RPMDrate Manual

Yury V. Suleimanov

Release 1.0

October 25, 2013

CONTENTS

1 Introduction to RPMDrate 1
- 1.1 What is RPMD? . . . 1
- 1.2 License . . . 3
- 1.3 References . . . 3

2 Installing RPMDrate 5
- 2.1 Installation . . . 5
- 2.2 Dependencies . . . 5
- 2.3 Compilation . . . 6
- 2.4 Local installation of Python . . . 6

3 Creating an RPMDrate input file 9
- 3.1 Define the potential energy surface . . . 9
- 3.2 Define the bimolecular reactants . . . 10
- 3.3 Define the transition state dividing surface . . . 10
- 3.4 Define the thermostat . . . 11
- 3.5 Generate initial umbrella configurations . . . 13
- 3.6 Define the umbrella integration parameters . . . 13
- 3.7 Biased sampling . . . 14
- 3.8 Calculate the potential of mean force . . . 15
- 3.9 Calculate the transmission coefficient . . . 15
- 3.10 Compute the ring polymer rate coefficient . . . 16

4 Exposing an external potential to Python 19
- 4.1 Define the Fortran interface . . . 19
- 4.2 Define the f2py wrapper module . . . 20
- 4.3 Compile the Python wrapper . . . 21

5 Running RPMDrate 23
- 5.1 Browsing RPMDrate output files . . . 23

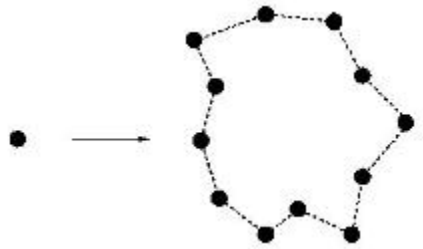
CHAPTER

ONE

INTRODUCTION TO RPMDRATE

1.1 What is RPMD?

RPMD is short for Ring Polymer Molecular Dynamics: an approximate quantum mechanical simulation technique [1] to compute approximate values of various dynamics properties, such as chemical reaction rate coefficients [2], [3], [4] and diffusion coefficients [5]. RPMD is based on the isomorphism between the quantum statistical mechanics of the physical system and the classical statistical mechanics of a fictious ring polymer consisting of n copies of the system connected by harmonic springs [6].



The resulting RPMD reaction rate theory is essentially a classical rate theory in an extended (discretized imaginary time path integral) phase space, and thus gives it some very desitable features:

- First, the RPMD rate becomes exact in the high temperature limit, where the ring polymer collapses to a single bead. It is also exact for a parabolic barrier bilinearly coupled to a bath of harmonic oscillators at all temperatures for which a rate coefficient can be defined.

[1] Craig, I. R.; Manolopoulos, D. E. Quantum statistics and classical mechanics: Real time correlation functions from ring polymer molecular dynamics, J. Chem. Phys. 2004, 121, 3368.

[2] Collepardo-Guevara, R.; Suleimanov, Yu. V.; Manolopoulos, D. E. Bimolecular reaction rates from ring polymer molecular dynamics, J. Chem. Phys. 2009, 130, 174713.

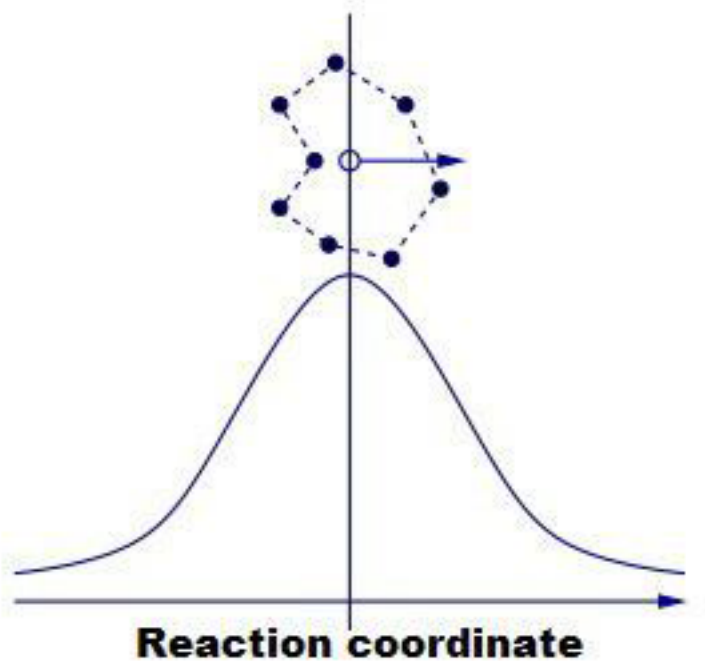
[3] Suleimanov, Yu. V.; Collepardo-Guevara, R.; Manolopoulos, D. E.; Bimolecular reaction rates from ring polymer molecular dynamics: Application to $CH_4 + H \rightarrow CH_3 + H_2$, J. Chem. Phys. 2011, 134, 044131.

[4] Perez de Tudela, R.; Aoiz, F. J.; Suleimanov, Yu. V.; Manolopoulos, D. E. Chemical reaction rates from ring polymer molecular dynamics: Zero point energy conservation in $Mu + H_2 \rightarrow MuH + H$, J. Phys. Chem. Lett. 2012, 3, 493.

[5] Suleimanov, Yu. V. Surface Diffusion of Hydrogen on Ni(100) from Ring Polymer Molecular Dynamics, J. Phys. Chem. C 2012, 116, 11141.

[6] Chandler, D. Exploiting the Isomorphism between Quantum Theory and Classical Statistical Mechanics of Polyatomic Fluids, J. Chem. Phys. 1981, 74, 4078.

- Second, the theory has a well-defined short-time limit that provides an upper bound on the RPMD rate, in the same way as classical transition state theory provides an upper bound on the classical rate. Indeed when the dividing surface is defined in terms of the centroid of the ring polymer the short-time limit of the RPMD rate coincides with a well-known (centroid-density) version of the quantum transition state theory (QTST).
- Finally, and perhaps most importantly, the RPMD rate coefficient is rigorously independent of the choice of the transition state dividing surface that is used to compute it. This is a highly desirable feature of the theory for applications to multidimensional reactions for which the optimum dividing surface can be very difficult to determine.



The application of RPMD to the study of thermally activated gas-phase bimolecular reactions is one of the most recent developments [2], [3], [4]. The rate coefficients obtained so far with RPMD

- are reliable (predictive) at high temperatures;
- correctly capture the zero-point energy effects;
- are within a factor of 2-3 of accurate QM results even at very low temperatures in the deep tunneling regime (when such comparison is available).

RPMDrate is the name of this software package, which provides functionality for conducting RPMD simulations in order to compute the bimolecular reaction rate coefficients for thermally activated processes in the gas phase. The RPMD rate methodology has been developed in Refs. [2], [3]. The RPMD codebase is split into a Fortran 90/95 core used to efficiently conduct the RPMD simulations, and a Python layer that provides a more user-friendly, scriptable interface.

1.2 License

RPMDrate is distributed under the terms of the MIT license[7]. This is a permissive license, meaning that you are generally free to use, redistribute, and modify RPMDrate at your discretion. If you use all or part of RPMDrate in your work, we only ask that proper attribution be given (in addition to following the terms of the license as stated below).

The MIT license is reproduced in its entirety below:

1.3 References

1.3.1 RPMDrate references

The recommended reference for the current version of the RPMDrate code is to give Reference [1]. Reference [2] is also recommended in certain cases because it provides published explanations of the methodology used in the code.

[1] Suleimanov, Yu. V.; Allen, J. W.; Green, W. H. RPMDrate: bimolecular chemical reaction rates for ring polymer molecular dynamics, Comp. Phys. Comm. 2013, 184, 833.

[2] Suleimanov, Yu. V.; Collepardo-Guevara, R.; Manolopoulos, D. E.; Bimolecular reaction rates from ring polymer molecular dynamics: Application to $CH_4 + H \rightarrow CH_3 + H_2$, J. Chem. Phys. 2011, 134, 044131.

[7] http://www.opensource.org/licenses/mit-license

1.3.2 References for incorporated algorithms

1. Bennett-Chandler factorization of the rate coeffiicent.

[3] Bennett, C. H. In Algorithms for Chemical Computations, ACS Symposium Series No. 46; Christofferson, R. E., Ed.; American Chemical Society: Washington DC, 1977; p 63.

[4] Chandler, D. Statistical mechanics of isomerization dynamics in liquids and the transition state approximation, J. Chem. Phys. 1978, 68, 2959.

2. Umbrella integration along the reaction coordinate.

[5] Kästner, J.; Thiel, W. Bridging the gap between thermodynamic integration and umbrella sampling provides a novel analysis method: "Umbrella integration" J. Chem. Phys. 2005, 123, 144104.

[6] Kästner, J.; Thiel, W. Analysis of the statistical error in umbrella sampling simulations by umbrella integration J. Chem. Phys. 2006, 124, 234106.

[7] Kästner, J. Umbrella integration in two or more reaction coordinates, J. Chem. Phys. 2009, 131, 034109.

3. RATTLE algorithm for constrained molecular dynamics simulations.

[8] Andersen, H. C. Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations, J. Comput. Phys. 1983, 52, 24.

4. Andersen thermostat.

[9] Andersen, H. C. Molecular dynamics simulations at constant pressure and/or temperature, J. Chem. Phys. 1980, 72, 2384.

5. Colored-Noise, generalized Langevin equation thermostats.

[10] Ceriotti, M.; Bussi, G.; Parrinello, M. Langevin equation with colored noise for constant-temperature molecular dynamics simulations, Phys. Rev. Lett. 2009, 102, 020601.

[11] Ceriotti, M.; Bussi, G.; Parrinello, M. Nuclear quantum effects in solids using a colored-noise thermostat, Phys. Rev. Lett. 2009, 103, 030603.

[12] Ceriotti, M.; Bussi, G.; Parrinello, M. Colored-noise thermostats a la carte, J. Chem. Theory Comput. 2010, 6, 1170.

[13] Ceriotti, M.; Manolopoulos, D. E.; Parrinello, M. Accelerating the convergence of path integral dynamics with a generalized Langevin equation, J. Chem. Phys. 2011, 134, 084104.

CHAPTER

TWO

INSTALLING RPMDRATE

2.1 Installation

There are multiple ways to download RPMDrate to your computer:

- Download and compile a stable release package as a tar gzipped file directly from the Computer Physics Communications Program Library (http://cpc.cs.qub.ac.uk).
- Clone our git repository from

  https://github.com/GreenGroup/RPMDrate.git

  and easily follow along with the bleeding-edge development as it happens.
- Use one of the automatic Python package installation tools, such as `pip` (recommended) or `easy_install`

  ```
  $ pip install rpmdrate
  ```

  or

  ```
  $ easy_install rpmdrate
  ```

The benefit of this approach is that all of the dependencies will be handled automatically for you. All you need is a reasonably recent Fortran 90/95 compiler for compiling the Fortran modules.

2.2 Dependencies

RPMDrate depends on several other packages in order to provide its full functional capabilities. The following dependencies are required to compile and run RPMDrate:

- Python[1] (version 2.6.x or later, including any version of Python 3, is recommended)
- NumPy[2] (version 1.5.0 or later is recommended)
- FFTW[3] (version 3.3 or later is recommended)

[1] http://www.python.org/
[2] http://numpy.scipy.org/
[3] http://www.fftw.org/

- A standard Fortran 90/95 compiler (e.g. `gfortran`, `g95`, `ifort`, etc.)

2.3 Compilation

Once you have obtained a copy of RPMDrate, either from a released package or from cloning the git repository, RPMDrate can be installed by invoking the `setup.py` script as usual:

```
$ python setup.py install
```

This will compile the Fortran source and Python wrapper code, which may take some time. Note that you may need administrator privileges to install RPMDrate.

If you wish to use RPMDrate without installing, simply add the folder containing this file to your `PYTHONPATH` environment variable and compile the source code in-place using the command

```
$ python setup.py build_ext --inplace
```

A Makefile that wraps these commands has been provided. The Makefile also provides a clean target for deleting all files created during compilation.

RPMDrate uses the `f2py` tool (bundled with NumPy) to expose its Fortran code to Python. The default compiling option for the FFTW library (which is required to link against when building Fortran extension modules using `f2py`) is "-lfftw3". These options can be modified at the end of the `setup.py` file as the `libraries` options of the `rpmdrate._main` Fortran extension module. The libraries are assumed to be located in the standard library search path on target systems. If you need to link with these libraries in a non-standard location, you'll have to specify the directories to search for those libraries. The `library_dirs` option is a list of directories to search for libraries at link-time, and the `runtime_library_dirs` option - for shared (dynamically loaded) libraries at run-time. (More information is available at http://docs.python.org/distutils.)

2.4 Local installation of Python

Below is the procedure to install `Python 2.7` to your local account. This will place `Python 2.7` in `$HOME/local directory` (e.g. /home/username/local). The whole procedure should not take more than 10-15 minutes.

1. Install `Python 2.7`

```
$ mkdir $HOME/local

$ wget http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2

$ tar -xjvf Python-2.7.3.tar.bz2

$ cd Python-2.7.3

$ ./configure --prefix=$HOME/local
```

```
$ make

$ make install
```

2. Edit `.bash_profile` to add these lines

```
$ export PATH=$HOME/local/bin:$PATH

$ export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
```

3. Log out and in again so modified `.bash_profile` takes effect
4. Run `Python` once to ensure default is 2.7 3. Install `setuptools`

```
$ wget http://pypi.python.org/packages/source/s/setuptools/setuptools-0.6c11.tar.gz

$ tar -xzvf setuptools-0.6c11.tar.gz

$ cd setuptools-0.6c11

$ python setup.py install
```

5. Install `pip`

```
$ wget http://pypi.python.org/packages/source/p/pip/pip-1.1.tar.gz

$ tar -xzvf pip-1.1.tar.gz

$ cd pip-1.1

$ python setup.py install
```

5. Install `numpy` (and `f2py`)

```
$ pip install numpy
```

2.4.1 Developing for RPMDrate

If you would like to follow along with the bleeding-edge development of RPMDrate, the best way to do so is to clone the repository using our version control system, git[4]:

```
$ git clone git://github.com/GreenGroup/RPMDrate.git
```

This enables you to easy update your local copy with the latest changes. If you intend to contribute to RPMDrate - and you are welcome to do so! - you should fork the project on GitHub first, and clone from your fork.

Once you have cloned the repository, follow the compilation instructions for building RPMDrate in-place. You may also wish to add the folder containing the RPMDrate repository to your `PYTHONPATH` environment variable.

[4] http://git-scm.com/

2.4.2 Running the Unit Tests

RPMDrate comes with a large suite of unit tests that ensure functionality is working as intended. To run these tests, first install RPMDrate or compile it from source in-place using the directions in the previous section. Then, simply invoke the entire suite of unit tests using

```
$ python setup.py test
```

This will run all of the unit tests in sequence, which may take some time. If one or more unit tests fail, please report them on the GitHub issue tracker (if they aren't already reported).

CHAPTER

THREE

CREATING AN RPMDRATE INPUT FILE

The format of RPMDrate input files is based on Python syntax. In fact, the RPMDrate input file is a valid Python source code, and this is used to facilitate reading of the file.

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

3.1 Define the potential energy surface

The RPMDrate code must be provided with an external function that computes the Born-Oppenheimer potential energy with gradients for a given geometry defined in the atomic Cartesian coordinates. In principle, this can be any Python callable object of the form

```
V, dVdq = get_potential(q)
```

where the lone parameter `q` is the $3 \times N_{\text{atoms}} \times N_{\text{beads}}$ array of Cartesian coordinates of N_{beads} beads of N_{atoms} atoms, and the outputs are the corresponding potentials and forces. The input file requires the name to be `get_potential`. Input and output parameters should be in atomic units.

In practice, evaluating the potential energy and gradients is computationally expensive, and so the global potential energy function is likely to be prepared as an analytic function evaluated by subroutines written in another language (such as Fortran) and must therefore be exposed to Python. The process of doing this is explained in the next chapter Exposing an external potential to Python.

If your implementation of the potential function requires any initialization, you can also provide a Python function `initialize_potential` for this purpose. This function is assumed not to accept any parameters, and any return value is ignored. If the `initialize_potential` identifier is not found, the code will assume that no initialization of the potential is required.

A recommended practice is to define the potential function (and its initialization subroutine, if applicable) in a separate module `PES` placed in the same directory as the input file. If this is done correctly, then the input file need only import them:

```
from PES import get_potential, initialize_potential
```

If no initialization subroutine is required, then the syntax is

```
from PES import get_potential
```

3.2 Define the bimolecular reactants

The next step is to provide information about the reactants. This is done by calling the `reactants()` function with the following arguments, all of which are required:

- `atoms` - A list of the atom symbols for all atoms in the reactant pair. RPMDrate recognizes a variety of atom symbols; the full list is available in the `rpmd.element` module.
- `reactant1Atoms` - A list of the indices of each atom in the first reactant molecule. The indices correspond to those of the `atoms` list, except that the first index is one, not zero.
- `reactant2Atoms` - A list of the indices of each atom in the second reactant molecule. The indices correspond to those of the `atoms` list, except that the first index is one, not zero.
- `Rinf` - The center-of-mass distance which should be chosen to be sufficiently large as to make interaction between the reactant molecules negligible. This is specified using a 2-tuple of the form `(value,units)`, where `value` is a number and `units` is a string with the corresponding units (of length). RPMDrate can handle a variety of input units for length, including `m`, `angstrom`, and `bohr` (atomic units).

The following example illustrates the structure of the `reactants()` function for the bimolecular reaction $CH_4 + H \rightarrow CH_3 + H_2$:

```
reactants(
    atoms = ['H', 'C', 'H', 'H', 'H', 'H'],
    reactant1Atoms = [1,2,3,4,5],
    reactant2Atoms = [6],
    Rinf = (15,"angstrom"),
)
```

3.3 Define the transition state dividing surface

Next step is to define parameters of the transition state dividing surface. The easiest way to do this is to specify a transition state geometry and lists of the bonds that are forming and breaking. RPMDrate will use this information to extract the relevant transition state bond lengths to use in the transition state dividing surface. Note that the final RPMD rate coefficient is independent of the choice of the initial transition state configuration. However, it is computationally advantageous to provide a reasonable configuration so as to minimize the recrossings of the transition state dividing surface.

- The `geometry` is specified as a 2-tuple of the form `(value,units)`, where `value` is a $N_{\text{atoms}} \times 3$ matrix (list of lists) of coordinates and `units` is a string with the corresponding units (of length). The coordinates must be given in the same order as the list of atom symbols `atoms` was given in the `reactants()` block.
- The `formingBonds` and `breakingBonds` are lists of 2-tuples containing the indices of the atoms involved in each forming or breaking bond. As before, the indices correspond to the list

of atom symbols `atoms` was given in the `reactants()` block, and the indices start from one instead of zero.

For the $CH_4 + H \rightarrow CH_3 + H_2$ reaction, the transition state can be defined as follows:

```
transitionState(
    geometry = (
        [[-4.68413503,   -0.43825460,   -0.07250839],
         [-5.04748906,    0.58950601,   -0.07250840],
         [-4.68411607,    1.10337961,    0.81755453],
         [-4.58404767,    1.24489401,   -1.20768359],
         [-6.13758906,    0.58951941,   -0.07250839],
         [-4.29480419,    1.61876150,   -1.94140095]],
        "angstrom",
    ),
    formingBonds = [(4,6)],
    breakingBonds = [(2,4)],
)
```

In our example, the H radical can abstract any of the H atoms from methane. These equivalent transition states can be added with one or more `equivalentTransitionState()` blocks. In these blocks you need only specify the forming and breaking bonds in an order corresponding to that of the main `transitionState()` block. RPMDrate will automatically copy the relevant transition state distances from the main transition state to use in each of the equivalent transition states.

In our example we must add three equivalent transition states to obtain the correct reaction-path degeneracy of four:

```
equivalentTransitionState(
    formingBonds=[(1,6)],
    breakingBonds=[(2,1)],
)
equivalentTransitionState(
    formingBonds=[(3,6)],
    breakingBonds=[(2,3)],
)
equivalentTransitionState(
    formingBonds=[(5,6)],
    breakingBonds=[(2,5)],
)
```

Some systems will have several combinations of breaking and forming bonds. In this case, you need to specify all these combinations in the `formingBonds` and `breakingBonds` lists. For instance, in the case of 3 bonds it has the following form:

```
formingBonds=[(A1,A2),(B1,B2),(C1,C2)],
breakingBonds=[(A3,A1),(B3,B1),(C3,C1)]
```

3.4 Define the thermostat

RPMDrate provides two options for temperature control of time-independent part of molecular dynamics simulations: Andersen thermostatting scheme and colored noise, generalized Langevin

equation (GLE) thermostats.

3.4.1 Andersen thermostat

The Andersen thermostat is the simplest stochastic thermostat which does correctly sample the NVT ensemble. In this method, atoms at each integration step are subject to a small probability to experience collision with the heat bath, which is simulated by resampling the momenta from a Boltzmann distribution at the desired temperature. The following sets up an Andersen thermostat with an automatically-determined sampling time:

```
thermostat('Andersen')
```

The user also has an option to fix the time between collisions using the `samplingTime` parameter. This parameter is given as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` is a string with the corresponding units (of time). RPMDrate can handle a variety of input units for time, including `s`, `ms`, `us`, `ns`, `ps`, and `fs`. An example of fixing the sampling time is given below:

```
thermostat('Andersen', samplingTime=(100,'fs'))
```

3.4.2 GLE thermostats

The more elaborate colored-noise, generalized Langevin equation (GLE) thermostats can be used to enhance canonical sampling and to reduce the number of ring polymer beads required to achieve convergence in path integral dynamics. These thermostats require more effort to set up but also can provide much faster convergence for complex polyatomic systems. To use these thermostats, user must provide the drift matrix A in a separate text file. For example, the following sets up a GLE thermostat with the drift matrix read from the file `gle_A.txt` in the same directory as the input file:

```
thermostat('GLE', A=('gle_A.txt','s^-1')).
```

If non-FDT (fluctuation-dissipation theorem) dynamics is to be performed, a similar diffusion matrix C is required, containing the covariance matrix. In this case the input would look similar to

```
thermostat('GLE', A=('gle_A.txt','s^-1'), C=('gle_C_1000.txt','K')).
```

Here `A` and `C` are 2-tuples of the form `(file,units)`, where `file` is the name of the file containing the corresponding matrix and `units` are the corresponding units of inverse time (matrix A) and energy (matrix C). RPMDrate can handle a variety of input units for inverse time, including `s^-1`, `ps^-1`, and `fs^-1`. The diffusion matrix C should be given in units of K (Kelvin).

The files containing the A and C matrices should be structured such that each line contains one row of the matrix, with each element in that row separated by whitespace. The hash character `#` can be used to indicate comment lines.

An easy way to generate the A and C matrices is to use the online form at the GLE4MD[1] web page. On this page, select `Input` and then fill out the rest of the form with the values for your system.

[1] http://gle4md.berlios.de/

Be sure to check that the output units for the two matrices are the same as those indicated in the RPMDrate input file.

3.5 Generate initial umbrella configurations

At this point the reactive system is successfully set up. In the remainder of the RPMDrate input file we specify the parameters of the various steps involved in computing the ring polymer rate coefficient.

The first step is to generate the initial (classical, i.e. with number of beads set to 1) configurations required for umbrella sampling in each biasing window. This can accomplished by a brief biased sampling in each window using only one bead, using the result from the previous window as the initial configuration for the next window (starting from the initial transition state configuration). The resulting configurations can be reused in subsequent calculations (e.g., for different number of beads).

The generation of the umbrella configurations is invoked using a `generateUmbrellaConfigurations()` block, which accepts the following parameters:

- `dt` - The evolution time step, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `evolutionTime` - The total length of each trajectory, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `xi_list` - A list, tuple, or array containing the centers of windows centers placed along the reaction coordinate.
- `kforce` - The value of the umbrella force constant which defines the strength of the bias in the window, in atomic units.

An example of a `generateUmbrellaConfigurations()` block for windows evenly spaced in the interval between -0.05 and 1.05 along the reaction coordinate with a step of 0.01 is given below:

```
generateUmbrellaConfigurations(
    dt = (0.0001,"ps"),
    evolutionTime = (5,"ps"),
    xi_list = numpy.arange(-0.05, 1.05, 0.01),
    kforce = 0.1 * T,
)
```

3.6 Define the umbrella integration parameters

The next step is to define the umbrella integration parameters. As the RPMDrate input file is a Python script, this is very easily done with a small bit of Python code. The idea is to create a list of the windows, each one a `Window()` object with the following parameters:

- `xi` - The position of the center of the window.

- `kforce` - The value of the umbrella force constant which defines the strength of the bias in the window, in atomic units.
- `trajectories` - The number of biased sampling trajectories to run for the window. These trajectores can be run in parallel (see Running RPMDrate).
- `equilibrationTime` - The equilibration period for each trajectory before sampling is initiated. This value is specified as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `evolutionTime` - The total length of each trajectory, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).

An example of code that generates a set of umbrella integration windows is given below:

```
windows = []
for xi in numpy.arange(-0.05, 1.05, 0.01):
    window = Window(xi=xi, kforce=0.1*T, trajectories=100, equilibrationTime=(20,"ps"),
    evolutionTime=(100,"ps"))
    windows.append(window)
```

Note that you do not necessarily have to space out the windows evenly, or to use the same force constant or sampling times in each window. For example, the following code uses a larger reaction coordinate step size and weaker bias at lower values of `xi`:

```
windows = []
for xi in numpy.arange(-0.05, 0.25, 0.05):
    window = Window(xi=xi, kforce=0.03*T, trajectories=100, equilibrationTime=(20,"ps"),
    evolutionTime=(100,"ps"))
    windows.append(window)
for xi in numpy.arange(-0.25, 0.55, 0.03):
    window = Window(xi=xi, kforce=0.05*T, trajectories=100, equilibrationTime=(20,"ps"),
    evolutionTime=(100,"ps"))
    windows.append(window)
for xi in numpy.arange( 0.55, 1.15, 0.01):
    window = Window(xi=xi, kforce=0.1*T, trajectories=100, equilibrationTime=(20,"ps"),
    evolutionTime=(100,"ps"))
    windows.append(window)
```

3.7 Biased sampling

Once we have defined the biasing windows, we can conduct the biased sampling via a `conductUmbrellaSampling()` block, which accepts the following parameters:

- `dt` - The evolution time step, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `windows` - The list of umbrella integration windows, as generated in the previous section.

An example of a `conductUmbrellaSampling()` block is given below:

```
conductUmbrellaSampling(
    dt = (0.0001,"ps"),
```

```
    windows = windows,
)
```

For each window, the mean $\overline{\xi}$ and variance σ_{ξ}^2 of the reaction coordinate are accumulated and stored after running each trajectory.

3.8 Calculate the potential of mean force

When biased sampling is complete in all windows, the potential of mean force (free energy) along the reaction coordinate is calculated using umbrella integration. This is done using a `computePotentialOfMeanForce()` block with the following parameters all of which are required:

- `windows` - The list of biasing windows used to conduct the biased sampling.
- `xi_min` - The value of the reaction coordinate to use as a lower bound for the umbrella integration range.
- `xi_max` - The value of the reaction coordinate to use as an upper bound for the umbrella integration range.
- `bins` - The number of bins for the umbrella integration algorithm to use. Umbrella integration is performed using the trapezoidal rule between the bins.

An example of a `computePotentialOfMeanForce()` block is given below:

```
computePotentialOfMeanForce(windows=windows, xi_min=-0.02, xi_max=1.02, bins=5000)
```

The potential of mean force can also be calculated for the current state. The code will automatically find all previous files with the results of the biased sampling for given temperature and number of beads. In this case, `conductUmbrellaSampling` block and `windows=windows` should be removed:

```
computePotentialOfMeanForce(xi_min=-0.02, xi_max=1.02, bins=5000)
```

3.9 Calculate the transmission coefficient

The final step before calculating the ring polymer rate coefficient is the transmission coefficient (recrossing factor). Having obtained the optimal value of the reaction coordinate from the potential of mean force profile, it is evaluated by performing a constrained RPMD simulation in the presence of a thermostat (parent trajectory) to generate a series of configurations at the transition state dividing surface. For each of these constrained configurations, a series of recrossing trajectories are evolved forward in time without a thermostat or a dividing surface constraint (child trajectories), and the fraction of these trajectories resulting in products is determined.

The computation of the recrossing factor is invoked using a `computeRecrossingFactor()` block, which accepts the following parameters:

- `dt` - The evolution time step, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).

- `equilibrationTime` - The equilibration period in the parent (constrained trajectory). This value is specified as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `childTrajectories` - The total number of child trajectories to run.
- `childSamplingTime` - The length of the parent trajectory between each set of child trajectories, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).
- `childrenPerSampling` - The number of child trajectores to sample after each evolution of the parent trajectory.
- `childEvolutionTime` - The length of each child trajectory, as a 2-tuple of the form `(value,units)`, where `value` is a number and `units` are the corresponding units (of time).

An example of a `computeRecrossingFactor()` block is given below:

```
computeRecrossingFactor(
    dt = (0.0001,"ps"),
    equilibrationTime = (20,"ps"),
    childTrajectories = 100000,
    childSamplingTime = (2,"ps"),
    childrenPerSampling = 100,
    childEvolutionTime = (0.05,"ps"),
)
```

By default, the `computeRecrossingFactor()` block calculates the recrossing factor at the position of the reaction coordinate which corresponds to the maximum along the potential of mean force profile. This choice of the reaction coordinate can be overridden by specifying an additional parameter `xi_current` with the value of the reaction coordinate you wish to use for the recrossing factor calculation. An example of this is given below:

```
computeRecrossingFactor(
    dt = (0.0001,"ps"),
    equilibrationTime = (20,"ps"),
    childTrajectories = 100000,
    childSamplingTime = (2,"ps"),
    childrenPerSampling = 100,
    childEvolutionTime = (0.05,"ps"),
    xi_current = 1.016,
)
```

3.10 Compute the ring polymer rate coefficient

The final portion of the calculation is the determination of the bimolecular rate coefficient using the Bennett-Chandler method. This is done using a `computeRateCoefficient()` block, as shown below:

```
computeRateCoefficient()
```

The `computeRateCoefficient()` block does not require any parameters. It uses the position of the reaction coordinate from the most recently computed recrossing factor to compute the value of the centroid density quantum transition state theory (QTST) rate coefficient. If you have only done the potential of mean force calculation, you can still use this block to generate an estimate of the rate coefficient; in this case, a value of unity will be assumed for the recrossing factor and the centroid density QTST rate coefficient will be calculated at the top of the barrier.

CHAPTER

FOUR

EXPOSING AN EXTERNAL POTENTIAL TO PYTHON

The most time-consuming step of RPMD simulation (as well as of classical molecular dynamics) is the evaluation of the energy gradients. Usually, the analytic potential energy surface is provided in the form of a subroutine in a compiled language, such as Fortran. In this section, we demonstrate how to expose a global potential function defined as a Fortran code to Python, and, by extension, RPMDrate.

4.1 Define the Fortran interface

The first step is to define the functions that Python will invoke. The function `get_potential()` computes the potential and forces for a given configuration. The parameters for `get_potential()` are as follows:

- `q` - The $3 \times N_{\text{atoms}} \times N_{\text{beads}}$ array of coordinates of each bead for each atom in atomic units.
- `Natoms` - The number of atoms in the system.
- `Nbeads` - The number of ring polymer beads in the system.
- `V` - The `Nbeads` of potential energies for each bead in atomic units.
- `dVdq` - The $3 \times N_{\text{atoms}} \times N_{\text{beads}}$ array of computed energy gradients for each atom/bead combination in atomic units.
- `info` - An output flag; use a nonzero value to indicate that the coordinates are outside of the valid range of the potential energy function. This will generally cause the RPMD trajectory to be restarted using the previous known good coordinates.

A template for `get_potential()` is given below:

```
subroutine get_potential(q, Natoms, Nbeads, V, dVdq, info)

    implicit none
    integer, intent(in) :: Natoms
    integer, intent(in) :: Nbeads
    double precision, intent(in) :: q(3,Natoms,Nbeads)
    double precision, intent(out) :: V(Nbeads)
```

```
    double precision, intent(out) :: dVdq(Nbeads)
    integer, intent(out) :: info

    integer :: k

    info = 0

    ! Iterate over the beads, computing the potential and forces of each separately
    do k = 1, Nbeads

        ! Fill in the code for calling the Fortran potential function here.
        ! Use q(:,:,k) for the position
        ! Place the computed potential in V(k) and forces in dVdq(:,:,k)

    end do

end subroutine
```

Many potential energy functions require some parameter initialization (for instance, potential energy surfaces from the POTLIB[1] online library). In this case, a second function `initialize_potential()` that RPMDrate can call to perform the initialization should be prepared. A template for this function is given below:

```
subroutine initialize_potential()

    ! Fill in the code for initializing the Fortran potential function here.

end subroutine
```

4.2 Define the f2py wrapper module

The Fortran functions defined above are exposed to Python using the f2py[2] utility. This utility has been shipped with NumPy since 2007, and should therefore not require any additional setup if you are using a recent version of NumPy. Note that you will need a reasonably recent Fortran compiler, such as gfortran[3], g95[4], or ifort[5].

If the `get_potential()` and `initialize_potential()` Fortran subroutines have been created as shown in the previous section, then the following f2py module definition file should suffice:

```
python module PES
    interface
        subroutine initialize_potential()
        end subroutine initialize_potential
        subroutine get_potential(q,Natoms,Nbeads,V,dVdq,info)
            integer, intent(in) :: Natoms
```

[1] http://comp.chem.umn.edu/potlib/
[2] http://www.scipy.org/f2py
[3] http://gcc.gnu.org/fortran/
[4] http://www.g95.org/
[5] http://software.intel.com/en-us/articles/intel-compilers/

```
            integer, intent(in) :: Nbeads
            double precision dimension(1:3,1:Natoms,1:Nbeads), intent(in) :: q
            double precision dimension(1:Nbeads), intent(out) :: V
            double precision dimension(1:3,1:Natoms,1:Nbeads), intent(out) :: dVdq
            integer, intent(out) :: info
        end subroutine get_potential
    end interface
end python module PES
```

Place the above in a file called `PES.pyf` in the same directory as the RPMDrate input file and the other Fortran source files for your potential.

4.3 Compile the Python wrapper

All that remains is to perform the actual compilation. We assume that the external potential is given by two files:

- `PES.f` - The Fortran source file containing the potential, without any modifications for use by RPMDrate.
- `PES_wrapper.f90` - The Fortran source file containing the `get_potential()` and `initialize_potential()` Fortran subroutines that interface between RPMDrate and the potential function.

With these two files and the f2py module definition file `PES.pyf`, we can invoke the compilation using the following command:

```
$ f2py -c PES.f PES_wrapper.f90 PES.pyf
```

You will see a large amount of compiler messages scroll by, including possibly some warnings depending on how compliant your Fortran potential function is. If the compilation is successful, you should see a new file – `PES.so` on Mac/Linux/etc., `PES.pyd` on Windows – that represents the wrapped Fortran code. This file is ready for use by RPMDrate; this line from the input file tells RPMDrate about the `get_potential()` and `initialize_potential()` subroutines (see Define the potential energy surface):

```
from PES import get_potential, initialize_potential
```

CHAPTER

FIVE

RUNNING RPMDRATE

Once the input file and exposed the Fortran potential function to Python have been set up, everything is ready to run the RPMD calculation. To do this, invoke the main `rpmdrate.py` script, passing the input file, temperature, and number of beads as command-line arguments:

```
$ python rpmdrate.py examples/H+CH4/input.py Temp Nbeads
```

where `Temp` is the temperature and `Nbeads` is the number of ring polymer beads. RPMDrate is also designed to exploit multiprocessor systems. To specify the number of processors, use the `-p` flag. For example, the following indicates that eight processors are available for RPMDrate to use:

```
$ python rpmdrate.py examples/H+CH4/input.py Temp Nbeads 1 -p Nprocs
```

where `Nprocs` is the number of requested processors.

5.1 Browsing RPMDrate output files

RPMDrate generates a large number of output files for each calculation. These are sorted into subfolders `Temp`/`Nbeads` by temperature and number of beads, so that the same input file can be used by multiple RPMD calculations. RPMDrate will also use these output files for restarting incomplete jobs, to avoid repeating already-finished steps in the calculation. (Biased sampling and transmission coefficient calculations are the most time consuming steps. During these steps, the program creates a series of checkpoints which allow RPMDrate to be safely stopped at any point in its execution and then to be restarted later.)

1. Initial configurations for umbrella integration

The computed initial umbrella configurations are saved to `umbrella_configurations.dat` in the top-level directory, as these configurations can be used in subsequent calculations. This file contains the Cartesian coordinates in atomic units for each configuration.

2. Biased sampling

The computed values for the mean $\overline{\xi}$ and variance σ^2 of the reaction coordinate are saved to a set of files named `umbrella_sampling_*.dat` after each trajectory run, where `*` represents the position of center of the window along the reaction coordinate ξ.

3. Potential of mean force

The potential of mean force as a function of the reaction coordinate is saved to the file `potential_of_mean_force.dat`.

4. Transmission coefficient

The computed value of the transmission coefficient (recrossing factor) is saved to a file named `recrossing_factor_*.dat`, where `*` represents the value of the reaction coordinate ξ at which this factor was computed.

5. Ring polymer rate coefficient

The final value of the RPMD rate coefficient, along with a summary of the various intermediate values used to compute it, is saved to a file named `rate_coefficient_*.dat`, where `*` represents the value of the reaction coordinate ξ at which the transmission coefficient was computed. Note that the same value of the reaction coordinate ξ is used to obtain the centroid-density quantum transition state theory rate coefficient and the recrossing factor.