complete the model generation.

generate_cantera_files(chemkin_file, **kwargs)

Convert a chemkin mechanism *chem.inp* file to a cantera mechanism file *chem.yaml* 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.

initialize_seed_mech()

Initialize the process of saving the seed mechanism by performing the following:

1. Create the initial seed mechanism folder (the seed from a previous iterations will be deleted)
2. Save the restart-from-seed file (unless the current job is itself a restart job)
3. Ensure that we don't overwrite existing libraries in the database that have the same name as this job
4. Create the `previous_seeds` directory to save intermediate seeds if the user gives a value for `saveSeed-Modulus`

load_input(path=None)

Load an RMG job from the input file located at `input_file`, or from the `input_file` attribute if not given as a parameter.

load_thermo_input(path=None)

Load an Thermo Estimation job from a thermo input file located at `input_file`, or from the `input_file` attribute if not given as a parameter.

log_header(level=20)

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

make_seed_mech()

Save a seed mechanism (both core and edge) in the 'seed' sub-folder of the output directory. Additionally, save the filter tensors to the 'seed/filters' sub-folder so that the RMG job can be restarted from a seed mechanism. If `self.save_seed_to_database` is True then the seed mechanism is also saved as libraries (one each for the core and edge) in the RMG-database.

Notes

`initialize_seed_mech` should be called one time before this function is ever called.

make_species_labels_independent(species)

This method looks at the core species labels and makes sure none of them conflict. If a conflict occurs, the second occurrence will have '-2' added. Returns a list of the old labels.

process_pdep_networks(obj)

properly processes PDepNetwork objects and lists of PDepNetwork objects returned from simulate

process_reactions_to_species(obj)

properly processes Reaction objects and lists of Reaction objects returned from simulate

process_to_species_networks(obj)

breaks down the objects returned by simulate into Species and PDepNetwork components

react_init_tuples(requires_rms=False)

Reacts tuples given in the react block

register_listeners(requires_rms=False)

Attaches listener classes depending on the options found in the RMG input file.

run_model_analysis(number=10)

Run sensitivity and uncertainty analysis if requested.

run_uncertainty_analysis()

Run uncertainty analysis if proper settings are available.

save_everything()

Saves the output HTML and the Chemkin file. If the job is being profiled this is saved as well.

save_input(path=None)

Save an RMG job to the input file located at *path*.

update_reaction_threshold_and_react_flags(rxn_sys_unimol_threshold=None, rxn_sys_bimol_threshold=None, rxn_sys_trimol_threshold=None, skip_update=False)

updates the length and boolean value of the unimolecular and bimolecular react and threshold flags

rmgpy.rmg.main.initialize_log(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.make_profile_graph(stats_file, force_graph_generation=False)

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.

Will only generate a graph if a display is present as errors can occur otherwise. If *force_graph_generation* is True then the graph generation will be attempted either way

rmgpy.rmg.main.process_profile_stats(stats_file, log_file)

Saving RMG output

rmgpy.rmg.output.save_output_html(path, reaction_model, part_core_edge='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.save_diff_html(path, common_species_list, species_list1, species_list2, common_reactions, unique_reactions1, unique_reactions2)

This function outputs the species and reactions on an HTML page for the comparison of two RMG models.

rmgpy.rmg.pdep.PDepNetwork

class rmgpy.rmg.pdep.PDepNetwork(index=-1, source=None)

A representation of a *partial* unimolecular reaction network. Each partial network has a single *source* isomer or reactant channel, and is responsible only for $k(T, P)$ values for net reactions with source as the reactant. Multiple partial networks can have the same source, but networks with the same source and any explored isomers must be combined.

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

add_path_reaction(*newReaction*)

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

apply_chemically_significant_eigenvalues_method(*lumping_order=None, method='allen', neglect_high_energy_collisions=False, high_energy_rate_tol=0.01*)

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the chemically-significant eigenvalues method. If a *lumping_order* 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.

apply_modified_strong_collision_method(*efficiency_model='default'*)

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the modified strong collision method.

apply_reservoir_state_method()

Compute the phenomenological rate coefficients $k(T, P)$ at the current conditions using the reservoir state method.

calculate_collision_model()

Calculate the matrix of first-order rate coefficients for collisional population transfer between grains for each isomer, including the corresponding collision frequencies.

calculate_densities_of_states()

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.

calculate_equilibrium_ratios()

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.

calculate_microcanonical_rates()

Calculate and return arrays containing the microcanonical rate coefficients $k(E)$ for the isomerization, dissociation, and association path reactions in the network.

cleanup()

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

explore_isomer(*isomer*)

Explore a previously-unexplored unimolecular *isomer* in this partial network using the provided core-edge reaction model *reaction_model*, returning the new reactions and new species.

get_all_species()

Return a list of all unique species in the network, including all isomers, reactant and product channels, and bath gas species.

get_energy_filtered_reactions(*T, tol*)

Returns a list of products and isomers that are greater in Free Energy than $a \cdot R \cdot T + G_{\text{source}}(T)$

get_leak_branching_ratios(*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.

get_leak_coefficient(*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.

get_maximum_leak_species(*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_rate_filtered_products(*T*, *P*, *tol*)

determines the set of path_reactions that have fluxes less than *tol* at steady state where all $A \Rightarrow B + C$ reactions are irreversible and there is a constant flux from/to the source configuration of 1.0

initialize(*Tmin*, *Tmax*, *Pmin*, *Pmax*, *maximum_grain_size*=0.0, *minimum_grain_count*=0, *active_j_rotor*=True, *active_k_rotor*=True, *rmgmode*=False)

Initialize a pressure dependence calculation by computing several quantities that are independent of the conditions. You must specify the temperature and pressure ranges of interest using *Tmin* and *Tmax* in K and *Pmin* and *Pmax* in Pa. You must also specify the maximum energy grain size *grain_size* in J/mol and/or the minimum number of grains *grain_count*.

invalidate()

Mark the network as in need of a new calculation to determine the pressure-dependent rate coefficients

log_summary(*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.

map_densities_of_states()

Map the overall densities of states to the current energy grains. Semi-logarithmic interpolation will be used if the grain sizes of *Elist0* and *e_list* 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.

remove_disconnected_reactions()

gets rid of reactions/isomers/products not connected to the source by a reaction sequence

remove_reactions(*reaction_model*, *networks*, *rxns*=None, *prods*=None)

removes a list of reactions from the network and all reactions/products left disconnected by removing those reactions

select_energy_grains(*T*, *grain_size*=0.0, *grain_count*=0)

Select a suitable list of energies to use for subsequent calculations. This is done by finding the minimum and maximum energies on the potential energy surface, then adding a multiple of $k_B T$ onto the maximum energy.

You must specify either the desired grain spacing *grain_size* in J/mol or the desired number of grains *n_grains*, as well as a temperature *T* in K to use for the equilibrium calculation. You can specify both *grain_size* and *grain_count*, 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.

set_conditions(*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.

solve_full_me(*tlist*, *x0*)

Directly solve the full master equation using a stiff ODE solver. Pass the reaction *network* to solve, the temperature *T* in K and pressure *P* in Pa to solve at, the energies *e_list* 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 *dens_states* in mol/J of each isomer. Returns the times in s, population distributions for each isomer, and total population profiles for each configuration.

solve_reduced_me(*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 $A \Rightarrow B + C$ reactions are irreversible and the flux from/to the source configuration is 1.0

update(*reaction_model*, *pdep_settings*, *requires_rms=False*)

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

update_configurations(*reaction_model*)

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 *reaction_model* 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,
                                specific_collider=None, network=None, kinetics=None,
                                network_kinetics=None, reversible=True, transition_state=None,
                                duplicate=False, degeneracy=1, pairs=None)
```

apply_solvent_correction(*solvent*)

apply kinetic solvent correction

calculate_coll_limit(*temp*, *reverse=False*)

Calculate the collision limit rate in m³/mol-s for the given temperature implemented as recommended in Wang et al. doi 10.1016/j.combustflame.2017.08.005 (Eq. 1)

calculate_microcanonical_rate_coefficient(*e_list*, *j_list*, *reac_dens_states*,
prod_dens_states=None, *T=0.0*)

Calculate the microcanonical rate coefficient $k(E)$ for the reaction *reaction* at the energies *e_list* in J/mol. *reac_dens_states* and *prod_dens_states* 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 *prod_dens_states* and the temperature of interest *T* in K can also be provided. For isomerization and association reactions *prod_dens_states* 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.

calculate_tst_rate_coefficient(*T*)

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

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

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

can_tst()

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(*t_min*, *t_max*, *p_min*, *p_max*)

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.

copy()

Create a deep copy of the current reaction.

degeneracy

The reaction path degeneracy for this reaction.

If the reaction has kinetics, changing the degeneracy will adjust the reaction rate by a ratio of the new degeneracy to the old degeneracy.

draw(*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.

ensure_species(*reactant_resonance=False*, *product_resonance=False*, *save_order=False*)

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. If *save_order* is `True` the atom order is reset after performing atom isomorphism.

fix_barrier_height(*force_positive=False*, *solvent=""*, *apply_solvation_correction=True*)

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 *force_positive* is `True`, then all reactions are forced to have a non-negative barrier.

fix_diffusion_limited_a_factor(*T*)

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

generate_3d_ts(*reactants, products*)

Generate the 3D structure of the transition state. Called from `model.generate_kinetics()`.

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

generate_high_p_limit_kinetics()

Used for incorporating library reactions with pressure-dependent kinetics in PDep networks. Only implemented for `LibraryReaction`

generate_pairs()

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. This should work most of the time, but a more rigorous algorithm may be needed for some cases.

generate_reverse_rate_coefficient(*network_kinetics=False, Tmin=None, Tmax=None, surface_site_density=0.0*)

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.

If the reaction kinetics model is Sticking Coefficient, please provide a nonzero surface site density in mol/m^2 which is required to evaluate the rate coefficient.

get_enthalpies_of_reaction(*Tlist*)

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

get_enthalpy_of_reaction(*T*)

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

get_entropies_of_reaction(*Tlist*)

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

get_entropy_of_reaction(*T*)

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

get_equilibrium_constant(*T, potential=0.0, type='Kc', surface_site_density=2.5e-05*)

Return the equilibrium constant for the reaction at the specified temperature *T* in K and reference *surface_site_density* in mol/m^2 (2.5e-05 default) The *type* parameter lets you specify the quantities used in the equilibrium constant: *Ka* for activities, *Kc* for concentrations (default), or *Kp* for pressures. This function assumes a reference pressure of 1e5 Pa for gas phases species and uses the ideal gas law to determine reference concentrations. For surface species, the *surface_site_density* is the assumed reference. For protons (H⁺), a reference concentration of 1000 mol/m^3 (1 mol/L) is assumed

get_equilibrium_constants(*Tlist, potential=0.0, type='Kc'*)

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.

get_free_energies_of_reaction(*Tlist*, *potential=0.0*)

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

get_free_energy_of_reaction(*T*, *potential=0.0*)

Return the Gibbs free energy of reaction in J/mol evaluated at temperature *T* in K and potential in Volts (if applicable)

get_mean_sigma_and_epsilon(*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_rate_coefficient(*T*, *P=0.0*, *surface_site_density=0.0*, *potential=0.0*)

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 *diffusion_limiter* 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.

If the reaction has sticking coefficient kinetics, a nonzero surface site density in *mol/m^2* must be provided

get_reduced_mass(*reverse=False*)

Returns the reduced mass of the reactants if *reverse* is **False** Returns the reduced mass of the products if *reverse* is **True**

get_reversible_potential(*T*)

Get the Potential in V at *T* in 'K' at which the charge transfer reaction is at equilibrium

get_source()

Get the source of this PDepReaction

get_stoichiometric_coefficient(*spec*)

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.

get_surface_rate_coefficient(*T*, *surface_site_density*, *potential=0.0*)

Return the overall surface rate coefficient for the forward reaction at temperature *T* in K with surface site density *surface_site_density* in mol/m2. Value is returned in combination of [m,mol,s]

get_url()

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

has_template(*reactants*, *products*)

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

is_association()

Return **True** if the reaction represents an association reaction $A + B \rightleftharpoons C$ or **False** if not.

is_balanced()

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

is_charge_transfer_reaction()

Return **True** if one or more reactants or products are electrons

is_dissociation()

Return **True** if the reaction represents a dissociation reaction $A \rightleftharpoons B + C$ or **False** if not.

is_isomerization()

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

is_isomorphic(*other*, *either_direction=True*, *check_identical=False*, *check_only_label=False*, *check_template_rxn_products=False*, *generate_initial_map=False*, *strict=True*, *save_order=False*)

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.

Parameters

- **either_direction** (*bool*, *optional*) – if False, then the reaction direction must match.
- **check_identical** (*bool*, *optional*) – if True, check that atom ID's match (used for checking degeneracy)
- **check_only_label** (*bool*, *optional*) – if True, only check the string representation, ignoring molecular structure comparisons
- **check_template_rxn_products** (*bool*, *optional*) – if True, only check isomorphism of reaction products (used when we know the reactants are identical, i.e. in generating reactions)
- **generate_initial_map** (*bool*, *optional*) – if True, initialize map by pairing atoms with same labels
- **strict** (*bool*, *optional*) – if False, perform isomorphism ignoring electrons
- **save_order** (*bool*, *optional*) – if True, perform isomorphism saving atom order

is_surface_charge_transfer_reaction()

Return True if one or more reactants or products are electrons

is_surface_reaction()

Return True if one or more reactants or products are surface species (or surface sites)

is_unimolecular()

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

matches_species(*reactants*, *products=None*)

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

protons

The stoichiometric coeff for protons in charge transfer reactions

reverse_arrhenius_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

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

reverse_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, 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).

reverse_sticking_coeff_rate(*k_forward*, *reverse_units*, *surface_site_density*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `StickingCoefficient` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self). The *surface_site_density* in mol/m^2 is used to evaluate the forward rate constant.

reverse_surface_arrhenius_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceArrhenius` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

reverse_surface_charge_transfer_rate(*k_forward*, *reverse_units*, *Tmin=None*, *Tmax=None*)

Reverses the given *k_forward*, which must be a `SurfaceChargeTransfer` type. You must supply the correct units for the reverse rate. The equilibrium constant is evaluated from the current reaction instance (self).

set_reference_potential(*T*)

Set the reference Potential of the `SurfaceChargeTransfer` kinetics model to the reversible potential of the reaction

to_cantera(*species_list=None*, *use_chemkin_identifier=False*)

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

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

to_chemkin(*species_list=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 *species_list* to figure out third body colliders.) Otherwise, only the reaction string will be returned.

to_labeled_str(*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

1.12 Reaction system simulation (rmgpy.solver)

The `rmgpy.solver` module contains classes used to represent and simulate reaction systems.

1.12.1 Reaction systems

Class	Description
<code>ReactionSystem</code>	Base class for all reaction systems
<code>SimpleReactor</code>	A simple isothermal, isobaric, well-mixed batch reactor
<code>LiquidReactor</code>	A homogeneous, isothermal, isobaric liquid batch reactor
<code>SurfaceReactor</code>	A heterogeneous, isothermal, isochoric batch reactor
<code>MBSampledReactor</code>	<code>SimpleReactor</code> with sampling delay for simulating molecular beam experiments

1.12.2 Termination criteria

Class	Description
<i>TerminationTime</i>	Represent a time at which the simulation should be terminated
<i>TerminationConversion</i>	Represent a species conversion at which the simulation should be terminated
<i>TerminationRateRatio</i>	Represent a fraction of the maximum characteristic rate at which the simulation should be terminated

rmgpy.solver.ReactionSystem

class rmgpy.solver.ReactionSystem

A base class for all RMG reaction systems.

add_reactions_to_surface(*new_surface_reactions*, *new_surface_reaction_inds*, *surface_species*, *surface_reactions*, *edge_species*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

advance(*tout*)

Simulate from the current value of the independent variable to a specified value *tout*, taking as many steps as necessary. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

compute_network_variables(*pdep_networks=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.

compute_rate_derivative()

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

generate_reactant_product_indices(*core_reactions*, *edge_reactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*core_reactions*, *edge_reactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*core_species*, *edge_species*)

Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_layering_indices()

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(*spc*)

Retrieves the index that is associated with the parameter species from the species index dictionary.

initialize(*t0*, *y0*, *dydt0=None*, *senpar=None*, *atol=1e-16*, *rtol=1e-08*)

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.

initialize_model(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species*=None, *surface_reactions*=None, *pdep_networks*=None, *atol*=1e-16, *rtol*=1e-08, *sensitivity*=False, *sens_atol*=1e-06, *sens_rtol*=0.0001, *filter_reactions*=False, *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_surface(*core_species*, *core_reactions*, *surface_species*, *surface_reactions*)

removes surface_species and surface_reactions 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(*atol*=1e-16, *rtol*=1e-08, *sensitivity*=False, *sens_atol*=1e-06, *sens_rtol*=0.0001)

Computes the number of differential equations and initializes the tolerance arrays.

log_conversions(*species_index*, *y0*)

Log information about the current conversion values.

log_rates(*char_rate*, *species*, *species_rate*, *max_dif_ln_accum_num*, *network*, *network_rate*)

Log information about the current maximum species and network rates.

reset_max_edge_species_rate_ratios()

This function sets *max_edge_species_rate_ratios* back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*t*, *y*, *dydt*, *senpar*)

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).

set_initial_conditions()

Sets the common initial conditions of the rate equations that represent the reaction system.

- Sets the initial time of the reaction system to 0
- Initializes the species moles to a *n* x 1 array with zeros

set_initial_derivative()

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species*, *surface_reactions*, *pdep_networks*=None, *prune*=False, *sensitivity*=False, *sens_worksheet*=None, *model_settings*=None, *simulator_settings*=None, *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.

step(*tout*)

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value *tout*. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

rmgpy.solver.SimpleReactor**class rmgpy.solver.SimpleReactor**

A reaction system consisting of a homogeneous, isothermal, isobaric batch reactor. These assumptions allow for a number of optimizations that enable this solver to complete very rapidly, even for large kinetic models.

add_reactions_to_surface(*new_surface_reactions*, *new_surface_reaction_inds*, *surface_species*, *surface_reactions*, *edge_species*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

advance(*tout*)

Simulate from the current value of the independent variable to a specified value *tout*, taking as many steps as necessary. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

calculate_effective_pressure(*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:

$$P_{eff} = \frac{P * y_{specific_collider}}{\sum_j y_j}$$

if a *specific_collider* is mentioned.

compute_network_variables(*pdep_networks=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.

compute_rate_derivative()

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

convert_initial_keys_to_species_objects(*species_dict*)

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

generate_rate_coefficients(*core_reactions*, *edge_reactions*)

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(*core_reactions*, *edge_reactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*core_reactions*, *edge_reactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*core_species*, *edge_species*)

Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_const_spc_indices(*core_species*)

Allow to identify constant Species position in solver

get_layering_indices()

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(*spc*)

Retrieves the index that is associated with the parameter species from the species index dictionary.

get_threshold_rate_constants(*model_settings*)

Get the threshold rate constants for reaction filtering.

initialize(*t0*, *y0*, *dydt0*=None, *senpar*=None, *atol*=1e-16, *rtol*=1e-08)

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.

initialize_model(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species*=None, *surface_reactions*=None, *pdep_networks*=None, *atol*=1e-16, *rtol*=1e-08, *sensitivity*=False, *sens_atol*=1e-06, *sens_rtol*=0.0001, *filter_reactions*=False, *conditions*=None)

Initialize a simulation of the simple reactor using the provided kinetic model.

initialize_surface(*core_species*, *core_reactions*, *surface_species*, *surface_reactions*)

removes surface_species and surface_reactions 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(*atol*=1e-16, *rtol*=1e-08, *sensitivity*=False, *sens_atol*=1e-06, *sens_rtol*=0.0001)

Computes the number of differential equations and initializes the tolerance arrays.

jacobian(*t*, *y*, *dydt*, *cj*, *senpar*=array([0.]))

Return the analytical Jacobian for the reaction system.

log_conversions(*species_index*, *y0*)

Log information about the current conversion values.

log_rates(*char_rate*, *species*, *species_rate*, *max_dif_ln_accum_num*, *network*, *network_rate*)

Log information about the current maximum species and network rates.

reset_max_edge_species_rate_ratios()

This function sets *max_edge_species_rate_ratios* back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*t*, *y*, *dydt*, *senpar*=array([0.]))

Return the residual function for the governing DAE system for the simple reaction system.

set_colliders(*core_reactions*, *edge_reactions*, *core_species*)

Store collider efficiencies and reaction indices for pdep reactions that have collider efficiencies, and store specific collider indices

set_initial_conditions()

Sets the initial conditions of the rate equations that represent the current reactor model.

The volume is set to the value derived from the ideal gas law, using the user-defined pressure, temperature, and the number of moles of initial species.

The species moles array (*y0*) is set to the values stored in the initial mole fractions dictionary.

The initial species concentration is computed and stored in the *core_species_concentrations* array.

set_initial_derivative()

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species*, *surface_reactions*, *pdep_networks*=None, *prune*=False, *sensitivity*=False, *sens_worksheet*=None, *model_settings*=None, *simulator_settings*=None, *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.

step(*tout*)

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value *tout*. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

rmgpy.solver.LiquidReactor

class rmgpy.solver.LiquidReactor

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.

add_reactions_to_surface(*new_surface_reactions*, *new_surface_reaction_inds*, *surface_species*, *surface_reactions*, *edge_species*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

advance(*tout*)

Simulate from the current value of the independent variable to a specified value *tout*, taking as many steps as necessary. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

compute_network_variables(*pdep_networks*=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.

compute_rate_derivative()

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

convert_initial_keys_to_species_objects(*species_dict*)

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

generate_rate_coefficients(*core_reactions*, *edge_reactions*)

Populates the `forwardRateCoefficients`, `reverseRateCoefficients` and `equilibriumConstants` arrays with the values computed at the temperature and (effective) pressure of the reaction system.

generate_reactant_product_indices(*core_reactions*, *edge_reactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*core_reactions*, *edge_reactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*core_species*, *edge_species*)

Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_const_spc_indices(*core_species*)

Allow to identify constant Species position in solver

get_layering_indices()

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(*spc*)

Retrieves the index that is associated with the parameter species from the species index dictionary.

get_threshold_rate_constants(*model_settings*)

Get the threshold rate constants for reaction filtering.

`model_settings` is not used here, but is needed so that the method matches the one in `simpleReactor`.

initialize(*t0*, *y0*, *dydt0=None*, *senpar=None*, *atol=1e-16*, *rtol=1e-08*)

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.

initialize_model(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species=None*, *surface_reactions=None*, *pdep_networks=None*, *atol=1e-16*, *rtol=1e-08*, *sensitivity=False*, *sens_atol=1e-06*, *sens_rtol=0.0001*, *filter_reactions=False*, *conditions=None*)

Initialize a simulation of the liquid reactor using the provided kinetic model.

initialize_surface(*core_species*, *core_reactions*, *surface_species*, *surface_reactions*)

removes surface_species and surface_reactions 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(*atol=1e-16*, *rtol=1e-08*, *sensitivity=False*, *sens_atol=1e-06*, *sens_rtol=0.0001*)

Computes the number of differential equations and initializes the tolerance arrays.

jacobian(*t, y, dydt, cj, senpar=array([0.])*)

Return the analytical Jacobian for the reaction system.

log_conversions(*species_index, y0*)

Log information about the current conversion values.

log_rates(*char_rate, species, species_rate, max_dif_ln_accum_num, network, network_rate*)

Log information about the current maximum species and network rates.

reset_max_edge_species_rate_ratios()

This function sets max_edge_species_rate_ratios back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*t, y, dydt, senpar=array([0.])*)

Return the residual function for the governing DAE system for the liquid reaction system.

set_initial_conditions()

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 core_species_concentrations 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()

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate(*core_species, core_reactions, edge_species, edge_reactions, surface_species, surface_reactions, pdep_networks=None, prune=False, sensitivity=False, sens_worksheet=None, model_settings=None, simulator_settings=None, 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.

step(*tout*)

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value *tout*. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

rmgpy.solver.SurfaceReactor

class rmgpy.solver.SurfaceReactor

A reaction system consisting of a heterogeneous, isothermal, constant volume batch reactor.

add_reactions_to_surface(*new_surface_reactions, new_surface_reaction_inds, surface_species, surface_reactions, edge_species*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

advance(*tout*)

Simulate from the current value of the independent variable to a specified value *tout*, taking as many steps as necessary. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

compute_network_variables(*pdep_networks=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.

compute_rate_derivative()

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

convert_initial_keys_to_species_objects(*species_dict*)

Convert the `initial_gas_mole_fractions` and `initial_surface_coverages` dictionaries from species names into species objects, using the given dictionary of species.

generate_rate_coefficients(*core_reactions, edge_reactions*)

Populates the `kf`, `kb` and `equilibriumConstants` arrays with the values computed at the temperature and (effective) pressure of the reaction system.

generate_reactant_product_indices(*core_reactions, edge_reactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*core_reactions, edge_reactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*core_species, edge_species*)

Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_layering_indices()

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(*spc*)

Retrieves the index that is associated with the parameter species from the species index dictionary.

get_threshold_rate_constants(*model_settings*)

Get the threshold rate constants for reaction filtering.

initialize(*t0, y0, dydt0=None, senpar=None, atol=1e-16, rtol=1e-08*)

Initialize the DASPK solver by setting the initial values of the independent variable t_0 , dependent variables y_0 , and first derivatives $dydt_0$. If provided, the derivatives must be consistent with the other initial conditions; if not provided, DASPK will attempt to estimate a consistent set of initial values for the derivatives. You can also set the absolute and relative tolerances $atol$ and $rtol$, respectively, either as single values for all dependent variables or individual values for each dependent variable.

initialize_model(*core_species, core_reactions, edge_species, edge_reactions, surface_species=[], surface_reactions=[], pdep_networks=None, atol=1e-16, rtol=1e-08, sensitivity=False, sens_atol=1e-06, sens_rtol=0.0001, filter_reactions=False, conditions=None*)

Initialize a simulation of the simple reactor using the provided kinetic model.

initialize_surface(*core_species, core_reactions, surface_species, surface_reactions*)

removes surface_species and surface_reactions 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(*atol=1e-16, rtol=1e-08, sensitivity=False, sens_atol=1e-06, sens_rtol=0.0001*)

Computes the number of differential equations and initializes the tolerance arrays.

log_conversions(*species_index, y0*)

Log information about the current conversion values.

log_initial_conditions(*number=None*)

Log to the console some information about this reaction system.

Should correspond to the calculations done in `set_initial_conditions`.

log_rates(*char_rate, species, species_rate, max_dif_ln_accum_num, network, network_rate*)

Log information about the current maximum species and network rates.

reset_max_edge_species_rate_ratios()

This function sets `max_edge_species_rate_ratios` back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*t, N, dNdt, senpar=array([0.])*)

Return the residual function for the governing DAE system for the simple reaction system.

set_initial_conditions()

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 gas phase core species at start.

The total surface sites are calculated from `surface_volume_ratio` and `surface_site_density` allowing `initial_surface_coverages` to determine the number of moles of surface species. The number of moles of gas phase species is taken from `initial_gas_mole_fractions`.

The `core_species_concentrations` array is then determined, in `mol/m3` for gas phase and `mol/m2` for surface species.

The initial number of moles of a species `j` in the reactor is computed and stored in the `y0` instance attribute.

set_initial_derivative()

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate(*core_species, core_reactions, edge_species, edge_reactions, surface_species, surface_reactions, pdep_networks=None, prune=False, sensitivity=False, sens_worksheet=None, model_settings=None, simulator_settings=None, 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.

step(*tout*)

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value `tout`. The resulting values of `t`, `y`, and $\frac{dy}{dt}$ can then be accessed via the `t`, `y`, and `dydt` attributes.

rmgpy.solver.MBSampledReactor**class rmgpy.solver.MBSampledReactor**

A reaction system consisting of a homogeneous, isothermal, isobaric batch reactor that is being sampled by a molecular beam. The sampling process is modeled as a unimolecular reaction. These assumptions allow for a number of optimizations that enable this solver to complete very rapidly, even for large kinetic models.

This is currently only intended for use with the `simulate.py` script, and cannot be used for a standard RMG job.

add_reactions_to_surface(*new_surface_reactions*, *new_surface_reaction_inds*, *surface_species*, *surface_reactions*, *edge_species*)

moves new surface reactions to the surface done after the while loop before the simulate call ends

advance(*tout*)

Simulate from the current value of the independent variable to a specified value *tout*, taking as many steps as necessary. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

calculate_effective_pressure(*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:

$$P_{eff} = \frac{P * y_{specific_collider}}{\sum_j y_j}$$

if a *specific_collider* is mentioned.

compute_network_variables(*pdep_networks=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.

compute_rate_derivative()

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

convert_initial_keys_to_species_objects(*species_dict*)

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

generate_rate_coefficients(*core_reactions*, *edge_reactions*)

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(*core_reactions*, *edge_reactions*)

Creates a matrix for the reactants and products.

generate_reaction_indices(*core_reactions*, *edge_reactions*)

Assign an index to each reaction (core first, then edge) and store the (reaction, index) pair in a dictionary.

generate_species_indices(*core_species*, *edge_species*)

Assign an index to each species (core first, then edge) and store the (species, index) pair in a dictionary.

get_layering_indices()

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(*spc*)

Retrieves the index that is associated with the parameter species from the species index dictionary.

initialize(*t0*, *y0*, *dydt0=None*, *senpar=None*, *atol=1e-16*, *rtol=1e-08*)

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.

initialize_model(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species=None*, *surface_reactions=None*, *pdep_networks=None*, *atol=1e-16*, *rtol=1e-08*, *sensitivity=False*, *sens_atol=1e-06*, *sens_rtol=0.0001*, *filter_reactions=False*, *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_surface(*core_species*, *core_reactions*, *surface_species*, *surface_reactions*)

removes surface_species and surface_reactions 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(*atol=1e-16*, *rtol=1e-08*, *sensitivity=False*, *sens_atol=1e-06*, *sens_rtol=0.0001*)

Computes the number of differential equations and initializes the tolerance arrays.

log_conversions(*species_index*, *y0*)

Log information about the current conversion values.

log_rates(*char_rate*, *species*, *species_rate*, *max_dif_ln_accum_num*, *network*, *network_rate*)

Log information about the current maximum species and network rates.

reset_max_edge_species_rate_ratios()

This function sets *max_edge_species_rate_ratios* back to zero for pruning of ranged reactors it is important to avoid doing this every initialization

residual(*t*, *y*, *dydt*, *senpar=array([0.])*)

Return the residual function for the governing DAE system for the simple reaction system.

set_colliders(*core_reactions*, *edge_reactions*, *core_species*)

Store collider efficiencies and reaction indices for pdep reactions that have collider efficiencies, and store specific collider indices

set_initial_conditions()

Sets the initial conditions of the rate equations that represent the current reactor model.

The volume is set to the value derived from the ideal gas law, using the user-defined pressure, temperature, and the number of moles of initial species.

The species moles array (*y0*) is set to the values stored in the initial mole fractions dictionary.

The initial species concentration is computed and stored in the *core_species_concentrations* array.

set_initial_derivative()

Sets the derivative of the species moles with respect to the independent variable (time) equal to the residual.

simulate(*core_species*, *core_reactions*, *edge_species*, *edge_reactions*, *surface_species*, *surface_reactions*,
pdep_networks=None, *prune*=False, *sensitivity*=False, *sens_worksheet*=None,
model_settings=None, *simulator_settings*=None, *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.

step(*tout*)

Perform one simulation step from the current value of the independent variable toward (but not past) a specified value *tout*. The resulting values of *t*, *y*, and $\frac{dy}{dt}$ can then be accessed via the *t*, *y*, and *dydt* attributes.

Termination criteria

class *rmgpy.solver.TerminationTime*(*time*=(0.0, 's'))

Represent a time at which the simulation should be terminated. This class has one attribute: the termination *time* in seconds.

class *rmgpy.solver.TerminationConversion*(*spec*=None, *conv*=0.0)

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.

class *rmgpy.solver.TerminationRateRatio*(*ratio*=0.01)

Represent a fraction of the maximum characteristic rate of the simulation at which the simulation should be terminated. This class has one attribute the ratio between the current and maximum characteristic rates at which to terminate

1.13 Species (*rmgpy.species*)

The *rmgpy.species* subpackage contains classes and functions for working with chemical species.

1.13.1 Species

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

1.13.2 Transition state

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

rmgpy.species.Species

class rmgpy.species.Species

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

Attribute	Description
<i>index</i>	A unique nonnegative integer index
<i>label</i>	A descriptive string label
<i>thermo</i>	The heat capacity model for the species
<i>conformer</i>	The molecular conformer for the species
<i>molecule</i>	A list of the <code>Molecule</code> objects describing the molecular structure
<i>transport_data</i>	A set of transport collision parameters
<i>molecular_weight</i>	The molecular weight of the species
<i>energy_transfer_model</i>	The collisional energy transfer model to use
<i>reactive</i>	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
<i>props</i>	A generic 'properties' dictionary to store user-defined flags
<i>aug_inchi</i>	Unique augmented inchi
<i>symmetry_number</i>	Estimated symmetry number of the species, using the resonance hybrid
<i>creation_iteration</i>	Iteration which the species is created within the reaction mechanism generation algorithm
<i>explicitly_allowed</i>	Flag to exempt species from forbidden structure checks

calculate_cp0()

Return the value of the heat capacity at zero temperature in J/mol*K.

calculate_cpinf()

Return the value of the heat capacity at infinite temperature in J/mol*K.

contains_surface_site()

Return True if the species is adsorbed on a surface (or is itself a site), else False.

copy(*deep=False*)

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.

fingerprint

Fingerprint of this species, taken from molecule attribute. Read-only.

from_adjacency_list(*adjlist*, *raise_atomtype_exception=True*, *raise_charge_exception=False*)

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.

from_smiles(*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.

generate_energy_transfer_model()

Generate the collisional energy transfer model parameters for the species. This "algorithm" is very much in need of improvement.

generate_resonance_structures(*keep_isomorphic=True*, *filter_structures=True*, *save_order=False*)

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. If *save_order* is True the atom order is reset after performing atom isomorphism.

generate_statmech()

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

generate_transport_data()

Generate the transport_data parameters for the species.

get_density_of_states(*e_list*)

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state.

get_enthalpy(*T*)

Return the enthalpy in J/mol for the species at the specified temperature *T* in K.

get_entropy(*T*)

Return the entropy in J/mol*K for the species at the specified temperature *T* in K.

get_free_energy(*T*)

Return the Gibbs free energy in J/mol for the species at the specified temperature *T* in K.

get_heat_capacity(*T*)

Return the heat capacity in J/mol*K for the species at the specified temperature *T* in K.

get_net_charge()

Iterate through the atoms in the structure and calculate the net charge on the overall molecule.

get_partition_function(*T*)

Return the partition function for the species at the specified temperature *T* in K.

get_resonance_hybrid()

Returns a molecule object with bond orders that are the average of all the resonance structures.

get_sum_of_states(*e_list*)

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol.

get_symmetry_number()

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 calculate_symmetry_number calls.

get_thermo_data(solvent_name="")

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.

get_transport_data()

Returns the transport data associated with this species, and calculates it if it is not yet available.

has_reactive_molecule()

True if the species has at least one reactive molecule, *False* otherwise

has_statmech()

Return *True* if the species has statistical mechanical parameters, or *False* otherwise.

has_thermo()

Return *True* if the species has thermodynamic parameters, or *False* otherwise.

inchi

InChI string representation of this species. Read-only.

is_electron()

Return *True* if the species is an electron

is_identical(*other*, *strict=True*)

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.

If *strict=False*, performs the check ignoring electrons and resonance structures.

is_isomorphic(*other*, *generate_initial_map=False*, *save_order=False*, *strict=True*)

Return *True* if the species is isomorphic to *other*, which can be either a *Molecule* object or a *Species* object.

Parameters

- **generate_initial_map** (*bool*, *optional*) – If *True*, make initial map by matching labeled atoms
- **save_order** (*bool*, *optional*) – if *True*, reset atom order after performing atom isomorphism
- **strict** (*bool*, *optional*) – If *False*, perform isomorphism ignoring electrons.

is_proton()

Return True if the species is a proton

is_structure_in_list(*species_list*)

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.

is_surface_site()

Return True if the species is a vacant surface site.

molecular_weight

value_si is in kg/molecule not kg/mol)

Type

The molecular weight of the species. (Note

multiplicity

Fingerprint of this species, taken from molecule attribute. Read-only.

number_of_surface_sites()

Return the number of surface sites for a species. eg. 2 for bidentate.

set_e0_with_thermo()

Helper method that sets species' E0 using the species' thermo data

set_structure(*structure*)

Set self.molecule from *structure* which could be either a SMILES string or an adjacency list multi-line string

smiles

SMILES string representation of this species. Read-only.

Note that SMILES representations for different resonance structures of the same species may be different.

sorting_key

Returns a sorting key for comparing Species objects. Read-only

to_adjacency_list()

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

to_cantera(*use_chemkin_identifier=False*)

Converts the RMG Species object to a Cantera Species object with the appropriate thermo data.

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

to_chemkin()

Return the chemkin-formatted string for this species.

rmgpy.species.TransitionState**class rmgpy.species.TransitionState**

A chemical transition state, representing a first-order saddle point on a potential energy surface. The attributes are:

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

calculate_tunneling_factor(*T*)

Calculate and return the value of the canonical tunneling correction factor for the reaction at the given temperature *T* in K.

calculate_tunneling_function(*e_list*)

Calculate and return the value of the microcanonical tunneling correction for the reaction at the given energies *e_list* in J/mol.

frequency

The negative frequency of the first-order saddle point.

get_density_of_states(*e_list*)

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state.

get_enthalpy(*T*)

Return the enthalpy in J/mol for the transition state at the specified temperature *T* in K.

get_entropy(*T*)

Return the entropy in J/mol*K for the transition state at the specified temperature *T* in K.

get_free_energy(*T*)

Return the Gibbs free energy in J/mol for the transition state at the specified temperature *T* in K.

get_heat_capacity(*T*)

Return the heat capacity in J/mol*K for the transition state at the specified temperature *T* in K.

get_partition_function(*T*)

Return the partition function for the transition state at the specified temperature *T* in K.

get_sum_of_states(*e_list*)

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol.

1.14 Statistical mechanics (rmgpy.statmech)

The *rmgpy.statmech* subpackage contains classes that represent various statistical mechanical models of molecular degrees of freedom. These models enable the computation of macroscopic parameters (e.g. thermodynamics, kinetics, etc.) from microscopic parameters.

A molecular system consisting of *N* atoms is described by $3N$ molecular degrees of freedom. Three of these modes involve translation of the system as a whole. Another three of these modes involve rotation of the system as a whole, unless the system is linear (e.g. diatomics), for which there are only two rotational modes. The remaining $3N - 6$ (or

$3N - 5$ if linear) modes involve internal motion of the atoms within the system. Many of these modes are well-described as harmonic oscillations, while others are better modeled as torsional rotations around a bond within the system.

Molecular degrees of freedom are mathematically represented using the Schrodinger equation $\hat{H}\Psi = E\Psi$. By solving the Schrodinger equation, we can determine the available energy states of the molecular system, which enables computation of macroscopic parameters. Depending on the temperature of interest, some modes (e.g. vibrations) require a quantum mechanical treatment, while others (e.g. translation, rotation) can be described using a classical solution.

1.14.1 Translational degrees of freedom

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

1.14.2 Rotational degrees of freedom

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

1.14.3 Vibrational degrees of freedom

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

1.14.4 Torsional degrees of freedom

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

1.14.5 The Schrodinger equation

Class	Description
<i>get_partition_function()</i>	Calculate the partition function at a given temperature from energy levels and degeneracies
<i>get_heat_capacity()</i>	Calculate the dimensionless heat capacity at a given temperature from energy levels and degeneracies
<i>get_enthalpy()</i>	Calculate the enthalpy at a given temperature from energy levels and degeneracies
<i>get_entropy()</i>	Calculate the entropy at a given temperature from energy levels and degeneracies
<i>get_sum_of_states()</i>	Calculate the sum of states for a given energy domain from energy levels and degeneracies
<i>get_density_of_states()</i>	Calculate the density of states for a given energy domain from energy levels and degeneracies

1.14.6 Convolution

Class	Description
<code>convolve()</code>	Return the convolution of two arrays
<code>convolve_bs()</code>	Convolve a degree of freedom into a density or sum of states using the Beyer-Swinehart (BS) direct count algorithm
<code>convolve_bssr()</code>	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
<code>Conformer</code>	A model of a molecular conformation

Translational degrees of freedom

class `rmgpy.statmech.IdealGasTranslation`(*mass=None, quantum=False*)

A statistical mechanical model of translation in an 3-dimensional infinite square well by an ideal gas. The attributes are:

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

Translational energies are much smaller than $k_B T$ except for temperatures approaching absolute zero, so a classical treatment of translation is more than adequate.

The translation of an *ideal gas* – a gas composed of randomly-moving, noninteracting particles of negligible size – in three dimensions can be modeled using the particle-in-a-box model. In this model, a gas particle is confined to a three-dimensional box of size $L_x L_y L_z = V$ with the following potential:

$$V(x, y, z) = \begin{cases} 0 & 0 \leq x \leq L_x, 0 \leq y \leq L_y, 0 \leq z \leq L_z \\ \infty & \text{otherwise} \end{cases}$$

The time-independent Schrodinger equation for this system (within the box) is given by

$$-\frac{\hbar^2}{2M} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \Psi(x, y, z) = E \Psi(x, y, z)$$

where M is the total mass of the particle. Because the box is finite in all dimensions, the solution of the above is quantized with the following energy levels:

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

Above we have introduced n_x , n_y , and n_z as quantum numbers. The quantum mechanical partition function is obtained by summing over the above energy levels:

$$Q_{\text{trans}}(T) = \sum_{n_x=1}^{\infty} \sum_{n_y=1}^{\infty} \sum_{n_z=1}^{\infty} \exp \left(-\frac{E_{n_x, n_y, n_z}}{k_B T} \right)$$

In almost all cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the translational partition function in the classical limit:

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

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

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

get_sum_of_states(*self*, ndarray *e_list*, ndarray *sum_states_0*=None) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

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

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default.

A linear rigid rotor is modeled as a pair of point masses m_1 and m_2 separated by a distance R . Since we are modeling the rotation of this system, we choose to work in spherical coordinates. Following the physics convention – where $0 \leq \theta \leq \pi$ is the zenith angle and $0 \leq \phi \leq 2\pi$ is the azimuth – the Schrodinger equation for the rotor is given by

$$-\frac{\hbar^2}{2I} \left[\frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} \right] \Psi(\theta, \phi) = E \Psi(\theta, \phi)$$

where $I \equiv \mu R^2$ is the moment of inertia of the rotating body, and $\mu \equiv m_1 m_2 / (m_1 + m_2)$ is the reduced mass. Note that there is no potential term in the above expression; for this reason, a rigid rotor is often referred to as a *free* rotor. Solving the Schrodinger equation gives the energy levels E_J and corresponding degeneracies g_J for the linear rigid rotor as

$$\begin{aligned} E_J &= B J(J+1) & J &= 0, 1, 2, \dots \\ g_J &= 2J+1 \end{aligned}$$

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:

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

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the linear rotor partition function in the classical limit:

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

Above we have also introduced σ as the symmetry number of the rigid rotor.

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

Return the degeneracy of level *J*.

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

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

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

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

get_sum_of_states(*self*, ndarray *e_list*, ndarray *sum_states_0*=None) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

inertia

The moment of inertia of the rotor.

make_object(*self*, dict *data*, dict *class_dict*)

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

Note that the moments of inertia and the rotational constants are simply two ways of representing the same quantity; only one set of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default. In the current implementation, the quantum mechanical model has not been implemented, and a `NotImplementedError` will be raised if you try to use it.

A nonlinear rigid rotor is the generalization of the linear rotor to a nonlinear polyatomic system. Such a system is characterized by three moments of inertia I_A , I_B , and I_C instead of just one. The solution to the Schrodinger equation for the quantum nonlinear rotor is not well defined, so we will simply show the classical result instead:

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

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

get_sum_of_states(*self*, ndarray *e_list*, ndarray *sum_states_0*=None) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

inertia

The moments of inertia of the rotor.

make_object(*self*, dict *data*, dict *class_dict*)

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

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the K-rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default.

The energy levels E_K of the K-rotor are given by

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

where K is the quantum number for the rotor and $B \equiv \hbar^2/2I$ is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition function for the K-rotor:

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

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the linear rotor partition function in the classical limit:

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

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

get_density_of_states(*self, ndarray e_list, ndarray dens_states_0=None*) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

Return the degeneracy of level *J*.

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

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

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

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

get_sum_of_states(*self*, *ndarray e_list*, *ndarray sum_states_0=None*) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

inertia

The moment of inertia of the rotor.

make_object(*self*, *dict data*, *dict class_dict*)

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

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

In the majority of chemical applications, the energies involved in the rigid rotor place it very nearly in the classical limit at all relevant temperatures; therefore, the classical model is used by default.

A spherical top rotor is simply the three-dimensional equivalent of a linear rigid rotor. Unlike the nonlinear rotor, all three moments of inertia of a spherical top are equal, i.e. $I_A = I_B = I_C = I$. The energy levels E_J and corresponding degeneracies g_J of the spherical top rotor are given by

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

where J is the quantum number for the rotor and $B \equiv \hbar^2/2I$ is the rotational constant.

Using these expressions for the energy levels and corresponding degeneracies, we can evaluate the partition function for the spherical top rotor:

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

In many cases the temperature of interest is large relative to the energy spacing; in this limit we can obtain a closed-form analytical expression for the linear rotor partition function in the classical limit:

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

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

Return the degeneracy of level *J*.

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

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

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

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

get_sum_of_states(*self*, *ndarray e_list*, *ndarray sum_states_0=None*) → *ndarray*

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

inertia

The moment of inertia of the rotor.

make_object(*self*, *dict data*, *dict class_dict*)

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

In the majority of chemical applications, the energy levels of the harmonic oscillator are of similar magnitude to $k_{\text{B}}T$, requiring a quantum mechanical treatment. Fortunately, the harmonic oscillator has an analytical quantum mechanical solution.

Many vibrational motions are well-described as one-dimensional quantum harmonic oscillators. The time-independent Schrodinger equation for such an oscillator is given by

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

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

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

Above we have introduced n as the quantum number. Note that, even in the ground state ($n = 0$), the harmonic oscillator has an energy that is not zero; this energy is called the *zero-point energy*.

The harmonic oscillator partition function is obtained by summing over the above energy levels:

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

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

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

In RMG the convention is to place the zero-point energy in with the ground-state energy of the system instead of the numerator of the vibrational partition function, which gives

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

The energy levels of the harmonic oscillator in chemical systems are often significant compared to the temperature of interest, so we usually use the quantum result. However, the classical limit is provided here for completeness:

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

frequencies

The vibrational frequencies of the oscillators.

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

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

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

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

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

get_sum_of_states(*self*, ndarray *e_list*, ndarray *sum_states_0*=None) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

quantum

‘bool’

Type

quantum

Torsional degrees of freedom

class `rmgpy.statmech.HinderedRotor`(*inertia=None*, *symmetry=1*, *barrier=None*, *fourier=None*, *rotationalConstant=None*, *quantum=True*, *semiclassical=True*, *frequency=None*, *energies=None*)

A statistical mechanical model of a one-dimensional hindered rotor. The attributes are:

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

Note that the moment of inertia and the rotational constant are simply two ways of representing the same quantity; only one of these can be specified independently.

The Schrodinger equation for a one-dimensional hindered rotor is given by

$$-\frac{\hbar^2}{2I} \frac{d^2}{d\phi^2} \Psi(\phi) + V(\phi) \Psi(\phi) = E \Psi(\phi)$$

where I is the reduced moment of inertia of the torsion and $V(\phi)$ describes the potential of the torsion. There are two common forms for the potential: a simple cosine of the form

$$V(\phi) = \frac{1}{2} V_0 (1 - \cos \sigma \phi)$$

where V_0 is the barrier height and σ is the symmetry number, or a more general Fourier series of the form

$$V(\phi) = A + \sum_{k=1}^C (a_k \cos k\phi + b_k \sin k\phi)$$

where A , a_k and b_k are fitted coefficients. Both potentials are typically defined such that the minimum of the potential is zero and is found at $\phi = 0$.

For either the cosine or Fourier series potentials, the energy levels of the quantum hindered rotor must be determined numerically. The cosine potential does permit a closed-form representation of the classical partition function, however:

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

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

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

Above we have defined ν as the vibrational frequency of the hindered rotor:

$$\nu \equiv \frac{\sigma}{2\pi} \sqrt{\frac{V_0}{2I}}$$

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

barrier

The barrier height of the cosine potential.

energies

numpy.ndarray

Type

energies

fit_cosine_potential_to_data(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.

fit_fourier_potential_to_data(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

get_density_of_states(*self*, ndarray *e_list*, ndarray *dens_states_0*=None) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* in J/mol above the ground state. If an initial density of states *dens_states_0* is given, the rotor density of states will be convoluted into these states.

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

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

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

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

get_frequency(*self*) → double

Return the frequency of vibration in cm⁻¹ corresponding to the limit of harmonic oscillation.

get_hamiltonian(*self*, int *n_basis*) → ndarray

Return the to the Hamiltonian matrix for the hindered rotor for the given number of basis functions *n_basis*. The Hamiltonian matrix is returned in banded lower triangular form and with units of J/mol.

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

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

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

Return the degeneracy of level *J*.

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

Return the energy of level *J* in J.

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

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

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

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

get_sum_of_states(*self*, ndarray *e_list*, ndarray *sum_states_0*=None) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in J/mol above the ground state. If an initial sum of states *sum_states_0* is given, the rotor sum of states will be convoluted into these states.

inertia

The moment of inertia of the rotor.

make_object(*self*, dict *data*, dict *class_dict*)

quantum

‘bool’

Type

quantum

rotationalConstant

The rotational constant of the rotor.

semiclassical

‘bool’

Type

semiclassical

solve_schrodinger_equation(*self*, *int n_basis=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 *n_basis* 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.convolve_bs(*ndarray e_list*, *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 *e_list* (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.convolve_bssr(*ndarray e_list*, *ndarray rho0*, *energy*,
degeneracy=unit_degeneracy, *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 *e_list* (in J/mol) and the energy levels corresponding to the solution of the Schrodinger equation.

rmgpy.statmech.schrodinger.get_density_of_states(*ndarray e_list*, *energy*,
degeneracy=unit_degeneracy, *int n0=0*, *ndarray dens_states_0=None*) → ndarray

Return the values of the dimensionless density of states $\rho(E) dE$ for a given set of energies *e_list* in J/mol above the ground state using an initial density of states *dens_states_0*. 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.get_enthalpy(*double T*, *energy*, *degeneracy=unit_degeneracy*, *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.get_entropy(*double T*, *energy*, *degeneracy=unit_degeneracy*, *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.get_heat_capacity(double T, energy, degeneracy=unit_degeneracy, int n0=0, int nmax=10000, double tol=1e-12) → double`

Return the value of the dimensionless heat capacity $C_v(T)/R$ at a given temperature T in K. The solution to the Schrodinger equation is given using functions *energy* and *degeneracy* that accept as argument a quantum number and return the corresponding energy in J/mol and degeneracy of that level. The quantum number always begins at $n0$ and increases by ones. You can also change the relative tolerance tol and the maximum allowed value of the quantum number $nmax$.

`rmgpy.statmech.schrodinger.get_partition_function(double T, energy, degeneracy=unit_degeneracy, 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.get_sum_of_states(ndarray e_list, energy, degeneracy=unit_degeneracy, int n0=0, ndarray sum_states_0=None) → ndarray`

Return the values of the sum of states $N(E)$ for a given set of energies *e_list* in J/mol above the ground state using an initial sum of states *sum_states_0*. 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.unit_degeneracy(n)`

rmgpy.statmech.Conformer

class `rmgpy.statmech.Conformer(E0=None, modes=None, spin_multiplicity=1, optical_isomers=1, number=None, mass=None, coordinates=None)`

A representation of an individual molecular conformation. The attributes are:

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

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

E0

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

coordinates

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

get_active_modes(*self*, *bool active_j_rotor=False*, *bool active_k_rotor=True*) → list

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

get_center_of_mass(*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.

get_density_of_states(*self*, *ndarray e_list*) → ndarray

Return the density of states $\rho(E) dE$ at the specified energies *e_list* above the ground state.

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

Return the enthalpy in J/mol for the system at the specified temperature *T* in K.

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

Return the entropy in J/mol*K for the system at the specified temperature *T* in K.

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

Return the Gibbs free energy in J/mol for the system at the specified temperature *T* in K.

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

Return the heat capacity in J/mol*K for the system at the specified temperature *T* in K.

get_internal_reduced_moment_of_inertia(*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 Pitzer¹, which is described as $I^{(2,option)}$ by East and Radom². In this procedure, the molecule is divided into two tops: those at either end of the hindered rotor bond. The moment of inertia of each top is evaluated using an axis 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 is an integer corresponding to one of three 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	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

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

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).

get_moment_of_inertia_tensor(*self*) → ndarray

Calculate and return the moment of inertia tensor for the conformer in $\text{kg}\cdot\text{m}^2$. If the coordinates are not at the center of mass, they are temporarily shifted there for the purposes of this calculation.

get_number_degrees_of_freedom(*self*)

Return the number of degrees of freedom in a species object, which should be 3N, and raises an exception if it is not.

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

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

get_principal_moments_of_inertia(*self*)

Calculate and return the principal moments of inertia and corresponding principal axes for the conformer. The moments of inertia are in $\text{kg}\cdot\text{m}^2$, while the principal axes have unit length.

get_sum_of_states(*self*, ndarray *e_list*) → ndarray

Return the sum of states $N(E)$ at the specified energies *e_list* in kJ/mol above the ground state.

get_symmetric_top_rotors(*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.

get_total_mass(*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.

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

mass

An array of masses of each atom in the conformer.

modes

list

Type

modes

number

An array of atomic numbers of each atom in the conformer.

optical_isomers

'int'

Type

optical_isomers

spin_multiplicity

‘int’

Type

spin_multiplicity

1.15 Thermodynamics (rmgpy.thermo)

The *rmgpy.thermo* subpackage contains classes that represent various thermodynamic models of heat capacity.

1.15.1 Heat capacity models

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

rmgpy.thermo.ThermoData

```
class rmgpy.thermo.ThermoData(Tdata=None, Cpdata=None, H298=None, S298=None, Cp0=None,  
                               CpInf=None, Tmin=None, Tmax=None, E0=None, label="", comment="")
```

A heat capacity model based on a set of discrete heat capacity data points. The attributes are:

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

Cp0

The heat capacity at zero temperature.

CpInf

The heat capacity at infinite temperature.

Cpdata

An array of heat capacities at the given temperatures.

E0

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

H298

The standard enthalpy of formation at 298 K.

S298

The standard entropy of formation at 298 K.

Tdata

An array of temperatures at which the heat capacity is known.

Tmax

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

Tmin

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

as_dict()

A helper function for dumping objects as dictionaries for YAML files

Returns

A dictionary representation of the object

Return type

dict

comment

unicode

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

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

Return the enthalpy in J/mol at the specified temperature *T* in K.

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

Return the entropy in J/mol*K at the specified temperature *T* in K.

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

Return the Gibbs free energy in J/mol at the specified temperature *T* in K.

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

Return the constant-pressure heat capacity in J/mol*K at the specified temperature *T* in K.

is_all_zeros(*self*) → bool

Check whether a ThermoData object has all zero values

e.g.:

```
ThermoData(
    Tdata=[300, 400, 500, 600, 800, 1000, 1500], "K"),
    Cpdata=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], "J/(mol*K)"),
    H298=(0.0, "kJ/mol"),
    S298=(0.0, "J/(mol*K)"),
)
```

Returns

Whether all values are zeroes or not.

Return type

bool

is_identical_to(*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.

is_similar_to(*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.

is_temperature_valid(*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

unicode

Type

label

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

to_nasa(*self*, double *Tmin*, double *Tmax*, double *Tint*, bool *fixedTint=False*, bool *weighting=True*, int *continuity=3*) → [NASA](#)

Convert the object to a [NASA](#) object. You must specify the minimum and maximum temperatures of the fit *Tmin* and *Tmax* in K, as well as the intermediate temperature *Tint* in K to use as the bridge between the two fitted polynomials. The remaining parameters can be used to modify the fitting algorithm used:

- *fixedTint* - False to allow *Tint* to vary in order to improve the fit, or True to keep it fixed
- *weighting* - True to weight the fit by T^{-1} to emphasize good fit at lower temperatures, or False to not use weighting
- *continuity* - The number of continuity constraints to enforce at *Tint*:
 - 0: no constraints on continuity of $C_p(T)$ at *Tint*
 - 1: constrain $C_p(T)$ to be continuous at *Tint*
 - 2: constrain $C_p(T)$ and $\frac{dC_p}{dT}$ to be continuous at *Tint*
 - 3: constrain $C_p(T)$, $\frac{dC_p}{dT}$, and $\frac{d^2C_p}{dT^2}$ to be continuous at *Tint*
 - 4: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, and $\frac{d^3C_p}{dT^3}$ to be continuous at *Tint*
 - 5: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, $\frac{d^3C_p}{dT^3}$, and $\frac{d^4C_p}{dT^4}$ to be continuous at *Tint*

Note that values of *continuity* of 5 or higher effectively constrain all the coefficients to be equal and should be equivalent to fitting only one polynomial (rather than two).

Returns the fitted [NASA](#) object containing the two fitted [NASAPolynomial](#) objects.

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

The Wilhoit polynomial is an expression for heat capacity that is guaranteed to give the correct limits at zero and infinite temperature, and gives a very reasonable shape to the heat capacity profile in between:

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

Above, $y \equiv T/(T + B)$ is a scaled temperature that ranges from zero to one based on the value of the coefficient *B*, and a_0 , a_1 , a_2 , and a_3 are the Wilhoit polynomial coefficients.

The enthalpy is given by

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

where $f_{ij} = 3 + j$ if $i = j$, $f_{ij} = 1$ if $i > j$, and $f_{ij} = 0$ if $i < j$.

The entropy is given by

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

The low-temperature limit $C_p(0)$ is $3.5R$ for linear molecules and $4R$ for nonlinear molecules. The high-temperature limit $C_p(\infty)$ is taken to be $[3N_{\text{atoms}} - 1.5]R$ for linear molecules and $[3N_{\text{atoms}} - (2 + 0.5N_{\text{rotors}})]R$ for nonlinear molecules, for a molecule composed of N_{atoms} atoms and N_{rotors} internal rotors.

B

The Wilhoit scaled temperature coefficient.

Cp0

The heat capacity at zero temperature.

CpInf

The heat capacity at infinite temperature.

E0

The ground state energy (J/mol) at zero Kelvin, including zero point energy.

For the Wilhoit class, this is calculated as the Enthalpy at 0.001 Kelvin.

H0

The integration constant for enthalpy.

NB. this is not equal to the enthalpy at 0 Kelvin, which you can access via E0

S0

The integration constant for entropy.

Tmax

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

Tmin

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

a0

'double'

Type

a0

a1

'double'

Type

a1

a2

'double'

Type

a2

a3

'double'

Type

a3

as_dict(*self*) → dict

A helper function for YAML parsing

comment

unicode

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

fit_to_data(*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.

fit_to_data_for_constant_b(*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.

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

Return the enthalpy in J/mol at the specified temperature *T* in K.

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

Return the entropy in J/mol*K at the specified temperature *T* in K.

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

Return the Gibbs free energy in J/mol at the specified temperature *T* in K.

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

Return the constant-pressure heat capacity in J/mol*K at the specified temperature *T* in K.

is_identical_to(*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.

is_similar_to(*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.

is_temperature_valid(*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

unicode

Type

label

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

to_nasa(*self*, *double Tmin*, *double Tmax*, *double Tint*, *bool fixedTint=False*, *bool weighting=True*, *int continuity=3*) → *NASA*

Convert the Wilhoit object to a *NASA* object. You must specify the minimum and maximum temperatures of the fit *Tmin* and *Tmax* in K, as well as the intermediate temperature *Tint* in K to use as the bridge between the two fitted polynomials. The remaining parameters can be used to modify the fitting algorithm used:

- *fixedTint* - *False* to allow *Tint* to vary in order to improve the fit, or *True* to keep it fixed
- *weighting* - *True* to weight the fit by T^{-1} to emphasize good fit at lower temperatures, or *False* to not use weighting
- *continuity* - The number of continuity constraints to enforce at *Tint*:
 - 0: no constraints on continuity of $C_p(T)$ at *Tint*
 - 1: constrain $C_p(T)$ to be continuous at *Tint*
 - 2: constrain $C_p(T)$ and $\frac{dC_p}{dT}$ to be continuous at *Tint*
 - 3: constrain $C_p(T)$, $\frac{dC_p}{dT}$, and $\frac{d^2C_p}{dT^2}$ to be continuous at *Tint*
 - 4: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, and $\frac{d^3C_p}{dT^3}$ to be continuous at *Tint*
 - 5: constrain $C_p(T)$, $\frac{dC_p}{dT}$, $\frac{d^2C_p}{dT^2}$, $\frac{d^3C_p}{dT^3}$, and $\frac{d^4C_p}{dT^4}$ to be continuous at *Tint*

Note that values of *continuity* of 5 or higher effectively constrain all the coefficients to be equal and should be equivalent to fitting only one polynomial (rather than two).

Returns the fitted *NASA* object containing the two fitted *NASAPolynomial* objects.

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

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

expressions

$$\frac{C_p(T)}{R} = a_{-2}T^{-2} + a_{-1}T^{-1} + a_0 + a_1T + a_2T^2 + a_3T^3 + a_4T^4$$

$$\frac{H(T)}{RT} = -a_{-2}T^{-2} + a_{-1}T^{-1} \ln T + a_0 + \frac{1}{2}a_1T + \frac{1}{3}a_2T^2 + \frac{1}{4}a_3T^3 + \frac{1}{5}a_4T^4 + \frac{a_5}{T}$$

$$\frac{S(T)}{R} = -\frac{1}{2}a_{-2}T^{-2} - a_{-1}T^{-1} + a_0 \ln T + a_1T + \frac{1}{2}a_2T^2 + \frac{1}{3}a_3T^3 + \frac{1}{4}a_4T^4 + a_6$$

In the seven-coefficient version, $a_{-2} = a_{-1} = 0$.

As simple polynomial expressions, the NASA polynomial is faster to evaluate when compared to the Wilhoit model; however, it does not have the nice physical behavior of the Wilhoit representation. Often multiple NASA polynomials are used to accurately represent the thermodynamics of a system over a wide temperature range.

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.

as_dict(*self*) → dict

A helper function for YAML dumping

change_base_enthalpy(*self*, *double deltaH*) → [NASA](#)

Add deltaH in J/mol to the base enthalpy of formation H298 and return the modified NASA object.

change_base_entropy(*self*, *double deltaS*) → [NASA](#)

Add deltaS in J/molK to the base entropy of formation S298 and return the modified NASA object

comment

unicode

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

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

Return the enthalpy $H(T)$ in J/mol at the specified temperature T in K.

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

Return the entropy $S(T)$ in J/mol*K at the specified temperature T in K.

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

Return the Gibbs free energy $G(T)$ in J/mol at the specified temperature T in K.

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

Return the constant-pressure heat capacity $C_p(T)$ in J/mol*K at the specified temperature T in K.

is_identical_to(*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.

is_similar_to(*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.

is_temperature_valid(*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

unicode

Type

label

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

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.

select_polynomial(*self*, *double T*) → *NASAPolynomial*

to_cantera(*self*)

Return the cantera equivalent NasaPoly2 object from this NASA object.

to_thermo_data(*self*) → *ThermoData*

Convert the NASAPolynomial model to a *ThermoData* object.

If Cp0 and CpInf are omitted or 0, they are None in the returned ThermoData.

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

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

$$\frac{C_p(T)}{R} = a_{-2}T^{-2} + a_{-1}T^{-1} + a_0 + a_1T + a_2T^2 + a_3T^3 + a_4T^4$$

$$\frac{H(T)}{RT} = -a_{-2}T^{-2} + a_{-1}T^{-1} \ln T + a_0 + \frac{1}{2}a_1T + \frac{1}{3}a_2T^2 + \frac{1}{4}a_3T^3 + \frac{1}{5}a_4T^4 + \frac{a_5}{T}$$

$$\frac{S(T)}{R} = -\frac{1}{2}a_{-2}T^{-2} - a_{-1}T^{-1} + a_0 \ln T + a_1T + \frac{1}{2}a_2T^2 + \frac{1}{3}a_3T^3 + \frac{1}{4}a_4T^4 + a_6$$

In the seven-coefficient version, $a_{-2} = a_{-1} = 0$.

As simple polynomial expressions, the NASA polynomial is faster to evaluate when compared to the Wilhoit model; however, it does not have the nice physical behavior of the Wilhoit representation. Often multiple NASA polynomials are used to accurately represent the thermodynamics of a system over a wide temperature range; the [NASA](#) class is available for this purpose.

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.

as_dict(*self*) → dict

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`**change_base_enthalpy**(*self*, *double deltaH*)

Add deltaH in J/mol to the base enthalpy of formation H298.

change_base_entropy(*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`unicode`

Type

comment

discrepancy(*self*, *HeatCapacityModel other*) → doubleReturn some measure of how dissimilar *self* is from *other*.

The measure is arbitrary, but hopefully useful for sorting purposes. Discrepancy of 0 means they are identical

get_enthalpy(*self*, double *T*) → doubleReturn the enthalpy in J/mol at the specified temperature *T* in K.**get_entropy**(*self*, double *T*) → doubleReturn the entropy in J/mol*K at the specified temperature *T* in K.**get_free_energy**(*self*, double *T*) → doubleReturn the Gibbs free energy in J/mol at the specified temperature *T* in K.**get_heat_capacity**(*self*, double *T*) → doubleReturn the constant-pressure heat capacity in J/mol*K at the specified temperature *T* in K.**is_identical_to**(*self*, *HeatCapacityModel other*) → boolReturns True if *self* and *other* report very similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.**is_similar_to**(*self*, *HeatCapacityModel other*) → boolReturns True if *self* and *other* report similar thermo values for heat capacity, enthalpy, entropy, and free energy over a wide range of temperatures, or False otherwise.**is_temperature_valid**(*self*, double *T*) → boolReturn True if the temperature *T* in K is within the valid temperature range of the thermodynamic data, or False if not. If the minimum and maximum temperature are not defined, True is returned.**label**

unicode

Type

label

make_object(*data*, *class_dict*)

A helper function for constructing objects from a dictionary (used when loading YAML files)

Parameters

- **data** (*dict*) – The dictionary representation of the object
- **class_dict** (*dict*) – A mapping of class names to the classes themselves

Returns

None

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.ChemkinError

An exception class for exceptional behavior involving Chemkin files. Pass a string describing the circumstances that caused the exceptional behavior.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.CoreError

An exception raised if there is a problem within the model core

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.DatabaseError

A exception that occurs when working with an RMG database. Pass a string giving specifics about the exceptional behavior.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.**InchiException**

An exception used when encountering a non-valid Inchi expression are encountered. Pass a string describing the error.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.**InvalidActionError**

An exception to be raised when an invalid action is encountered in a reaction recipe.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception rmgpy.exceptions.NetworkError

Raised when an error occurs while working with a pressure-dependent reaction network

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.ReactionError`

An exception class for exceptional behavior involving `Reaction` objects. Pass a string describing the circumstances that caused the exceptional behavior.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.ReactionPairsError`

An exception to be raised when an error occurs while working with reaction pairs.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.ResonanceError`

An exception class for when RMG is unable to generate resonance structures.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.SettingsError`

An exception raised when dealing with settings.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.StatmechError`

An exception used when an error occurs in estimating Statmech.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.UndeterminableKineticsError`(*reaction*, *message*=")

An exception raised when attempts to estimate appropriate kinetic parameters for a chemical reaction are unsuccessful.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.UnexpectedChargeError`(*graph*)

An exception class when encountering a group/molecule with unexpected charge. Currently in RMG, we never expect to see -2/+2 or greater magnitude charge, we only expect +1/-1 charges on nitrogen, oxygen, sulfur or specifically carbon monoxide/monosulfide.

Attributes: *graph* is the molecule or group object with the unexpected charge

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `rmgpy.exceptions.VF2Error`

An exception raised if an error occurs within the VF2 graph isomorphism algorithm. Pass a string describing the error.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

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

PYTHON MODULE INDEX

a

arkane, 3
arkane.common, 20
arkane.output, 14
arkane.sensitivity, 19

r

rmgpy.chemkin, 23
rmgpy.constants, 26
rmgpy.data, 27
rmgpy.exceptions, 287
rmgpy.kinetics, 95
rmgpy.molecule, 125
rmgpy.molecule.adjlist, 175
rmgpy.molecule.converter, 173
rmgpy.molecule.filtration, 168
rmgpy.molecule.kekulize, 167
rmgpy.molecule.pathfinder, 170
rmgpy.molecule.resonance, 164
rmgpy.molecule.translator, 173
rmgpy.pdep, 179
rmgpy.qm, 189
rmgpy.quantity, 210
rmgpy.reaction, 215
rmgpy.rmg, 221
rmgpy.solver, 238
rmgpy.species, 250
rmgpy.statmech, 255
rmgpy.statmech.schrodinger, 270
rmgpy.thermo, 274

A

- A* (*rmgpy.kinetics.Arrhenius* attribute), 97
- A* (*rmgpy.kinetics.StickingCoefficient* attribute), 123
- A* (*rmgpy.kinetics.SurfaceArrhenius* attribute), 120
- a0* (*rmgpy.thermo.Wilhoit* attribute), 278
- a1* (*rmgpy.thermo.Wilhoit* attribute), 278
- a2* (*rmgpy.thermo.Wilhoit* attribute), 278
- a3* (*rmgpy.thermo.Wilhoit* attribute), 278
- ActionError*, 287
- add_action()* (*rmgpy.data.kinetics.ReactionRecipe* method), 65
- add_allyls()* (in module *rmgpy.molecule.pathfinder*), 170
- add_atom()* (*rmgpy.molecule.Group* method), 156
- add_atom()* (*rmgpy.molecule.Molecule* method), 142
- add_atom_labels_for_reaction()* (*rmgpy.data.kinetics.KineticsFamily* method), 42
- add_bond()* (*rmgpy.molecule.Group* method), 157
- add_bond()* (*rmgpy.molecule.Molecule* method), 142
- add_edge()* (*rmgpy.molecule.graph.Graph* method), 128
- add_edge()* (*rmgpy.molecule.Group* method), 157
- add_edge()* (*rmgpy.molecule.Molecule* method), 142
- add_entry()* (*rmgpy.data.kinetics.KineticsFamily* method), 42
- add_explicit_ligands()* (*rmgpy.molecule.Group* method), 157
- add_implicit_atoms_from_atomtype()* (*rmgpy.molecule.Group* method), 157
- add_implicit_benzene()* (*rmgpy.molecule.Group* method), 157
- add_inverse_allyls()* (in module *rmgpy.molecule.pathfinder*), 170
- add_new_surface_objects()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_path_reaction()* (*rmgpy.rmg.pdep.PDepNetwork* method), 230
- add_reaction_library_to_edge()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_reaction_library_to_output()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_reaction_to_core()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_reaction_to_edge()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_reaction_to_unimolecular_networks()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_reactions_to_surface()* (*rmgpy.solver.LiquidReactor* method), 243
- add_reactions_to_surface()* (*rmgpy.solver.MBSampledReactor* method), 248
- add_reactions_to_surface()* (*rmgpy.solver.ReactionSystem* method), 239
- add_reactions_to_surface()* (*rmgpy.solver.SimpleReactor* method), 241
- add_reactions_to_surface()* (*rmgpy.solver.SurfaceReactor* method), 245
- add_reverse_attribute()* (*rmgpy.data.kinetics.KineticsFamily* method), 42
- add_rules_from_training()* (*rmgpy.data.kinetics.KineticsFamily* method), 42
- add_seed_mechanism_to_core()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_species_to_core()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_species_to_edge()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 223
- add_unsaturated_bonds()* (in module *rmgpy.molecule.pathfinder*), 170
- add_vertex()* (*rmgpy.molecule.graph.Graph* method),

- 128
- `add_vertex()` (*rmgpy.molecule.Group* method), 157
- `add_vertex()` (*rmgpy.molecule.Molecule* method), 142
- `adjust_surface()` (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
- `advance()` (*rmgpy.solver.LiquidReactor* method), 243
- `advance()` (*rmgpy.solver.MBSampledReactor* method), 248
- `advance()` (*rmgpy.solver.ReactionSystem* method), 239
- `advance()` (*rmgpy.solver.SimpleReactor* method), 241
- `advance()` (*rmgpy.solver.SurfaceReactor* method), 245
- `alpha` (*rmgpy.kinetics.Troe* attribute), 116
- `alpha0` (*rmgpy.pdep.SingleExponentialDown* attribute), 181
- `analyze_molecule()` (in module *rmgpy.molecule.resonance*), 164
- `ancestors()` (*rmgpy.data.base.Database* method), 28
- `ancestors()` (*rmgpy.data.kinetics.KineticsDepository* method), 39
- `ancestors()` (*rmgpy.data.kinetics.KineticsFamily* method), 42
- `ancestors()` (*rmgpy.data.kinetics.KineticsGroups* method), 50
- `ancestors()` (*rmgpy.data.kinetics.KineticsLibrary* method), 53
- `ancestors()` (*rmgpy.data.kinetics.KineticsRules* method), 55
- `ancestors()` (*rmgpy.data.statmech.StatmechDepository* method), 66
- `ancestors()` (*rmgpy.data.statmech.StatmechGroups* method), 73
- `ancestors()` (*rmgpy.data.statmech.StatmechLibrary* method), 76
- `ancestors()` (*rmgpy.data.thermo.ThermoDepository* method), 87
- `ancestors()` (*rmgpy.data.thermo.ThermoGroups* method), 89
- `ancestors()` (*rmgpy.data.thermo.ThermoLibrary* method), 92
- `apply_action()` (*rmgpy.molecule.Atom* method), 137
- `apply_action()` (*rmgpy.molecule.Bond* method), 139
- `apply_action()` (*rmgpy.molecule.GroupAtom* method), 153
- `apply_action()` (*rmgpy.molecule.GroupBond* method), 155
- `apply_chemically_significant_eigenvalues_method()` (in module *rmgpy.pdep.cse*), 188
- `apply_chemically_significant_eigenvalues_method()` (in module *rmgpy.pdep.Network* method), 185
- `apply_chemically_significant_eigenvalues_method()` (in module *rmgpy.rmg.pdep.PDepNetwork* method), 231
- `apply_forward()` (*rmgpy.data.kinetics.ReactionRecipe* method), 65
- `apply_inverse_laplace_transform_method()` (in module *rmgpy.pdep.reaction*), 183
- `apply_kinetics_to_reaction()` (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
- `apply_modified_strong_collision_method()` (in module *rmgpy.pdep.msc*), 187
- `apply_modified_strong_collision_method()` (*rmgpy.pdep.Network* method), 185
- `apply_modified_strong_collision_method()` (*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `apply_recipe()` (*rmgpy.data.kinetics.KineticsFamily* method), 42
- `apply_reservoir_state_method()` (in module *rmgpy.pdep.rs*), 188
- `apply_reservoir_state_method()` (*rmgpy.pdep.Network* method), 185
- `apply_reservoir_state_method()` (*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `apply_reverse()` (*rmgpy.data.kinetics.ReactionRecipe* method), 65
- `apply_rrkm_theory()` (in module *rmgpy.pdep.reaction*), 182
- `apply_solvent_correction()` (*rmgpy.data.kinetics.DepositoryReaction* method), 30
- `apply_solvent_correction()` (*rmgpy.data.kinetics.LibraryReaction* method), 58
- `apply_solvent_correction()` (*rmgpy.data.kinetics.TemplateReaction* method), 78
- `apply_solvent_correction()` (*rmgpy.reaction.Reaction* method), 216
- `apply_solvent_correction()` (*rmgpy.rmg.pdep.PDepReaction* method), 233
- `apply_thermo_to_species()` (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
- `are_siblings()` (*rmgpy.data.base.Database* method), 28
- `are_siblings()` (*rmgpy.data.kinetics.KineticsDepository* method), 39
- `are_siblings()` (*rmgpy.data.kinetics.KineticsFamily* method), 43
- `are_siblings()` (*rmgpy.data.kinetics.KineticsGroups* method), 50
- `are_siblings()` (*rmgpy.data.kinetics.KineticsLibrary* method), 53
- `are_siblings()` (*rmgpy.data.kinetics.KineticsRules* method), 55
- `are_siblings()` (*rmgpy.data.statmech.StatmechDepository* method), 66

- [are_siblings\(\)](#) (*rmgpy.data.statmech.StatmechGroups* method), 73
[are_siblings\(\)](#) (*rmgpy.data.statmech.StatmechLibrary* method), 76
[are_siblings\(\)](#) (*rmgpy.data.thermo.ThermoDepository* method), 87
[are_siblings\(\)](#) (*rmgpy.data.thermo.ThermoGroups* method), 89
[are_siblings\(\)](#) (*rmgpy.data.thermo.ThermoLibrary* method), 92
[arkane](#)
 module, 3
[Arkane](#) (class in *arkane*), 14
[arkane.common](#)
 module, 20
[arkane.output](#)
 module, 14
[arkane.sensitivity](#)
 module, 19
[ArkaneSpecies](#) (class in *arkane.common*), 20
[AromaticBond](#) (class in *rmgpy.molecule.kekulize*), 167
[aromaticity_filtration\(\)](#) (in module *rmgpy.molecule.filtration*), 168
[AromaticRing](#) (class in *rmgpy.molecule.kekulize*), 167
[ArrayQuantity](#) (class in *rmgpy.quantity*), 213
[Arrhenius](#) (class in *rmgpy.kinetics*), 97
[arrhenius](#) (*rmgpy.kinetics.MultiArrhenius* attribute), 100
[arrhenius](#) (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 105
[arrhenius](#) (*rmgpy.kinetics.PDepArrhenius* attribute), 103
[arrheniusHigh](#) (*rmgpy.kinetics.Lindemann* attribute), 113
[arrheniusHigh](#) (*rmgpy.kinetics.Troe* attribute), 116
[arrheniusLow](#) (*rmgpy.kinetics.Lindemann* attribute), 113
[arrheniusLow](#) (*rmgpy.kinetics.ThirdBody* attribute), 110
[arrheniusLow](#) (*rmgpy.kinetics.Troe* attribute), 116
[as_dict\(\)](#) (*arkane.common.ArkaneSpecies* method), 21
[as_dict\(\)](#) (*rmgpy.pdep.SingleExponentialDown* method), 181
[as_dict\(\)](#) (*rmgpy.quantity.ArrayQuantity* method), 213
[as_dict\(\)](#) (*rmgpy.quantity.ScalarQuantity* method), 212
[as_dict\(\)](#) (*rmgpy.statmech.Conformer* method), 271
[as_dict\(\)](#) (*rmgpy.statmech.HarmonicOscillator* method), 266
[as_dict\(\)](#) (*rmgpy.statmech.HinderedRotor* method), 268
[as_dict\(\)](#) (*rmgpy.statmech.IdealGasTranslation* method), 258
[as_dict\(\)](#) (*rmgpy.statmech.KRotor* method), 262
[as_dict\(\)](#) (*rmgpy.statmech.LinearRotor* method), 259
[as_dict\(\)](#) (*rmgpy.statmech.NonlinearRotor* method), 261
[as_dict\(\)](#) (*rmgpy.statmech.SphericalTopRotor* method), 264
[as_dict\(\)](#) (*rmgpy.thermo.NASA* method), 281
[as_dict\(\)](#) (*rmgpy.thermo.NASAPolynomial* method), 284
[as_dict\(\)](#) (*rmgpy.thermo.ThermoData* method), 275
[as_dict\(\)](#) (*rmgpy.thermo.Wilhoit* method), 278
[assign_atom_ids\(\)](#) (*rmgpy.molecule.Molecule* method), 142
[Atom](#) (class in *rmgpy.molecule*), 137
[atom_ids_valid\(\)](#) (*rmgpy.molecule.Molecule* method), 142
[atoms](#) (*rmgpy.molecule.Group* attribute), 157
[atoms](#) (*rmgpy.molecule.Molecule* attribute), 142
[AtomType](#) (class in *rmgpy.molecule*), 132
[AtomTypeError](#), 287
- ## B
- [B](#) (*rmgpy.thermo.Wilhoit* attribute), 277
[badness\(\)](#) (*rmgpy.exceptions.InvalidMicrocanonicalRateError* method), 288
[barrier](#) (*rmgpy.statmech.HinderedRotor* attribute), 268
[Bond](#) (class in *rmgpy.molecule*), 139
- ## C
- [c0](#) (*rmgpy.thermo.NASAPolynomial* attribute), 284
[c1](#) (*rmgpy.thermo.NASAPolynomial* attribute), 284
[c2](#) (*rmgpy.thermo.NASAPolynomial* attribute), 285
[c3](#) (*rmgpy.thermo.NASAPolynomial* attribute), 285
[c4](#) (*rmgpy.thermo.NASAPolynomial* attribute), 285
[c5](#) (*rmgpy.thermo.NASAPolynomial* attribute), 285
[c6](#) (*rmgpy.thermo.NASAPolynomial* attribute), 285
[calculate\(\)](#) (*rmgpy.qm.symmetry.SymmetryJob* method), 194
[calculate_atom_symmetry_number\(\)](#) (in module *rmgpy.molecule.symmetry*), 177
[calculate_axis_symmetry_number\(\)](#) (in module *rmgpy.molecule.symmetry*), 177
[calculate_bond_symmetry_number\(\)](#) (in module *rmgpy.molecule.symmetry*), 177
[calculate_chirality_correction\(\)](#) (*rmgpy.qm.gaussian.GaussianMol* method), 195
[calculate_chirality_correction\(\)](#) (*rmgpy.qm.gaussian.GaussianMolPM3* method), 197
[calculate_chirality_correction\(\)](#) (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
[calculate_chirality_correction\(\)](#) (*rmgpy.qm.molecule.QMMolecule* method),

- 192
- `calculate_chirality_correction()`
(*rmgpy.qm.mopac.MopacMol* method), 202
- `calculate_chirality_correction()`
(*rmgpy.qm.mopac.MopacMolPM3* method), 204
- `calculate_chirality_correction()`
(*rmgpy.qm.mopac.MopacMolPM6* method), 206
- `calculate_chirality_correction()`
(*rmgpy.qm.mopac.MopacMolPM7* method), 208
- `calculate_coll_limit()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 30
- `calculate_coll_limit()`
(*rmgpy.data.kinetics.LibraryReaction* method), 58
- `calculate_coll_limit()`
(*rmgpy.data.kinetics.TemplateReaction* method), 78
- `calculate_coll_limit()` (*rmgpy.reaction.Reaction* method), 216
- `calculate_coll_limit()`
(*rmgpy.rmg.pdep.PDepReaction* method), 233
- `calculate_collision_efficiency()`
(*rmgpy.pdep.SingleExponentialDown* method), 181
- `calculate_collision_frequency()`
(*rmgpy.pdep.Configuration* method), 183
- `calculate_collision_model()`
(*rmgpy.pdep.Network* method), 185
- `calculate_collision_model()`
(*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `calculate_cp0()` (*rmgpy.molecule.Molecule* method), 143
- `calculate_cp0()` (*rmgpy.species.Species* method), 251
- `calculate_cpinf()` (*rmgpy.molecule.Molecule* method), 143
- `calculate_cpinf()` (*rmgpy.species.Species* method), 251
- `calculate_cyclic_symmetry_number()` (in module *rmgpy.molecule.symmetry*), 178
- `calculate_degeneracy()`
(*rmgpy.data.kinetics.KineticsFamily* method), 43
- `calculate_densities_of_states()`
(*rmgpy.pdep.Network* method), 185
- `calculate_densities_of_states()`
(*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `calculate_density_of_states()`
(*rmgpy.pdep.Configuration* method), 183
- `calculate_effective_pressure()`
(*rmgpy.solver.MBSampledReactor* method), 248
- `calculate_effective_pressure()`
(*rmgpy.solver.SimpleReactor* method), 241
- `calculate_equilibrium_ratios()`
(*rmgpy.pdep.Network* method), 186
- `calculate_equilibrium_ratios()`
(*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `calculate_microcanonical_rate_coefficient()`
(in module *rmgpy.pdep.reaction*), 182
- `calculate_microcanonical_rate_coefficient()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 31
- `calculate_microcanonical_rate_coefficient()`
(*rmgpy.data.kinetics.LibraryReaction* method), 58
- `calculate_microcanonical_rate_coefficient()`
(*rmgpy.data.kinetics.TemplateReaction* method), 79
- `calculate_microcanonical_rate_coefficient()`
(*rmgpy.reaction.Reaction* method), 216
- `calculate_microcanonical_rate_coefficient()`
(*rmgpy.rmg.pdep.PDepReaction* method), 233
- `calculate_microcanonical_rates()`
(*rmgpy.pdep.Network* method), 186
- `calculate_microcanonical_rates()`
(*rmgpy.rmg.pdep.PDepNetwork* method), 231
- `calculate_symmetry_number()` (in module *rmgpy.molecule.symmetry*), 178
- `calculate_symmetry_number()`
(*rmgpy.molecule.Molecule* method), 143
- `calculate_thermo_data()`
(*rmgpy.qm.gaussian.GaussianMol* method), 195
- `calculate_thermo_data()`
(*rmgpy.qm.gaussian.GaussianMolPM3* method), 197
- `calculate_thermo_data()`
(*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
- `calculate_thermo_data()`
(*rmgpy.qm.molecule.QMMolecule* method), 192
- `calculate_thermo_data()`
(*rmgpy.qm.mopac.MopacMol* method), 202
- `calculate_thermo_data()`
(*rmgpy.qm.mopac.MopacMolPM3* method), 204
- `calculate_thermo_data()`
(*rmgpy.qm.mopac.MopacMolPM6* method),

- 206
`calculate_thermo_data()`
 (*rmgpy.qm.mopac.MopacMolPM7* method), 208
`calculate_tst_rate_coefficient()`
 (*rmgpy.data.kinetics.DepositoryReaction* method), 31
`calculate_tst_rate_coefficient()`
 (*rmgpy.data.kinetics.LibraryReaction* method), 59
`calculate_tst_rate_coefficient()`
 (*rmgpy.data.kinetics.TemplateReaction* method), 79
`calculate_tst_rate_coefficient()`
 (*rmgpy.reaction.Reaction* method), 217
`calculate_tst_rate_coefficient()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 234
`calculate_tunneling_factor()`
 (*rmgpy.kinetics.Eckart* method), 119
`calculate_tunneling_factor()`
 (*rmgpy.kinetics.Wigner* method), 118
`calculate_tunneling_factor()`
 (*rmgpy.species.TransitionState* method), 255
`calculate_tunneling_function()`
 (*rmgpy.kinetics.Eckart* method), 119
`calculate_tunneling_function()`
 (*rmgpy.kinetics.Wigner* method), 118
`calculate_tunneling_function()`
 (*rmgpy.species.TransitionState* method), 255
`can_tst()` (*rmgpy.data.kinetics.DepositoryReaction* method), 31
`can_tst()` (*rmgpy.data.kinetics.LibraryReaction* method), 59
`can_tst()` (*rmgpy.data.kinetics.TemplateReaction* method), 79
`can_tst()` (*rmgpy.reaction.Reaction* method), 217
`can_tst()` (*rmgpy.rmg.pdep.PDepReaction* method), 234
`change_base_enthalpy()` (*rmgpy.thermo.NASA* method), 281
`change_base_enthalpy()`
 (*rmgpy.thermo.NASAPolynomial* method), 285
`change_base_entropy()` (*rmgpy.thermo.NASA* method), 281
`change_base_entropy()`
 (*rmgpy.thermo.NASAPolynomial* method), 285
`change_rate()` (*rmgpy.kinetics.Arrhenius* method), 98
`change_rate()` (*rmgpy.kinetics.Chebyshev* method), 108
`change_rate()` (*rmgpy.kinetics.Lindemann* method), 113
`change_rate()` (*rmgpy.kinetics.MultiArrhenius* method), 100
`change_rate()` (*rmgpy.kinetics.MultiPDepArrhenius* method), 106
`change_rate()` (*rmgpy.kinetics.PDepArrhenius* method), 103
`change_rate()` (*rmgpy.kinetics.StickingCoefficient* method), 124
`change_rate()` (*rmgpy.kinetics.SurfaceArrhenius* method), 121
`change_rate()` (*rmgpy.kinetics.ThirdBody* method), 110
`change_rate()` (*rmgpy.kinetics.Troe* method), 116
`change_t0()` (*rmgpy.kinetics.Arrhenius* method), 98
`change_t0()` (*rmgpy.kinetics.StickingCoefficient* method), 124
`change_t0()` (*rmgpy.kinetics.SurfaceArrhenius* method), 121
`charge_filtration()` (in module *rmgpy.molecule.filtration*), 168
`Chebyshev` (class in *rmgpy.kinetics*), 107
`chebyshev()` (*rmgpy.kinetics.Chebyshev* method), 108
`check_all_set()` (*rmgpy.qm.main.QMSettings* method), 190
`check_collision_limit_violation()`
 (*rmgpy.data.kinetics.DepositoryReaction* method), 31
`check_collision_limit_violation()`
 (*rmgpy.data.kinetics.LibraryReaction* method), 59
`check_collision_limit_violation()`
 (*rmgpy.data.kinetics.TemplateReaction* method), 79
`check_collision_limit_violation()`
 (*rmgpy.reaction.Reaction* method), 217
`check_collision_limit_violation()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 234
`check_conformer_energy()` (in module *arkane.common*), 21
`check_for_duplicates()`
 (*rmgpy.data.kinetics.KineticsLibrary* method), 53
`check_for_errors()` (*arkane.ess.ESSAdapter* method), 5
`check_for_errors()` (*arkane.ess.GaussianLog* method), 7
`check_for_errors()` (*arkane.ess.MolproLog* method), 8
`check_for_errors()` (*arkane.ess.OrcaLog* method), 9
`check_for_errors()` (*arkane.ess.QChemLog* method), 10

`check_for_errors()` (*arkane.ess.TeraChemLog method*), 11
`check_for_existing_reaction()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 224
`check_for_existing_species()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 224
`check_for_inchi_key_collision()` (*rmgpy.qm.qmverifier.QMVerifier method*), 194
`check_input()` (*rmgpy.rmg.main.RMG method*), 228
`check_libraries()` (*rmgpy.rmg.main.RMG method*), 228
`check_model()` (*rmgpy.rmg.main.RMG method*), 228
`check_paths()` (*rmgpy.qm.gaussian.GaussianMol method*), 195
`check_paths()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 197
`check_paths()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 200
`check_paths()` (*rmgpy.qm.main.QMCalculator method*), 190
`check_paths()` (*rmgpy.qm.molecule.QMMolecule method*), 192
`check_paths()` (*rmgpy.qm.mopac.MopacMol method*), 202
`check_paths()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 204
`check_paths()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 206
`check_paths()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 208
`check_reactive()` (*in module rmgpy.molecule.filtration*), 169
`check_ready()` (*rmgpy.qm.gaussian.GaussianMol method*), 196
`check_ready()` (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 197
`check_ready()` (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 200
`check_ready()` (*rmgpy.qm.main.QMCalculator method*), 190
`check_ready()` (*rmgpy.qm.molecule.QMMolecule method*), 192
`check_ready()` (*rmgpy.qm.mopac.MopacMol method*), 202
`check_ready()` (*rmgpy.qm.mopac.MopacMolPM3 method*), 204
`check_ready()` (*rmgpy.qm.mopac.MopacMolPM6 method*), 206
`check_ready()` (*rmgpy.qm.mopac.MopacMolPM7 method*), 208
`ChemicallySignificantEigenvaluesError`, 287
`ChemkinError`, 287
`classify_benzene_carbons()` (*rmgpy.molecule.Group method*), 157
`clean_dir()` (*in module arkane.common*), 21
`clean_tree_groups()` (*rmgpy.data.kinetics.KineticsFamily method*), 43
`cleanup()` (*rmgpy.pdep.Configuration method*), 183
`cleanup()` (*rmgpy.rmg.pdep.PDepNetwork method*), 231
`clear()` (*rmgpy.rmg.main.RMG method*), 228
`clear_labeled_atoms()` (*rmgpy.molecule.Group method*), 157
`clear_labeled_atoms()` (*rmgpy.molecule.Molecule method*), 143
`clear_reg_dims()` (*rmgpy.molecule.Group method*), 157
`clear_surface_adjustments()` (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 224
`cm1` (*rmgpy.thermo.NASAPolynomial attribute*), 285
`cm2` (*rmgpy.thermo.NASAPolynomial attribute*), 285
`coeffs` (*rmgpy.kinetics.Chebyshev attribute*), 108
`coeffs` (*rmgpy.thermo.NASAPolynomial attribute*), 285
`CollisionError`, 287
`comment` (*rmgpy.kinetics.Arrhenius attribute*), 98
`comment` (*rmgpy.kinetics.Chebyshev attribute*), 108
`comment` (*rmgpy.kinetics.KineticsData attribute*), 96
`comment` (*rmgpy.kinetics.Lindemann attribute*), 113
`comment` (*rmgpy.kinetics.MultiArrhenius attribute*), 100
`comment` (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 106
`comment` (*rmgpy.kinetics.PDepArrhenius attribute*), 103
`comment` (*rmgpy.kinetics.PDepKineticsData attribute*), 101
`comment` (*rmgpy.kinetics.StickingCoefficient attribute*), 124
`comment` (*rmgpy.kinetics.SurfaceArrhenius attribute*), 121
`comment` (*rmgpy.kinetics.ThirdBody attribute*), 111
`comment` (*rmgpy.kinetics.Troe attribute*), 116
`comment` (*rmgpy.thermo.NASA attribute*), 281
`comment` (*rmgpy.thermo.NASAPolynomial attribute*), 285
`comment` (*rmgpy.thermo.ThermoData attribute*), 275
`comment` (*rmgpy.thermo.Wilhoit attribute*), 278
`compute_atom_distance()` (*in module rmgpy.molecule.pathfinder*), 170
`compute_group_additivity_thermo()` (*rmgpy.data.thermo.ThermoDatabase method*), 84
`compute_network_variables()` (*rmgpy.solver.LiquidReactor method*), 243
`compute_network_variables()` (*rmgpy.solver.MBSampledReactor method*),

- 248
- `compute_network_variables()`
(*rmgpy.solver.ReactionSystem* method), 239
- `compute_network_variables()`
(*rmgpy.solver.SimpleReactor* method), 241
- `compute_network_variables()`
(*rmgpy.solver.SurfaceReactor* method), 245
- `compute_rate_derivative()`
(*rmgpy.solver.LiquidReactor* method), 243
- `compute_rate_derivative()`
(*rmgpy.solver.MBSampledReactor* method), 248
- `compute_rate_derivative()`
(*rmgpy.solver.ReactionSystem* method), 239
- `compute_rate_derivative()`
(*rmgpy.solver.SimpleReactor* method), 241
- `compute_rate_derivative()`
(*rmgpy.solver.SurfaceReactor* method), 246
- `Configuration` (class in *rmgpy.pdep*), 183
- `Conformer` (class in *rmgpy.statmech*), 271
- `connect_the_dots()` (*rmgpy.molecule.Molecule* method), 143
- `contains_labeled_atom()` (*rmgpy.molecule.Group* method), 157
- `contains_labeled_atom()`
(*rmgpy.molecule.Molecule* method), 143
- `contains_surface_site()` (*rmgpy.molecule.Group* method), 158
- `contains_surface_site()`
(*rmgpy.molecule.Molecule* method), 143
- `contains_surface_site()` (*rmgpy.species.Species* method), 251
- `convert_duplicates_to_multi()`
(*rmgpy.data.kinetics.KineticsLibrary* method), 53
- `convert_imaginary_freq_to_negative_float()`
(in module *arkane.common*), 22
- `convert_initial_keys_to_species_objects()`
(*rmgpy.solver.LiquidReactor* method), 244
- `convert_initial_keys_to_species_objects()`
(*rmgpy.solver.MBSampledReactor* method), 248
- `convert_initial_keys_to_species_objects()`
(*rmgpy.solver.SimpleReactor* method), 241
- `convert_initial_keys_to_species_objects()`
(*rmgpy.solver.SurfaceReactor* method), 246
- `convolve()` (in module *rmgpy.statmech.schrodinger*), 270
- `convolve_bs()` (in module *rmgpy.statmech.schrodinger*), 270
- `convolve_bssr()` (in module *rmgpy.statmech.schrodinger*), 270
- `coordinates` (*rmgpy.statmech.Conformer* attribute), 272
- `copy()` (*arkane.ExplorerJob* method), 19
- `copy()` (*arkane.PressureDependenceJob* method), 17
- `copy()` (*rmgpy.data.kinetics.DepositoryReaction* method), 31
- `copy()` (*rmgpy.data.kinetics.LibraryReaction* method), 59
- `copy()` (*rmgpy.data.kinetics.TemplateReaction* method), 79
- `copy()` (*rmgpy.molecule.Atom* method), 137
- `copy()` (*rmgpy.molecule.Bond* method), 139
- `copy()` (*rmgpy.molecule.graph.Edge* method), 127
- `copy()` (*rmgpy.molecule.graph.Graph* method), 128
- `copy()` (*rmgpy.molecule.graph.Vertex* method), 127
- `copy()` (*rmgpy.molecule.Group* method), 158
- `copy()` (*rmgpy.molecule.GroupAtom* method), 153
- `copy()` (*rmgpy.molecule.GroupBond* method), 155
- `copy()` (*rmgpy.molecule.Molecule* method), 143
- `copy()` (*rmgpy.quantity.ArrayQuantity* method), 213
- `copy()` (*rmgpy.quantity.ScalarQuantity* method), 212
- `copy()` (*rmgpy.reaction.Reaction* method), 217
- `copy()` (*rmgpy.rmg.pdep.PDepReaction* method), 234
- `copy()` (*rmgpy.species.Species* method), 252
- `copy()` (*rmgpy.thermo.Wilhoit* method), 278
- `copy_and_map()` (*rmgpy.molecule.graph.Graph* method), 128
- `copy_and_map()` (*rmgpy.molecule.Group* method), 158
- `copy_and_map()` (*rmgpy.molecule.Molecule* method), 143
- `copy_data()` (*rmgpy.data.thermo.ThermoGroups* method), 90
- `CoreEdgeReactionModel` (class in *rmgpy.rmg.model*), 222
- `CoreError`, 287
- `correct_binding_energy()`
(*rmgpy.data.thermo.ThermoDatabase* method), 84
- `count_aromatic_rings()` (*rmgpy.molecule.Molecule* method), 143
- `count_bonds()` (*rmgpy.molecule.GroupAtom* method), 153
- `count_internal_rotors()`
(*rmgpy.molecule.Molecule* method), 143
- `coverage_dependence`
(*rmgpy.kinetics.StickingCoefficient* attribute), 124
- `coverage_dependence`
(*rmgpy.kinetics.SurfaceArrhenius* attribute), 121
- `Cp0` (*rmgpy.thermo.NASA* attribute), 281
- `Cp0` (*rmgpy.thermo.NASAPolynomial* attribute), 284
- `Cp0` (*rmgpy.thermo.ThermoData* attribute), 274
- `Cp0` (*rmgpy.thermo.Wilhoit* attribute), 278
- `Cpdata` (*rmgpy.thermo.ThermoData* attribute), 274
- `CpInf` (*rmgpy.thermo.NASA* attribute), 281

- CpInf (*rmgpy.thermo.NASAPolynomial* attribute), 284
 CpInf (*rmgpy.thermo.ThermoData* attribute), 274
 CpInf (*rmgpy.thermo.Wilhoit* attribute), 278
 create_and_connect_atom() (*rmgpy.molecule.Group* method), 158
 create_geometry() (*rmgpy.qm.gaussian.GaussianMol* method), 196
 create_geometry() (*rmgpy.qm.gaussian.GaussianMolPM6* method), 197
 create_geometry() (*rmgpy.qm.gaussian.GaussianMolPM7* method), 200
 create_geometry() (*rmgpy.qm.molecule.QMMolecule* method), 192
 create_geometry() (*rmgpy.qm.mopac.MopacMol* method), 203
 create_geometry() (*rmgpy.qm.mopac.MopacMolPM3* method), 205
 create_geometry() (*rmgpy.qm.mopac.MopacMolPM6* method), 207
 create_geometry() (*rmgpy.qm.mopac.MopacMolPM7* method), 209
 create_hindered_rotor_figure() (*arkane.StatMechJob* method), 18
 cross_validate() (*rmgpy.data.kinetics.KineticsFamily* method), 43
 cross_validate_old() (*rmgpy.data.kinetics.KineticsFamily* method), 43
- ## D
- Database (*class in rmgpy.data.base*), 28
 DatabaseError, 287
 debug_rdkit_mol() (*in module rmgpy.molecule.converter*), 173
 decrement_charge() (*rmgpy.molecule.Atom* method), 137
 decrement_lone_pairs() (*rmgpy.molecule.Atom* method), 137
 decrement_order() (*rmgpy.molecule.Bond* method), 139
 decrement_radical() (*rmgpy.molecule.Atom* method), 137
 degeneracy (*rmgpy.data.kinetics.DepositoryReaction* attribute), 31
 degeneracy (*rmgpy.data.kinetics.LibraryReaction* attribute), 59
 degeneracy (*rmgpy.data.kinetics.TemplateReaction* attribute), 79
 degeneracy (*rmgpy.reaction.Reaction* attribute), 217
 degeneracy (*rmgpy.rmg.pdep.PDepReaction* attribute), 234
 degreeP (*rmgpy.kinetics.Chebyshev* attribute), 108
 degreeT (*rmgpy.kinetics.Chebyshev* attribute), 108
 delete_hydrogens() (*rmgpy.molecule.Molecule* method), 143
 DependencyError, 287
 DepositoryReaction (*class in rmgpy.data.kinetics*), 30
 descend_tree() (*rmgpy.data.base.Database* method), 28
 descend_tree() (*rmgpy.data.kinetics.KineticsDepository* method), 39
 descend_tree() (*rmgpy.data.kinetics.KineticsFamily* method), 43
 descend_tree() (*rmgpy.data.kinetics.KineticsGroups* method), 50
 descend_tree() (*rmgpy.data.kinetics.KineticsLibrary* method), 53
 descend_tree() (*rmgpy.data.kinetics.KineticsRules* method), 56
 descend_tree() (*rmgpy.data.statmech.StatmechDepository* method), 66
 descend_tree() (*rmgpy.data.statmech.StatmechGroups* method), 73
 descend_tree() (*rmgpy.data.statmech.StatmechLibrary* method), 76
 descend_tree() (*rmgpy.data.thermo.ThermoDepository* method), 87
 descend_tree() (*rmgpy.data.thermo.ThermoGroups* method), 90
 descend_tree() (*rmgpy.data.thermo.ThermoLibrary* method), 92
 descendants() (*rmgpy.data.base.Database* method), 28
 descendants() (*rmgpy.data.kinetics.KineticsDepository* method), 39
 descendants() (*rmgpy.data.kinetics.KineticsFamily* method), 43
 descendants() (*rmgpy.data.kinetics.KineticsGroups* method), 51
 descendants() (*rmgpy.data.kinetics.KineticsLibrary* method), 53
 descendants() (*rmgpy.data.kinetics.KineticsRules* method), 56
 descendants() (*rmgpy.data.statmech.StatmechDepository* method), 67
 descendants() (*rmgpy.data.statmech.StatmechGroups* method), 73
 descendants() (*rmgpy.data.statmech.StatmechLibrary* method), 76
 descendants() (*rmgpy.data.thermo.ThermoDepository* method), 88
 descendants() (*rmgpy.data.thermo.ThermoGroups* method), 90
 descendants() (*rmgpy.data.thermo.ThermoLibrary* method), 92
 determine_point_group()

- (*rmgpy.qm.gaussian.GaussianMol* method), 196
determine_point_group() (*rmgpy.qm.gaussian.GaussianMolPM3* method), 197
determine_point_group() (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
determine_point_group() (*rmgpy.qm.molecule.QMMolecule* method), 192
determine_point_group() (*rmgpy.qm.mopac.MopacMol* method), 203
determine_point_group() (*rmgpy.qm.mopac.MopacMolPM3* method), 205
determine_point_group() (*rmgpy.qm.mopac.MopacMolPM6* method), 207
determine_point_group() (*rmgpy.qm.mopac.MopacMolPM7* method), 209
DirectFit (class in *rmgpy.data.statmechfit*), 70
discrepancy() (*rmgpy.kinetics.Arrhenius* method), 98
discrepancy() (*rmgpy.kinetics.Chebyshev* method), 108
discrepancy() (*rmgpy.kinetics.KineticsData* method), 96
discrepancy() (*rmgpy.kinetics.Lindemann* method), 113
discrepancy() (*rmgpy.kinetics.MultiArrhenius* method), 100
discrepancy() (*rmgpy.kinetics.MultiPDepArrhenius* method), 106
discrepancy() (*rmgpy.kinetics.PDepArrhenius* method), 104
discrepancy() (*rmgpy.kinetics.PDepKineticsData* method), 101
discrepancy() (*rmgpy.kinetics.StickingCoefficient* method), 124
discrepancy() (*rmgpy.kinetics.SurfaceArrhenius* method), 121
discrepancy() (*rmgpy.kinetics.ThirdBody* method), 111
discrepancy() (*rmgpy.kinetics.Troe* method), 116
discrepancy() (*rmgpy.thermo.NASA* method), 281
discrepancy() (*rmgpy.thermo.NASAPolynomial* method), 286
discrepancy() (*rmgpy.thermo.ThermoData* method), 275
discrepancy() (*rmgpy.thermo.Wilhoit* method), 279
distribute_tree_distances() (*rmgpy.data.kinetics.KineticsFamily* method), 43
draw() (*arkane.KineticsJob* method), 13
draw() (*arkane.PressureDependenceJob* method), 17
draw() (*rmgpy.data.kinetics.DepositoryReaction* method), 31
draw() (*rmgpy.data.kinetics.LibraryReaction* method), 59
draw() (*rmgpy.data.kinetics.TemplateReaction* method), 79
draw() (*rmgpy.molecule.draw.MoleculeDrawer* method), 178
draw() (*rmgpy.molecule.draw.ReactionDrawer* method), 179
draw() (*rmgpy.molecule.Group* method), 158
draw() (*rmgpy.molecule.Molecule* method), 143
draw() (*rmgpy.reaction.Reaction* method), 217
draw() (*rmgpy.rmg.pdep.PDepReaction* method), 234
- ## E
- E0* (*rmgpy.pdep.Configuration* attribute), 183
E0 (*rmgpy.statmech.Conformer* attribute), 271
E0 (*rmgpy.thermo.NASA* attribute), 281
E0 (*rmgpy.thermo.NASAPolynomial* attribute), 284
E0 (*rmgpy.thermo.ThermoData* attribute), 274
E0 (*rmgpy.thermo.Wilhoit* attribute), 278
E0_prod (*rmgpy.kinetics.Eckart* attribute), 119
E0_reac (*rmgpy.kinetics.Eckart* attribute), 119
E0_TS (*rmgpy.kinetics.Eckart* attribute), 119
Ea (*rmgpy.kinetics.Arrhenius* attribute), 97
Ea (*rmgpy.kinetics.StickingCoefficient* attribute), 123
Ea (*rmgpy.kinetics.SurfaceArrhenius* attribute), 120
Eckart (class in *rmgpy.kinetics*), 118
Edge (class in *rmgpy.molecule.graph*), 127
efficiencies (*rmgpy.kinetics.Chebyshev* attribute), 108
efficiencies (*rmgpy.kinetics.Lindemann* attribute), 113
efficiencies (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 106
efficiencies (*rmgpy.kinetics.PDepArrhenius* attribute), 104
efficiencies (*rmgpy.kinetics.PDepKineticsData* attribute), 101
efficiencies (*rmgpy.kinetics.ThirdBody* attribute), 111
efficiencies (*rmgpy.kinetics.Troe* attribute), 116
Element (class in *rmgpy.molecule*), 132
element_count_from_conformer() (*arkane.ThermoJob* method), 18
ElementError, 288
energies (*rmgpy.statmech.HinderedRotor* attribute), 268
enlarge() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224

`ensure_species()` (*rmgpy.data.kinetics.DepositoryReaction method*), 19
`ensure_species()` (*rmgpy.data.kinetics.LibraryReaction method*), 31
`ensure_species()` (*rmgpy.data.kinetics.TemplateReaction method*), 59
`ensure_species()` (*rmgpy.data.kinetics.TemplateReaction method*), 79
`ensure_species()` (*rmgpy.reaction.Reaction method*), 217
`ensure_species()` (*rmgpy.rmg.pdep.PDepReaction method*), 234
`Entry` (*class in rmgpy.data.base*), 36
`enumerate_bonds()` (*rmgpy.molecule.Molecule method*), 143
`equals()` (*rmgpy.quantity.ArrayQuantity method*), 213
`equals()` (*rmgpy.quantity.ScalarQuantity method*), 212
`equivalent()` (*rmgpy.molecule.Atom method*), 137
`equivalent()` (*rmgpy.molecule.AtomType method*), 133
`equivalent()` (*rmgpy.molecule.Bond method*), 139
`equivalent()` (*rmgpy.molecule.graph.Edge method*), 127
`equivalent()` (*rmgpy.molecule.graph.Vertex method*), 127
`equivalent()` (*rmgpy.molecule.GroupAtom method*), 154
`equivalent()` (*rmgpy.molecule.GroupBond method*), 155
`ess_factory()` (*in module arkane.ess.factory*), 6
`ESSAdapter` (*class in arkane.ess*), 5
`estimate_kinetics()` (*rmgpy.data.kinetics.KineticsRules method*), 56
`estimate_kinetics_using_rate_rules()` (*rmgpy.data.kinetics.KineticsFamily method*), 43
`estimate_radical_thermo_via_hbi()` (*rmgpy.data.thermo.ThermoDatabase method*), 84
`estimate_thermo_via_group_additivity()` (*rmgpy.data.thermo.ThermoDatabase method*), 84
`eval_ext()` (*rmgpy.data.kinetics.KineticsFamily method*), 43
`evaluate()` (*rmgpy.data.statmechfit.DirectFit method*), 70
`evaluate()` (*rmgpy.data.statmechfit.PseudoFit method*), 72
`evaluate()` (*rmgpy.data.statmechfit.PseudoRotorFit method*), 71
`execute()` (*arkane.Arkane method*), 14
`execute()` (*arkane.ExplorerJob method*), 19
`execute()` (*arkane.KineticsJob method*), 13
`execute()` (*arkane.PressureDependenceJob method*), 17
`execute()` (*arkane.sensitivity.KineticsSensitivity method*), 19
`execute()` (*arkane.sensitivity.PDepSensitivity method*), 20
`execute()` (*arkane.StatMechJob method*), 18
`execute()` (*arkane.ThermoJob method*), 18
`execute()` (*rmgpy.rmg.main.RMG method*), 228
`explore_isomer()` (*rmgpy.rmg.pdep.PDepNetwork method*), 231
`ExplorerJob` (*class in arkane*), 19
`extend_node()` (*rmgpy.data.kinetics.KineticsFamily method*), 43
`extract_source_from_comments()` (*rmgpy.data.kinetics.KineticsDatabase method*), 37
`extract_source_from_comments()` (*rmgpy.data.kinetics.KineticsFamily method*), 44
`extract_source_from_comments()` (*rmgpy.data.thermo.ThermoDatabase method*), 84

F

`failure_keys` (*rmgpy.qm.gaussian.Gaussian attribute*), 195
`failure_keys` (*rmgpy.qm.gaussian.GaussianMol attribute*), 196
`failure_keys` (*rmgpy.qm.gaussian.GaussianMolPM3 attribute*), 198
`failure_keys` (*rmgpy.qm.gaussian.GaussianMolPM6 attribute*), 200
`failure_keys` (*rmgpy.qm.mopac.Mopac attribute*), 202
`failure_keys` (*rmgpy.qm.mopac.MopacMol attribute*), 203
`failure_keys` (*rmgpy.qm.mopac.MopacMolPM3 attribute*), 205
`failure_keys` (*rmgpy.qm.mopac.MopacMolPM6 attribute*), 207
`failure_keys` (*rmgpy.qm.mopac.MopacMolPM7 attribute*), 209
`feasible()` (*rmgpy.molecule.vf2.VF2 method*), 131
`fill_rules_by_averaging_up()` (*rmgpy.data.kinetics.KineticsFamily method*), 44
`fill_rules_by_averaging_up()` (*rmgpy.data.kinetics.KineticsRules method*), 56
`filter_structures()` (*in module rmgpy.molecule.filtration*), 169
`find_adj_lone_pair_multiple_bond_delocalization_paths()` (*in module rmgpy.molecule.pathfinder*), 171
`find_adj_lone_pair_radical_delocalization_paths()` (*in module rmgpy.molecule.pathfinder*), 171
`find_adj_lone_pair_radical_multiple_bond_delocalization_paths()` (*in module rmgpy.molecule.pathfinder*), 172

[find_adsorbate_conjugate_delocalization_paths\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_adsorbate_delocalization_paths\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_allyl_delocalization_paths\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_allyl_end_with_charge\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_butadiene\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_butadiene_end_with_charge\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_h_bonds\(\)](#) ([rmgpy.molecule.Molecule](#) method), 144
[find_isomorphism\(\)](#) ([rmgpy.molecule.graph.Graph](#) method), 128
[find_isomorphism\(\)](#) ([rmgpy.molecule.Group](#) method), 158
[find_isomorphism\(\)](#) ([rmgpy.molecule.Molecule](#) method), 144
[find_isomorphism\(\)](#) ([rmgpy.molecule.vf2.VF2](#) method), 131
[find_lone_pair_multiple_bond_paths\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 172
[find_N5dc_radical_delocalization_paths\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 170
[find_subgraph_isomorphisms\(\)](#) ([rmgpy.molecule.graph.Graph](#) method), 128
[find_subgraph_isomorphisms\(\)](#) ([rmgpy.molecule.Group](#) method), 158
[find_subgraph_isomorphisms\(\)](#) ([rmgpy.molecule.Molecule](#) method), 144
[find_subgraph_isomorphisms\(\)](#) ([rmgpy.molecule.vf2.VF2](#) method), 131
[find_unique_sites_in_charged_list\(\)](#) (in module [rmgpy.molecule.filtration](#)), 169
[fingerprint\(\)](#) ([rmgpy.molecule.Molecule](#) attribute), 144
[fingerprint\(\)](#) ([rmgpy.species.Species](#) attribute), 252
[finish\(\)](#) ([rmgpy.rmg.main.RMG](#) method), 228
[fit_cosine_potential_to_data\(\)](#) ([rmgpy.statmech.HinderedRotor](#) method), 268
[fit_fourier_potential_to_data\(\)](#) ([rmgpy.statmech.HinderedRotor](#) method), 268
[fit_interpolation_model\(\)](#) ([arkane.PressureDependenceJob](#) method), 17
[fit_interpolation_models\(\)](#) ([arkane.PressureDependenceJob](#) method), 17
[fit_statmech_direct\(\)](#) (in module [rmgpy.data.statmechfit](#)), 69
[fit_statmech_pseudo\(\)](#) (in module [rmgpy.data.statmechfit](#)), 69
[fit_statmech_pseudo_rotors\(\)](#) (in module [rmgpy.data.statmechfit](#)), 69
[fit_statmech_to_heat_capacity\(\)](#) (in module [rmgpy.data.statmechfit](#)), 69
[fit_to_data\(\)](#) ([rmgpy.kinetics.Arrhenius](#) method), 98
[fit_to_data\(\)](#) ([rmgpy.kinetics.Chebyshev](#) method), 108
[fit_to_data\(\)](#) ([rmgpy.kinetics.PDepArrhenius](#) method), 104
[fit_to_data\(\)](#) ([rmgpy.kinetics.StickingCoefficient](#) method), 124
[fit_to_data\(\)](#) ([rmgpy.kinetics.SurfaceArrhenius](#) method), 121
[fit_to_data\(\)](#) ([rmgpy.thermo.Wilhoit](#) method), 279
[fit_to_data_for_constant_b\(\)](#) ([rmgpy.thermo.Wilhoit](#) method), 279
[fix_barrier_height\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 32
[fix_barrier_height\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 59
[fix_barrier_height\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 80
[fix_barrier_height\(\)](#) ([rmgpy.reaction.Reaction](#) method), 217
[fix_barrier_height\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 234
[fix_diffusion_limited_a_factor\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 32
[fix_diffusion_limited_a_factor\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 59
[fix_diffusion_limited_a_factor\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 80
[fix_diffusion_limited_a_factor\(\)](#) ([rmgpy.reaction.Reaction](#) method), 218
[fix_diffusion_limited_a_factor\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 234
[ForbiddenStructureException](#), 288
[fourier\(\)](#) ([rmgpy.statmech.HinderedRotor](#) attribute), 268
[frequencies](#) ([rmgpy.statmech.HarmonicOscillator](#) attribute), 266
[frequency](#) ([rmgpy.kinetics.Eckart](#) attribute), 119
[frequency](#) ([rmgpy.kinetics.Wigner](#) attribute), 118
[frequency](#) ([rmgpy.species.TransitionState](#) attribute), 255

- frequency (*rmgpy.statmech.HinderedRotor* attribute), 268
- from_adjacency_list() (in module *rmgpy.molecule.adjlist*), 177
- from_adjacency_list() (*rmgpy.molecule.Group* method), 158
- from_adjacency_list() (*rmgpy.molecule.Molecule* method), 144
- from_adjacency_list() (*rmgpy.species.Species* method), 252
- from_augmented_inchi() (in module *rmgpy.molecule.translator*), 173
- from_augmented_inchi() (*rmgpy.molecule.Molecule* method), 144
- from_inchi() (in module *rmgpy.molecule.translator*), 174
- from_inchi() (*rmgpy.molecule.Molecule* method), 144
- from_ob_mol() (in module *rmgpy.molecule.converter*), 173
- from_rdkit_mol() (in module *rmgpy.molecule.converter*), 173
- from_smarts() (in module *rmgpy.molecule.translator*), 174
- from_smarts() (*rmgpy.molecule.Molecule* method), 145
- from_smiles() (in module *rmgpy.molecule.translator*), 174
- from_smiles() (*rmgpy.molecule.Molecule* method), 145
- from_smiles() (*rmgpy.species.Species* method), 252
- from_xyz() (*rmgpy.molecule.Molecule* method), 145
- G**
- Gaussian (class in *rmgpy.qm.gaussian*), 195
- GaussianLog (class in *arkane.ess*), 7
- GaussianMol (class in *rmgpy.qm.gaussian*), 195
- GaussianMolPM3 (class in *rmgpy.qm.gaussian*), 197
- GaussianMolPM6 (class in *rmgpy.qm.gaussian*), 199
- generate_3d_ts() (*rmgpy.data.kinetics.DepositoryReaction* method), 32
- generate_3d_ts() (*rmgpy.data.kinetics.LibraryReaction* method), 60
- generate_3d_ts() (*rmgpy.data.kinetics.TemplateReaction* method), 80
- generate_3d_ts() (*rmgpy.reaction.Reaction* method), 218
- generate_3d_ts() (*rmgpy.rmg.pdep.PDepReaction* method), 234
- generate_adj_lone_pair_multiple_bond_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_adj_lone_pair_radical_multiple_bond_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_adj_lone_pair_radical_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_adsorbate_conjugate_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_adsorbate_shift_down_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_adsorbate_shift_up_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_allyl_delocalization_resonance_structures() (in module *rmgpy.molecule.resonance*), 165
- generate_aromatic_resonance_structure() (in module *rmgpy.molecule.resonance*), 165
- generate_aryne_resonance_structures() (in module *rmgpy.molecule.resonance*), 166
- generate_cantera_files() (*rmgpy.rmg.main.RMG* method), 228
- generate_clar_structures() (in module *rmgpy.molecule.resonance*), 166
- generate_collision_matrix() (*rmgpy.pdep.Configuration* method), 183
- generate_collision_matrix() (*rmgpy.pdep.SingleExponentialDown* method), 181
- generate_energy_transfer_model() (*rmgpy.species.Species* method), 252
- generate_frequencies() (*rmgpy.data.statmech.GroupFrequencies* method), 36
- generate_full_me_matrix() (in module *rmgpy.pdep.me*), 187
- generate_h_bonded_structures() (*rmgpy.molecule.Molecule* method), 145
- generate_high_p_limit_kinetics() (*rmgpy.data.kinetics.DepositoryReaction* method), 32
- generate_high_p_limit_kinetics() (*rmgpy.data.kinetics.LibraryReaction* method), 60
- generate_high_p_limit_kinetics() (*rmgpy.data.kinetics.TemplateReaction* method), 80
- generate_high_p_limit_kinetics() (*rmgpy.reaction.Reaction* method), 218
- generate_high_p_limit_kinetics() (*rmgpy.rmg.pdep.PDepReaction* method), 235
- generate_isomorphic_resonance_structures() (in module *rmgpy.molecule.resonance*), 166
- generate_kekule_structure() (in module *rmgpy.molecule.resonance*), 166
- generate_kinetics() (*arkane.KineticsJob* method), 13
- generate_kinetics() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
- generate_lone_pair_multiple_bond_resonance_structures()

(in module *rmgpy.molecule.resonance*), 166
generate_N5dc_radical_resonance_structures() (in module *rmgpy.molecule.resonance*), 164
generate_old_library_entry() (*rmgpy.data.statmech.StatmechGroups* method), 73
generate_old_library_entry() (*rmgpy.data.statmech.StatmechLibrary* method), 76
generate_old_library_entry() (*rmgpy.data.thermo.ThermoGroups* method), 90
generate_old_library_entry() (*rmgpy.data.thermo.ThermoLibrary* method), 92
generate_old_tree() (*rmgpy.data.base.Database* method), 29
generate_old_tree() (*rmgpy.data.kinetics.KineticsDepository* method), 39
generate_old_tree() (*rmgpy.data.kinetics.KineticsFamily* method), 44
generate_old_tree() (*rmgpy.data.kinetics.KineticsGroups* method), 51
generate_old_tree() (*rmgpy.data.kinetics.KineticsLibrary* method), 53
generate_old_tree() (*rmgpy.data.kinetics.KineticsRules* method), 56
generate_old_tree() (*rmgpy.data.statmech.StatmechDepository* method), 67
generate_old_tree() (*rmgpy.data.statmech.StatmechGroups* method), 74
generate_old_tree() (*rmgpy.data.statmech.StatmechLibrary* method), 76
generate_old_tree() (*rmgpy.data.thermo.ThermoDepository* method), 88
generate_old_tree() (*rmgpy.data.thermo.ThermoGroups* method), 90
generate_old_tree() (*rmgpy.data.thermo.ThermoLibrary* method), 92
generate_optimal_aromatic_resonance_structures() (in module *rmgpy.molecule.resonance*), 166
generate_P_list() (*arkane.PressureDependenceJob* method), 17
generate_pairs() (*rmgpy.data.kinetics.DepositoryReaction* method), 32
generate_pairs() (*rmgpy.data.kinetics.LibraryReaction* method), 60
generate_pairs() (*rmgpy.data.kinetics.TemplateReaction* method), 80
generate_pairs() (*rmgpy.reaction.Reaction* method), 218
generate_pairs() (*rmgpy.rmg.pdep.PDepReaction* method), 235
generate_product_template() (*rmgpy.data.kinetics.KineticsFamily* method), 44
generate_qm_data() (*rmgpy.qm.gaussian.GaussianMol* method), 196
generate_qm_data() (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
generate_qm_data() (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
generate_qm_data() (*rmgpy.qm.molecule.QMMolecule* method), 192
generate_qm_data() (*rmgpy.qm.mopac.MopacMol* method), 203
generate_qm_data() (*rmgpy.qm.mopac.MopacMolPM3* method), 205
generate_qm_data() (*rmgpy.qm.mopac.MopacMolPM6* method), 207
generate_qm_data() (*rmgpy.qm.mopac.MopacMolPM7* method), 209
generate_rate_coefficients() (*rmgpy.solver.LiquidReactor* method), 244
generate_rate_coefficients() (*rmgpy.solver.MBSampledReactor* method), 248
generate_rate_coefficients() (*rmgpy.solver.SimpleReactor* method), 241
generate_rate_coefficients() (*rmgpy.solver.SurfaceReactor* method), 246
generate_rdkit_geometries() (*rmgpy.qm.molecule.Geometry* method), 191
generate_reactant_product_indices() (*rmgpy.solver.LiquidReactor* method), 244
generate_reactant_product_indices() (*rmgpy.solver.MBSampledReactor* method), 248
generate_reactant_product_indices() (*rmgpy.solver.ReactionSystem* method), 239
generate_reactant_product_indices() (*rmgpy.solver.SimpleReactor* method), 241
generate_reactant_product_indices() (*rmgpy.solver.SurfaceReactor* method), 246
generate_reaction_indices() (*rmgpy.solver.LiquidReactor* method), 244
generate_reaction_indices()

- (*rmgpy.solver.MBSampledReactor* method), 249
- generate_reaction_indices()* (*rmgpy.solver.ReactionSystem* method), 239
- generate_reaction_indices()* (*rmgpy.solver.SimpleReactor* method), 241
- generate_reaction_indices()* (*rmgpy.solver.SurfaceReactor* method), 246
- generate_reactions()* (*rmgpy.data.kinetics.KineticsDatabase* method), 37
- generate_reactions()* (*rmgpy.data.kinetics.KineticsFamily* method), 44
- generate_reactions_from_families()* (*rmgpy.data.kinetics.KineticsDatabase* method), 37
- generate_reactions_from_libraries()* (*rmgpy.data.kinetics.KineticsDatabase* method), 37
- generate_reactions_from_library()* (*rmgpy.data.kinetics.KineticsDatabase* method), 37
- generate_resonance_structures()* (in module *rmgpy.molecule.resonance*), 166
- generate_resonance_structures()* (*rmgpy.molecule.Molecule* method), 145
- generate_resonance_structures()* (*rmgpy.species.Species* method), 252
- generate_reverse_rate_coefficient()* (*rmgpy.data.kinetics.DepositoryReaction* method), 32
- generate_reverse_rate_coefficient()* (*rmgpy.data.kinetics.LibraryReaction* method), 60
- generate_reverse_rate_coefficient()* (*rmgpy.data.kinetics.TemplateReaction* method), 80
- generate_reverse_rate_coefficient()* (*rmgpy.reaction.Reaction* method), 218
- generate_reverse_rate_coefficient()* (*rmgpy.rmg.pdep.PDepReaction* method), 235
- generate_species_indices()* (*rmgpy.solver.LiquidReactor* method), 244
- generate_species_indices()* (*rmgpy.solver.MBSampledReactor* method), 249
- generate_species_indices()* (*rmgpy.solver.ReactionSystem* method), 239
- generate_species_indices()* (*rmgpy.solver.SimpleReactor* method), 242
- generate_species_indices()* (*rmgpy.solver.SurfaceReactor* method), 246
- generate_statmech()* (*rmgpy.species.Species* method), 252
- generate_T_list()* (*arkane.PressureDependenceJob* method), 17
- generate_thermo()* (*arkane.ThermoJob* method), 18
- generate_thermo()* (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
- generate_thermo_data()* (*rmgpy.qm.gaussian.GaussianMol* method), 196
- generate_thermo_data()* (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
- generate_thermo_data()* (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
- generate_thermo_data()* (*rmgpy.qm.molecule.QMMolecule* method), 192
- generate_thermo_data()* (*rmgpy.qm.mopac.MopacMol* method), 203
- generate_thermo_data()* (*rmgpy.qm.mopac.MopacMolPM3* method), 205
- generate_thermo_data()* (*rmgpy.qm.mopac.MopacMolPM6* method), 207
- generate_thermo_data()* (*rmgpy.qm.mopac.MopacMolPM7* method), 209
- generate_transport_data()* (*rmgpy.species.Species* method), 252
- generate_tree()* (*rmgpy.data.kinetics.KineticsFamily* method), 45
- generic_visit()* (*arkane.output.PrettifyVisitor* method), 14
- Geometry* (class in *rmgpy.qm.molecule*), 191
- get_active_modes()* (*rmgpy.statmech.Conformer* method), 272
- get_adatoms()* (*rmgpy.molecule.Molecule* method), 145
- get_all_cycles()* (*rmgpy.molecule.graph.Graph* method), 128
- get_all_cycles()* (*rmgpy.molecule.Group* method), 158
- get_all_cycles()* (*rmgpy.molecule.Molecule* method), 145
- get_all_cycles_of_size()* (*rmgpy.molecule.graph.Graph* method), 128
- get_all_cycles_of_size()* (*rmgpy.molecule.Group* method), 158
- get_all_cycles_of_size()* (*rmgpy.molecule.Molecule* method), 145

`get_all_cyclic_vertices()`
 (*rmgpy.molecule.graph.Graph* method), 129
`get_all_cyclic_vertices()`
 (*rmgpy.molecule.Group* method), 159
`get_all_cyclic_vertices()`
 (*rmgpy.molecule.Molecule* method), 145
`get_all_descendants()` (*rmgpy.data.base.Entry* method), 36
`get_all_edges()` (*rmgpy.molecule.graph.Graph* method), 129
`get_all_edges()` (*rmgpy.molecule.Group* method), 159
`get_all_edges()` (*rmgpy.molecule.Molecule* method), 145
`get_all_labeled_atoms()` (*rmgpy.molecule.Group* method), 159
`get_all_labeled_atoms()`
 (*rmgpy.molecule.Molecule* method), 145
`get_all_polycyclic_vertices()`
 (*rmgpy.molecule.graph.Graph* method), 129
`get_all_polycyclic_vertices()`
 (*rmgpy.molecule.Group* method), 159
`get_all_polycyclic_vertices()`
 (*rmgpy.molecule.Molecule* method), 145
`get_all_rules()` (*rmgpy.data.kinetics.KineticsRules* method), 56
`get_all_simple_cycles_of_size()`
 (*rmgpy.molecule.graph.Graph* method), 129
`get_all_simple_cycles_of_size()`
 (*rmgpy.molecule.Group* method), 159
`get_all_simple_cycles_of_size()`
 (*rmgpy.molecule.Molecule* method), 145
`get_all_species()` (*rmgpy.pdep.Network* method), 186
`get_all_species()` (*rmgpy.rmg.pdep.PDepNetwork* method), 231
`get_all_thermo_data()`
 (*rmgpy.data.thermo.ThermoDatabase* method), 84
`get_alpha()` (*rmgpy.pdep.SingleExponentialDown* method), 181
`get_aromatic_rings()` (*rmgpy.molecule.Molecule* method), 146
`get_atomtype()` (*in module rmgpy.molecule*), 133
`get_augmented_inchi_key()`
 (*rmgpy.qm.gaussian.GaussianMol* method), 196
`get_augmented_inchi_key()`
 (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
`get_augmented_inchi_key()`
 (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
`get_augmented_inchi_key()`
 (*rmgpy.qm.molecule.QMMolecule* method), 192
`get_augmented_inchi_key()`
 (*rmgpy.qm.mopac.MopacMol* method), 203
`get_augmented_inchi_key()`
 (*rmgpy.qm.mopac.MopacMolPM3* method), 205
`get_augmented_inchi_key()`
 (*rmgpy.qm.mopac.MopacMolPM6* method), 207
`get_augmented_inchi_key()`
 (*rmgpy.qm.mopac.MopacMolPM7* method), 209
`get_backbone_roots()`
 (*rmgpy.data.kinetics.KineticsFamily* method), 45
`get_bde()` (*rmgpy.molecule.Bond* method), 139
`get_bond()` (*rmgpy.molecule.Group* method), 159
`get_bond()` (*rmgpy.molecule.Molecule* method), 146
`get_bond_string()` (*rmgpy.molecule.Bond* method), 140
`get_bonds()` (*rmgpy.molecule.Group* method), 159
`get_bonds()` (*rmgpy.molecule.Molecule* method), 146
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.Chebyshev* method), 108
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.Lindemann* method), 113
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.MultiPDepArrhenius* method), 106
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.PDepArrhenius* method), 104
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.PDepKineticsData* method), 102
`get_cantera_efficiencies()`
 (*rmgpy.kinetics.ThirdBody* method), 111
`get_cantera_efficiencies()` (*rmgpy.kinetics.Troe* method), 116
`get_center_of_mass()` (*in module arkane.common*), 22
`get_center_of_mass()` (*rmgpy.statmech.Conformer* method), 272
`get_charge_span()` (*rmgpy.molecule.Molecule* method), 146
`get_charge_span_list()` (*in module rmgpy.molecule.filtration*), 169
`get_const_spc_indices()`
 (*rmgpy.solver.LiquidReactor* method), 244
`get_const_spc_indices()`

`(rmgpy.solver.SimpleReactor method)`, 242
`get_conversion_factor_from_si()`
`(rmgpy.quantity.ArrayQuantity method)`, 213
`get_conversion_factor_from_si()`
`(rmgpy.quantity.ScalarQuantity method)`, 212
`get_conversion_factor_from_si_to_cm_mol_s()`
`(rmgpy.quantity.ArrayQuantity method)`, 214
`get_conversion_factor_from_si_to_cm_mol_s()`
`(rmgpy.quantity.ScalarQuantity method)`, 212
`get_conversion_factor_to_si()`
`(rmgpy.quantity.ArrayQuantity method)`, 214
`get_conversion_factor_to_si()`
`(rmgpy.quantity.ScalarQuantity method)`, 212
`get_crude_mol_file_path()`
`(rmgpy.qm.molecule.Geometry method)`, 191
`get_D1_diagnostic()` (*arkane.ess.ESSAdapter method*), 5
`get_D1_diagnostic()` (*arkane.ess.GaussianLog method*), 7
`get_D1_diagnostic()` (*arkane.ess.MolproLog method*), 8
`get_D1_diagnostic()` (*arkane.ess.OrcaLog method*), 9
`get_D1_diagnostic()` (*arkane.ess.QChemLog method*), 10
`get_D1_diagnostic()` (*arkane.ess.TeraChemLog method*), 11
`get_density_of_states()` (*in module rmgpy.statmech.schrodinger*), 270
`get_density_of_states()` (*rmgpy.species.Species method*), 252
`get_density_of_states()`
`(rmgpy.species.TransitionState method)`, 255
`get_density_of_states()`
`(rmgpy.statmech.Conformer method)`, 272
`get_density_of_states()`
`(rmgpy.statmech.HarmonicOscillator method)`, 266
`get_density_of_states()`
`(rmgpy.statmech.HinderedRotor method)`, 268
`get_density_of_states()`
`(rmgpy.statmech.IdealGasTranslation method)`, 258
`get_density_of_states()` (*rmgpy.statmech.KRotor method*), 262
`get_density_of_states()`
`(rmgpy.statmech.LinearRotor method)`, 259
`get_density_of_states()`
`(rmgpy.statmech.NonlinearRotor method)`, 261
`get_density_of_states()`
`(rmgpy.statmech.SphericalTopRotor method)`, 264
`get_desorbed_molecules()`
`(rmgpy.molecule.Molecule method)`, 146
`get_deterministic_sssr()`
`(rmgpy.molecule.Molecule method)`, 146
`get_disparate_cycles()`
`(rmgpy.molecule.graph.Graph method)`, 129
`get_disparate_cycles()` (*rmgpy.molecule.Group method*), 159
`get_disparate_cycles()` (*rmgpy.molecule.Molecule method*), 146
`get_edge()` (*rmgpy.molecule.graph.Graph method*), 129
`get_edge()` (*rmgpy.molecule.Group method*), 159
`get_edge()` (*rmgpy.molecule.Molecule method*), 146
`get_edges()` (*rmgpy.molecule.graph.Graph method*), 129
`get_edges()` (*rmgpy.molecule.Group method*), 159
`get_edges()` (*rmgpy.molecule.Molecule method*), 146
`get_edges_in_cycle()`
`(rmgpy.molecule.graph.Graph method)`, 129
`get_edges_in_cycle()` (*rmgpy.molecule.Group method*), 159
`get_edges_in_cycle()` (*rmgpy.molecule.Molecule method*), 147
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.Chebyshev method)`, 109
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.Lindemann method)`, 113
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.MultiPDepArrhenius method)`, 106
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.PDepArrhenius method)`, 104
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.PDepKineticsData method)`, 102
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.ThirdBody method)`, 111
`get_effective_collider_efficiencies()`
`(rmgpy.kinetics.Troe method)`, 116
`get_effective_pressure()`
`(rmgpy.kinetics.Chebyshev method)`, 109
`get_effective_pressure()`
`(rmgpy.kinetics.Lindemann method)`, 113
`get_effective_pressure()`
`(rmgpy.kinetics.MultiPDepArrhenius method)`,

- 106
 get_effective_pressure() (rmgpy.kinetics.PDepArrhenius method), 104
 get_effective_pressure() (rmgpy.kinetics.PDepKineticsData method), 102
 get_effective_pressure() (rmgpy.kinetics.ThirdBody method), 111
 get_effective_pressure() (rmgpy.kinetics.Troe method), 116
 get_element() (in module rmgpy.molecule), 132
 get_element_count() (rmgpy.molecule.Group method), 159
 get_element_count() (rmgpy.molecule.Molecule method), 147
 get_element_mass() (in module arkane.common), 22
 get_end_roots() (rmgpy.data.kinetics.KineticsFamily method), 45
 get_energy_filtered_reactions() (rmgpy.rmg.pdep.PDepNetwork method), 231
 get_enthalpies_of_reaction() (rmgpy.data.kinetics.DepositoryReaction method), 32
 get_enthalpies_of_reaction() (rmgpy.data.kinetics.LibraryReaction method), 60
 get_enthalpies_of_reaction() (rmgpy.data.kinetics.TemplateReaction method), 80
 get_enthalpies_of_reaction() (rmgpy.reaction.Reaction method), 218
 get_enthalpies_of_reaction() (rmgpy.rmg.pdep.PDepReaction method), 235
 get_enthalpy() (in module rmgpy.statmech.schrodinger), 270
 get_enthalpy() (rmgpy.pdep.Configuration method), 184
 get_enthalpy() (rmgpy.species.Species method), 252
 get_enthalpy() (rmgpy.species.TransitionState method), 255
 get_enthalpy() (rmgpy.statmech.Conformer method), 272
 get_enthalpy() (rmgpy.statmech.HarmonicOscillator method), 266
 get_enthalpy() (rmgpy.statmech.HinderedRotor method), 269
 get_enthalpy() (rmgpy.statmech.IdealGasTranslation method), 258
 get_enthalpy() (rmgpy.statmech.KRotor method), 262
 get_enthalpy() (rmgpy.statmech.LinearRotor method), 260
 get_enthalpy() (rmgpy.statmech.NonlinearRotor method), 261
 get_enthalpy() (rmgpy.statmech.SphericalTopRotor method), 264
 get_enthalpy() (rmgpy.thermo.NASA method), 281
 get_enthalpy() (rmgpy.thermo.NASAPolynomial method), 286
 get_enthalpy() (rmgpy.thermo.ThermoData method), 275
 get_enthalpy() (rmgpy.thermo.Wilhoit method), 279
 get_enthalpy_of_reaction() (rmgpy.data.kinetics.DepositoryReaction method), 32
 get_enthalpy_of_reaction() (rmgpy.data.kinetics.LibraryReaction method), 60
 get_enthalpy_of_reaction() (rmgpy.data.kinetics.TemplateReaction method), 80
 get_enthalpy_of_reaction() (rmgpy.reaction.Reaction method), 218
 get_enthalpy_of_reaction() (rmgpy.rmg.pdep.PDepReaction method), 235
 get_entries() (rmgpy.data.kinetics.KineticsRules method), 56
 get_entries_to_save() (rmgpy.data.base.Database method), 29
 get_entries_to_save() (rmgpy.data.kinetics.KineticsDepository method), 39
 get_entries_to_save() (rmgpy.data.kinetics.KineticsFamily method), 45
 get_entries_to_save() (rmgpy.data.kinetics.KineticsGroups method), 51
 get_entries_to_save() (rmgpy.data.kinetics.KineticsLibrary method), 53
 get_entries_to_save() (rmgpy.data.kinetics.KineticsRules method), 56
 get_entries_to_save() (rmgpy.data.statmech.StatmechDepository method), 67
 get_entries_to_save() (rmgpy.data.statmech.StatmechGroups method), 74
 get_entries_to_save() (rmgpy.data.statmech.StatmechLibrary method), 76
 get_entries_to_save() (rmgpy.data.thermo.ThermoDepository method), 88

`get_entries_to_save()`
(*rmgpy.data.thermo.ThermoGroups* method), 90

`get_entries_to_save()`
(*rmgpy.data.thermo.ThermoLibrary* method), 92

`get_entropies_of_reaction()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 32

`get_entropies_of_reaction()`
(*rmgpy.data.kinetics.LibraryReaction* method), 60

`get_entropies_of_reaction()`
(*rmgpy.data.kinetics.TemplateReaction* method), 80

`get_entropies_of_reaction()`
(*rmgpy.reaction.Reaction* method), 218

`get_entropies_of_reaction()`
(*rmgpy.rmg.pdep.PDepReaction* method), 235

`get_entropy()` (in module *rmgpy.statmech.schrodinger*), 270

`get_entropy()` (*rmgpy.pdep.Configuration* method), 184

`get_entropy()` (*rmgpy.species.Species* method), 252

`get_entropy()` (*rmgpy.species.TransitionState* method), 255

`get_entropy()` (*rmgpy.statmech.Conformer* method), 272

`get_entropy()` (*rmgpy.statmech.HarmonicOscillator* method), 266

`get_entropy()` (*rmgpy.statmech.HinderedRotor* method), 269

`get_entropy()` (*rmgpy.statmech.IdealGasTranslation* method), 258

`get_entropy()` (*rmgpy.statmech.KRotor* method), 263

`get_entropy()` (*rmgpy.statmech.LinearRotor* method), 260

`get_entropy()` (*rmgpy.statmech.NonlinearRotor* method), 261

`get_entropy()` (*rmgpy.statmech.SphericalTopRotor* method), 264

`get_entropy()` (*rmgpy.thermo.NASA* method), 281

`get_entropy()` (*rmgpy.thermo.NASAPolynomial* method), 286

`get_entropy()` (*rmgpy.thermo.ThermoData* method), 275

`get_entropy()` (*rmgpy.thermo.Wilhoit* method), 279

`get_entropy_of_reaction()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 32

`get_entropy_of_reaction()`
(*rmgpy.data.kinetics.LibraryReaction* method), 60

`get_entropy_of_reaction()`
(*rmgpy.data.kinetics.TemplateReaction* method), 80

`get_entropy_of_reaction()`
(*rmgpy.reaction.Reaction* method), 218

`get_entropy_of_reaction()`
(*rmgpy.rmg.pdep.PDepReaction* method), 235

`get_equilibrium_constant()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 33

`get_equilibrium_constant()`
(*rmgpy.data.kinetics.LibraryReaction* method), 60

`get_equilibrium_constant()`
(*rmgpy.data.kinetics.TemplateReaction* method), 81

`get_equilibrium_constant()`
(*rmgpy.reaction.Reaction* method), 218

`get_equilibrium_constant()`
(*rmgpy.rmg.pdep.PDepReaction* method), 235

`get_equilibrium_constants()`
(*rmgpy.data.kinetics.DepositoryReaction* method), 33

`get_equilibrium_constants()`
(*rmgpy.data.kinetics.LibraryReaction* method), 61

`get_equilibrium_constants()`
(*rmgpy.data.kinetics.TemplateReaction* method), 81

`get_equilibrium_constants()`
(*rmgpy.reaction.Reaction* method), 219

`get_equilibrium_constants()`
(*rmgpy.rmg.pdep.PDepReaction* method), 235

`get_extension_edge()`
(*rmgpy.data.kinetics.KineticsFamily* method), 45

`get_extensions()` (*rmgpy.molecule.Group* method), 159

`get_features()` (*rmgpy.molecule.AtomType* method), 133

`get_file_path()` (*rmgpy.qm.gaussian.GaussianMol* method), 196

`get_file_path()` (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198

`get_file_path()` (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200

`get_file_path()` (*rmgpy.qm.molecule.Geometry* method), 191

`get_file_path()` (*rmgpy.qm.molecule.QMMolecule* method), 192

`get_file_path()` (*rmgpy.qm.mopac.MopacMol*

`method`), 203
`get_file_path()` (`rmgpy.qm.mopac.MopacMolPM3` `method`), 205
`get_file_path()` (`rmgpy.qm.mopac.MopacMolPM6` `method`), 207
`get_file_path()` (`rmgpy.qm.mopac.MopacMolPM7` `method`), 209
`get_formula()` (`rmgpy.molecule.Molecule` `method`), 147
`get_forward_reaction_for_family_entry()` (`rmgpy.data.kinetics.KineticsDatabase` `method`), 37
`get_free_energies_of_reaction()` (`rmgpy.data.kinetics.DepositoryReaction` `method`), 33
`get_free_energies_of_reaction()` (`rmgpy.data.kinetics.LibraryReaction` `method`), 61
`get_free_energies_of_reaction()` (`rmgpy.data.kinetics.TemplateReaction` `method`), 81
`get_free_energies_of_reaction()` (`rmgpy.reaction.Reaction` `method`), 219
`get_free_energies_of_reaction()` (`rmgpy.rmg.pdep.PDepReaction` `method`), 235
`get_free_energy()` (`rmgpy.pdep.Configuration` `method`), 184
`get_free_energy()` (`rmgpy.species.Species` `method`), 252
`get_free_energy()` (`rmgpy.species.TransitionState` `method`), 255
`get_free_energy()` (`rmgpy.statmech.Conformer` `method`), 272
`get_free_energy()` (`rmgpy.thermo.NASA` `method`), 281
`get_free_energy()` (`rmgpy.thermo.NASAPolynomial` `method`), 286
`get_free_energy()` (`rmgpy.thermo.ThermoData` `method`), 275
`get_free_energy()` (`rmgpy.thermo.Wilhoit` `method`), 279
`get_free_energy_of_reaction()` (`rmgpy.data.kinetics.DepositoryReaction` `method`), 33
`get_free_energy_of_reaction()` (`rmgpy.data.kinetics.LibraryReaction` `method`), 61
`get_free_energy_of_reaction()` (`rmgpy.data.kinetics.TemplateReaction` `method`), 81
`get_free_energy_of_reaction()` (`rmgpy.reaction.Reaction` `method`), 219
`get_free_energy_of_reaction()` (`rmgpy.rmg.pdep.PDepReaction` `method`), 236
`get_frequency()` (`rmgpy.statmech.HinderedRotor` `method`), 269
`get_frequency_groups()` (`rmgpy.data.statmech.StatmechGroups` `method`), 74
`get_hamiltonian()` (`rmgpy.statmech.HinderedRotor` `method`), 269
`get_heat_capacity()` (in `rmgpy.statmech.schrodinger` module), 271
`get_heat_capacity()` (`rmgpy.pdep.Configuration` `method`), 184
`get_heat_capacity()` (`rmgpy.species.Species` `method`), 252
`get_heat_capacity()` (`rmgpy.species.TransitionState` `method`), 255
`get_heat_capacity()` (`rmgpy.statmech.Conformer` `method`), 272
`get_heat_capacity()` (`rmgpy.statmech.HarmonicOscillator` `method`), 266
`get_heat_capacity()` (`rmgpy.statmech.HinderedRotor` `method`), 269
`get_heat_capacity()` (`rmgpy.statmech.IdealGasTranslation` `method`), 258
`get_heat_capacity()` (`rmgpy.statmech.KRotor` `method`), 263
`get_heat_capacity()` (`rmgpy.statmech.LinearRotor` `method`), 260
`get_heat_capacity()` (`rmgpy.statmech.NonlinearRotor` `method`), 261
`get_heat_capacity()` (`rmgpy.statmech.SphericalTopRotor` `method`), 264
`get_heat_capacity()` (`rmgpy.thermo.NASA` `method`), 281
`get_heat_capacity()` (`rmgpy.thermo.NASAPolynomial` `method`), 286
`get_heat_capacity()` (`rmgpy.thermo.ThermoData` `method`), 275
`get_heat_capacity()` (`rmgpy.thermo.Wilhoit` `method`), 279
`get_internal_reduced_moment_of_inertia()` (`rmgpy.statmech.Conformer` `method`), 272
`get_kinetics()` (`rmgpy.data.kinetics.KineticsFamily` `method`), 46
`get_kinetics_for_template()` (`rmgpy.data.kinetics.KineticsFamily` `method`), 46

`get_kinetics_from_depository()`
 (*rmgpy.data.kinetics.KineticsFamily* method), 46
`get_labeled_atoms()` (*rmgpy.molecule.Group* method), 159
`get_labeled_atoms()` (*rmgpy.molecule.Molecule* method), 147
`get_labeled_reactants_and_products()`
 (*rmgpy.data.kinetics.KineticsFamily* method), 46
`get_largest_ring()` (*rmgpy.molecule.graph.Graph* method), 129
`get_largest_ring()` (*rmgpy.molecule.Group* method), 159
`get_largest_ring()` (*rmgpy.molecule.Molecule* method), 147
`get_layering_indices()`
 (*rmgpy.solver.LiquidReactor* method), 244
`get_layering_indices()`
 (*rmgpy.solver.MBSampledReactor* method), 249
`get_layering_indices()`
 (*rmgpy.solver.ReactionSystem* method), 239
`get_layering_indices()`
 (*rmgpy.solver.SimpleReactor* method), 242
`get_layering_indices()`
 (*rmgpy.solver.SurfaceReactor* method), 246
`get_leak_branching_ratios()`
 (*rmgpy.rmg.pdep.PDepNetwork* method), 231
`get_leak_coefficient()`
 (*rmgpy.rmg.pdep.PDepNetwork* method), 231
`get_level_degeneracy()`
 (*rmgpy.statmech.HinderedRotor* method), 269
`get_level_degeneracy()` (*rmgpy.statmech.KRotor* method), 263
`get_level_degeneracy()`
 (*rmgpy.statmech.LinearRotor* method), 260
`get_level_degeneracy()`
 (*rmgpy.statmech.SphericalTopRotor* method), 264
`get_level_energy()` (*rmgpy.statmech.HinderedRotor* method), 269
`get_level_energy()` (*rmgpy.statmech.KRotor* method), 263
`get_level_energy()` (*rmgpy.statmech.LinearRotor* method), 260
`get_level_energy()` (*rmgpy.statmech.SphericalTopRotor* method), 264
`get_libraries()` (*arkane.Arkane* method), 14
`get_library_reactions()`
 (*rmgpy.data.kinetics.KineticsLibrary* method), 53
`get_max_cycle_overlap()`
 (*rmgpy.molecule.graph.Graph* method), 129
`get_max_cycle_overlap()` (*rmgpy.molecule.Group* method), 160
`get_max_cycle_overlap()`
 (*rmgpy.molecule.Molecule* method), 147
`get_maximum_leak_species()`
 (*rmgpy.rmg.pdep.PDepNetwork* method), 231
`get_mean_sigma_and_epsilon()`
 (*rmgpy.data.kinetics.DepositoryReaction* method), 33
`get_mean_sigma_and_epsilon()`
 (*rmgpy.data.kinetics.LibraryReaction* method), 61
`get_mean_sigma_and_epsilon()`
 (*rmgpy.data.kinetics.TemplateReaction* method), 81
`get_mean_sigma_and_epsilon()`
 (*rmgpy.reaction.Reaction* method), 219
`get_mean_sigma_and_epsilon()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 236
`get_model_size()` (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 224
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.gaussian.GaussianMol* method), 196
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.molecule.QMMolecule* method), 193
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.mopac.MopacMol* method), 203
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.mopac.MopacMolPM3* method), 205
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.mopac.MopacMolPM6* method), 207
`get_mol_file_path_for_calculation()`
 (*rmgpy.qm.mopac.MopacMolPM7* method), 209
`get_molecular_weight()` (*rmgpy.molecule.Molecule* method), 147
`get_moment_of_inertia_tensor()` (in module *arkane.common*), 22

- [get_moment_of_inertia_tensor\(\)](#) (*rmgpy.statmech.Conformer* method), 272
[get_monocycles\(\)](#) (*rmgpy.molecule.graph.Graph* method), 129
[get_monocycles\(\)](#) (*rmgpy.molecule.Group* method), 160
[get_monocycles\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_net_charge\(\)](#) (*rmgpy.molecule.Group* method), 160
[get_net_charge\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_net_charge\(\)](#) (*rmgpy.species.Species* method), 252
[get_nth_neighbor\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_num_atoms\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_number_degrees_of_freedom\(\)](#) (*rmgpy.statmech.Conformer* method), 273
[get_number_of_atoms\(\)](#) (*arkane.ess.ESSAdapter* method), 5
[get_number_of_atoms\(\)](#) (*arkane.ess.GaussianLog* method), 7
[get_number_of_atoms\(\)](#) (*arkane.ess.MolproLog* method), 8
[get_number_of_atoms\(\)](#) (*arkane.ess.OrcaLog* method), 9
[get_number_of_atoms\(\)](#) (*arkane.ess.QChemLog* method), 10
[get_number_of_atoms\(\)](#) (*arkane.ess.TeraChemLog* method), 11
[get_octet_deviation\(\)](#) (in module *rmgpy.molecule.filtration*), 169
[get_octet_deviation_list\(\)](#) (in module *rmgpy.molecule.filtration*), 169
[get_order_num\(\)](#) (*rmgpy.molecule.Bond* method), 140
[get_order_num\(\)](#) (*rmgpy.molecule.GroupBond* method), 155
[get_order_str\(\)](#) (*rmgpy.molecule.Bond* method), 140
[get_order_str\(\)](#) (*rmgpy.molecule.GroupBond* method), 155
[get_other_vertex\(\)](#) (*rmgpy.molecule.Bond* method), 140
[get_other_vertex\(\)](#) (*rmgpy.molecule.graph.Edge* method), 127
[get_other_vertex\(\)](#) (*rmgpy.molecule.GroupBond* method), 155
[get_parser\(\)](#) (*rmgpy.qm.gaussian.GaussianMol* method), 196
[get_parser\(\)](#) (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
[get_parser\(\)](#) (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
[get_parser\(\)](#) (*rmgpy.qm.mopac.Mopac* method), 202
[get_parser\(\)](#) (*rmgpy.qm.mopac.MopacMol* method), 203
[get_parser\(\)](#) (*rmgpy.qm.mopac.MopacMolPM3* method), 205
[get_parser\(\)](#) (*rmgpy.qm.mopac.MopacMolPM6* method), 207
[get_parser\(\)](#) (*rmgpy.qm.mopac.MopacMolPM7* method), 209
[get_partition_function\(\)](#) (in module *rmgpy.statmech.schrodinger*), 271
[get_partition_function\(\)](#) (*rmgpy.species.Species* method), 252
[get_partition_function\(\)](#) (*rmgpy.species.TransitionState* method), 255
[get_partition_function\(\)](#) (*rmgpy.statmech.Conformer* method), 273
[get_partition_function\(\)](#) (*rmgpy.statmech.HarmonicOscillator* method), 266
[get_partition_function\(\)](#) (*rmgpy.statmech.HinderedRotor* method), 269
[get_partition_function\(\)](#) (*rmgpy.statmech.IdealGasTranslation* method), 258
[get_partition_function\(\)](#) (*rmgpy.statmech.KRotor* method), 263
[get_partition_function\(\)](#) (*rmgpy.statmech.LinearRotor* method), 260
[get_partition_function\(\)](#) (*rmgpy.statmech.NonlinearRotor* method), 261
[get_partition_function\(\)](#) (*rmgpy.statmech.SphericalTopRotor* method), 264
[get_polycycles\(\)](#) (*rmgpy.molecule.graph.Graph* method), 129
[get_polycycles\(\)](#) (*rmgpy.molecule.Group* method), 160
[get_polycycles\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_possible_structures\(\)](#) (*rmgpy.data.base.LogicOr* method), 64
[get_potential\(\)](#) (*rmgpy.statmech.HinderedRotor* method), 269
[get_principal_moments_of_inertia\(\)](#) (in module *arkane.common*), 22
[get_principal_moments_of_inertia\(\)](#) (*rmgpy.statmech.Conformer* method), 273
[get_radical_atoms\(\)](#) (*rmgpy.molecule.Molecule* method), 147
[get_radical_count\(\)](#) (*rmgpy.molecule.Molecule*

`method`), 147
`get_rate_coefficient()`
 (`rmgpy.data.kinetics.DepositoryReaction`
 `method`), 33
`get_rate_coefficient()`
 (`rmgpy.data.kinetics.LibraryReaction` `method`),
 61
`get_rate_coefficient()`
 (`rmgpy.data.kinetics.TemplateReaction`
 `method`), 81
`get_rate_coefficient()` (`rmgpy.kinetics.Arrhenius`
 `method`), 98
`get_rate_coefficient()` (`rmgpy.kinetics.Chebyshev`
 `method`), 109
`get_rate_coefficient()`
 (`rmgpy.kinetics.KineticsData` `method`), 96
`get_rate_coefficient()`
 (`rmgpy.kinetics.Lindemann` `method`), 113
`get_rate_coefficient()`
 (`rmgpy.kinetics.MultiArrhenius` `method`),
 100
`get_rate_coefficient()`
 (`rmgpy.kinetics.MultiPDepArrhenius` `method`),
 106
`get_rate_coefficient()`
 (`rmgpy.kinetics.PDepArrhenius` `method`),
 104
`get_rate_coefficient()`
 (`rmgpy.kinetics.PDepKineticsData` `method`),
 102
`get_rate_coefficient()`
 (`rmgpy.kinetics.StickingCoefficient` `method`),
 124
`get_rate_coefficient()`
 (`rmgpy.kinetics.SurfaceArrhenius` `method`),
 121
`get_rate_coefficient()` (`rmgpy.kinetics.ThirdBody`
 `method`), 111
`get_rate_coefficient()` (`rmgpy.kinetics.Troe`
 `method`), 116
`get_rate_coefficient()` (`rmgpy.reaction.Reaction`
 `method`), 219
`get_rate_coefficient()`
 (`rmgpy.rmg.pdep.PDepReaction` `method`),
 236
`get_rate_filtered_products()`
 (`rmgpy.rmg.pdep.PDepNetwork` `method`),
 232
`get_rate_rule()` (`rmgpy.data.kinetics.KineticsFamily`
 `method`), 46
`get_reaction_matches()`
 (`rmgpy.data.kinetics.KineticsFamily` `method`),
 46
`get_reaction_pairs()`
 (`rmgpy.data.kinetics.KineticsFamily` `method`),
 46
`get_reaction_template()`
 (`rmgpy.data.kinetics.KineticsFamily` `method`),
 46
`get_reaction_template()`
 (`rmgpy.data.kinetics.KineticsGroups` `method`),
 51
`get_reaction_template_labels()`
 (`rmgpy.data.kinetics.KineticsFamily` `method`),
 47
`get_reduced_mass()` (`rmgpy.data.kinetics.DepositoryReaction`
 `method`), 33
`get_reduced_mass()` (`rmgpy.data.kinetics.LibraryReaction`
 `method`), 61
`get_reduced_mass()` (`rmgpy.data.kinetics.TemplateReaction`
 `method`), 81
`get_reduced_mass()` (`rmgpy.reaction.Reaction`
 `method`), 219
`get_reduced_mass()` (`rmgpy.rmg.pdep.PDepReaction`
 `method`), 236
`get_reduced_pressure()` (`rmgpy.kinetics.Chebyshev`
 `method`), 109
`get_reduced_temperature()`
 (`rmgpy.kinetics.Chebyshev` `method`), 109
`get_refined_mol_file_path()`
 (`rmgpy.qm.molecule.Geometry` `method`),
 191
`get_relevant_cycles()`
 (`rmgpy.molecule.graph.Graph` `method`),
 129
`get_relevant_cycles()` (`rmgpy.molecule.Group`
 `method`), 160
`get_relevant_cycles()` (`rmgpy.molecule.Molecule`
 `method`), 148
`get_resonance_hybrid()` (`rmgpy.species.Species`
 `method`), 252
`get_reverse()` (`rmgpy.data.kinetics.ReactionRecipe`
 `method`), 65
`get_reversible_potential()`
 (`rmgpy.data.kinetics.DepositoryReaction`
 `method`), 33
`get_reversible_potential()`
 (`rmgpy.data.kinetics.LibraryReaction` `method`),
 61
`get_reversible_potential()`
 (`rmgpy.data.kinetics.TemplateReaction`
 `method`), 81
`get_reversible_potential()`
 (`rmgpy.reaction.Reaction` `method`), 219
`get_reversible_potential()`
 (`rmgpy.rmg.pdep.PDepReaction` `method`),
 236
`get_ring_groups_from_comments()`

(rmgpy.data.thermo.ThermoDatabase method), 85
 get_root_template() (rmgpy.data.kinetics.KineticsFamily method), 47
 get_rule() (rmgpy.data.kinetics.KineticsRules method), 56
 get_rxn_batches() (rmgpy.data.kinetics.KineticsFamily method), 47
 get_singlet_carbene_count() (rmgpy.molecule.Molecule method), 148
 get_smallest_set_of_smallest_rings() (rmgpy.molecule.graph.Graph method), 130
 get_smallest_set_of_smallest_rings() (rmgpy.molecule.Group method), 160
 get_smallest_set_of_smallest_rings() (rmgpy.molecule.Molecule method), 148
 get_software() (arkane.ess.ESSAdapter method), 5
 get_software() (arkane.ess.GaussianLog method), 7
 get_software() (arkane.ess.MolproLog method), 8
 get_software() (arkane.ess.OrcaLog method), 9
 get_software() (arkane.ess.QChemLog method), 10
 get_software() (arkane.ess.TeraChemLog method), 12
 get_source() (rmgpy.data.kinetics.DepositoryReaction method), 33
 get_source() (rmgpy.data.kinetics.LibraryReaction method), 61
 get_source() (rmgpy.data.kinetics.TemplateReaction method), 81
 get_source() (rmgpy.rmg.pdep.PDepReaction method), 236
 get_sources_for_template() (rmgpy.data.kinetics.KineticsFamily method), 47
 get_species() (rmgpy.data.base.Database method), 29
 get_species() (rmgpy.data.kinetics.KineticsDepository method), 39
 get_species() (rmgpy.data.kinetics.KineticsFamily method), 47
 get_species() (rmgpy.data.kinetics.KineticsGroups method), 51
 get_species() (rmgpy.data.kinetics.KineticsLibrary method), 53
 get_species() (rmgpy.data.kinetics.KineticsRules method), 56
 get_species() (rmgpy.data.statmech.StatmechDepository method), 67
 get_species() (rmgpy.data.statmech.StatmechGroups method), 74
 get_species() (rmgpy.data.statmech.StatmechLibrary method), 76
 get_species() (rmgpy.data.thermo.ThermoDepository method), 88
 get_species() (rmgpy.data.thermo.ThermoGroups method), 90
 get_species() (rmgpy.data.thermo.ThermoLibrary method), 92
 get_species_identifier() (in module rmgpy.chemkin), 26
 get_species_index() (rmgpy.solver.LiquidReactor method), 244
 get_species_index() (rmgpy.solver.MBSampledReactor method), 249
 get_species_index() (rmgpy.solver.ReactionSystem method), 239
 get_species_index() (rmgpy.solver.SimpleReactor method), 242
 get_species_index() (rmgpy.solver.SurfaceReactor method), 246
 get_species_reaction_lists() (rmgpy.rmg.model.CoreEdgeReactionModel method), 224
 get_statmech_data() (rmgpy.data.statmech.StatmechDatabase method), 65
 get_statmech_data() (rmgpy.data.statmech.StatmechGroups method), 74
 get_statmech_data_from_depository() (rmgpy.data.statmech.StatmechDatabase method), 65
 get_statmech_data_from_groups() (rmgpy.data.statmech.StatmechDatabase method), 65
 get_statmech_data_from_library() (rmgpy.data.statmech.StatmechDatabase method), 65
 get_sticking_coefficient() (rmgpy.data.kinetics.LibraryReaction method), 61
 get_sticking_coefficient() (rmgpy.kinetics.StickingCoefficient method), 124
 get_stoichiometric_coefficient() (rmgpy.data.kinetics.DepositoryReaction method), 33
 get_stoichiometric_coefficient() (rmgpy.data.kinetics.LibraryReaction method), 61
 get_stoichiometric_coefficient() (rmgpy.data.kinetics.TemplateReaction method), 81
 get_stoichiometric_coefficient() (rmgpy.reaction.Reaction method), 219

`get_stoichiometric_coefficient()` (*rmgpy.rmg.pdep.PDepReaction* method), 236

`get_stoichiometry_matrix()` (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 225

`get_str_xyz()` (in module *arkane.output*), 15

`get_sum_of_states()` (in module *rmgpy.statmech.schrodinger*), 271

`get_sum_of_states()` (*rmgpy.species.Species* method), 253

`get_sum_of_states()` (*rmgpy.species.TransitionState* method), 255

`get_sum_of_states()` (*rmgpy.statmech.Conformer* method), 273

`get_sum_of_states()` (*rmgpy.statmech.HarmonicOscillator* method), 266

`get_sum_of_states()` (*rmgpy.statmech.HinderedRotor* method), 269

`get_sum_of_states()` (*rmgpy.statmech.IdealGasTranslation* method), 258

`get_sum_of_states()` (*rmgpy.statmech.KRotor* method), 263

`get_sum_of_states()` (*rmgpy.statmech.LinearRotor* method), 260

`get_sum_of_states()` (*rmgpy.statmech.NonlinearRotor* method), 261

`get_sum_of_states()` (*rmgpy.statmech.SphericalTopRotor* method), 265

`get_surface_rate_coefficient()` (*rmgpy.data.kinetics.DepositoryReaction* method), 33

`get_surface_rate_coefficient()` (*rmgpy.data.kinetics.LibraryReaction* method), 61

`get_surface_rate_coefficient()` (*rmgpy.data.kinetics.TemplateReaction* method), 81

`get_surface_rate_coefficient()` (*rmgpy.reaction.Reaction* method), 219

`get_surface_rate_coefficient()` (*rmgpy.rmg.pdep.PDepReaction* method), 236

`get_surface_sites()` (*rmgpy.molecule.Group* method), 160

`get_surface_sites()` (*rmgpy.molecule.Molecule* method), 148

`get_symmetric_top_rotors()` (*rmgpy.statmech.Conformer* method), 273

`get_symmetry_number()` (*rmgpy.molecule.Molecule* method), 148

`get_symmetry_number()` (*rmgpy.species.Species* method), 253

`get_symmetry_properties()` (*arkane.ess.ESSAdapter* method), 5

`get_symmetry_properties()` (*arkane.ess.GaussianLog* method), 7

`get_symmetry_properties()` (*arkane.ess.MolproLog* method), 8

`get_symmetry_properties()` (*arkane.ess.OrcaLog* method), 9

`get_symmetry_properties()` (*arkane.ess.QChemLog* method), 10

`get_symmetry_properties()` (*arkane.ess.TeraChemLog* method), 12

`get_T1_diagnostic()` (*arkane.ess.ESSAdapter* method), 5

`get_T1_diagnostic()` (*arkane.ess.GaussianLog* method), 7

`get_T1_diagnostic()` (*arkane.ess.MolproLog* method), 8

`get_T1_diagnostic()` (*arkane.ess.OrcaLog* method), 9

`get_T1_diagnostic()` (*arkane.ess.QChemLog* method), 10

`get_T1_diagnostic()` (*arkane.ess.TeraChemLog* method), 11

`get_thermo_data()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data()` (*rmgpy.qm.main.QMCalculator* method), 190

`get_thermo_data()` (*rmgpy.species.Species* method), 253

`get_thermo_data_for_surface_species()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data_from_depository()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data_from_groups()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data_from_libraries()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data_from_library()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_data_from_ml()` (*rmgpy.data.thermo.ThermoDatabase* method), 85

`get_thermo_file_path()` (*rmgpy.qm.gaussian.GaussianMol* method),

- 196
- `get_thermo_file_path()`
(*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
- `get_thermo_file_path()`
(*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
- `get_thermo_file_path()`
(*rmgpy.qm.molecule.QMMolecule* method), 193
- `get_thermo_file_path()`
(*rmgpy.qm.mopac.MopacMol* method), 203
- `get_thermo_file_path()`
(*rmgpy.qm.mopac.MopacMolPM3* method), 205
- `get_thermo_file_path()`
(*rmgpy.qm.mopac.MopacMolPM6* method), 207
- `get_thermo_file_path()`
(*rmgpy.qm.mopac.MopacMolPM7* method), 209
- `get_threshold_rate_constants()`
(*rmgpy.solver.LiquidReactor* method), 244
- `get_threshold_rate_constants()`
(*rmgpy.solver.SimpleReactor* method), 242
- `get_threshold_rate_constants()`
(*rmgpy.solver.SurfaceReactor* method), 246
- `get_top_level_groups()`
(*rmgpy.data.kinetics.KineticsFamily* method), 47
- `get_total_bond_order()` (*rmgpy.molecule.Atom* method), 137
- `get_total_mass()` (*rmgpy.statmech.Conformer* method), 273
- `get_training_depository()`
(*rmgpy.data.kinetics.KineticsFamily* method), 47
- `get_training_set()` (*rmgpy.data.kinetics.KineticsFamily* method), 47
- `get_transport_data()` (*rmgpy.species.Species* method), 253
- `get_url()` (*rmgpy.data.kinetics.DepositoryReaction* method), 33
- `get_url()` (*rmgpy.data.kinetics.LibraryReaction* method), 61
- `get_url()` (*rmgpy.data.kinetics.TemplateReaction* method), 81
- `get_url()` (*rmgpy.molecule.Molecule* method), 148
- `get_url()` (*rmgpy.reaction.Reaction* method), 219
- `get_url()` (*rmgpy.rmg.pdep.PDepReaction* method), 236
- `Graph` (class in *rmgpy.molecule.graph*), 128
- `groundStateDegeneracy` (*rmgpy.qm.qmdata.QMData* attribute), 193
- `Group` (class in *rmgpy.molecule*), 156
- `GroupAtom` (class in *rmgpy.molecule*), 153
- `GroupBond` (class in *rmgpy.molecule*), 155
- `GroupFrequencies` (class in *rmgpy.data.statmech*), 36
- ## H
- `H0` (*rmgpy.thermo.Wilhoit* attribute), 278
- `H298` (*rmgpy.thermo.ThermoData* attribute), 274
- `harmonic_oscillator_d_heat_capacity_d_freq()`
(in module *rmgpy.data.statmechfit*), 69
- `harmonic_oscillator_heat_capacity()` (in module *rmgpy.data.statmechfit*), 69
- `HarmonicOscillator` (class in *rmgpy.statmech*), 265
- `has_atom()` (*rmgpy.molecule.Group* method), 160
- `has_atom()` (*rmgpy.molecule.Molecule* method), 148
- `has_bond()` (*rmgpy.molecule.Group* method), 160
- `has_bond()` (*rmgpy.molecule.Molecule* method), 148
- `has_edge()` (*rmgpy.molecule.graph.Graph* method), 130
- `has_edge()` (*rmgpy.molecule.Group* method), 160
- `has_edge()` (*rmgpy.molecule.Molecule* method), 148
- `has_halogen()` (*rmgpy.molecule.Molecule* method), 148
- `has_lone_pairs()` (*rmgpy.molecule.Molecule* method), 148
- `has_rate_rule()` (*rmgpy.data.kinetics.KineticsFamily* method), 47
- `has_reactive_molecule()` (*rmgpy.species.Species* method), 253
- `has_rule()` (*rmgpy.data.kinetics.KineticsRules* method), 56
- `has_statmech()` (*rmgpy.pdep.Configuration* method), 184
- `has_statmech()` (*rmgpy.species.Species* method), 253
- `has_template()` (*rmgpy.data.kinetics.DepositoryReaction* method), 33
- `has_template()` (*rmgpy.data.kinetics.LibraryReaction* method), 61
- `has_template()` (*rmgpy.data.kinetics.TemplateReaction* method), 81
- `has_template()` (*rmgpy.reaction.Reaction* method), 219
- `has_template()` (*rmgpy.rmg.pdep.PDepReaction* method), 236
- `has_thermo()` (*rmgpy.pdep.Configuration* method), 184
- `has_thermo()` (*rmgpy.species.Species* method), 253
- `has_vertex()` (*rmgpy.molecule.graph.Graph* method), 130
- `has_vertex()` (*rmgpy.molecule.Group* method), 160
- `has_vertex()` (*rmgpy.molecule.Molecule* method), 148
- `has_wildcards()` (*rmgpy.molecule.Group* method), 160
- `has_wildcards()` (*rmgpy.molecule.GroupAtom* method), 154

- highPlimit (*rmgpy.kinetics.Chebyshev* attribute), 109
 highPlimit (*rmgpy.kinetics.Lindemann* attribute), 113
 highPlimit (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 106
 highPlimit (*rmgpy.kinetics.PDepArrhenius* attribute), 104
 highPlimit (*rmgpy.kinetics.PDepKineticsData* attribute), 102
 highPlimit (*rmgpy.kinetics.ThirdBody* attribute), 111
 highPlimit (*rmgpy.kinetics.Troe* attribute), 117
 hindered_rotor_d_heat_capacity_d_barr() (in module *rmgpy.data.statmechfit*), 69
 hindered_rotor_d_heat_capacity_d_freq() (in module *rmgpy.data.statmechfit*), 69
 hindered_rotor_heat_capacity() (in module *rmgpy.data.statmechfit*), 69
 HinderedRotor (class in *rmgpy.statmech*), 267
 |
 IdealGasTranslation (class in *rmgpy.statmech*), 257
 identify_ring_membership() (*rmgpy.molecule.Molecule* method), 148
 ImplicitBenzeneError, 288
 inchi (*rmgpy.molecule.Molecule* attribute), 148
 inchi (*rmgpy.species.Species* attribute), 253
 InchiException, 288
 increment_charge() (*rmgpy.molecule.Atom* method), 137
 increment_lone_pairs() (*rmgpy.molecule.Atom* method), 138
 increment_order() (*rmgpy.molecule.Bond* method), 140
 increment_radical() (*rmgpy.molecule.Atom* method), 138
 inertia (*rmgpy.statmech.HinderedRotor* attribute), 269
 inertia (*rmgpy.statmech.KRotor* attribute), 263
 inertia (*rmgpy.statmech.LinearRotor* attribute), 260
 inertia (*rmgpy.statmech.NonlinearRotor* attribute), 261
 inertia (*rmgpy.statmech.SphericalTopRotor* attribute), 265
 initialize() (*arkane.PressureDependenceJob* method), 17
 initialize() (*rmgpy.data.statmechfit.DirectFit* method), 70
 initialize() (*rmgpy.data.statmechfit.PseudoFit* method), 72
 initialize() (*rmgpy.data.statmechfit.PseudoRotorFit* method), 71
 initialize() (*rmgpy.pdep.Network* method), 186
 initialize() (*rmgpy.qm.gaussian.GaussianMol* method), 196
 initialize() (*rmgpy.qm.gaussian.GaussianMolPM3* method), 198
 initialize() (*rmgpy.qm.gaussian.GaussianMolPM6* method), 200
 initialize() (*rmgpy.qm.main.QMCalculator* method), 191
 initialize() (*rmgpy.qm.molecule.QMMolecule* method), 193
 initialize() (*rmgpy.qm.mopac.MopacMol* method), 203
 initialize() (*rmgpy.qm.mopac.MopacMolPM3* method), 205
 initialize() (*rmgpy.qm.mopac.MopacMolPM6* method), 207
 initialize() (*rmgpy.qm.mopac.MopacMolPM7* method), 209
 initialize() (*rmgpy.rmg.main.RMG* method), 228
 initialize() (*rmgpy.rmg.pdep.PDepNetwork* method), 232
 initialize() (*rmgpy.solver.LiquidReactor* method), 244
 initialize() (*rmgpy.solver.MBSampledReactor* method), 249
 initialize() (*rmgpy.solver.ReactionSystem* method), 239
 initialize() (*rmgpy.solver.SimpleReactor* method), 242
 initialize() (*rmgpy.solver.SurfaceReactor* method), 246
 initialize_index_species_dict() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 225
 initialize_log() (in module *rmgpy.rmg.main*), 230
 initialize_model() (*rmgpy.solver.LiquidReactor* method), 244
 initialize_model() (*rmgpy.solver.MBSampledReactor* method), 249
 initialize_model() (*rmgpy.solver.ReactionSystem* method), 240
 initialize_model() (*rmgpy.solver.SimpleReactor* method), 242
 initialize_model() (*rmgpy.solver.SurfaceReactor* method), 246
 initialize_seed_mech() (*rmgpy.rmg.main.RMG* method), 228
 initialize_surface() (*rmgpy.solver.LiquidReactor* method), 244
 initialize_surface() (*rmgpy.solver.MBSampledReactor* method), 249
 initialize_surface() (*rmgpy.solver.ReactionSystem* method), 240
 initialize_surface() (*rmgpy.solver.SimpleReactor* method), 242
 initialize_surface() (*rmgpy.solver.SurfaceReactor* method), 246

[initiate_tolerances\(\)](#)
 ([rmgpy.solver.LiquidReactor](#) method), 244
[initiate_tolerances\(\)](#)
 ([rmgpy.solver.MBSampledReactor](#) method), 249
[initiate_tolerances\(\)](#)
 ([rmgpy.solver.ReactionSystem](#) method), 240
[initiate_tolerances\(\)](#)
 ([rmgpy.solver.SimpleReactor](#) method), 242
[initiate_tolerances\(\)](#)
 ([rmgpy.solver.SurfaceReactor](#) method), 247
[input_file_keywords\(\)](#)
 ([rmgpy.qm.gaussian.GaussianMol](#) method), 196
[input_file_keywords\(\)](#)
 ([rmgpy.qm.gaussian.GaussianMolPM3](#) method), 198
[input_file_keywords\(\)](#)
 ([rmgpy.qm.gaussian.GaussianMolPM6](#) method), 201
[input_file_keywords\(\)](#)
 ([rmgpy.qm.mopac.MopacMol](#) method), 203
[input_file_keywords\(\)](#)
 ([rmgpy.qm.mopac.MopacMolPM3](#) method), 205
[input_file_keywords\(\)](#)
 ([rmgpy.qm.mopac.MopacMolPM6](#) method), 207
[input_file_keywords\(\)](#)
 ([rmgpy.qm.mopac.MopacMolPM7](#) method), 209
[input_file_path](#) ([rmgpy.qm.gaussian.GaussianMol](#) property), 196
[input_file_path](#) ([rmgpy.qm.gaussian.GaussianMolPM3](#) property), 198
[input_file_path](#) ([rmgpy.qm.gaussian.GaussianMolPM6](#) property), 201
[input_file_path](#) ([rmgpy.qm.molecule.QMMolecule](#) property), 193
[input_file_path](#) ([rmgpy.qm.mopac.MopacMol](#) property), 203
[input_file_path](#) ([rmgpy.qm.mopac.MopacMolPM3](#) property), 205
[input_file_path](#) ([rmgpy.qm.mopac.MopacMolPM6](#) property), 207
[input_file_path](#) ([rmgpy.qm.mopac.MopacMolPM7](#) property), 209
[input_file_path](#) ([rmgpy.qm.symmetry.SymmetryJob](#) property), 195
[InputError](#), 288
[InvalidActionError](#), 288
[InvalidAdjacencyListError](#), 288
[invalidate\(\)](#) ([rmgpy.pdep.Network](#) method), 186
[invalidate\(\)](#) ([rmgpy.rmg.pdep.PDepNetwork](#) method), 232
[InvalidMicrocanonicalRateError](#), 288
[ionic_bond_filteration\(\)](#) (in module [rmgpy.molecule.filtration](#)), 169
[is_all_zeros\(\)](#) ([rmgpy.thermo.ThermoData](#) method), 275
[is_aromatic\(\)](#) ([rmgpy.molecule.Molecule](#) method), 149
[is_aromatic_ring\(\)](#) ([rmgpy.molecule.Group](#) method), 160
[is_aryl_radical\(\)](#) ([rmgpy.molecule.Molecule](#) method), 149
[is_association\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 34
[is_association\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 62
[is_association\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 82
[is_association\(\)](#) ([rmgpy.reaction.Reaction](#) method), 219
[is_association\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 236
[is_atom_able_to_gain_lone_pair\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 173
[is_atom_able_to_lose_lone_pair\(\)](#) (in module [rmgpy.molecule.pathfinder](#)), 173
[is_atom_in_cycle\(\)](#) ([rmgpy.molecule.Molecule](#) method), 149
[is_balanced\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 34
[is_balanced\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 62
[is_balanced\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 82
[is_balanced\(\)](#) ([rmgpy.reaction.Reaction](#) method), 219
[is_balanced\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 236
[is_benzene\(\)](#) ([rmgpy.molecule.Bond](#) method), 140
[is_benzene\(\)](#) ([rmgpy.molecule.GroupBond](#) method), 155
[is_benzene_explicit\(\)](#) ([rmgpy.molecule.Group](#) method), 161
[is_bimolecular\(\)](#) ([rmgpy.pdep.Configuration](#) method), 184
[is_bond_in_cycle\(\)](#) ([rmgpy.molecule.Molecule](#) method), 149
[is_bonded_to_halogen\(\)](#) ([rmgpy.molecule.Atom](#) method), 138
[is_bonded_to_surface\(\)](#) ([rmgpy.molecule.Atom](#) method), 138
[is_bonded_to_surface\(\)](#) ([rmgpy.molecule.GroupAtom](#) method), 154
[is_bromine\(\)](#) ([rmgpy.molecule.Atom](#) method), 138
[is_bromine\(\)](#) ([rmgpy.molecule.GroupAtom](#) method),

- 154
- `is_carbon()` (*rmgpy.molecule.Atom* method), 138
- `is_carbon()` (*rmgpy.molecule.GroupAtom* method), 154
- `is_charge_transfer_reaction()` (*rmgpy.data.kinetics.DepositoryReaction* method), 34
- `is_charge_transfer_reaction()` (*rmgpy.data.kinetics.LibraryReaction* method), 62
- `is_charge_transfer_reaction()` (*rmgpy.data.kinetics.TemplateReaction* method), 82
- `is_charge_transfer_reaction()` (*rmgpy.reaction.Reaction* method), 220
- `is_charge_transfer_reaction()` (*rmgpy.rmg.pdep.PDepReaction* method), 236
- `is_chlorine()` (*rmgpy.molecule.Atom* method), 138
- `is_chlorine()` (*rmgpy.molecule.GroupAtom* method), 154
- `is_cyclic()` (*rmgpy.molecule.graph.Graph* method), 130
- `is_cyclic()` (*rmgpy.molecule.Group* method), 161
- `is_cyclic()` (*rmgpy.molecule.Molecule* method), 149
- `is_dissociation()` (*rmgpy.data.kinetics.DepositoryReaction* method), 34
- `is_dissociation()` (*rmgpy.data.kinetics.LibraryReaction* method), 62
- `is_dissociation()` (*rmgpy.data.kinetics.TemplateReaction* method), 82
- `is_dissociation()` (*rmgpy.reaction.Reaction* method), 220
- `is_dissociation()` (*rmgpy.rmg.pdep.PDepReaction* method), 236
- `is_double()` (*rmgpy.molecule.Bond* method), 140
- `is_double()` (*rmgpy.molecule.GroupBond* method), 155
- `is_double_or_triple()` (*rmgpy.molecule.Bond* method), 140
- `is_edge_in_cycle()` (*rmgpy.molecule.graph.Graph* method), 130
- `is_edge_in_cycle()` (*rmgpy.molecule.Group* method), 161
- `is_edge_in_cycle()` (*rmgpy.molecule.Molecule* method), 149
- `is_electron()` (*rmgpy.molecule.Atom* method), 138
- `is_electron()` (*rmgpy.molecule.Group* method), 161
- `is_electron()` (*rmgpy.molecule.GroupAtom* method), 154
- `is_electron()` (*rmgpy.molecule.Molecule* method), 149
- `is_electron()` (*rmgpy.species.Species* method), 253
- `is_entry_match()` (*rmgpy.data.kinetics.KineticsFamily* method), 47
- `is_fluorine()` (*rmgpy.molecule.Atom* method), 138
- `is_fluorine()` (*rmgpy.molecule.GroupAtom* method), 154
- `is_halogen()` (*rmgpy.molecule.Atom* method), 138
- `is_heterocyclic()` (*rmgpy.molecule.Molecule* method), 149
- `is_hydrogen()` (*rmgpy.molecule.Atom* method), 138
- `is_hydrogen_bond()` (*rmgpy.molecule.Bond* method), 140
- `is_hydrogen_bond()` (*rmgpy.molecule.GroupBond* method), 155
- `is_identical()` (*rmgpy.molecule.Group* method), 161
- `is_identical()` (*rmgpy.molecule.Molecule* method), 149
- `is_identical()` (*rmgpy.species.Species* method), 253
- `is_identical_to()` (*rmgpy.kinetics.Arrhenius* method), 98
- `is_identical_to()` (*rmgpy.kinetics.Chebyshev* method), 109
- `is_identical_to()` (*rmgpy.kinetics.KineticsData* method), 96
- `is_identical_to()` (*rmgpy.kinetics.Lindemann* method), 114
- `is_identical_to()` (*rmgpy.kinetics.MultiArrhenius* method), 100
- `is_identical_to()` (*rmgpy.kinetics.MultiPDepArrhenius* method), 106
- `is_identical_to()` (*rmgpy.kinetics.PDepArrhenius* method), 104
- `is_identical_to()` (*rmgpy.kinetics.PDepKineticsData* method), 102
- `is_identical_to()` (*rmgpy.kinetics.StickingCoefficient* method), 124
- `is_identical_to()` (*rmgpy.kinetics.SurfaceArrhenius* method), 121
- `is_identical_to()` (*rmgpy.kinetics.ThirdBody* method), 111
- `is_identical_to()` (*rmgpy.kinetics.Troe* method), 117
- `is_identical_to()` (*rmgpy.thermo.NASA* method), 282
- `is_identical_to()` (*rmgpy.thermo.NASAPolynomial* method), 286
- `is_identical_to()` (*rmgpy.thermo.ThermoData* method), 276
- `is_identical_to()` (*rmgpy.thermo.Wilhoit* method), 279
- `is_iodine()` (*rmgpy.molecule.Atom* method), 138
- `is_isomerization()` (*rmgpy.data.kinetics.DepositoryReaction* method), 34
- `is_isomerization()` (*rmgpy.data.kinetics.LibraryReaction* method), 62
- `is_isomerization()` (*rmgpy.data.kinetics.TemplateReaction* method), 82

`is_isomerization()` (*rmgpy.reaction.Reaction method*), 220
`is_isomerization()` (*rmgpy.rmg.pdep.PDepReaction method*), 236
`is_isomorphic()` (*rmgpy.data.kinetics.DepositoryReaction method*), 34
`is_isomorphic()` (*rmgpy.data.kinetics.LibraryReaction method*), 62
`is_isomorphic()` (*rmgpy.data.kinetics.TemplateReaction method*), 82
`is_isomorphic()` (*rmgpy.molecule.graph.Graph method*), 130
`is_isomorphic()` (*rmgpy.molecule.Group method*), 161
`is_isomorphic()` (*rmgpy.molecule.Molecule method*), 149
`is_isomorphic()` (*rmgpy.molecule.vf2.VF2 method*), 131
`is_isomorphic()` (*rmgpy.reaction.Reaction method*), 220
`is_isomorphic()` (*rmgpy.rmg.pdep.PDepReaction method*), 237
`is_isomorphic()` (*rmgpy.species.Species method*), 253
`is_linear()` (*rmgpy.molecule.Molecule method*), 149
`is_lithium()` (*rmgpy.molecule.Atom method*), 138
`is_lithium()` (*rmgpy.molecule.GroupAtom method*), 154
`is_mapping_valid()` (*rmgpy.molecule.graph.Graph method*), 130
`is_mapping_valid()` (*rmgpy.molecule.Group method*), 161
`is_mapping_valid()` (*rmgpy.molecule.Molecule method*), 150
`is_molecule_forbidden()` (*rmgpy.data.kinetics.KineticsFamily method*), 47
`is_multidentate()` (*rmgpy.molecule.Molecule method*), 150
`is_nitrogen()` (*rmgpy.molecule.Atom method*), 138
`is_nitrogen()` (*rmgpy.molecule.GroupAtom method*), 154
`is_non_hydrogen()` (*rmgpy.molecule.Atom method*), 138
`is_nos()` (*rmgpy.molecule.Atom method*), 138
`is_order()` (*rmgpy.molecule.Bond method*), 140
`is_oxygen()` (*rmgpy.molecule.Atom method*), 138
`is_oxygen()` (*rmgpy.molecule.GroupAtom method*), 154
`is_pdep()` (*in module arkane.common*), 23
`is_phosphorus()` (*rmgpy.molecule.Atom method*), 138
`is_pressure_dependent()` (*rmgpy.kinetics.Arrhenius method*), 98
`is_pressure_dependent()` (*rmgpy.kinetics.Chebyshev method*), 109
`is_pressure_dependent()` (*rmgpy.kinetics.KineticsData method*), 96
`is_pressure_dependent()` (*rmgpy.kinetics.Lindemann method*), 114
`is_pressure_dependent()` (*rmgpy.kinetics.MultiArrhenius method*), 100
`is_pressure_dependent()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 106
`is_pressure_dependent()` (*rmgpy.kinetics.PDepArrhenius method*), 104
`is_pressure_dependent()` (*rmgpy.kinetics.PDepKineticsData method*), 102
`is_pressure_dependent()` (*rmgpy.kinetics.StickingCoefficient method*), 124
`is_pressure_dependent()` (*rmgpy.kinetics.SurfaceArrhenius method*), 121
`is_pressure_dependent()` (*rmgpy.kinetics.ThirdBody method*), 111
`is_pressure_dependent()` (*rmgpy.kinetics.Troe method*), 117
`is_pressure_valid()` (*rmgpy.kinetics.Chebyshev method*), 109
`is_pressure_valid()` (*rmgpy.kinetics.Lindemann method*), 114
`is_pressure_valid()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 106
`is_pressure_valid()` (*rmgpy.kinetics.PDepArrhenius method*), 104
`is_pressure_valid()` (*rmgpy.kinetics.PDepKineticsData method*), 102
`is_pressure_valid()` (*rmgpy.kinetics.ThirdBody method*), 111
`is_pressure_valid()` (*rmgpy.kinetics.Troe method*), 117
`is_proton()` (*rmgpy.molecule.Atom method*), 138
`is_proton()` (*rmgpy.molecule.Group method*), 161
`is_proton()` (*rmgpy.molecule.GroupAtom method*), 154
`is_proton()` (*rmgpy.molecule.Molecule method*), 150
`is_proton()` (*rmgpy.species.Species method*), 253
`is_quadruple()` (*rmgpy.molecule.Bond method*), 140
`is_quadruple()` (*rmgpy.molecule.GroupBond method*), 155
`is_radical()` (*rmgpy.molecule.Molecule method*), 150
`is_reaction_bond()` (*rmgpy.molecule.Bond method*),

- 140
- `is_reaction_bond()` (*rmgpy.molecule.GroupBond method*), 156
- `is_silicon()` (*rmgpy.molecule.Atom method*), 139
- `is_similar_to()` (*rmgpy.kinetics.Arrhenius method*), 98
- `is_similar_to()` (*rmgpy.kinetics.Chebyshev method*), 109
- `is_similar_to()` (*rmgpy.kinetics.KineticsData method*), 96
- `is_similar_to()` (*rmgpy.kinetics.Lindemann method*), 114
- `is_similar_to()` (*rmgpy.kinetics.MultiArrhenius method*), 100
- `is_similar_to()` (*rmgpy.kinetics.MultiPDepArrhenius method*), 106
- `is_similar_to()` (*rmgpy.kinetics.PDepArrhenius method*), 104
- `is_similar_to()` (*rmgpy.kinetics.PDepKineticsData method*), 102
- `is_similar_to()` (*rmgpy.kinetics.StickingCoefficient method*), 124
- `is_similar_to()` (*rmgpy.kinetics.SurfaceArrhenius method*), 121
- `is_similar_to()` (*rmgpy.kinetics.ThirdBody method*), 111
- `is_similar_to()` (*rmgpy.kinetics.Troe method*), 117
- `is_similar_to()` (*rmgpy.thermo.NASA method*), 282
- `is_similar_to()` (*rmgpy.thermo.NASAPolynomial method*), 286
- `is_similar_to()` (*rmgpy.thermo.ThermoData method*), 276
- `is_similar_to()` (*rmgpy.thermo.Wilhoit method*), 279
- `is_single()` (*rmgpy.molecule.Bond method*), 140
- `is_single()` (*rmgpy.molecule.GroupBond method*), 156
- `is_specific_case_of()` (*rmgpy.molecule.Atom method*), 139
- `is_specific_case_of()` (*rmgpy.molecule.AtomType method*), 133
- `is_specific_case_of()` (*rmgpy.molecule.Bond method*), 140
- `is_specific_case_of()` (*rmgpy.molecule.graph.Edge method*), 128
- `is_specific_case_of()` (*rmgpy.molecule.graph.Vertex method*), 127
- `is_specific_case_of()` (*rmgpy.molecule.GroupAtom method*), 154
- `is_specific_case_of()` (*rmgpy.molecule.GroupBond method*), 156
- `is_structure_in_list()` (*rmgpy.species.Species method*), 254
- `is_subgraph_isomorphic()` (*rmgpy.molecule.graph.Graph method*), 130
- `is_subgraph_isomorphic()` (*rmgpy.molecule.Group method*), 161
- `is_subgraph_isomorphic()` (*rmgpy.molecule.Molecule method*), 150
- `is_subgraph_isomorphic()` (*rmgpy.molecule.vf2.VF2 method*), 131
- `is_sulfur()` (*rmgpy.molecule.Atom method*), 139
- `is_sulfur()` (*rmgpy.molecule.GroupAtom method*), 154
- `is_surface_charge_transfer_reaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 34
- `is_surface_charge_transfer_reaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 62
- `is_surface_charge_transfer_reaction()` (*rmgpy.data.kinetics.TemplateReaction method*), 82
- `is_surface_charge_transfer_reaction()` (*rmgpy.reaction.Reaction method*), 220
- `is_surface_charge_transfer_reaction()` (*rmgpy.rmg.pdep.PDepReaction method*), 237
- `is_surface_reaction()` (*rmgpy.data.kinetics.DepositoryReaction method*), 34
- `is_surface_reaction()` (*rmgpy.data.kinetics.LibraryReaction method*), 62
- `is_surface_reaction()` (*rmgpy.data.kinetics.TemplateReaction method*), 82
- `is_surface_reaction()` (*rmgpy.reaction.Reaction method*), 220
- `is_surface_reaction()` (*rmgpy.rmg.pdep.PDepReaction method*), 237
- `is_surface_site()` (*rmgpy.molecule.Atom method*), 139
- `is_surface_site()` (*rmgpy.molecule.Group method*), 161
- `is_surface_site()` (*rmgpy.molecule.GroupAtom method*), 154
- `is_surface_site()` (*rmgpy.molecule.Molecule method*), 150
- `is_surface_site()` (*rmgpy.species.Species method*), 254
- `is_temperature_valid()` (*rmgpy.kinetics.Arrhenius method*), 98
- `is_temperature_valid()` (*rmgpy.kinetics.Chebyshev method*), 109
- `is_temperature_valid()` (*rmgpy.kinetics.KineticsData method*), 96
- `is_temperature_valid()`

- (*rmgpy.kinetics.Lindemann* method), 114
- is_temperature_valid()* (*rmgpy.kinetics.MultiArrhenius* method), 100
- is_temperature_valid()* (*rmgpy.kinetics.MultiPDepArrhenius* method), 107
- is_temperature_valid()* (*rmgpy.kinetics.PDepArrhenius* method), 104
- is_temperature_valid()* (*rmgpy.kinetics.PDepKineticsData* method), 102
- is_temperature_valid()* (*rmgpy.kinetics.StickingCoefficient* method), 124
- is_temperature_valid()* (*rmgpy.kinetics.SurfaceArrhenius* method), 121
- is_temperature_valid()* (*rmgpy.kinetics.ThirdBody* method), 111
- is_temperature_valid()* (*rmgpy.kinetics.Troe* method), 117
- is_temperature_valid()* (*rmgpy.thermo.NASA* method), 282
- is_temperature_valid()* (*rmgpy.thermo.NASAPolynomial* method), 286
- is_temperature_valid()* (*rmgpy.thermo.ThermoData* method), 276
- is_temperature_valid()* (*rmgpy.thermo.Wilhoit* method), 279
- is_termolecular()* (*rmgpy.pdep.Configuration* method), 184
- is_transition_state()* (*rmgpy.pdep.Configuration* method), 184
- is_triple()* (*rmgpy.molecule.Bond* method), 140
- is_triple()* (*rmgpy.molecule.GroupBond* method), 156
- is_uncertainty_additive()* (*rmgpy.quantity.ArrayQuantity* method), 214
- is_uncertainty_additive()* (*rmgpy.quantity.ScalarQuantity* method), 212
- is_uncertainty_multiplicative()* (*rmgpy.quantity.ArrayQuantity* method), 214
- is_uncertainty_multiplicative()* (*rmgpy.quantity.ScalarQuantity* method), 212
- is_unimolecular()* (*rmgpy.data.kinetics.DepositoryReaction* method), 34
- is_unimolecular()* (*rmgpy.data.kinetics.LibraryReaction* method), 62
- is_unimolecular()* (*rmgpy.data.kinetics.TemplateReaction* method), 82
- is_unimolecular()* (*rmgpy.pdep.Configuration* method), 184
- is_unimolecular()* (*rmgpy.reaction.Reaction* method), 220
- is_unimolecular()* (*rmgpy.rmg.pdep.PDepReaction* method), 237
- is_van_der_waals()* (*rmgpy.molecule.Bond* method), 140
- is_van_der_waals()* (*rmgpy.molecule.GroupBond* method), 156
- is_vertex_in_cycle()* (*rmgpy.molecule.graph.Graph* method), 130
- is_vertex_in_cycle()* (*rmgpy.molecule.Group* method), 161
- is_vertex_in_cycle()* (*rmgpy.molecule.Molecule* method), 150
- ## J
- jacobian()* (*rmgpy.solver.LiquidReactor* method), 244
- jacobian()* (*rmgpy.solver.SimpleReactor* method), 242

K

kdata (*rmgpy.kinetics.KineticsData* attribute), 96

kdata (*rmgpy.kinetics.PDepKineticsData* attribute), 102

KekulizationError, 289

kekulize() (in module *rmgpy.molecule.kekulize*), 168

kekulize() (*rmgpy.molecule.kekulize.AromaticRing* method), 168

kekulize() (*rmgpy.molecule.Molecule* method), 150

keywords (*rmgpy.qm.gaussian.GaussianMolPM3* attribute), 198

keywords (*rmgpy.qm.gaussian.GaussianMolPM6* attribute), 201

keywords (*rmgpy.qm.mopac.MopacMol* attribute), 203

keywords (*rmgpy.qm.mopac.MopacMolPM3* attribute), 205

keywords (*rmgpy.qm.mopac.MopacMolPM6* attribute), 207

keywords (*rmgpy.qm.mopac.MopacMolPM7* attribute), 209

KineticsData (class in *rmgpy.kinetics*), 96

KineticsDatabase (class in *rmgpy.data.kinetics*), 37

KineticsDepository (class in *rmgpy.data.kinetics*), 39

KineticsError, 289

KineticsFamily (class in *rmgpy.data.kinetics*), 41

KineticsGroups (class in *rmgpy.data.kinetics*), 50

KineticsJob (class in *arkane*), 13

KineticsLibrary (class in *rmgpy.data.kinetics*), 53

KineticsRules (class in *rmgpy.data.kinetics*), 55

KineticsSensitivity (class in *arkane.sensitivity*), 19
 KRotor (class in *rmgpy.statmech*), 262
 kunits (*rmgpy.kinetics.Chebyshev* attribute), 109

L

label (*rmgpy.thermo.NASA* attribute), 282
 label (*rmgpy.thermo.NASAPolynomial* attribute), 286
 label (*rmgpy.thermo.ThermoData* attribute), 276
 label (*rmgpy.thermo.Wilhoit* attribute), 279
 LibraryReaction (class in *rmgpy.data.kinetics*), 58
 Lindemann (class in *rmgpy.kinetics*), 112
 LinearRotor (class in *rmgpy.statmech*), 259
 LiquidReactor (class in *rmgpy.solver*), 243
 load() (*arkane.StatMechJob* method), 18
 load() (*rmgpy.data.base.Database* method), 29
 load() (*rmgpy.data.kinetics.KineticsDatabase* method), 38
 load() (*rmgpy.data.kinetics.KineticsDepository* method), 40
 load() (*rmgpy.data.kinetics.KineticsFamily* method), 47
 load() (*rmgpy.data.kinetics.KineticsGroups* method), 51
 load() (*rmgpy.data.kinetics.KineticsLibrary* method), 53
 load() (*rmgpy.data.kinetics.KineticsRules* method), 56
 load() (*rmgpy.data.statmech.StatmechDatabase* method), 65
 load() (*rmgpy.data.statmech.StatmechDepository* method), 67
 load() (*rmgpy.data.statmech.StatmechGroups* method), 74
 load() (*rmgpy.data.statmech.StatmechLibrary* method), 76
 load() (*rmgpy.data.thermo.ThermoDatabase* method), 86
 load() (*rmgpy.data.thermo.ThermoDepository* method), 88
 load() (*rmgpy.data.thermo.ThermoGroups* method), 90
 load() (*rmgpy.data.thermo.ThermoLibrary* method), 93
 load_chemkin_file() (in module *rmgpy.chemkin*), 24
 load_conformer() (*arkane.ess.ESSAdapter* method), 5
 load_conformer() (*arkane.ess.GaussianLog* method), 7
 load_conformer() (*arkane.ess.MolproLog* method), 8
 load_conformer() (*arkane.ess.OrcaLog* method), 9
 load_conformer() (*arkane.ess.QChemLog* method), 10
 load_conformer() (*arkane.ess.TeraChemLog* method), 12
 load_depository() (*rmgpy.data.statmech.StatmechDatabase* method), 66
 load_depository() (*rmgpy.data.thermo.ThermoDatabase* method), 86
 load_energy() (*arkane.ess.ESSAdapter* method), 5
 load_energy() (*arkane.ess.GaussianLog* method), 7
 load_energy() (*arkane.ess.MolproLog* method), 8
 load_energy() (*arkane.ess.OrcaLog* method), 9
 load_energy() (*arkane.ess.QChemLog* method), 11
 load_energy() (*arkane.ess.TeraChemLog* method), 12
 load_entry() (*rmgpy.data.kinetics.KineticsDepository* method), 40
 load_entry() (*rmgpy.data.kinetics.KineticsGroups* method), 51
 load_entry() (*rmgpy.data.kinetics.KineticsLibrary* method), 54
 load_entry() (*rmgpy.data.kinetics.KineticsRules* method), 56
 load_entry() (*rmgpy.data.statmech.StatmechDepository* method), 67
 load_entry() (*rmgpy.data.statmech.StatmechGroups* method), 74
 load_entry() (*rmgpy.data.statmech.StatmechLibrary* method), 76
 load_entry() (*rmgpy.data.thermo.ThermoDepository* method), 88
 load_entry() (*rmgpy.data.thermo.ThermoGroups* method), 90
 load_entry() (*rmgpy.data.thermo.ThermoLibrary* method), 93
 load_families() (*rmgpy.data.kinetics.KineticsDatabase* method), 38
 load_forbidden() (*rmgpy.data.kinetics.KineticsFamily* method), 48
 load_force_constant_matrix() (*arkane.ess.ESSAdapter* method), 5
 load_force_constant_matrix() (*arkane.ess.GaussianLog* method), 7
 load_force_constant_matrix() (*arkane.ess.MolproLog* method), 8
 load_force_constant_matrix() (*arkane.ess.OrcaLog* method), 9
 load_force_constant_matrix() (*arkane.ess.QChemLog* method), 11
 load_force_constant_matrix() (*arkane.ess.TeraChemLog* method), 12
 load_geometry() (*arkane.ess.ESSAdapter* method), 5
 load_geometry() (*arkane.ess.GaussianLog* method), 7
 load_geometry() (*arkane.ess.MolproLog* method), 8
 load_geometry() (*arkane.ess.OrcaLog* method), 10
 load_geometry() (*arkane.ess.QChemLog* method), 11
 load_geometry() (*arkane.ess.TeraChemLog* method), 12
 load_groups() (*rmgpy.data.statmech.StatmechDatabase* method), 66
 load_groups() (*rmgpy.data.thermo.ThermoDatabase* method), 86
 load_input() (*rmgpy.rmg.main.RMG* method), 229
 load_input_file() (*arkane.Arkane* method), 14

<code>load_input_file()</code> (in module <code>arkane.input</code>), 13	<code>(rmgpy.data.kinetics.KineticsFamily method)</code> , 48
<code>load_libraries()</code> (<code>rmgpy.data.kinetics.KineticsDatabase</code> method), 38	<code>load_old_dictionary()</code>
<code>load_libraries()</code> (<code>rmgpy.data.statmech.StatmechDatabase</code> method), 66	<code>(rmgpy.data.kinetics.KineticsGroups method)</code> , 51
<code>load_libraries()</code> (<code>rmgpy.data.thermo.ThermoDatabase</code> method), 86	<code>load_old_dictionary()</code>
<code>load_negative_frequency()</code> (<code>arkane.ess.ESSAdapter</code> method), 6	<code>(rmgpy.data.kinetics.KineticsLibrary method)</code> , 54
<code>load_negative_frequency()</code> (<code>arkane.ess.GaussianLog</code> method), 7	<code>load_old_dictionary()</code>
<code>load_negative_frequency()</code> (<code>arkane.ess.MolproLog</code> method), 8	<code>(rmgpy.data.kinetics.KineticsRules method)</code> , 57
<code>load_negative_frequency()</code> (<code>arkane.ess.OrcaLog</code> method), 10	<code>load_old_dictionary()</code>
<code>load_negative_frequency()</code> (<code>arkane.ess.QChemLog</code> method), 11	<code>(rmgpy.data.statmech.StatmechDepository method)</code> , 67
<code>load_negative_frequency()</code> (<code>arkane.ess.TeraChemLog</code> method), 12	<code>load_old_dictionary()</code>
<code>load_old()</code> (<code>rmgpy.data.base.Database</code> method), 29	<code>(rmgpy.data.statmech.StatmechGroups method)</code> , 74
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsDatabase</code> method), 38	<code>load_old_dictionary()</code>
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsDepository</code> method), 40	<code>(rmgpy.data.statmech.StatmechLibrary method)</code> , 77
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsFamily</code> method), 48	<code>load_old_dictionary()</code>
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsGroups</code> method), 51	<code>(rmgpy.data.thermo.ThermoDepository method)</code> , 88
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsLibrary</code> method), 54	<code>load_old_dictionary()</code>
<code>load_old()</code> (<code>rmgpy.data.kinetics.KineticsRules</code> method), 57	<code>(rmgpy.data.thermo.ThermoGroups method)</code> , 90
<code>load_old()</code> (<code>rmgpy.data.statmech.StatmechDatabase</code> method), 66	<code>load_old_dictionary()</code>
<code>load_old()</code> (<code>rmgpy.data.statmech.StatmechDepository</code> method), 67	<code>(rmgpy.data.thermo.ThermoLibrary method)</code> , 93
<code>load_old()</code> (<code>rmgpy.data.statmech.StatmechGroups</code> method), 74	<code>load_old_library()</code> (<code>rmgpy.data.base.Database</code> method), 29
<code>load_old()</code> (<code>rmgpy.data.statmech.StatmechLibrary</code> method), 76	<code>load_old_library()</code> (<code>rmgpy.data.kinetics.KineticsDepository</code> method), 40
<code>load_old()</code> (<code>rmgpy.data.thermo.ThermoDatabase</code> method), 86	<code>load_old_library()</code> (<code>rmgpy.data.kinetics.KineticsFamily</code> method), 48
<code>load_old()</code> (<code>rmgpy.data.thermo.ThermoDepository</code> method), 88	<code>load_old_library()</code> (<code>rmgpy.data.kinetics.KineticsGroups</code> method), 51
<code>load_old()</code> (<code>rmgpy.data.thermo.ThermoGroups</code> method), 90	<code>load_old_library()</code> (<code>rmgpy.data.kinetics.KineticsLibrary</code> method), 54
<code>load_old()</code> (<code>rmgpy.data.thermo.ThermoLibrary</code> method), 93	<code>load_old_library()</code> (<code>rmgpy.data.kinetics.KineticsRules</code> method), 57
<code>load_old_dictionary()</code> (<code>rmgpy.data.base.Database</code> method), 29	<code>load_old_library()</code> (<code>rmgpy.data.statmech.StatmechDepository</code> method), 67
<code>load_old_dictionary()</code> (<code>rmgpy.data.kinetics.KineticsDepository</code> method), 40	<code>load_old_library()</code> (<code>rmgpy.data.statmech.StatmechGroups</code> method), 74
<code>load_old_dictionary()</code>	<code>load_old_library()</code> (<code>rmgpy.data.statmech.StatmechLibrary</code> method), 77
	<code>load_old_library()</code> (<code>rmgpy.data.thermo.ThermoDepository</code> method), 88
	<code>load_old_library()</code> (<code>rmgpy.data.thermo.ThermoGroups</code> method), 91
	<code>load_old_library()</code> (<code>rmgpy.data.thermo.ThermoLibrary</code> method), 93
	<code>load_old_tree()</code> (<code>rmgpy.data.base.Database</code> method), 29

`load_old_tree()` (*rmgpy.data.kinetics.KineticsDepository* method), 8
`load_old_tree()` (*rmgpy.data.kinetics.KineticsFamily* method), 40
`load_old_tree()` (*rmgpy.data.kinetics.KineticsGroups* method), 48
`load_old_tree()` (*rmgpy.data.kinetics.KineticsLibrary* method), 51
`load_old_tree()` (*rmgpy.data.kinetics.KineticsRules* method), 54
`load_old_tree()` (*rmgpy.data.kinetics.KineticsRules* method), 57
`load_old_tree()` (*rmgpy.data.statmech.StatmechDepository* method), 67
`load_old_tree()` (*rmgpy.data.statmech.StatmechGroups* method), 74
`load_old_tree()` (*rmgpy.data.statmech.StatmechLibrary* method), 77
`load_old_tree()` (*rmgpy.data.thermo.ThermoDepository* method), 88
`load_old_tree()` (*rmgpy.data.thermo.ThermoGroups* method), 91
`load_old_tree()` (*rmgpy.data.thermo.ThermoLibrary* method), 93
`load_recipe()` (*rmgpy.data.kinetics.KineticsFamily* method), 48
`load_recommended_families()` (*rmgpy.data.kinetics.KineticsDatabase* method), 38
`load_scan_energies()` (*arkane.ess.ESSAdapter* method), 6
`load_scan_energies()` (*arkane.ess.GaussianLog* method), 7
`load_scan_energies()` (*arkane.ess.MolproLog* method), 9
`load_scan_energies()` (*arkane.ess.OrcaLog* method), 10
`load_scan_energies()` (*arkane.ess.QChemLog* method), 11
`load_scan_energies()` (*arkane.ess.TeraChemLog* method), 12
`load_scan_frozen_atoms()` (*arkane.ess.ESSAdapter* method), 6
`load_scan_frozen_atoms()` (*arkane.ess.GaussianLog* method), 7
`load_scan_frozen_atoms()` (*arkane.ess.MolproLog* method), 9
`load_scan_frozen_atoms()` (*arkane.ess.OrcaLog* method), 10
`load_scan_frozen_atoms()` (*arkane.ess.QChemLog* method), 11
`load_scan_frozen_atoms()` (*arkane.ess.TeraChemLog* method), 12
`load_scan_pivot_atoms()` (*arkane.ess.ESSAdapter* method), 6
`load_scan_pivot_atoms()` (*arkane.ess.GaussianLog* method), 8
`load_scan_pivot_atoms()` (*arkane.ess.MolproLog* method), 9
`load_scan_pivot_atoms()` (*arkane.ess.OrcaLog* method), 10
`load_scan_pivot_atoms()` (*arkane.ess.QChemLog* method), 11
`load_scan_pivot_atoms()` (*arkane.ess.TeraChemLog* method), 12
`load_species_dictionary()` (*in module rmgpy.chemkin*), 24
`load_surface()` (*rmgpy.data.thermo.ThermoDatabase* method), 86
`load_template()` (*rmgpy.data.kinetics.KineticsFamily* method), 48
`load_thermo_data()` (*rmgpy.qm.gaussian.GaussianMol* method), 196
`load_thermo_data()` (*rmgpy.qm.gaussian.GaussianMolPM3* method), 199
`load_thermo_data()` (*rmgpy.qm.gaussian.GaussianMolPM6* method), 201
`load_thermo_data()` (*rmgpy.qm.molecule.QMMolecule* method), 193
`load_thermo_data()` (*rmgpy.qm.mopac.MopacMol* method), 204
`load_thermo_data()` (*rmgpy.qm.mopac.MopacMolPM3* method), 206
`load_thermo_data()` (*rmgpy.qm.mopac.MopacMolPM6* method), 208
`load_thermo_data()` (*rmgpy.qm.mopac.MopacMolPM7* method), 210
`load_thermo_input()` (*rmgpy.rmg.main.RMG* method), 229
`load_transport_file()` (*in module rmgpy.chemkin*), 24
`load_yaml()` (*arkane.common.ArkaneSpecies* method), 21
`load_zero_point_energy()` (*arkane.ess.ESSAdapter* method), 6
`load_zero_point_energy()` (*arkane.ess.GaussianLog* method), 8
`load_zero_point_energy()` (*arkane.ess.MolproLog* method), 9
`load_zero_point_energy()` (*arkane.ess.OrcaLog* method), 10
`load_zero_point_energy()` (*arkane.ess.QChemLog* method), 11
`load_zero_point_energy()` (*arkane.ess.TeraChemLog* method), 12
`log_conversions()` (*rmgpy.solver.LiquidReactor* method), 245
`log_conversions()` (*rmgpy.solver.MBSampledReactor* method), 249
`log_conversions()` (*rmgpy.solver.ReactionSystem* method), 8

- method), 240
- log_conversions() (rmgpy.solver.SimpleReactor method), 242
- log_conversions() (rmgpy.solver.SurfaceReactor method), 247
- log_enlarge_summary() (rmgpy.rmg.model.CoreEdgeReactionModel method), 225
- log_header() (rmgpy.rmg.main.RMG method), 229
- log_initial_conditions() (rmgpy.solver.SurfaceReactor method), 247
- log_rates() (rmgpy.solver.LiquidReactor method), 245
- log_rates() (rmgpy.solver.MBSampledReactor method), 249
- log_rates() (rmgpy.solver.ReactionSystem method), 240
- log_rates() (rmgpy.solver.SimpleReactor method), 242
- log_rates() (rmgpy.solver.SurfaceReactor method), 247
- log_summary() (rmgpy.pdep.Network method), 186
- log_summary() (rmgpy.rmg.pdep.PDepNetwork method), 232
- LogicAnd (class in rmgpy.data.base), 64
- LogicNode (class in rmgpy.data.base), 64
- LogicOr (class in rmgpy.data.base), 64
- ## M
- make_bond() (rmgpy.molecule.GroupBond method), 156
- make_logic_node() (in module rmgpy.data.base), 64
- make_new_pdep_reaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 225
- make_new_reaction() (rmgpy.rmg.model.CoreEdgeReactionModel method), 225
- make_new_species() (rmgpy.rmg.model.CoreEdgeReactionModel method), 225
- make_object() (arkane.common.ArkaneSpecies method), 21
- make_object() (rmgpy.pdep.SingleExponentialDown method), 181
- make_object() (rmgpy.quantity.ArrayQuantity method), 214
- make_object() (rmgpy.quantity.ScalarQuantity method), 212
- make_object() (rmgpy.statmech.Conformer method), 273
- make_object() (rmgpy.statmech.HarmonicOscillator method), 266
- make_object() (rmgpy.statmech.HinderedRotor method), 269
- make_object() (rmgpy.statmech.IdealGasTranslation method), 258
- make_object() (rmgpy.statmech.KRotor method), 263
- make_object() (rmgpy.statmech.LinearRotor method), 260
- make_object() (rmgpy.statmech.NonlinearRotor method), 261
- make_object() (rmgpy.statmech.SphericalTopRotor method), 265
- make_object() (rmgpy.thermo.NASA method), 282
- make_object() (rmgpy.thermo.NASAPolynomial method), 286
- make_object() (rmgpy.thermo.ThermoData method), 276
- make_object() (rmgpy.thermo.Wilhoit method), 279
- make_profile_graph() (in module rmgpy.rmg.main), 230
- make_sample_atom() (rmgpy.molecule.GroupAtom method), 154
- make_sample_molecule() (rmgpy.molecule.Group method), 161
- make_seed_mech() (rmgpy.rmg.main.RMG method), 229
- make_species_labels_independent() (rmgpy.rmg.main.RMG method), 229
- make_tree() (rmgpy.data.kinetics.KineticsFamily method), 48
- map_densities_of_states() (rmgpy.pdep.Network method), 186
- map_densities_of_states() (rmgpy.rmg.pdep.PDepNetwork method), 232
- map_density_of_states() (rmgpy.pdep.Configuration method), 184
- map_sum_of_states() (rmgpy.pdep.Configuration method), 184
- mark_chemkin_duplicates() (rmgpy.rmg.model.CoreEdgeReactionModel method), 225
- mark_duplicate_reactions() (in module rmgpy.chemkin), 26
- mark_unreactive_structures() (in module rmgpy.molecule.filtration), 169
- mark_valid_duplicates() (rmgpy.data.kinetics.KineticsLibrary method), 54
- mass (rmgpy.statmech.Conformer attribute), 273
- mass (rmgpy.statmech.IdealGasTranslation attribute), 258
- match_logic_or() (rmgpy.data.base.LogicOr method), 64
- match_node_to_child() (rmgpy.data.base.Database method), 29
- match_node_to_child()

<code>(rmgpy.data.kinetics.KineticsDepository method), 40</code>	<code>(rmgpy.data.statmech.StatmechLibrary method), 77</code>
<code>match_node_to_child() (rmgpy.data.kinetics.KineticsFamily method), 48</code>	<code>match_node_to_node() (rmgpy.data.thermo.ThermoDepository method), 88</code>
<code>match_node_to_child() (rmgpy.data.kinetics.KineticsGroups method), 52</code>	<code>match_node_to_node() (rmgpy.data.thermo.ThermoGroups method), 91</code>
<code>match_node_to_child() (rmgpy.data.kinetics.KineticsLibrary method), 54</code>	<code>match_node_to_node() (rmgpy.data.thermo.ThermoLibrary method), 93</code>
<code>match_node_to_child() (rmgpy.data.kinetics.KineticsRules method), 57</code>	<code>match_node_to_structure() (rmgpy.data.base.Database method), 29</code>
<code>match_node_to_child() (rmgpy.data.statmech.StatmechDepository method), 67</code>	<code>match_node_to_structure() (rmgpy.data.kinetics.KineticsDepository method), 40</code>
<code>match_node_to_child() (rmgpy.data.statmech.StatmechGroups method), 74</code>	<code>match_node_to_structure() (rmgpy.data.kinetics.KineticsFamily method), 48</code>
<code>match_node_to_child() (rmgpy.data.statmech.StatmechLibrary method), 77</code>	<code>match_node_to_structure() (rmgpy.data.kinetics.KineticsGroups method), 52</code>
<code>match_node_to_child() (rmgpy.data.thermo.ThermoDepository method), 88</code>	<code>match_node_to_structure() (rmgpy.data.kinetics.KineticsLibrary method), 54</code>
<code>match_node_to_child() (rmgpy.data.thermo.ThermoGroups method), 91</code>	<code>match_node_to_structure() (rmgpy.data.kinetics.KineticsRules method), 57</code>
<code>match_node_to_child() (rmgpy.data.thermo.ThermoLibrary method), 93</code>	<code>match_node_to_structure() (rmgpy.data.statmech.StatmechDepository method), 67</code>
<code>match_node_to_node() (rmgpy.data.base.Database method), 29</code>	<code>match_node_to_structure() (rmgpy.data.statmech.StatmechGroups method), 75</code>
<code>match_node_to_node() (rmgpy.data.kinetics.KineticsDepository method), 40</code>	<code>match_node_to_structure() (rmgpy.data.statmech.StatmechLibrary method), 77</code>
<code>match_node_to_node() (rmgpy.data.kinetics.KineticsFamily method), 48</code>	<code>match_node_to_structure() (rmgpy.data.thermo.ThermoDepository method), 88</code>
<code>match_node_to_node() (rmgpy.data.kinetics.KineticsGroups method), 52</code>	<code>match_node_to_structure() (rmgpy.data.thermo.ThermoGroups method), 91</code>
<code>match_node_to_node() (rmgpy.data.kinetics.KineticsLibrary method), 54</code>	<code>match_node_to_structure() (rmgpy.data.thermo.ThermoLibrary method), 93</code>
<code>match_node_to_node() (rmgpy.data.kinetics.KineticsRules method), 57</code>	<code>match_to_structure() (rmgpy.data.base.LogicAnd method), 64</code>
<code>match_node_to_node() (rmgpy.data.statmech.StatmechDepository method), 67</code>	<code>match_to_structure() (rmgpy.data.base.LogicOr method), 64</code>
<code>match_node_to_node() (rmgpy.data.statmech.StatmechGroups method), 74</code>	<code>matches_species() (rmgpy.data.kinetics.DepositoryReaction method), 34</code>
<code>match_node_to_node()</code>	<code>matches_species() (rmgpy.data.kinetics.LibraryReaction method), 62</code>
	<code>matches_species() (rmgpy.data.kinetics.TemplateReaction</code>

method), 82
`matches_species()` (*rmgpy.reaction.Reaction method*), 220
`matches_species()` (*rmgpy.rmg.pdep.PDepReaction method*), 237
`max_attempts` (*rmgpy.qm.gaussian.GaussianMol property*), 196
`max_attempts` (*rmgpy.qm.gaussian.GaussianMolPM3 property*), 199
`max_attempts` (*rmgpy.qm.gaussian.GaussianMolPM6 property*), 201
`max_attempts` (*rmgpy.qm.molecule.QMMolecule property*), 193
`max_attempts` (*rmgpy.qm.mopac.MopacMol property*), 204
`max_attempts` (*rmgpy.qm.mopac.MopacMolPM3 property*), 206
`max_attempts` (*rmgpy.qm.mopac.MopacMolPM6 property*), 208
`max_attempts` (*rmgpy.qm.mopac.MopacMolPM7 property*), 210
`maximum_grain_size` (*arkane.PressureDependenceJob property*), 17
`MBSampledReactor` (*class in rmgpy.solver*), 248
`merge()` (*rmgpy.molecule.graph.Graph method*), 130
`merge()` (*rmgpy.molecule.Group method*), 161
`merge()` (*rmgpy.molecule.Molecule method*), 150
`merge()` (*rmgpy.rmg.model.ReactionModel method*), 227
`merge()` (*rmgpy.rmg.pdep.PDepNetwork method*), 232
`merge_groups()` (*rmgpy.molecule.Group method*), 161
`modes` (*rmgpy.statmech.Conformer attribute*), 273
`ModifiedStrongCollisionError`, 289
`module`
 `arkane`, 3
 `arkane.common`, 20
 `arkane.output`, 14
 `arkane.sensitivity`, 19
 `rmgpy.chemkin`, 23
 `rmgpy.constants`, 26
 `rmgpy.data`, 27
 `rmgpy.exceptions`, 287
 `rmgpy.kinetics`, 95
 `rmgpy.molecule`, 125
 `rmgpy.molecule.adjlist`, 175
 `rmgpy.molecule.converter`, 173
 `rmgpy.molecule.filtration`, 168
 `rmgpy.molecule.kekulize`, 167
 `rmgpy.molecule.pathfinder`, 170
 `rmgpy.molecule.resonance`, 164
 `rmgpy.molecule.translator`, 173
 `rmgpy.pdep`, 179
 `rmgpy.qm`, 189
 `rmgpy.quantity`, 210

`rmgpy.reaction`, 215
`rmgpy.rmg`, 221
`rmgpy.solver`, 238
`rmgpy.species`, 250
`rmgpy.statmech`, 255
`rmgpy.statmech.schrodinger`, 270
`rmgpy.thermo`, 274
`molecular_weight` (*rmgpy.species.Species attribute*), 254
`Molecule` (*class in rmgpy.molecule*), 141
`MoleculeDrawer` (*class in rmgpy.molecule.draw*), 178
`MolproLog` (*class in arkane.ess*), 8
`Mopac` (*class in rmgpy.qm.mopac*), 202
`MopacMol` (*class in rmgpy.qm.mopac*), 202
`MopacMolPM3` (*class in rmgpy.qm.mopac*), 204
`MopacMolPM6` (*class in rmgpy.qm.mopac*), 206
`MopacMolPM7` (*class in rmgpy.qm.mopac*), 208
`MultiArrhenius` (*class in rmgpy.kinetics*), 99
`MultiPDepArrhenius` (*class in rmgpy.kinetics*), 105
`multiplicity` (*rmgpy.species.Species attribute*), 254

N

`n` (*rmgpy.kinetics.Arrhenius attribute*), 98
`n` (*rmgpy.kinetics.StickingCoefficient attribute*), 124
`n` (*rmgpy.kinetics.SurfaceArrhenius attribute*), 121
`n` (*rmgpy.pdep.SingleExponentialDown attribute*), 181
`NASA` (*class in rmgpy.thermo*), 280
`NASAPolynomial` (*class in rmgpy.thermo*), 284
`NegativeBarrierException`, 289
`Network` (*class in rmgpy.pdep*), 184
`NetworkError`, 289
`NonlinearRotor` (*class in rmgpy.statmech*), 260
`number` (*rmgpy.statmech.Conformer attribute*), 273
`number_of_surface_sites()` (*rmgpy.molecule.Molecule method*), 150
`number_of_surface_sites()` (*rmgpy.species.Species method*), 254
`numberOfAtoms` (*rmgpy.qm.qmdata.QMData attribute*), 193

O

`octet_filtration()` (*in module rmgpy.molecule.filtration*), 170
`optical_isomers` (*rmgpy.statmech.Conformer attribute*), 273
`OrcaLog` (*class in arkane.ess*), 9
`output_file_path` (*rmgpy.qm.gaussian.GaussianMol property*), 197
`output_file_path` (*rmgpy.qm.gaussian.GaussianMolPM3 property*), 199
`output_file_path` (*rmgpy.qm.gaussian.GaussianMolPM6 property*), 201
`output_file_path` (*rmgpy.qm.molecule.QMMolecule property*), 193

output_file_path (rmgpy.qm.mopac.MopacMol property), 204
 output_file_path (rmgpy.qm.mopac.MopacMolPM3 property), 206
 output_file_path (rmgpy.qm.mopac.MopacMolPM6 property), 208
 output_file_path (rmgpy.qm.mopac.MopacMolPM7 property), 210
 OutputError, 289

P

parse() (rmgpy.qm.gaussian.Gaussian method), 195
 parse() (rmgpy.qm.gaussian.GaussianMol method), 197
 parse() (rmgpy.qm.gaussian.GaussianMolPM3 method), 199
 parse() (rmgpy.qm.gaussian.GaussianMolPM6 method), 201
 parse() (rmgpy.qm.molecule.QMMolecule method), 193
 parse() (rmgpy.qm.mopac.MopacMol method), 204
 parse() (rmgpy.qm.mopac.MopacMolPM3 method), 206
 parse() (rmgpy.qm.mopac.MopacMolPM6 method), 208
 parse() (rmgpy.qm.mopac.MopacMolPM7 method), 210
 parse() (rmgpy.qm.symmetry.SymmetryJob method), 195
 parse_command_line_arguments() (arkane.Arkane method), 14
 parse_old_library() (rmgpy.data.base.Database method), 30
 parse_old_library() (rmgpy.data.kinetics.KineticsDepository method), 41
 parse_old_library() (rmgpy.data.kinetics.KineticsFamily method), 49
 parse_old_library() (rmgpy.data.kinetics.KineticsGroups method), 52
 parse_old_library() (rmgpy.data.kinetics.KineticsLibrary method), 55
 parse_old_library() (rmgpy.data.kinetics.KineticsRules method), 57
 parse_old_library() (rmgpy.data.statmech.StatmechDepository method), 68
 parse_old_library() (rmgpy.data.statmech.StatmechGroups method), 75
 parse_old_library() (rmgpy.data.statmech.StatmechLibrary method), 77
 parse_old_library() (rmgpy.data.thermo.ThermoDepository method), 89
 parse_old_library() (rmgpy.data.thermo.ThermoGroups method), 91
 parse_old_library() (rmgpy.data.thermo.ThermoLibrary method), 94
 Pdata (rmgpy.kinetics.PDepKineticsData attribute), 101
 PDepArrhenius (class in rmgpy.kinetics), 103
 PDepKineticsData (class in rmgpy.kinetics), 101
 PDepNetwork (class in rmgpy.rmg.pdep), 230
 PDepReaction (class in rmgpy.rmg.pdep), 233
 PDepSensitivity (class in arkane.sensitivity), 20
 perturb() (arkane.sensitivity.KineticsSensitivity method), 19
 perturb() (arkane.sensitivity.PDepSensitivity method), 20
 pick_wildcards() (rmgpy.molecule.Group method), 162
 Plist (arkane.PressureDependenceJob property), 16
 plot() (arkane.KineticsJob method), 13
 plot() (arkane.PressureDependenceJob method), 17
 plot() (arkane.sensitivity.KineticsSensitivity method), 19
 plot() (arkane.sensitivity.PDepSensitivity method), 20
 plot() (arkane.ThermoJob method), 18
 Pmax (arkane.PressureDependenceJob property), 16
 Pmax (rmgpy.kinetics.Arrhenius attribute), 97
 Pmax (rmgpy.kinetics.Chebyshev attribute), 108
 Pmax (rmgpy.kinetics.KineticsData attribute), 96
 Pmax (rmgpy.kinetics.Lindemann attribute), 112
 Pmax (rmgpy.kinetics.MultiArrhenius attribute), 99
 Pmax (rmgpy.kinetics.MultiPDepArrhenius attribute), 105
 Pmax (rmgpy.kinetics.PDepArrhenius attribute), 103
 Pmax (rmgpy.kinetics.PDepKineticsData attribute), 101
 Pmax (rmgpy.kinetics.StickingCoefficient attribute), 123
 Pmax (rmgpy.kinetics.SurfaceArrhenius attribute), 120
 Pmax (rmgpy.kinetics.ThirdBody attribute), 110
 Pmax (rmgpy.kinetics.Troe attribute), 115
 Pmin (arkane.PressureDependenceJob property), 16
 Pmin (rmgpy.kinetics.Arrhenius attribute), 97
 Pmin (rmgpy.kinetics.Chebyshev attribute), 108
 Pmin (rmgpy.kinetics.KineticsData attribute), 96
 Pmin (rmgpy.kinetics.Lindemann attribute), 113
 Pmin (rmgpy.kinetics.MultiArrhenius attribute), 99
 Pmin (rmgpy.kinetics.MultiPDepArrhenius attribute), 105
 Pmin (rmgpy.kinetics.PDepArrhenius attribute), 103
 Pmin (rmgpy.kinetics.PDepKineticsData attribute), 101

- Pmin (*rmgpy.kinetics.StickingCoefficient* attribute), 123
 Pmin (*rmgpy.kinetics.SurfaceArrhenius* attribute), 120
 Pmin (*rmgpy.kinetics.ThirdBody* attribute), 110
 Pmin (*rmgpy.kinetics.Troe* attribute), 115
 PointGroup (class in *rmgpy.qm.symmetry*), 194
 PointGroupCalculator (class in *rmgpy.qm.symmetry*), 194
 poly1 (*rmgpy.thermo.NASA* attribute), 282
 poly2 (*rmgpy.thermo.NASA* attribute), 282
 poly3 (*rmgpy.thermo.NASA* attribute), 282
 polynomials (*rmgpy.thermo.NASA* attribute), 282
 populate_resonance_algorithms() (in module *rmgpy.molecule.resonance*), 167
 PressureDependenceError, 289
 PressureDependenceJob (class in *arkane*), 16
 pressures (*rmgpy.kinetics.PDepArrhenius* attribute), 105
 prettify() (in module *arkane.output*), 15
 PrettifyVisitor (class in *arkane.output*), 14
 prioritize_thermo() (*rmgpy.data.thermo.ThermoDatabase* method), 86
 process_bonds() (*rmgpy.molecule.kekulize.AromaticRing* method), 168
 process_coverage_dependence() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 225
 process_new_reactions() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 225
 process_old_library_entry() (*rmgpy.data.statmech.StatmechGroups* method), 75
 process_old_library_entry() (*rmgpy.data.statmech.StatmechLibrary* method), 77
 process_old_library_entry() (*rmgpy.data.thermo.ThermoGroups* method), 91
 process_old_library_entry() (*rmgpy.data.thermo.ThermoLibrary* method), 94
 process_pdep_networks() (*rmgpy.rmg.main.RMG* method), 229
 process_profile_stats() (in module *rmgpy.rmg.main*), 230
 process_reactions_to_species() (*rmgpy.rmg.main.RMG* method), 229
 process_to_species_networks() (*rmgpy.rmg.main.RMG* method), 229
 protons (*rmgpy.data.kinetics.DepositoryReaction* attribute), 35
 protons (*rmgpy.data.kinetics.LibraryReaction* attribute), 63
 protons (*rmgpy.data.kinetics.TemplateReaction* attribute), 83
 protons (*rmgpy.reaction.Reaction* attribute), 220
 protons (*rmgpy.rmg.pdep.PDepReaction* attribute), 237
 prune() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 226
 prune_heteroatoms() (*rmgpy.data.thermo.ThermoDatabase* method), 86
 prune_tree() (*rmgpy.data.kinetics.KineticsFamily* method), 49
 PseudoFit (class in *rmgpy.data.statmechfit*), 72
 PseudoRotorFit (class in *rmgpy.data.statmechfit*), 71
- ## Q
- QChemLog (class in *arkane.ess*), 10
 QMCalculator (class in *rmgpy.qm.main*), 190
 QMData (class in *rmgpy.qm.qmdata*), 193
 QMMolecule (class in *rmgpy.qm.molecule*), 192
 QMSettings (class in *rmgpy.qm.main*), 190
 QMVerifier (class in *rmgpy.qm.qmverifier*), 194
 Quantity() (in module *rmgpy.quantity*), 214
 QuantityError, 289
 quantum (*rmgpy.statmech.HarmonicOscillator* attribute), 267
 quantum (*rmgpy.statmech.HinderedRotor* attribute), 269
 quantum (*rmgpy.statmech.IdealGasTranslation* attribute), 258
 quantum (*rmgpy.statmech.KRotor* attribute), 263
 quantum (*rmgpy.statmech.LinearRotor* attribute), 260
 quantum (*rmgpy.statmech.NonlinearRotor* attribute), 261
 quantum (*rmgpy.statmech.SphericalTopRotor* attribute), 265
- ## R
- rd_build() (*rmgpy.qm.molecule.Geometry* method), 191
 rd_embed() (*rmgpy.qm.molecule.Geometry* method), 191
 react_init_tuples() (*rmgpy.rmg.main.RMG* method), 229
 react_molecules() (*rmgpy.data.kinetics.KineticsDatabase* method), 38
 Reaction (class in *rmgpy.reaction*), 215
 ReactionDrawer (class in *rmgpy.molecule.draw*), 179
 ReactionError, 290
 ReactionModel (class in *rmgpy.rmg.model*), 227
 ReactionPairsError, 290
 ReactionRecipe (class in *rmgpy.data.kinetics*), 64
 ReactionSystem (class in *rmgpy.solver*), 239
 read_input_file() (in module *rmgpy.rmg.input*), 227
 read_kinetics_entry() (in module *rmgpy.chemkin*), 24

`read_reaction_comments()` (in module `rmgpy.chemkin`), 24
`read_reactions_block()` (in module `rmgpy.chemkin`), 24
`read_thermo_entry()` (in module `rmgpy.chemkin`), 24
`reconstruct_kinetics_from_source()` (`rmgpy.data.kinetics.KineticsDatabase` method), 38
`record_polycyclic_generic_nodes()` (`rmgpy.data.thermo.ThermoDatabase` method), 86
`record_ring_generic_nodes()` (`rmgpy.data.thermo.ThermoDatabase` method), 86
`register_ess_adapter()` (in module `arkane.ess.factory`), 6
`register_listeners()` (`rmgpy.rmg.main.RMG` method), 229
`register_reaction()` (`rmgpy.rmg.model.CoreEdgeReactionModel` method), 226
`regularize()` (`rmgpy.data.kinetics.KineticsFamily` method), 49
`remove_atom()` (`rmgpy.molecule.Group` method), 162
`remove_atom()` (`rmgpy.molecule.Molecule` method), 150
`remove_bond()` (`rmgpy.molecule.Group` method), 162
`remove_bond()` (`rmgpy.molecule.Molecule` method), 150
`remove_comment_from_line()` (in module `rmgpy.chemkin`), 25
`remove_disconnected_reactions()` (`rmgpy.rmg.pdep.PDepNetwork` method), 232
`remove_edge()` (`rmgpy.molecule.graph.Graph` method), 131
`remove_edge()` (`rmgpy.molecule.Group` method), 162
`remove_edge()` (`rmgpy.molecule.Molecule` method), 150
`remove_empty_pdep_networks()` (`rmgpy.rmg.model.CoreEdgeReactionModel` method), 226
`remove_group()` (`rmgpy.data.base.Database` method), 30
`remove_group()` (`rmgpy.data.kinetics.KineticsDepository` method), 41
`remove_group()` (`rmgpy.data.kinetics.KineticsFamily` method), 49
`remove_group()` (`rmgpy.data.kinetics.KineticsGroups` method), 52
`remove_group()` (`rmgpy.data.kinetics.KineticsLibrary` method), 55
`remove_group()` (`rmgpy.data.kinetics.KineticsRules` method), 58
`remove_group()` (`rmgpy.data.statmech.StatmechDepository` method), 68
`remove_group()` (`rmgpy.data.statmech.StatmechGroups` method), 75
`remove_group()` (`rmgpy.data.statmech.StatmechLibrary` method), 77
`remove_group()` (`rmgpy.data.thermo.ThermoDepository` method), 89
`remove_group()` (`rmgpy.data.thermo.ThermoGroups` method), 91
`remove_group()` (`rmgpy.data.thermo.ThermoLibrary` method), 94
`remove_h_bonds()` (`rmgpy.molecule.Molecule` method), 150
`remove_reactions()` (`rmgpy.rmg.pdep.PDepNetwork` method), 232
`remove_species_from_edge()` (`rmgpy.rmg.model.CoreEdgeReactionModel` method), 226
`remove_van_der_waals_bonds()` (`rmgpy.molecule.Group` method), 162
`remove_van_der_waals_bonds()` (`rmgpy.molecule.Molecule` method), 150
`remove_vertex()` (`rmgpy.molecule.graph.Graph` method), 131
`remove_vertex()` (`rmgpy.molecule.Group` method), 162
`remove_vertex()` (`rmgpy.molecule.Molecule` method), 150
`render()` (`rmgpy.molecule.draw.MoleculeDrawer` method), 178
`replace_halogen_with_hydrogen()` (`rmgpy.molecule.Molecule` method), 151
`replace_yaml_syntax()` (in module `arkane.common`), 23
ReservoirStateError, 290
`reset_connectivity_values()` (`rmgpy.molecule.Atom` method), 139
`reset_connectivity_values()` (`rmgpy.molecule.graph.Graph` method), 131
`reset_connectivity_values()` (`rmgpy.molecule.graph.Vertex` method), 127
`reset_connectivity_values()` (`rmgpy.molecule.Group` method), 162
`reset_connectivity_values()` (`rmgpy.molecule.GroupAtom` method), 155
`reset_connectivity_values()` (`rmgpy.molecule.Molecule` method), 151
`reset_max_edge_species_rate_ratios()` (`rmgpy.solver.LiquidReactor` method), 245
`reset_max_edge_species_rate_ratios()` (`rmgpy.solver.MBSampledReactor` method), 249

`reset_max_edge_species_rate_ratios()`
 (*rmgpy.solver.ReactionSystem* method), 240
`reset_max_edge_species_rate_ratios()`
 (*rmgpy.solver.SimpleReactor* method), 242
`reset_max_edge_species_rate_ratios()`
 (*rmgpy.solver.SurfaceReactor* method), 247
`reset_ring_membership()` (*rmgpy.molecule.Group*
 method), 162
`residual()` (*rmgpy.solver.LiquidReactor* method), 245
`residual()` (*rmgpy.solver.MBSampledReactor*
 method), 249
`residual()` (*rmgpy.solver.ReactionSystem* method), 240
`residual()` (*rmgpy.solver.SimpleReactor* method), 242
`residual()` (*rmgpy.solver.SurfaceReactor* method), 247
`ResonanceError`, 290
`restore_vertex_order()`
 (*rmgpy.molecule.graph.Graph* method),
 131
`restore_vertex_order()` (*rmgpy.molecule.Group*
 method), 162
`restore_vertex_order()` (*rmgpy.molecule.Molecule*
 method), 151
`retrieve()` (*rmgpy.rmg.model.CoreEdgeReactionModel*
 method), 226
`retrieve_original_entry()`
 (*rmgpy.data.kinetics.KineticsFamily* method),
 49
`retrieve_template()`
 (*rmgpy.data.kinetics.KineticsFamily* method),
 49
`reverse_arrhenius_charge_transfer_rate()`
 (*rmgpy.data.kinetics.DepositoryReaction*
 method), 35
`reverse_arrhenius_charge_transfer_rate()`
 (*rmgpy.data.kinetics.LibraryReaction* method),
 63
`reverse_arrhenius_charge_transfer_rate()`
 (*rmgpy.data.kinetics.TemplateReaction*
 method), 83
`reverse_arrhenius_charge_transfer_rate()`
 (*rmgpy.reaction.Reaction* method), 221
`reverse_arrhenius_charge_transfer_rate()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 237
`reverse_arrhenius_rate()`
 (*rmgpy.data.kinetics.DepositoryReaction*
 method), 35
`reverse_arrhenius_rate()`
 (*rmgpy.data.kinetics.LibraryReaction* method),
 63
`reverse_arrhenius_rate()`
 (*rmgpy.data.kinetics.TemplateReaction*
 method), 83
`reverse_arrhenius_rate()`
 (*rmgpy.reaction.Reaction* method), 221
`reverse_arrhenius_rate()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 238
`reverse_sticking_coeff_rate()`
 (*rmgpy.data.kinetics.DepositoryReaction*
 method), 35
`reverse_sticking_coeff_rate()`
 (*rmgpy.data.kinetics.LibraryReaction* method),
 63
`reverse_sticking_coeff_rate()`
 (*rmgpy.data.kinetics.TemplateReaction*
 method), 83
`reverse_sticking_coeff_rate()`
 (*rmgpy.reaction.Reaction* method), 221
`reverse_sticking_coeff_rate()`
 (*rmgpy.rmg.pdep.PDepReaction* method),
 237
`reverse_surface_arrhenius_rate()`
 (*rmgpy.data.kinetics.DepositoryReaction*
 method), 35
`reverse_surface_arrhenius_rate()`
 (*rmgpy.data.kinetics.LibraryReaction* method),
 63
`reverse_surface_arrhenius_rate()`
 (*rmgpy.data.kinetics.TemplateReaction*
 method), 83
`reverse_surface_arrhenius_rate()`
 (*rmgpy.reaction.Reaction* method), 221
`reverse_surface_arrhenius_rate()`
 (*rmgpy.rmg.pdep.PDepReaction* method),
 238
`reverse_surface_charge_transfer_rate()`
 (*rmgpy.data.kinetics.DepositoryReaction*
 method), 35
`reverse_surface_charge_transfer_rate()`
 (*rmgpy.data.kinetics.LibraryReaction* method),
 63
`reverse_surface_charge_transfer_rate()`
 (*rmgpy.data.kinetics.TemplateReaction*
 method), 83
`reverse_surface_charge_transfer_rate()`
 (*rmgpy.reaction.Reaction* method), 221
`reverse_surface_charge_transfer_rate()`
 (*rmgpy.rmg.pdep.PDepReaction* method), 238
`RMG` (class in *rmgpy.rmg.main*), 227
`rmgpy.chemkin`
 module, 23
`rmgpy.constants`
 module, 26
`rmgpy.data`
 module, 27
`rmgpy.exceptions`
 module, 287
`rmgpy.kinetics`

module, 95
 rmgpy.molecule
 module, 125
 rmgpy.molecule.adjlist
 module, 175
 rmgpy.molecule.converter
 module, 173
 rmgpy.molecule.filtration
 module, 168
 rmgpy.molecule.kekulize
 module, 167
 rmgpy.molecule.pathfinder
 module, 170
 rmgpy.molecule.resonance
 module, 164
 rmgpy.molecule.translator
 module, 173
 rmgpy.pdep
 module, 179
 rmgpy.qm
 module, 189
 rmgpy.quantity
 module, 210
 rmgpy.reaction
 module, 215
 rmgpy.rmg
 module, 221
 rmgpy.solver
 module, 238
 rmgpy.species
 module, 250
 rmgpy.statmech
 module, 255
 rmgpy.statmech.schrodinger
 module, 270
 rmgpy.thermo
 module, 274
 rotationalConstant (rmgpy.statmech.HinderedRotor
 attribute), 269
 rotationalConstant (rmgpy.statmech.KRotor at-
 tribute), 263
 rotationalConstant (rmgpy.statmech.LinearRotor
 attribute), 260
 rotationalConstant (rmgpy.statmech.NonlinearRotor
 attribute), 261
 rotationalConstant (rmgpy.statmech.SphericalTopRotor
 attribute), 265
 run() (rmgpy.qm.symmetry.SymmetryJob method), 195
 run_jobs() (rmgpy.qm.main.QMCalculator method),
 191
 run_model_analysis() (rmgpy.rmg.main.RMG
 method), 229
 run_uncertainty_analysis()
 (rmgpy.rmg.main.RMG method), 229

S

S0 (rmgpy.thermo.Wilhoit attribute), 278
 S298 (rmgpy.thermo.ThermoData attribute), 274
 saturate_radicals() (rmgpy.molecule.Molecule
 method), 151
 saturate_unfilled_valence()
 (rmgpy.molecule.Molecule method), 151
 save() (arkane.PressureDependenceJob method), 17
 save() (arkane.sensitivity.KineticsSensitivity method),
 19
 save() (arkane.sensitivity.PDepSensitivity method), 20
 save() (rmgpy.data.base.Database method), 30
 save() (rmgpy.data.kinetics.KineticsDatabase method),
 38
 save() (rmgpy.data.kinetics.KineticsDepository
 method), 41
 save() (rmgpy.data.kinetics.KineticsFamily method), 49
 save() (rmgpy.data.kinetics.KineticsGroups method),
 52
 save() (rmgpy.data.kinetics.KineticsLibrary method),
 55
 save() (rmgpy.data.kinetics.KineticsRules method), 58
 save() (rmgpy.data.statmech.StatmechDatabase
 method), 66
 save() (rmgpy.data.statmech.StatmechDepository
 method), 68
 save() (rmgpy.data.statmech.StatmechGroups method),
 75
 save() (rmgpy.data.statmech.StatmechLibrary method),
 78
 save() (rmgpy.data.thermo.ThermoDatabase method),
 86
 save() (rmgpy.data.thermo.ThermoDepository method),
 89
 save() (rmgpy.data.thermo.ThermoGroups method), 91
 save() (rmgpy.data.thermo.ThermoLibrary method), 94
 save_chemkin_file() (in module rmgpy.chemkin), 25
 save_coordinates_from_qm_data()
 (rmgpy.qm.molecule.Geometry method),
 191
 save_depository() (rmgpy.data.kinetics.KineticsFamily
 method), 49
 save_depository() (rmgpy.data.statmech.StatmechDatabase
 method), 66
 save_depository() (rmgpy.data.thermo.ThermoDatabase
 method), 87
 save_dictionary() (rmgpy.data.base.Database
 method), 30
 save_dictionary() (rmgpy.data.kinetics.KineticsDepository
 method), 41
 save_dictionary() (rmgpy.data.kinetics.KineticsFamily
 method), 49
 save_dictionary() (rmgpy.data.kinetics.KineticsGroups
 method), 52

[save_dictionary\(\) \(rmgpy.data.kinetics.KineticsLibrary method\), 17](#)
[save_dictionary\(\) \(rmgpy.data.kinetics.KineticsRules method\), 58](#)
[save_dictionary\(\) \(rmgpy.data.statmech.StatmechDepository method\), 39](#)
[save_dictionary\(\) \(rmgpy.data.statmech.StatmechGroups method\), 66](#)
[save_dictionary\(\) \(rmgpy.data.statmech.StatmechLibrary method\), 87](#)
[save_dictionary\(\) \(rmgpy.data.thermo.ThermoDepository method\), 89](#)
[save_dictionary\(\) \(rmgpy.data.thermo.ThermoGroups method\), 91](#)
[save_dictionary\(\) \(rmgpy.data.thermo.ThermoLibrary method\), 94](#)
[save_diff_html\(\) \(in module rmgpy.rmg.output\), 230](#)
[save_entry\(\) \(rmgpy.data.kinetics.KineticsDepository method\), 41](#)
[save_entry\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 49](#)
[save_entry\(\) \(rmgpy.data.kinetics.KineticsLibrary method\), 55](#)
[save_entry\(\) \(rmgpy.data.kinetics.KineticsRules method\), 58](#)
[save_entry\(\) \(rmgpy.data.statmech.StatmechDepository method\), 68](#)
[save_entry\(\) \(rmgpy.data.statmech.StatmechGroups method\), 75](#)
[save_entry\(\) \(rmgpy.data.statmech.StatmechLibrary method\), 78](#)
[save_entry\(\) \(rmgpy.data.thermo.ThermoDepository method\), 89](#)
[save_entry\(\) \(rmgpy.data.thermo.ThermoGroups method\), 92](#)
[save_entry\(\) \(rmgpy.data.thermo.ThermoLibrary method\), 94](#)
[save_everything\(\) \(rmgpy.rmg.main.RMG method\), 229](#)
[save_families\(\) \(rmgpy.data.kinetics.KineticsDatabase method\), 39](#)
[save_generated_tree\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 49](#)
[save_groups\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 50](#)
[save_groups\(\) \(rmgpy.data.statmech.StatmechDatabase method\), 66](#)
[save_groups\(\) \(rmgpy.data.thermo.ThermoDatabase method\), 87](#)
[save_hindered_rotor_figures\(\) \(arkane.StatMechJob method\), 18](#)
[save_input\(\) \(rmgpy.rmg.main.RMG method\), 230](#)
[save_input_file\(\) \(arkane.PressureDependenceJob method\), 17](#)
[save_input_file\(\) \(in module rmgpy.rmg.input\), 227](#)
[save_kinetics_lib\(\) \(in module arkane.output\), 15](#)
[save_libraries\(\) \(rmgpy.data.kinetics.KineticsDatabase method\), 39](#)
[save_libraries\(\) \(rmgpy.data.statmech.StatmechDatabase method\), 66](#)
[save_libraries\(\) \(rmgpy.data.thermo.ThermoDatabase method\), 87](#)
[save_old\(\) \(rmgpy.data.base.Database method\), 30](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsDatabase method\), 39](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsDepository method\), 41](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 50](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsGroups method\), 52](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsLibrary method\), 55](#)
[save_old\(\) \(rmgpy.data.kinetics.KineticsRules method\), 58](#)
[save_old\(\) \(rmgpy.data.statmech.StatmechDatabase method\), 66](#)
[save_old\(\) \(rmgpy.data.statmech.StatmechDepository method\), 68](#)
[save_old\(\) \(rmgpy.data.statmech.StatmechGroups method\), 75](#)
[save_old\(\) \(rmgpy.data.statmech.StatmechLibrary method\), 78](#)
[save_old\(\) \(rmgpy.data.thermo.ThermoDatabase method\), 87](#)
[save_old\(\) \(rmgpy.data.thermo.ThermoDepository method\), 89](#)
[save_old\(\) \(rmgpy.data.thermo.ThermoGroups method\), 92](#)
[save_old\(\) \(rmgpy.data.thermo.ThermoLibrary method\), 94](#)
[save_old_dictionary\(\) \(rmgpy.data.base.Database method\), 30](#)
[save_old_dictionary\(\) \(rmgpy.data.kinetics.KineticsDepository method\), 41](#)
[save_old_dictionary\(\) \(rmgpy.data.kinetics.KineticsFamily method\), 50](#)
[save_old_dictionary\(\) \(rmgpy.data.kinetics.KineticsGroups method\), 52](#)
[save_old_dictionary\(\) \(rmgpy.data.kinetics.KineticsLibrary method\), 55](#)
[save_old_dictionary\(\) \(rmgpy.data.kinetics.KineticsRules method\), 58](#)

```

save_old_dictionary()
    (rmgpy.data.statmech.StatmechDepository
     method), 68
save_old_dictionary()
    (rmgpy.data.statmech.StatmechGroups
     method), 75
save_old_dictionary()
    (rmgpy.data.statmech.StatmechLibrary
     method), 78
save_old_dictionary()
    (rmgpy.data.thermo.ThermoDepository
     method), 89
save_old_dictionary()
    (rmgpy.data.thermo.ThermoGroups
     method), 92
save_old_dictionary()
    (rmgpy.data.thermo.ThermoLibrary
     method), 94
save_old_library()
    (rmgpy.data.base.Database
     method), 30
save_old_library()
    (rmgpy.data.kinetics.KineticsDepository
     method), 41
save_old_library()
    (rmgpy.data.kinetics.KineticsFamily
     method), 50
save_old_library()
    (rmgpy.data.kinetics.KineticsGroups
     method), 52
save_old_library()
    (rmgpy.data.kinetics.KineticsLibrary
     method), 55
save_old_library()
    (rmgpy.data.kinetics.KineticsRules
     method), 58
save_old_library()
    (rmgpy.data.statmech.StatmechDepository
     method), 68
save_old_library()
    (rmgpy.data.statmech.StatmechGroups
     method), 75
save_old_library()
    (rmgpy.data.statmech.StatmechLibrary
     method), 78
save_old_library()
    (rmgpy.data.thermo.ThermoDepository
     method), 89
save_old_library()
    (rmgpy.data.thermo.ThermoGroups
     method), 92
save_old_library()
    (rmgpy.data.thermo.ThermoLibrary
     method), 94
save_old_tree()
    (rmgpy.data.base.Database
     method), 30
save_old_tree()
    (rmgpy.data.kinetics.KineticsDepository
     method), 41
save_old_tree()
    (rmgpy.data.kinetics.KineticsFamily
     method), 50
save_old_tree()
    (rmgpy.data.kinetics.KineticsGroups
     method), 53
save_old_tree()
    (rmgpy.data.kinetics.KineticsLibrary
     method), 55
save_old_tree()
    (rmgpy.data.kinetics.KineticsRules
     method), 58
save_old_tree()
    (rmgpy.data.statmech.StatmechDepository
     method), 68
save_old_tree()
    (rmgpy.data.statmech.StatmechGroups
     method), 75
save_old_tree()
    (rmgpy.data.statmech.StatmechLibrary
     method), 78
save_old_tree()
    (rmgpy.data.thermo.ThermoDepository
     method), 89
save_old_tree()
    (rmgpy.data.thermo.ThermoGroups
     method), 92
save_old_tree()
    (rmgpy.data.thermo.ThermoLibrary
     method), 94
save_output_html()
    (in module rmgpy.rmg.output),
    230
save_recommended_families()
    (rmgpy.data.kinetics.KineticsDatabase
     method), 39
save_species_dictionary()
    (in module
     rmgpy.chemkin), 25
save_surface()
    (rmgpy.data.thermo.ThermoDatabase
     method), 87
save_thermo_data()
    (rmgpy.qm.gaussian.GaussianMol
     method), 197
save_thermo_data()
    (rmgpy.qm.gaussian.GaussianMolPM3
     method), 199
save_thermo_data()
    (rmgpy.qm.gaussian.GaussianMolPM6
     method), 201
save_thermo_data()
    (rmgpy.qm.gaussian.GaussianMolPM6
     method), 201
save_thermo_data()
    (rmgpy.qm.molecule.QMMolecule
     method), 193
save_thermo_data()
    (rmgpy.qm.mopac.MopacMol
     method), 204
save_thermo_data()
    (rmgpy.qm.mopac.MopacMolPM3
     method), 206
save_thermo_data()
    (rmgpy.qm.mopac.MopacMolPM6
     method), 208
save_thermo_data()
    (rmgpy.qm.mopac.MopacMolPM7
     method), 210
save_thermo_lib()
    (in module arkane.output), 15
save_training_reactions()
    (rmgpy.data.kinetics.KineticsFamily
     method),
    50
save_transport_file()
    (in module rmgpy.chemkin),
    25
save_yaml()
    (arkane.common.ArkaneSpecies
     method),
    21
save_yaml()
    (arkane.KineticsJob
     method), 13
ScalarQuantity
    (class in rmgpy.quantity), 212
script_attempts
    (rmgpy.qm.gaussian.GaussianMol
     property), 197
script_attempts
    (rmgpy.qm.gaussian.GaussianMolPM3
     property), 199
script_attempts
    (rmgpy.qm.gaussian.GaussianMolPM6
     property), 201
script_attempts
    (rmgpy.qm.molecule.QMMolecule
     property), 201

```


- property*), 193
- `script_attempts` (*rmgpy.qm.mopac.MopacMol property*), 204
- `script_attempts` (*rmgpy.qm.mopac.MopacMolPM3 property*), 206
- `script_attempts` (*rmgpy.qm.mopac.MopacMolPM6 property*), 208
- `script_attempts` (*rmgpy.qm.mopac.MopacMolPM7 property*), 210
- `search_retrieve_reactions()`
(*rmgpy.rmg.model.CoreEdgeReactionModel method*), 226
- `select_energy_grains()` (*rmgpy.pdep.Network method*), 186
- `select_energy_grains()`
(*rmgpy.rmg.pdep.PDepNetwork method*), 232
- `select_polynomial()` (*rmgpy.thermo.NASA method*), 282
- `semiclassical` (*rmgpy.statmech.HinderedRotor attribute*), 269
- `set_binding_energies()`
(*rmgpy.data.thermo.ThermoDatabase method*), 87
- `set_cantera_kinetics()` (*rmgpy.kinetics.Arrhenius method*), 98
- `set_cantera_kinetics()` (*rmgpy.kinetics.Chebyshev method*), 109
- `set_cantera_kinetics()`
(*rmgpy.kinetics.KineticsData method*), 97
- `set_cantera_kinetics()`
(*rmgpy.kinetics.Lindemann method*), 114
- `set_cantera_kinetics()`
(*rmgpy.kinetics.MultiArrhenius method*), 100
- `set_cantera_kinetics()`
(*rmgpy.kinetics.MultiPDepArrhenius method*), 107
- `set_cantera_kinetics()`
(*rmgpy.kinetics.PDepArrhenius method*), 105
- `set_cantera_kinetics()`
(*rmgpy.kinetics.PDepKineticsData method*), 102
- `set_cantera_kinetics()`
(*rmgpy.kinetics.StickingCoefficient method*), 124
- `set_cantera_kinetics()`
(*rmgpy.kinetics.SurfaceArrhenius method*), 122
- `set_cantera_kinetics()` (*rmgpy.kinetics.ThirdBody method*), 112
- `set_cantera_kinetics()` (*rmgpy.kinetics.Troe method*), 117
- `set_colliders()` (*rmgpy.solver.MBSampledReactor method*), 249
- `set_colliders()` (*rmgpy.solver.SimpleReactor method*), 242
- `set_conditions()` (*rmgpy.pdep.Network method*), 186
- `set_conditions()` (*rmgpy.rmg.pdep.PDepNetwork method*), 232
- `set_default_output_directory()`
(*rmgpy.qm.main.QMCalculator method*), 191
- `set_e0_with_thermo()` (*rmgpy.species.Species method*), 254
- `set_initial_conditions()`
(*rmgpy.solver.LiquidReactor method*), 245
- `set_initial_conditions()`
(*rmgpy.solver.MBSampledReactor method*), 249
- `set_initial_conditions()`
(*rmgpy.solver.ReactionSystem method*), 240
- `set_initial_conditions()`
(*rmgpy.solver.SimpleReactor method*), 243
- `set_initial_conditions()`
(*rmgpy.solver.SurfaceReactor method*), 247
- `set_initial_derivative()`
(*rmgpy.solver.LiquidReactor method*), 245
- `set_initial_derivative()`
(*rmgpy.solver.MBSampledReactor method*), 250
- `set_initial_derivative()`
(*rmgpy.solver.ReactionSystem method*), 240
- `set_initial_derivative()`
(*rmgpy.solver.SimpleReactor method*), 243
- `set_initial_derivative()`
(*rmgpy.solver.SurfaceReactor method*), 247
- `set_lone_pairs()` (*rmgpy.molecule.Atom method*), 139
- `set_order_num()` (*rmgpy.molecule.Bond method*), 140
- `set_order_num()` (*rmgpy.molecule.GroupBond method*), 156
- `set_order_str()` (*rmgpy.molecule.Bond method*), 141
- `set_order_str()` (*rmgpy.molecule.GroupBond method*), 156
- `set_reference_potential()`
(*rmgpy.data.kinetics.DepositoryReaction method*), 35
- `set_reference_potential()`
(*rmgpy.data.kinetics.LibraryReaction method*), 63
- `set_reference_potential()`
(*rmgpy.data.kinetics.TemplateReaction method*), 83
- `set_reference_potential()`
(*rmgpy.reaction.Reaction method*), 221
- `set_reference_potential()`

- (*rmgpy.rmg.pdep.PDepReaction* method), 238
- set_structure() (*rmgpy.species.Species* method), 254
- set_thermodynamic_filtering_parameters() (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 226
- SettingsError, 290
- simple_regularization() (*rmgpy.data.kinetics.KineticsFamily* method), 50
- SimpleReactor (class in *rmgpy.solver*), 241
- simulate() (*rmgpy.solver.LiquidReactor* method), 245
- simulate() (*rmgpy.solver.MBSampledReactor* method), 250
- simulate() (*rmgpy.solver.ReactionSystem* method), 240
- simulate() (*rmgpy.solver.SimpleReactor* method), 243
- simulate() (*rmgpy.solver.SurfaceReactor* method), 247
- SingleExponentialDown (class in *rmgpy.pdep*), 180
- smiles (*rmgpy.molecule.Molecule* attribute), 151
- smiles (*rmgpy.species.Species* attribute), 254
- solute (*rmgpy.kinetics.Arrhenius* attribute), 99
- solute (*rmgpy.kinetics.Chebyshev* attribute), 110
- solute (*rmgpy.kinetics.KineticsData* attribute), 97
- solute (*rmgpy.kinetics.Lindemann* attribute), 114
- solute (*rmgpy.kinetics.MultiArrhenius* attribute), 100
- solute (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 107
- solute (*rmgpy.kinetics.PDepArrhenius* attribute), 105
- solute (*rmgpy.kinetics.PDepKineticsData* attribute), 102
- solute (*rmgpy.kinetics.StickingCoefficient* attribute), 125
- solute (*rmgpy.kinetics.SurfaceArrhenius* attribute), 122
- solute (*rmgpy.kinetics.ThirdBody* attribute), 112
- solute (*rmgpy.kinetics.Troe* attribute), 117
- solve() (*rmgpy.data.statmechfit.DirectFit* method), 70
- solve() (*rmgpy.data.statmechfit.PseudoFit* method), 73
- solve() (*rmgpy.data.statmechfit.PseudoRotorFit* method), 71
- solve_full_me() (*rmgpy.pdep.Network* method), 186
- solve_full_me() (*rmgpy.rmg.pdep.PDepNetwork* method), 232
- solve_reduced_me() (*rmgpy.pdep.Network* method), 187
- solve_reduced_me() (*rmgpy.rmg.pdep.PDepNetwork* method), 233
- solve_schrodinger_equation() (*rmgpy.statmech.HinderedRotor* method), 269
- solve_ss_network() (*rmgpy.rmg.pdep.PDepNetwork* method), 233
- sort_atoms() (*rmgpy.molecule.Group* method), 162
- sort_atoms() (*rmgpy.molecule.Molecule* method), 151
- sort_by_connectivity() (*rmgpy.molecule.Group* method), 162
- sort_cyclic_vertices() (*rmgpy.molecule.graph.Graph* method), 131
- sort_cyclic_vertices() (*rmgpy.molecule.Group* method), 162
- sort_cyclic_vertices() (*rmgpy.molecule.Molecule* method), 151
- sort_vertices() (*rmgpy.molecule.graph.Graph* method), 131
- sort_vertices() (*rmgpy.molecule.Group* method), 162
- sort_vertices() (*rmgpy.molecule.Molecule* method), 151
- sorting_key (*rmgpy.molecule.Atom* attribute), 139
- sorting_key (*rmgpy.molecule.Bond* attribute), 141
- sorting_key (*rmgpy.molecule.Molecule* attribute), 151
- sorting_key (*rmgpy.species.Species* attribute), 254
- Species (class in *rmgpy.species*), 251
- SpeciesError, 290
- specify_atom_extensions() (*rmgpy.molecule.Group* method), 162
- specify_bond_extensions() (*rmgpy.molecule.Group* method), 163
- specify_external_new_bond_extensions() (*rmgpy.molecule.Group* method), 163
- specify_internal_new_bond_extensions() (*rmgpy.molecule.Group* method), 163
- specify_ring_extensions() (*rmgpy.molecule.Group* method), 163
- specify_unpaired_extensions() (*rmgpy.molecule.Group* method), 163
- SphericalTopRotor (class in *rmgpy.statmech*), 263
- spin_multiplicity (*rmgpy.statmech.Conformer* attribute), 273
- split() (*rmgpy.molecule.graph.Graph* method), 131
- split() (*rmgpy.molecule.Group* method), 163
- split() (*rmgpy.molecule.Molecule* method), 151
- stabilize_charges_by_electronegativity() (in module *rmgpy.molecule.filtration*), 170
- stabilize_charges_by_proximity() (in module *rmgpy.molecule.filtration*), 170
- standardize_atomtype() (*rmgpy.molecule.Group* method), 163
- standardize_group() (*rmgpy.molecule.Group* method), 163
- StatmechDatabase (class in *rmgpy.data.statmech*), 65
- StatmechDepository (class in *rmgpy.data.statmech*), 66
- StatmechError, 290
- StatmechFitError, 290
- StatmechGroups (class in *rmgpy.data.statmech*), 73
- StatMechJob (class in *arkane*), 18
- StatmechLibrary (class in *rmgpy.data.statmech*), 76

- `step()` (*rmgpy.solver.LiquidReactor* method), 245
`step()` (*rmgpy.solver.MBSampledReactor* method), 250
`step()` (*rmgpy.solver.ReactionSystem* method), 240
`step()` (*rmgpy.solver.SimpleReactor* method), 243
`step()` (*rmgpy.solver.SurfaceReactor* method), 247
`StickingCoefficient` (class in *rmgpy.kinetics*), 122
`str_repr()` (in module *arkane.common*), 23
`successful_job_exists()`
 (*rmgpy.qm.qmverifier.QMVerifier* method), 194
`success_keys` (*rmgpy.qm.gaussian.Gaussian* attribute), 195
`success_keys` (*rmgpy.qm.gaussian.GaussianMol* attribute), 197
`success_keys` (*rmgpy.qm.gaussian.GaussianMolPM3* attribute), 199
`success_keys` (*rmgpy.qm.gaussian.GaussianMolPM6* attribute), 201
`success_keys` (*rmgpy.qm.mopac.Mopac* attribute), 202
`success_keys` (*rmgpy.qm.mopac.MopacMol* attribute), 204
`success_keys` (*rmgpy.qm.mopac.MopacMolPM3* attribute), 206
`success_keys` (*rmgpy.qm.mopac.MopacMolPM6* attribute), 208
`success_keys` (*rmgpy.qm.mopac.MopacMolPM7* attribute), 210
`SurfaceArrhenius` (class in *rmgpy.kinetics*), 120
`SurfaceReactor` (class in *rmgpy.solver*), 245
`symmetry` (*rmgpy.statmech.HinderedRotor* attribute), 270
`symmetry` (*rmgpy.statmech.KRotor* attribute), 263
`symmetry` (*rmgpy.statmech.LinearRotor* attribute), 260
`symmetry` (*rmgpy.statmech.NonlinearRotor* attribute), 261
`symmetry` (*rmgpy.statmech.SphericalTopRotor* attribute), 265
`SymmetryJob` (class in *rmgpy.qm.symmetry*), 194
- ## T
- `T0` (*rmgpy.kinetics.Arrhenius* attribute), 98
`T0` (*rmgpy.kinetics.StickingCoefficient* attribute), 123
`T0` (*rmgpy.kinetics.SurfaceArrhenius* attribute), 121
`T0` (*rmgpy.pdep.SingleExponentialDown* attribute), 181
`T1` (*rmgpy.kinetics.Troe* attribute), 115
`T2` (*rmgpy.kinetics.Troe* attribute), 115
`T3` (*rmgpy.kinetics.Troe* attribute), 115
`Tdata` (*rmgpy.kinetics.KineticsData* attribute), 96
`Tdata` (*rmgpy.kinetics.PDepKineticsData* attribute), 101
`Tdata` (*rmgpy.thermo.ThermoData* attribute), 275
`TemplateReaction` (class in *rmgpy.data.kinetics*), 78
`TeraChemLog` (class in *arkane.ess*), 11
`TerminationConversion` (class in *rmgpy.solver*), 250
`TerminationRateRatio` (class in *rmgpy.solver*), 250
`TerminationTime` (class in *rmgpy.solver*), 250
`thermo_filter_down()`
 (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 226
`thermo_filter_species()`
 (*rmgpy.rmg.model.CoreEdgeReactionModel* method), 227
`ThermoData` (class in *rmgpy.thermo*), 274
`ThermoDatabase` (class in *rmgpy.data.thermo*), 84
`ThermoDepository` (class in *rmgpy.data.thermo*), 87
`ThermoGroups` (class in *rmgpy.data.thermo*), 89
`ThermoJob` (class in *arkane*), 18
`ThermoLibrary` (class in *rmgpy.data.thermo*), 92
`ThirdBody` (class in *rmgpy.kinetics*), 110
`Tlist` (*arkane.KineticsJob* property), 13
`Tlist` (*arkane.PressureDependenceJob* property), 17
`Tmax` (*arkane.KineticsJob* property), 13
`Tmax` (*arkane.PressureDependenceJob* property), 17
`Tmax` (*rmgpy.kinetics.Arrhenius* attribute), 98
`Tmax` (*rmgpy.kinetics.Chebyshev* attribute), 108
`Tmax` (*rmgpy.kinetics.KineticsData* attribute), 96
`Tmax` (*rmgpy.kinetics.Lindemann* attribute), 113
`Tmax` (*rmgpy.kinetics.MultiArrhenius* attribute), 99
`Tmax` (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 105
`Tmax` (*rmgpy.kinetics.PDepArrhenius* attribute), 103
`Tmax` (*rmgpy.kinetics.PDepKineticsData* attribute), 101
`Tmax` (*rmgpy.kinetics.StickingCoefficient* attribute), 123
`Tmax` (*rmgpy.kinetics.SurfaceArrhenius* attribute), 121
`Tmax` (*rmgpy.kinetics.ThirdBody* attribute), 110
`Tmax` (*rmgpy.kinetics.Troe* attribute), 116
`Tmax` (*rmgpy.thermo.NASA* attribute), 281
`Tmax` (*rmgpy.thermo.NASAPolynomial* attribute), 284
`Tmax` (*rmgpy.thermo.ThermoData* attribute), 275
`Tmax` (*rmgpy.thermo.Wilhoit* attribute), 278
`Tmin` (*arkane.KineticsJob* property), 13
`Tmin` (*arkane.PressureDependenceJob* property), 17
`Tmin` (*rmgpy.kinetics.Arrhenius* attribute), 98
`Tmin` (*rmgpy.kinetics.Chebyshev* attribute), 108
`Tmin` (*rmgpy.kinetics.KineticsData* attribute), 96
`Tmin` (*rmgpy.kinetics.Lindemann* attribute), 113
`Tmin` (*rmgpy.kinetics.MultiArrhenius* attribute), 99
`Tmin` (*rmgpy.kinetics.MultiPDepArrhenius* attribute), 105
`Tmin` (*rmgpy.kinetics.PDepArrhenius* attribute), 103
`Tmin` (*rmgpy.kinetics.PDepKineticsData* attribute), 101
`Tmin` (*rmgpy.kinetics.StickingCoefficient* attribute), 124
`Tmin` (*rmgpy.kinetics.SurfaceArrhenius* attribute), 121
`Tmin` (*rmgpy.kinetics.ThirdBody* attribute), 110
`Tmin` (*rmgpy.kinetics.Troe* attribute), 116
`Tmin` (*rmgpy.thermo.NASA* attribute), 281
`Tmin` (*rmgpy.thermo.NASAPolynomial* attribute), 284
`Tmin` (*rmgpy.thermo.ThermoData* attribute), 275
`Tmin` (*rmgpy.thermo.Wilhoit* attribute), 278

[to_adjacency_list\(\)](#) (in module [rmgpy.molecule.adjlist](#)), 177
[to_adjacency_list\(\)](#) ([rmgpy.molecule.Group](#) method), 163
[to_adjacency_list\(\)](#) ([rmgpy.molecule.Molecule](#) method), 151
[to_adjacency_list\(\)](#) ([rmgpy.species.Species](#) method), 254
[to_arrhenius\(\)](#) ([rmgpy.kinetics.MultiArrhenius](#) method), 100
[to_arrhenius_ep\(\)](#) ([rmgpy.kinetics.Arrhenius](#) method), 99
[to_arrhenius_ep\(\)](#) ([rmgpy.kinetics.SurfaceArrhenius](#) method), 122
[to_augmented_inchi\(\)](#) ([rmgpy.molecule.Molecule](#) method), 151
[to_augmented_inchi_key\(\)](#) ([rmgpy.molecule.Molecule](#) method), 151
[to_cantera\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 35
[to_cantera\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 63
[to_cantera\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 83
[to_cantera\(\)](#) ([rmgpy.reaction.Reaction](#) method), 221
[to_cantera\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 238
[to_cantera\(\)](#) ([rmgpy.species.Species](#) method), 254
[to_cantera\(\)](#) ([rmgpy.thermo.NASA](#) method), 282
[to_cantera_kinetics\(\)](#) ([rmgpy.kinetics.Arrhenius](#) method), 99
[to_cantera_kinetics\(\)](#) ([rmgpy.kinetics.Lindemann](#) method), 114
[to_cantera_kinetics\(\)](#) ([rmgpy.kinetics.StickingCoefficient](#) method), 125
[to_cantera_kinetics\(\)](#) ([rmgpy.kinetics.SurfaceArrhenius](#) method), 122
[to_cantera_kinetics\(\)](#) ([rmgpy.kinetics.Troe](#) method), 117
[to_chemkin\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 35
[to_chemkin\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 63
[to_chemkin\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 83
[to_chemkin\(\)](#) ([rmgpy.reaction.Reaction](#) method), 221
[to_chemkin\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 238
[to_chemkin\(\)](#) ([rmgpy.species.Species](#) method), 254
[to_group\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_html\(\)](#) ([rmgpy.kinetics.Arrhenius](#) method), 99
[to_html\(\)](#) ([rmgpy.kinetics.Chebyshev](#) method), 110
[to_html\(\)](#) ([rmgpy.kinetics.KineticsData](#) method), 97
[to_html\(\)](#) ([rmgpy.kinetics.Lindemann](#) method), 114
[to_html\(\)](#) ([rmgpy.kinetics.MultiArrhenius](#) method), 100
[to_html\(\)](#) ([rmgpy.kinetics.MultiPDepArrhenius](#) method), 107
[to_html\(\)](#) ([rmgpy.kinetics.PDepArrhenius](#) method), 105
[to_html\(\)](#) ([rmgpy.kinetics.PDepKineticsData](#) method), 102
[to_html\(\)](#) ([rmgpy.kinetics.StickingCoefficient](#) method), 125
[to_html\(\)](#) ([rmgpy.kinetics.SurfaceArrhenius](#) method), 122
[to_html\(\)](#) ([rmgpy.kinetics.ThirdBody](#) method), 112
[to_html\(\)](#) ([rmgpy.kinetics.Troe](#) method), 117
[to_inchi\(\)](#) (in module [rmgpy.molecule.translator](#)), 174
[to_inchi\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_inchi_key\(\)](#) (in module [rmgpy.molecule.translator](#)), 174
[to_inchi_key\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_labeled_str\(\)](#) ([rmgpy.data.kinetics.DepositoryReaction](#) method), 35
[to_labeled_str\(\)](#) ([rmgpy.data.kinetics.LibraryReaction](#) method), 63
[to_labeled_str\(\)](#) ([rmgpy.data.kinetics.TemplateReaction](#) method), 83
[to_labeled_str\(\)](#) ([rmgpy.reaction.Reaction](#) method), 221
[to_labeled_str\(\)](#) ([rmgpy.rmg.pdep.PDepReaction](#) method), 238
[to_nasa\(\)](#) ([rmgpy.thermo.ThermoData](#) method), 276
[to_nasa\(\)](#) ([rmgpy.thermo.Wilhoit](#) method), 280
[to_ob_mol\(\)](#) (in module [rmgpy.molecule.converter](#)), 173
[to_rdkit_mol\(\)](#) (in module [rmgpy.molecule.converter](#)), 173
[to_rdkit_mol\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_single_bonds\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_smarts\(\)](#) (in module [rmgpy.molecule.translator](#)), 174
[to_smarts\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_smiles\(\)](#) (in module [rmgpy.molecule.translator](#)), 175
[to_smiles\(\)](#) ([rmgpy.molecule.Molecule](#) method), 152
[to_surface_charge_transfer\(\)](#) ([rmgpy.kinetics.SurfaceArrhenius](#) method), 122
[to_thermo_data\(\)](#) ([rmgpy.thermo.NASA](#) method), 282
[to_thermo_data\(\)](#) ([rmgpy.thermo.Wilhoit](#) method), 280

to_wilhoit() (*rmgpy.thermo.NASA method*), 283
 to_wilhoit() (*rmgpy.thermo.ThermoData method*), 276
 TransitionState (*class in rmgpy.species*), 255
 Troe (*class in rmgpy.kinetics*), 114

U

uncertainty (*rmgpy.kinetics.Arrhenius attribute*), 99
 uncertainty (*rmgpy.kinetics.Chebyshev attribute*), 110
 uncertainty (*rmgpy.kinetics.KineticsData attribute*), 97
 uncertainty (*rmgpy.kinetics.Lindemann attribute*), 114
 uncertainty (*rmgpy.kinetics.MultiArrhenius attribute*), 101
 uncertainty (*rmgpy.kinetics.MultiPDepArrhenius attribute*), 107
 uncertainty (*rmgpy.kinetics.PDepArrhenius attribute*), 105
 uncertainty (*rmgpy.kinetics.PDepKineticsData attribute*), 102
 uncertainty (*rmgpy.kinetics.StickingCoefficient attribute*), 125
 uncertainty (*rmgpy.kinetics.SurfaceArrhenius attribute*), 122
 uncertainty (*rmgpy.kinetics.ThirdBody attribute*), 112
 uncertainty (*rmgpy.kinetics.Troe attribute*), 117
 uncertainty (*rmgpy.quantity.ArrayQuantity attribute*), 214
 uncertainty (*rmgpy.quantity.ScalarQuantity attribute*), 213
 uncertainty_type (*rmgpy.quantity.ArrayQuantity attribute*), 214
 uncertainty_type (*rmgpy.quantity.ScalarQuantity attribute*), 213
 UndeterminableKineticsError, 290
 UnexpectedChargeError, 291
 unique_id (*rmgpy.qm.molecule.Geometry attribute*), 191
 unique_id (*rmgpy.qm.symmetry.SymmetryJob attribute*), 195
 unique_id_long (*rmgpy.qm.molecule.Geometry attribute*), 192
 unit_degeneracy() (*in module rmgpy.statmech.schrodinger*), 271
 unperturb() (*arkane.sensitivity.KineticsSensitivity method*), 19
 unperturb() (*arkane.sensitivity.PDepSensitivity method*), 20
 update() (*rmgpy.molecule.kekulize.AromaticBond method*), 167
 update() (*rmgpy.molecule.kekulize.AromaticRing method*), 168
 update() (*rmgpy.molecule.Molecule method*), 152
 update() (*rmgpy.rmg.pdep.PDepNetwork method*), 233

update_atomtypes() (*rmgpy.molecule.Molecule method*), 152
 update_charge() (*rmgpy.molecule.Atom method*), 139
 update_charge() (*rmgpy.molecule.Group method*), 163
 update_configurations() (*rmgpy.rmg.pdep.PDepNetwork method*), 233
 update_connectivity_values() (*rmgpy.molecule.graph.Graph method*), 131
 update_connectivity_values() (*rmgpy.molecule.Group method*), 163
 update_connectivity_values() (*rmgpy.molecule.Molecule method*), 153
 update_fingerprint() (*rmgpy.molecule.Group method*), 163
 update_lone_pairs() (*rmgpy.molecule.Molecule method*), 153
 update_multiplicity() (*rmgpy.molecule.Molecule method*), 153
 update_reaction_threshold_and_react_flags() (*rmgpy.rmg.main.RMG method*), 230
 update_species_attributes() (*arkane.common.ArkaneSpecies method*), 21
 update_unimolecular_reaction_networks() (*rmgpy.rmg.model.CoreEdgeReactionModel method*), 227
 update_xyz_string() (*arkane.common.ArkaneSpecies method*), 21

V

value (*rmgpy.quantity.ArrayQuantity attribute*), 214
 value (*rmgpy.quantity.ScalarQuantity attribute*), 213
 verify_output_file() (*rmgpy.qm.gaussian.Gaussian method*), 195
 verify_output_file() (*rmgpy.qm.gaussian.GaussianMol method*), 197
 verify_output_file() (*rmgpy.qm.gaussian.GaussianMolPM3 method*), 199
 verify_output_file() (*rmgpy.qm.gaussian.GaussianMolPM6 method*), 201
 verify_output_file() (*rmgpy.qm.mopac.Mopac method*), 202
 verify_output_file() (*rmgpy.qm.mopac.MopacMol method*), 204
 verify_output_file() (*rmgpy.qm.mopac.MopacMolPM3 method*), 206

verify_output_file() (rmgpy.qm.mopac.MopacMolPM6 method), 208
 verify_output_file() (rmgpy.qm.mopac.MopacMolPM7 method), 210
 Vertex (class in rmgpy.molecule.graph), 127
 VF2 (class in rmgpy.molecule.vf2), 131
 VF2Error, 291
 visit() (arkane.output.PrettifyVisitor method), 14
 visit_Call() (arkane.output.PrettifyVisitor method), 14
 visit_Dict() (arkane.output.PrettifyVisitor method), 14
 visit_List() (arkane.output.PrettifyVisitor method), 14
 visit_Num() (arkane.output.PrettifyVisitor method), 14
 visit_Str() (arkane.output.PrettifyVisitor method), 15
 visit_Tuple() (arkane.output.PrettifyVisitor method), 15
 visit_UnaryOp() (arkane.output.PrettifyVisitor method), 15
W
 Wigner (class in rmgpy.kinetics), 118
 Wilhoit (class in rmgpy.thermo), 277
 with_traceback() (rmgpy.exceptions.ActionError method), 287
 with_traceback() (rmgpy.exceptions.AtomTypeError method), 287
 with_traceback() (rmgpy.exceptions.ChemicallySignificantEigenvalueError method), 287
 with_traceback() (rmgpy.exceptions.ChemkinError method), 287
 with_traceback() (rmgpy.exceptions.CollisionError method), 287
 with_traceback() (rmgpy.exceptions.CoreError method), 287
 with_traceback() (rmgpy.exceptions.DatabaseError method), 287
 with_traceback() (rmgpy.exceptions.DependencyError method), 287
 with_traceback() (rmgpy.exceptions.ElementError method), 288
 with_traceback() (rmgpy.exceptions.ForbiddenStructureException method), 288
 with_traceback() (rmgpy.exceptions.ImplicitBenzeneError method), 288
 with_traceback() (rmgpy.exceptions.InchiException method), 288
 with_traceback() (rmgpy.exceptions.InputError method), 288
 with_traceback() (rmgpy.exceptions.InvalidActionError method), 288
 with_traceback() (rmgpy.exceptions.InvalidAdjacencyListError method), 288
 with_traceback() (rmgpy.exceptions.InvalidMicrocanonicalRateError method), 289
 with_traceback() (rmgpy.exceptions.KekulizationError method), 289
 with_traceback() (rmgpy.exceptions.KineticsError method), 289
 with_traceback() (rmgpy.exceptions.ModifiedStrongCollisionError method), 289
 with_traceback() (rmgpy.exceptions.NegativeBarrierException method), 289
 with_traceback() (rmgpy.exceptions.NetworkError method), 289
 with_traceback() (rmgpy.exceptions.OutputError method), 289
 with_traceback() (rmgpy.exceptions.PressureDependenceError method), 289
 with_traceback() (rmgpy.exceptions.QuantityError method), 290
 with_traceback() (rmgpy.exceptions.ReactionError method), 290
 with_traceback() (rmgpy.exceptions.ReactionPairsError method), 290
 with_traceback() (rmgpy.exceptions.ReservoirStateError method), 290
 with_traceback() (rmgpy.exceptions.ResonanceError method), 290
 with_traceback() (rmgpy.exceptions.SettingsError method), 290
 with_traceback() (rmgpy.exceptions.SpeciesError method), 290
 with_traceback() (rmgpy.exceptions.StatmechError method), 290
 with_traceback() (rmgpy.exceptions.StatmechFitError method), 290
 with_traceback() (rmgpy.exceptions.UndeterminableKineticsError method), 291
 with_traceback() (rmgpy.exceptions.UnexpectedChargeError method), 291
 with_traceback() (rmgpy.exceptions.VF2Error method), 291
 write_chemkin() (arkane.KineticsJob method), 13
 write_chemkin() (arkane.ThermoJob method), 18
 write_input_file() (rmgpy.qm.gaussian.GaussianMol method), 197
 write_input_file() (rmgpy.qm.gaussian.GaussianMolPM3 method), 199
 write_input_file() (rmgpy.qm.gaussian.GaussianMolPM6 method), 202
 write_input_file() (rmgpy.qm.mopac.MopacMol method), 204
 write_input_file() (rmgpy.qm.mopac.MopacMolPM3 method), 206

`write_input_file()` (*rmgpy.qm.mopac.MopacMolPM6*
method), [208](#)
`write_input_file()` (*rmgpy.qm.mopac.MopacMolPM7*
method), [210](#)
`write_input_file()` (*rmgpy.qm.symmetry.SymmetryJob*
method), [195](#)
`write_kinetics_entry()` (*in module*
rmgpy.chemkin), [26](#)
`write_output()` (*arkane.KineticsJob method*), [13](#)
`write_output()` (*arkane.StatMechJob method*), [18](#)
`write_output()` (*arkane.ThermoJob method*), [19](#)
`write_thermo_entry()` (*in module rmgpy.chemkin*),
[26](#)