MAVARIC
Version 1.0
User Manual

Elliot Eklund

Cornell University
ece52@cornell.edu

Table of Contents

1 SECTION

Introduction

1.1 Overview

This user manual contains instructions on how to use the program MAVARIC: MApping VARiable Integration Code. MAVARIC is a program for running simple, one-dimensional Mapping Variable Ring Polymer Molecular Dynamics (MV-RPMD) calculations as described in *Ananth*[1]. It can be used to calculate position centroid auto-correlation functions of a general s-level system in which the user specifies the form of the potential energy surfaces and coupling. The position centroid auto-correlation function is calculated by propogating classical equations of motion (EOM) from the MV-RPMD Hamiltonian,

$$H = \sum_{\alpha=1}^{N} \left(\frac{P_\alpha^2}{2M} + \frac{M}{2\beta_N^2}(Q_\alpha - Q_{\alpha+1})^2 + V_0(Q_\alpha) + \frac{1}{\beta_N}(\mathbf{x}_\alpha \cdot \mathbf{x}_\alpha + \mathbf{p}_\alpha \cdot \mathbf{p}_\alpha) \right) - \frac{1}{\beta_N} \ln|\Theta| \qquad (1)$$

Here, $\{Q_\alpha\}$ and $\{P_\alpha\}$ are the set of nuclear position and momentum for each bead, respectively. N is the number of beads, $\beta = 1/kT$, and $\beta_N = \beta/N$. $\{\mathbf{x}_\alpha\}$ and $\{\mathbf{p}_\alpha\}$ are the set of electronic "position" and "momenta" for each bead, respectively. Each electronic variable is a vector of length s. Θ is defined as,

$$\Theta(\mathbf{Q}, \mathbf{x}, \mathbf{p}) = \text{Re}(\text{Tr}(\Gamma(\mathbf{Q}, \mathbf{x}, \mathbf{p}))) \qquad (2)$$

Γ is a matrix defined as,

$$\Gamma(\mathbf{Q},\mathbf{x},\mathbf{p}) = \prod_{\alpha=1}^{N} (C_\alpha(\mathbf{x}_\alpha,\mathbf{p}_\alpha) - \frac{1}{2}I)\mathcal{M}(Q_\alpha). \tag{3}$$

C_α, referred to as the "C Matrix", is defined as

$$C_\alpha(\mathbf{x}_\alpha,\mathbf{p}_\alpha) = (\mathbf{x}_\alpha + i\mathbf{p}_\alpha) \otimes (\mathbf{x}_\alpha - i\mathbf{p}_\alpha)^T \tag{4}$$

And finally, $\mathcal{M}(R_\alpha)$, referred to as the "M Matrix", is defined as

$$\mathcal{M}_{n,m}(Q_\alpha) = \begin{cases} e^{-\beta_N V_{nn}(Q_\alpha)} & n = m \\ -\beta_N V_{n,m}(Q_\alpha) e^{-\beta_N V_{nn}(Q_\alpha)} & n \neq m \end{cases} \tag{5}$$

It is not the intention of this manual to review the theory behind MV-RPMD in detail. For further information regarding MV-RPMD, please read *Ananth*[1].

The EOM are propogated using a fourth-order Adams-Bashforth-Moulton (ABM) predictor corrector integration scheme[2]. The initial steps of the ABM integrator are found using a fourth-order Runge-Kutta integrator[2].

1.2 Using MAVARIC

MAVARIC is designed so that a molecular dynamics calculation can be split into three manageable phases. These phases are Monte Carlo, Sampling, and Dynamics. All three phases can be run sequentially, or broken up into pieces depending on the user's needs. In brief, we outline what each phase is meant to accomplish in the overall computation.

The first phase, Monte Carlo, is used to establish convergence parameters at equilibrium. This includes performing bead convergence and energy estimator convergence. Monte Carlo performs a standard Monte Carlo simulation using the Metropolis-Hastings algorithm for acceptance/rejection criteria[3].

The Sampling phase is used after convergence of equilibrium parameters has been established. During Sampling, the user samples decorrelated trajectories that will eventually be used to perform dynamics calculations. These samples are generated via Monte Carlo simulation. However, unlike the Monte Carlo phase, Sampling does not compute the energy estimator. As a result, Sampling is much faster at running a Monte Carlo simulation because it is able to skip the energy estimator calculation. The sampled trajectories can be histogramed to check the position centroid distribution.

The final phase is Dynamics. Once a sufficient number of trajectories have been sampled, we are ready to run a dynamics calculation. During this phase, we can monitor how well our trajectories are conserving energy. This useful for deciding an appropriate time step to be used by the integrator during the simulation. We are also able to calculate position centroid auto-correlation functions during the Dynamics phase. As mentioned above, the EOM are propogated with a fourth-order ABM integrator, and the initial steps are found with a fourth-order Runge-Kutta integrator.

In the following sections, we describe each of these phases in detail. This is done as a tutorial that works through all the steps necessary to calculate the position centroid auto-correlation function of a model system.

1.3 Tutorial Model System

The model system we use in the tutorial is Model 1 of *Ananth*[1]. This is a two state system with state specific potential energy surfaces

$$V_{11}(Q) = aQ + c \tag{6}$$

$$V_{22}(Q) = -aQ \tag{7}$$

and off-diagonal coupling elements $V_{12} = V_{21} = \Delta$. The state independent potential is

$$V_0(R) = \frac{1}{2}kQ^2. \tag{8}$$

The values of the parameters are $c = 0$, $k = a = 1$, and $\Delta = 0.1$. Other parameters that are needed to fully define the Hamiltonian are $M = \beta = 1$ and $N = 4$. All units are in a.u. except for N, which is unitless. Throughout the rest of this manual, we will simply refer to this model as "Model 1".

Upon downloading MAVARIC, the potential energy surfaces and coupling will be configured to Model 1. The input files will also be configured for these parameters, however, you will be asked to change these parameters throughout the tutorial. At the end of the tutorial, in Section 5, we will explain how to change the potential energy surfaces to your system of interest.

1.4 Input Files

MAVARIC contains five input files through which calculations are requested and certain model parameters can be specified. These files are found in the directory *InputFiles.* It is crucial that the number of lines in each file never change. Adding or deleting a line will cause the program to crash. In addition, the format of each line should never change. The format should always be a descriptive phrase, followed by a colon (:), followed by a positive number. There are only primitive error handling measures in place to detect deviations from this format - proceed with caution!

Many of the parameters used to request a calculation take binary input. These parameters use `1` to request the specified calculation, and `0` otherwise. Here is a list of all binary parameters used by MAVARIC. The parameters are listed under the file they can be found in.

MonteCarlo

```
Run MC
Save PSV
Save MC Data
Read PSV
Read MC Data
```

Sampling

```
Run Sampling
Save Sampled Trajectories
Histogram Positions
Read PSV
```

Dynamics

```
Run Dynamics
Read Trajectories
Check Energy Conservation
```

Please keep these in mind when working with MAVARIC.

1.5 Compiling and Running MAVARIC

MAVARIC is intended for Linux and Mac OS X operating systems. MAVARIC is written in C++ and will only work with compilers compatible with C++ 11 or higher. MAVARIC provides a makefile for compiling the code which can be found in the directory *MAVARIC*. The default compiler in the makefile is set to Intel's distribution of the C++ compiler. To change the compiler, modify the second line of the makefile. Just make sure it is C++ 11 compatible! To compile the code, enter `make` into the terminal.

This version of MAVARIC is not parallelized. In the future, a version of MAVARIC wirtten in parallel will be released.

In Section 5 we will discuss how to configure user specified potential energy surfaces. Doing so will require the user to modify C++ code. Every time an edit is made to a file containing C++ code, MAVARIC must be re-compiled! The safest way to re-compile MAVARIC is navigate to the directory *MAVARIC* and enter `clean` into the terminal. After `clean` has finished executing, enter `make`. If successful, you should see no error messages. Forgetting to re-compile after altering a MAVARIC C++ file means that any changes made won't actually be implemented during run time.

To run MAVARIC, enter `./mavaric` into the terminal. An output message will always appear on the terminal with information regarding whether or not MAVARIC successfully completed the requested calculations.

2

SECTION

Monte Carlo

The Monte Carlo phase is used to determine equilibrium convergence parameters. You can request calculations for the Monte Carlo phase using the input file *MonteCarlo*, located in the *InputFiles* directory. After downloading MAVARIC, the file *MonteCarlo* should contain exactly seven lines in the following format:

MonteCarlo

```
Run MC:0
MC Moves:1e5
Estimator Rate:1e2
Save PSV:0
Save MC Data:0
Read PSV:0
Read MC Data:0
```

In the directory *InputFiles*, you should also find a file called *SystemParameters* and a file called *ElecParameters*. *SystemParameters* contains parameters for specifying the system model. *ElecParameters* contains parameters that specify the electronic states of our model. After download, *SystemParameters* should contain exactly four lines in the following format:

SystemParameters

```
Mass:1
Beads:4
Temperature:1
MC Step Size:1
```

ElecParameters should contain exactly two lines in the following format:

ElecParameters

```
States:2
MC step size:1
```

The parameters `Mass`, `Beads`, and `Temperature` correspond to M, N, and T in $\beta = 1/T$ of Eq. 1 in Section 1. Both `M` and `T` are in a.u. `States` is the number of electronic states in our model. Both files contain a parameter called `MC Step Size` which will be discussed below. Its units are also given in atomic units.

Let's start the tutorial by changing `Run MC:0` to `Run MC:1` in the *MonteCarlo* file. Run the code after you have made this change. If done successfully, you will see the following message:

```
Calculating...

Running Monte Carlo calculations ...

MonteCarlo Results:
System Acceptance Ratio:  25.8661
Electronic Acceptance Ratio:  22.9134
Average Energy:  0.619438
Successfully wrote energy_estimator file to Results.

Monte Carlo Simulation Run Time:  0.163655

Finished Calculations
```

Because Monte Carlo simulations are a stochastic process, your `System Acceptance Ratio`, `Electronic Acceptance Ratio`, and `Average Energy` will differ slightly from the example above. Additionally, `Monte Carlo Simulation Run Time` will be dependent on the machine executing MAVARIC.

Let's discuss this output. The lines `System Acceptance Ratio` and `Electronic Acceptance Ratio` tell us the percentage of moves that were successfully accepted for the system and electronic degrees of freedom, respectively. In this case, roughly 26% of the time, our system moves were accepted, and roughly 23% of the time, our electronic moves were accepted. Typically, we want both acceptance ratios to be around 50%. If we lower the parameter `MC Step Size` in the *SystemParameters* file, we should see the System Acceptance Ratio increase. Similarly, if we lower the parameter `MC Step Size` in the *ElecParameters* file, we should see the Electronic Acceptance Ratio increase.

Navigate to the *SystemParameters* file and change `MC Step Size` to `0.55`. Similarly, in *ElecParameters*, change `MC Step Size` to `0.48`. After making these changes, run MAVARIC. You should now find acceptance ratios close to those reported below:

```
System Acceptance Ratio:   49.7047
Electronic Acceptance Ratio:   49.4914
```

Once we have found good Monte Carlo step sizes, we can work on converging energy. In the directory *Results* is a file called *energy_estimator*. This file contains the energy estimator computed throughout the most recently requested Monte Carlo simulation. Every time the Monte Carlo phase is executed, *energy_estimator* is overwritten with the latest results. In Figure 1, we plot *energy_estimator*.

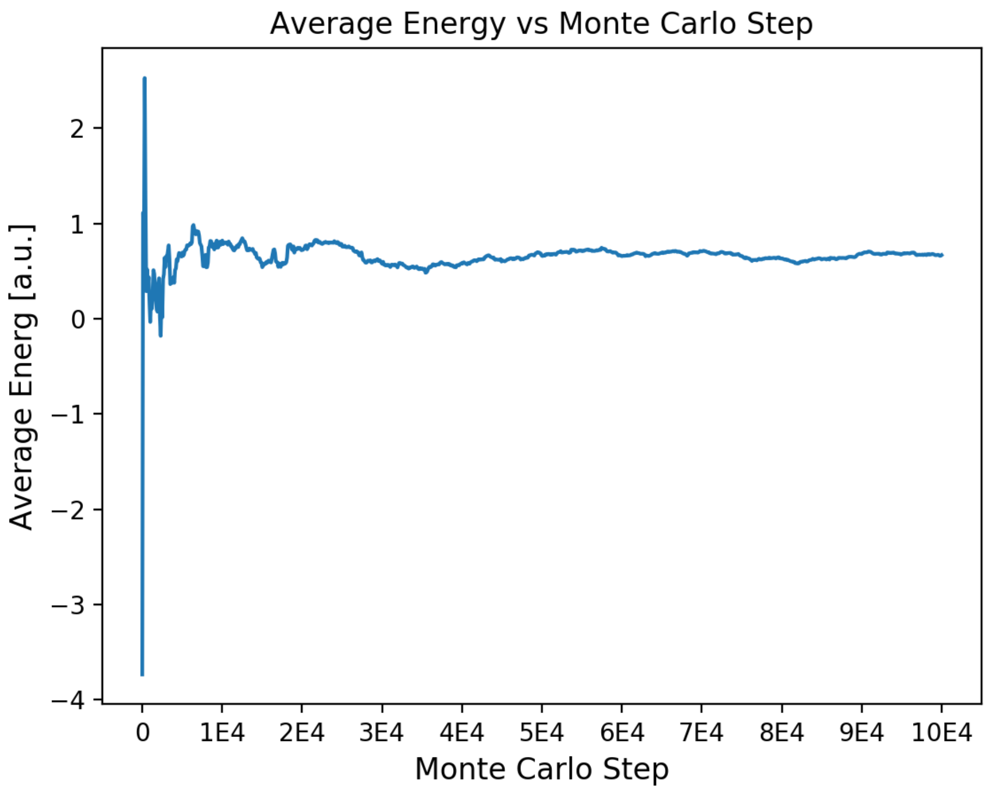

Figure 1: Average Energy throughout a short Monte Carlo simulation.

As one would expect, the average energy fluctuates rapidly at the beginning of the simulation, and gradually flattens toward the end as our system approaches equilibrium. Model 1, and most other models you will work with, take far more than 10^5 Monte Carlo steps for energy to converge. To change the number of Monte Carlo steps in a given simulation, modify `MC Moves` in the file *MonteCarlo*. For now, increase the value of `MC Moves` to 10^7. The next time we run MAVARIC, the Monte Carlo simulation will take 10^7 steps.

Typically, we do not need to write the average energy to file after every Monte Carlo step. For example, writing the average energy after every 10^3 steps is probably high enough resolution to understand how the average energy is converging. We can control how often the average energy is written to file with the parameter `Estimator Rate`. Change the value of `Estimator Rate` to 10^3. Now when we run the calculation using `MC steps:1e7` and `Estimator Rate:1e3`, we will write out the average energy after every 10^3 steps, giving us a total of 10^4 points being written to file. After making these changes, re-run MAVARIC and plot *energy_estimator*. It should look similar to Figure 2 below.

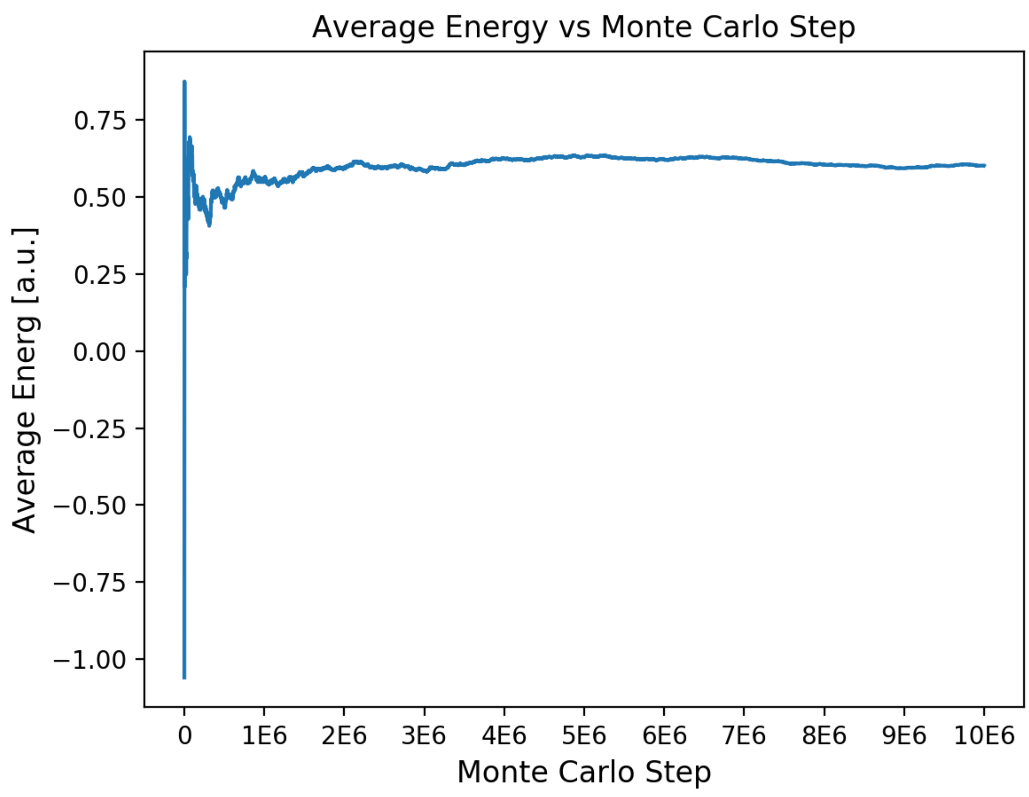

Figure 2: Average Energy throughout a longer Monte Carlo simulation.

Notice that Figure 2 is much flatter than Figure 1 toward the end of the simulation. Now we can be more confident that energy is converging.

After we run a Monte Carlo simulation, our system should be in a state that is much closer to equilibrium in comparison to its initial state. Unfortunately, every time we re-run a Monte Carlo simulation, our initial state is randomly generated, and we lose information about the equilibrium state gained from the previous simulation. Instead of starting from scratch each time a Monte Carlo simulation is run, MAVARIC allows the user to save the final state of the system following a Monte Carlo simulation, and use it as a starting point for later simulations.

To save the information from your Monte Carlo calculation, open the file *MonteCarlo* and change `Save PSV` to `1` and `Save MV Data` to `1` as well. Make these changes, then re-run the simulation. You should find the following print statement:

```
Calculating...

Running Monte Carlo calculations ...

MonteCarlo Results:
System Acceptance Ratio:  49.9421
Electronic Acceptance Ratio:  49.9112
Average Energy:  0.575902
Successfully wrote energy_estimator file to Results.
Successfully saved PSV to Results.
Successfully saved MC data to Results.

Monte Carlo Simulation Run Time:  14.4884

Finished Calculations
```

This indicates that the phase space variables (PSV) and other Monte Carlo data were successfully saved after the simulation. The directory *Results* will now contain two new files called *PSV* and *mc_data*. These files contain the saved data. Figure 3 below is a plot of the average energy for the simulation we just ran. We will say more about it in a moment.

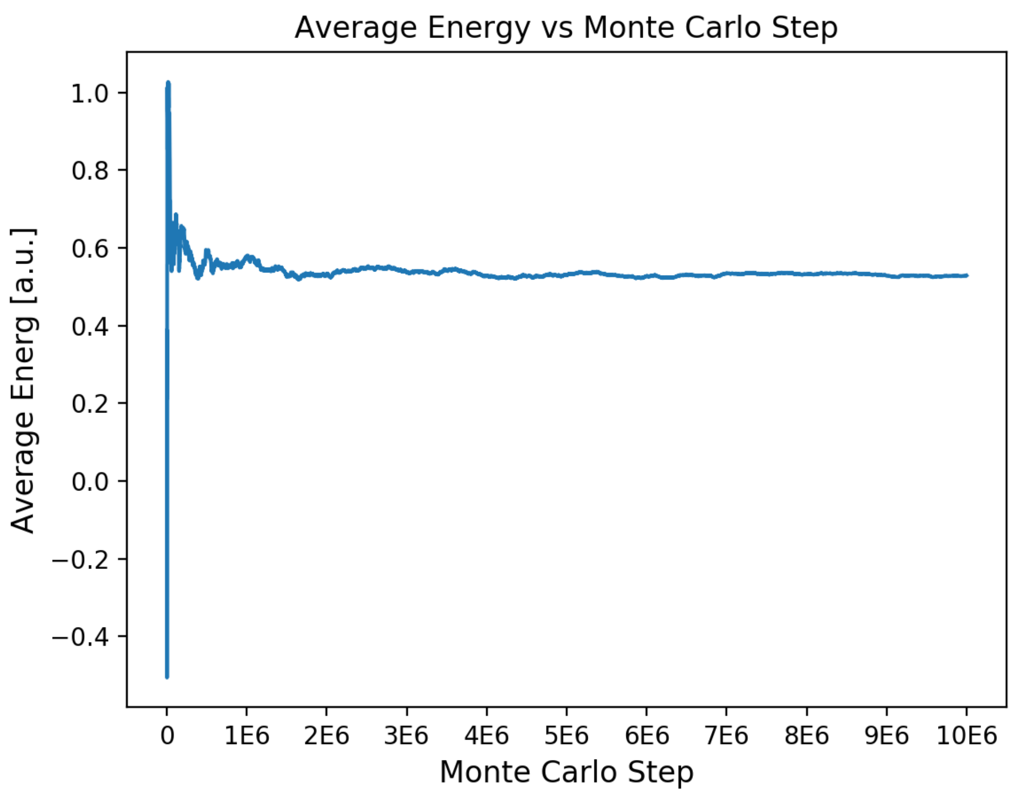

Figure 3: Monte Carlo simulation start from random initial state.

To read in data that has been saved , change `Read PSV` to `1` and `Read MC Data` to `1` in the *MonteCarlo* file. Make these changes, then re-run the MAVARIC. If done successfully, you should see the following message:

```
Calculating...

MonteCarlo Results:
Successfully read PSV file from Results.
Successfully read MC datat from Results.
System Acceptance Ratio:  49.9243
Electronic Acceptance Ratio:  49.9717
Average Energy:  0.550436
Successfully wrote energy_estimator file to Results.
Successfully saved PSV to Results.
Successfully saved MC data to Results.

Monte Carlo Simulation Run Time:  14.7865
```

This indicates that the files were successfully read. Let's look at the energy estimator plot for the simulation that we just ran in Figure 4 below.

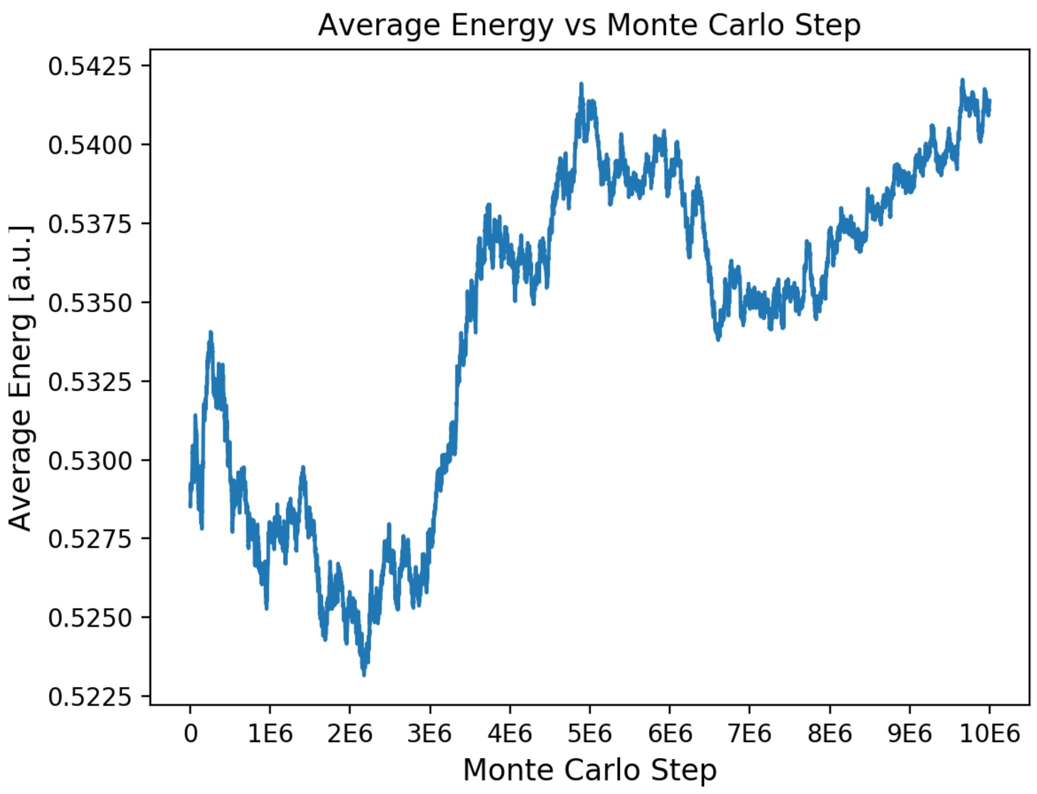

Figure 4: Monte Carlo simulation starting from saved initial state.

The fluctuations in Figure 4 are much smaller than Figure 3 and we no longer see massive fluctuations at the start of the simulation. This is because after running the simulation that generated Figure 3, we saved the final phase space variables and energy. Then, when we ran the simulation that produced Figure 4, we first read in those saved phase space variables and energy, then started the simulation. This means our starting point was much closer to equilibrium for the simulation that produced Figure 4 in comparison to Figure 3.

This demonstrates how running a Monte Carlo simulation can be broken up into chunks. If you find that your Monte Carlo simulation was not long enough to converge energy, you can simply start another simulation at the point where you left off, running the calculation for longer this time. Be careful when reading in files. If no files were saved previously, then trying to read in *PSV* and *mc_data* will cause an error.

Once you are satisfied with how energy is converging, you can increase the number of beads in the *SystemParameters* file and repeat the process outlined in this section. This is typically done for a range of bead numbers to establish bead convergence.

3

SECTION

Sampling

The Sampling phase is used to generate a distribution of trajectories that will eventually be used in the Dynamics Phase. The trajectories are sampled using a Monte Carlo simulation, however, the energy estimator is not calculated during Sampling because it is assumed that the average energy has already been converged. The step size of system and electronic moves are set by the parameter `MC Step Size` in the input files *SystemParameters* and *ElecParameters*, respectively. Requesting calculations for Sampling is done via the input file *Sampling*. After installation, the *Sampling* file should contain exactly seven lines in the following format:

Sampling

```
Run Sampling:0
Number of Trajectories:1e4
Decorrelation Length:1e2
Save Sampled Trajectories:0
Histogram Positions:0
Number of Bins:300
Read PSV:0
```

To use Sampling, simply change `Run Sampling` to `1`. In the current example, 10^4 trajectories will be sampled. Between each new trajectory that is sampled, we perform an additional 10^2 Monte Carlo steps to ensure the trajectories are sufficiently decorrelated. To change the number of sampled trajectories or decorrelation length, modify the parameters `Number of Trajectories` and `Decorrelation Length`, respectively.

Let's try running a Sampling calculation. In the *MonteCarlo* file, configure the parameters in the following way:

MonteCarlo

```
Run MC:1
MC Moves:1e6
Estimator Rate:1e3
Save PSV:0
Save MC Data:0
Read PSV:0
Read MC Data:0
```

In the *Sampling* file, only change `Run Sampling` to `1`. After making these changes, run the code. If everything worked correctly, you should see the following output:

```
Calculating...

Running Monte Carlo calculations ...

MonteCarlo Results:
System Acceptance Ratio:  50.0492
Electronic Acceptance Ratio:  49.9594
Average Energy:  0.6922
Successfully wrote energy_estimator file to Results.

Monte Carlo Simulation Run Time:  1.4755

Running Sampling ...

Sampling Run Time:  0.509911

Finished Calculations
```

MAVARIC began the program by running a Monte Carlo simulation, producing the corresponding Monte Carlo output messages when it finished. Then, it entered a Sampling calculation, also producing an output message upon completion.

Let's run another Sampling calculation, but this time, we will generate a histogram of the sampled position centroids. To do this, open the *Sampling* file and change the `Histogram Positions` parameters to `1`. Additionally, change `Number of Trajectories` to `1e5`. We can control the number of bins our histogram uses by modifying the `Number of Bins` parameter. For now, we will leave it at 300 bins. After making these changes, re-run the simulation. If done correctly, the Sampling output message should now include a histogram statistics report that looks like the following:

```
-- Histogram Statistics --
Mode:  -0.342636
Average:  -0.00551759
Standard Deviation:  1.24748
Skew:  0.0284287
```

To view the histogram, plot the file *pos_histogram* in the *Results* directory. Figure 5 below is an example of what the histogram should look like. Of course, increasing the number of trajectories will produce a smoother histogram.

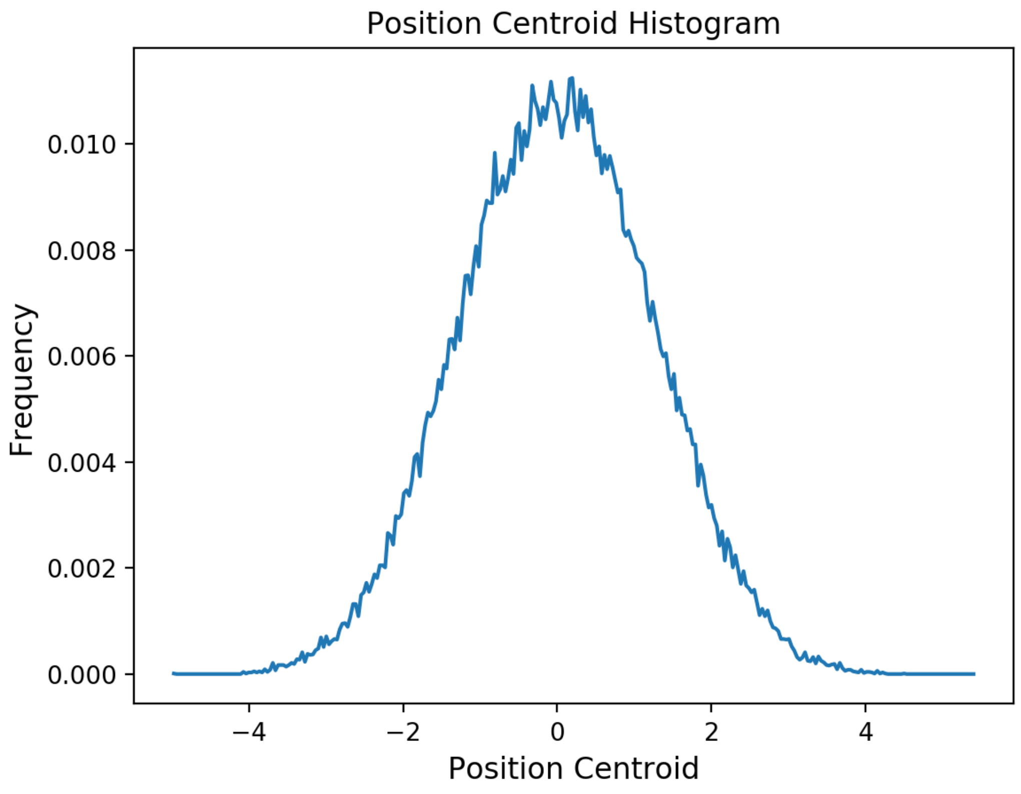

Figure 5: Histogram using 300 bins with 10^5 Trajectories

The Sampling calculation we have done so far have been have been after first running a Monte Carlo calculation. The final state of our system after the Monte Carlo calculation is what Sampling uses as its initial state. If equilibrium was successfully established during Monte Carlo, than all our sampled trajectories produced during Sampling should follow the equilibrium Boltzmann distribution.

Instead of starting with a Monte Carlo simulation, we can read in phase space variables that have been saved to the *PSV* file in the *Results* directory. This way, we do not need to run Sampling immediately after Monte Carlo. If we have saved our phase space variables after a Monte Carlo simulation, we can read them back in at a later time and start MAVARIC at Sampling. To read in saved phase space variables, change `Read PSV` to `1`.

In addition, we can save trajectories that we have sampled. To do this, change `Save Sampled Trajectories` to `1`. Now, after we run a sampling calculation, all the trajectories will be saved to the directory *Results/Trajectories*.

As a demonstration, run a Monte Carlo simulation and select to save both PSV and mc_data. Do this without running Sampling; i.e., set `Run Sampling` to `0`. If done successfully, you should see the following message:

```
Calculating...

Running Monte Carlo calculations ...

MonteCarlo Results:
System Acceptance Ratio:  49.8436
Electronic Acceptance Ratio:  50.1265
Average Energy:  0.622538
Successfully wrote energy_estimator file to
Results.
Successfully saved PSV to Results.
Successfully saved MC data to Results.

Monte Carlo Simulation Run Time:  1.46195

Finished Calculations
```

Now, run Sampling without doing a Monte Carlo simulation first; i.e. set `Run MC` to `0` and `Run Sampling` to `1`. In addition, set `Save Sampled Trajectories` and `Read PSV` both to `1`. If done successfully, you will see the following message:

```
Calculating...

Running Sampling ...
Successfully read PSV from Results.

-- Histogram Statistics --
Mode:  0.0924216
Average:  0.000271159
Standard Deviation:  1.24538
Skew:  -0.014498

Successfully saved sampled trajectories to Results/Trajectories.

Sampling Run Time:  11.1057

Finished Calculations
```

The directory *Results/Trajectories* should now contain four files titled *P, Q, xelec,* and *pelec.* These are the sampled trajectories that you have just saved. Saving sampled trajectories allows the user to read them in later when working with the Dynamics phase. That way, work can be done in chunks and one does not need to go directly from Sampling to Dynamics.

4

SECTION

Dynamics

The Dynamics phase calculates the position centroid auto-correlation function within the MV-RPMD framework. EOM are propogated using a 4^{th} order Adams-Bashforth-Moulton predictor-corrector integration scheme. Requesting calculations for Dynamics is done via the input file *Dynamics*. After download, the *Dynamics* file should contain exactly sixe lines in the following format:

Dynamics

```
Run Dynamics:0
Run Time:5
Time Step:0.01
Read Trajectories:0
Check Energy Conservation:0
Conservation Tolerance:0.1
```

To run Dynamics, change `Run Dynamics` to `1`. The parameter `Run Time` specifies the total time in atomic units for Dynamics to run. `Time Step` specifies the size of the time step, dt, used by the integrator.

As a first example, configure the *MonteCarlo*, *Sampling*, and *Dynamics* input files to matching the following.

Monte Carlo

```
Run MC:1
MC Moves:1e7
Estimator Rate:1e4
Save PSV:0
Save MC Data:0
Read PSV:0
Read MC Data:0
```

Sampling

```
Run Sampling:1
Number of Trajectories:1e4
Decorrelation Length:1e2
Save Sampled Trajectories:0
Histogram Positions:0
Number of Bins:300
Read PSV:0
```

Dynamics

```
Run Dynamics:1
Run Time:5
Time Step:0.005
Read Trajectories:0
Check Energy Conservation:0
Conservation Tolerance:0.1
```

Run MAVARIC with this set of configurations. You should see the following output after MAVARIC finishes:

```
Calculating...

Running Monte Carlo calculations ...

MonteCarlo Results:
System Acceptance Ratio:  49.9658
Electronic Acceptance Ratio:  50.0111
Average Energy:  0.562923
Successfully wrote energy_estimator file to Results.

Monte Carlo Simulation Run Time:  16.0427

Running Sampling ...

Sampling Run Time:  0.570696

Running Dynamics ...
Successfully wrote pos_auto_corr to Results.
Dynamics Run Time:  65.9003

Finished Calculations
```

After the calculation has finished, the *Results* directory will contain a file called *pos_auto_corr*. This file contains the position centroid auto-correlation function. In figure 6 Below is a plot of the auto-correlation function for the simulation we just ran.

To run a calculation using more or less trajectories, change the parameter `Number of Trajectories` in *Sampling*.

MAVARIC provides a way to monitor how well trajectories are conserving energy throughout the course of a calculation. If we change `Test Energy Conservation` to `1` in the *Dynamics* file, MAVARIC will now check how well energy is being conserved. The parameter `Conservation Tolerance` allows us to specify the tolerance to which we want each trajectory to conserve energy. For example, when `Conservation Tolerance` is set to `0.1`, this means a trajectory will be considered "broken" (not conserving energy) if its initial energy and current energy differ by more than 10%. Once a trajectory is broken, MAVARIC will stop propagating it and move on to the next trajectory. At the end of the Dynamics calculation, the percentage of broken trajectories is reported.

As a demonstration, run MAVARIC using the same input file configurations as previously, but with `Test Energy Conservation` set to `1`. If done correctly, the Dy-

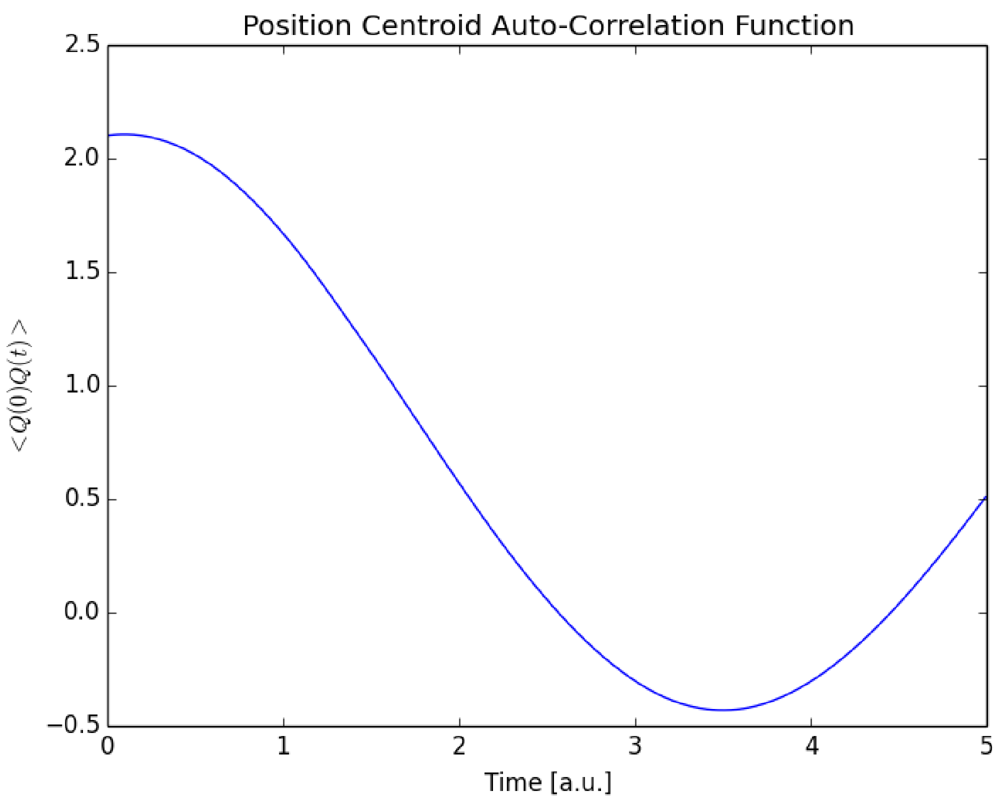

Figure 6: Position Centroid Auto-Correlation function using 10^4 trajectories.

namics output message should be similar to the following:

```
Running Dynamics ...

Percentage of Trajectories Broken:  30
Dynamics Run Time:  53.209
```

This indicates that 30% of our trajectories failed to conserve energy to within 10% of their initial energy. A proper Dynamics simulation should aim for a much lower percentage of broken trajectories. Lowering `Time Step` will decrease the number of trajectories that fail to conserve energy.

Finally, if sampled trajectories were saved during the Sampling phase, they can be read in during the Dynamics phase. To do this, change `Read Trajectories` to `1`. This will read in trajectories stored in the directors *Results/Trajectories*. If no trajectories have been saved, then trying to read them in will cause an error. Being able to read in trajectories means we do not need to run MonteCarlo, Sampling, and Dynamics in sequence. If sampled trajectories have been saved, then we can skip MonteCarlo and Sampling, and jump straight to Dynamics.

5

SECTION

Potentials

So far, this tutorial has only worked with one model that was pre-configured upon download. To change that model, you will need to change the potential energy surfaces and coupling. Making these changes is done in the file `Potentials.h` which is found in the directory *Potentials*. `Potentials.h` is a special file type called a C++ header file. This is the only C++ file a user who wishes to use MAVARIC as a black box device will have to worry about modifying.

`Potentials.h` makes use of six functions that need to be edited to fully specify a model. Throughout this section, we will discuss those functions in detail using Model I as an example. At the end of the section, we will consider a more complicated model and the changes that would need to be made to implement it.

Before discussing the different functions, we need to become somewhat familiar with two important C++ features that `Potentials.h` makes use of. These are macros and *valarrays*.

5.1 Macros

Typically, our potential energy surfaces and couplings contain parameters that need to be specified by the user. For example, if our system independent potential energy is the following,

$$V_0(Q) = \frac{1}{2}kQ^2, \tag{9}$$

then we need to be able to specify the parameter k in `Potentials.h`. We specify parameters using a C++ macro. The best way to understand a macro is to see an example. Toward the top of `Potentials.h` are the following three lines:

```
#define DELTA 0.1
#define A 1.0
#define D 0.0
```

Each of these defines a macro. Any new macro that you define must be placed before the line `struct Potentials{` . The general format for defining a macro is `#define NAME value`. It is common practice for the name of the macro to be in all caps. Once the macro is defined, you can use it anywhere in `Potentials.h` by typing its name. You should think of using a macro the same way you would a regular variable.

The only parameters that do not need to be defined as macros are the number of beads, number of electronic states, the product of the number of beads with number of electronic states, and mass. This is because `Potentials.h` always has access to these variables. They are represented by the following variables

```
int num_beads
int num_states
int elec_size
double mass
```

which are defined in the lines immediately following `struct Potentials{` . They can be used in any of the functions inside `Potentials.h`.

5.2 Valarray

The standard vector and array classes in C++ lack support for many vector features that are convenient when working with mathematical operations. There is a lesser known C++ vector class called *valarray* which does support these features. `Potentials.h` makes use of *valarray* when working with a vector of bead positions. We will now provide a brief discussion of how to work with a *valarray*, however, more information can be found at `http://en.cppreference.com/w/cpp/numeric/valarray`.

Without using *valarray*, mathematical operations on vectors in C++ would have to be performed with for loops. For example, element-wise multiplication between two vectors `v` and `w` would involve looping over all the elements in `v` and `w` and multiplying them one-by-one. If the two vectors are *valarrays*, we can make use of the element-wise multiplication function that *valarray* supports. The syntax for element-wise multiplication is `v * w`. As another example, let's consider summing all the elements in a vector. The *valarray* function for this operation is called `sum()`. To sum the elements of a *valarray* v, we simply type `v.sum()`.

In general, if there is a mathematical operation you want to perform over all the elements of a vector, *valarray* should have a function for doing so. As a final example, let's compute the dot product between `v` and `w`. To do this, we need to combine two operations, element-wise multiplication and summation. The syntax we would use is `(v * w).sum()`.

Vallary also supports array slicing - the ability to access chunks of an array at one time. For example, if we have an *valarray* `x` with ten elements, we can access the first five elements using the syntax `x[std::slice(0,5,1)]`. This syntax means starting at position `0`, grab the first `5` elements in intervals of `1`. In general, the syntax is `x[std::slice(a,b,n)]`. This means starting at position `a`, return the first `b` elements in intervals of `n`.

Now that we have covered the two main data types used by `Potentials.h`, we will turn to discussing the six functions that `Potentials.h` uses.

5.3 State Independent Potential

The state independent potential energy of a model is defined in the function `V0`. After download, `V0` should look like the following:

```
inline double V0(const std::valarray<double> Q){
    return 0.5*(Q*Q).sum();
}
```

Mathematically, `V0` corresponds to the term $\sum_{\alpha=1}^{N} V_0(Q_\alpha)$ in the MV-RPMD Hamiltonian. `V0` takes a *valarray* of bead positions as its argument, `Q`, and returns a single value, the energy corresponding $\sum_{\alpha=1}^{N} V_0(Q_\alpha)$. In Model I, V_0 is a harmonic oscillator, which means `V0` returns $\sum_{\alpha=1}^{N} \frac{1}{2} Q_\alpha^2$.

5.4 Gradient of State Independent Potential

The gradient of the state independent potential energy with respect to bead position is defined in the function dV0_dQ. After download, dV0_dQ should look like the following:

```
inline std::valarray<double> dV0_dQ
  (const std::valarray<double> Q){
     return Q;
}
```

Mathematically, dV0_dQ corresponds to $\nabla_{\mathbf{Q}} V_0$. We need this term to calculate the forces used in the integration scheme. dV0_dQ takes a *valarray* of bead positions as its argument, Q, and returns a *valarray* corresponding to $\nabla_{\mathbf{Q}} V_0$. The elements of the returned *valarray* are $(\frac{\partial V_0(Q_1)}{\partial Q_1}, \frac{\partial V_0(Q_2)}{\partial Q_2}, \ldots, \frac{\partial V_0(Q_N)}{\partial Q_N})$.

In Model I, V0 is a harmonic oscillator. Mathematically, its derivative with respect to a particular bead's position is $\frac{\partial V_0(Q_\alpha)}{\partial Q_\alpha} = Q_\alpha$. The *valarray* returned by dV0_dQ will be $(Q_1, Q_2, \ldots, Q_N)$.

5.5 State Specific Potential Energy Surfaces

The state specific potential energy surfaces are defined in the the function `Velec`. After download, `Velec` should look like the following:

```
inline void Velec(const std::valarray<double> &Q,
   std::valarray<double> &Vout){

    //state 1
    Vout[std::slice(0,num_beads,1)] = A*Q + D;

    //state 1
    Vout[std::slice(num_beads,num_beads,1)] = -A*Q;
}
```

Mathematically, `Velec` corresponds to the diagonal $V_{nn}(Q_\alpha)$ elements of the M Matrix (Eq 4 of Section 1). `Velec` takes two *valarrays* as arguments. The first argument, `Q`, is a *valarray* of bead positions: $(Q_1, Q_2, \ldots, Q_N)$. The second argument, `Vout`, corresponds to the evaluation of V_{nn} over $\mathbf{Q}$. `Vout` is a *valarray* of length `elec_size`. In Model I, the state specific potential energy surfaces are

$$V_{11}(Q_\alpha) = aQ_\alpha + d \tag{10}$$

$$V_{22}(Q_\alpha) = -aQ_\alpha \tag{11}$$

`Velec` needs to evaluate both V_{11} and V_{22} over every element of $(Q_1, Q_2, \ldots, Q_N)$. Let's think about how many computations we will need to do during this process. The number of beads in our system is `num_beads`, and the number of states is `num_states`. This means we need to perform `num_beads * num_states` number of computations. V_{11} will be computed `num_beads` times for each Q_α, and V_{22} will be computed `num_beads` times for each Q_α as well.

The first `num_beads` calculations will be to evaluate $V_{11}(Q_\alpha)$ over all `num_beads` positions and store the result in `Vout`. This means the first `num_beads` elements of `Vout` are now $(V_{11}(Q_1), V_{11}(Q_2), \ldots, V_{11}(Q_N))$.

Next, we evaluate state $V_{22}(Q_\alpha)$ over all `num_beads` positions and store the result in `Vout`. Similarly, the second `num_beads` elements of `Vout` are now $(V_{22}(Q_1), V_{22}(Q_2), \ldots, V_{22}(Q_N))$. In total, `Vout` will contain the elements $(V_{11}(Q_1), V_{11}(Q_2), \ldots, V_{11}(Q_N), V_{22}(Q_1), V_{22}(Q_2), \ldots, V_{22}(Q_N))$.

`Velec` uses *valarray's* slicing features to store elements in `Vout`. To access the first `num_beads` elements of `Vout`, we write `Vout[std::slice(0,num_beads,1)]`. To access the next `num_beads` elements. we write
`Vout[std::slice(num_beads,num_beads,1)]`.

5.6 Gradient of State Specific Potential Energy Surfaces

The gradient of the state specific potential energy surfaces with respect to bead position is defined in the function `dVelec`. After download, `dVelec` should look like the following:

```
inline void dVelec(const std::valarray<double> &Q,
   std::valarray<double> &Vout){

    //state 1
    Vout[std::slice(0,num_beads,1)] = A;

    //state 1
    Vout[std::slice(num_beads,num_beads,1)] = -A;
}
```

Mathematically, `dVelec` corresponds to $(\nabla_{\mathbf{Q}} V_{11}, \nabla_{\mathbf{Q}} V_{22})$. The layout of the function is identical to `Velec`, except the elements of `Vout` are now $\frac{\partial V_{nn}(Q_\alpha)}{\partial Q_\alpha}$. For Model I, we have

$$\frac{\partial V_{11}(Q_\alpha)}{\partial Q_\alpha} = a \tag{12}$$

$$\frac{\partial V_{22}(Q_\alpha)}{\partial Q_\alpha} = -a \tag{13}$$

The elements of `Vout` are $(V'_{11}(Q_1), V'_{11}(Q_2), \ldots, V'_{11}(Q_N), V'_{22}(Q_1), V'_{22}(Q_2), \ldots, V'_{22}(Q_N))$.

5.7 Potential Energy Coupling

The potential energy coupling is defined in the function `V_couple`. After download, `V_couple` should look like the following:

```
inline double V_couple(const double &Q, int state1, int state2)
{
     return DELTA;
}
```

Mathematically, `V_couple` corresponds to the off-diagonal potential energy coupling, $V_{n,m}(Q_\alpha)$, where $n \neq m$. This function takes three arguments: `Q`, `state1`, and `state2`. The first argument, `Q`, is the position of a single bead, Q_α. Note that this is not a *valarray*. The next two arguments indicate which two states we are calculating the coupling element between. For example, `state1 = 0` and `state2 = 1` would correspond to the element $V_{1,2}(Q_\alpha)$. Notice that the C++ syntax for the states and the indices of the mathematical description differ by one. This is because C++ starts its indexing at zero! Keep this in mind when working with `Potentials.h`.

In Model I, we have a two state system with constant coupling. As a consequence, we do not need to worry about which states we are computing the coupling between since they all have the same coupling. Below, we will consider a case where coupling is not constant and does depend on which states we are considering. In Model I, the coupling term `DELTA` is defined as a macro.

5.8 Derivative of Potential Energy Coupling

The derivative of the potential energy coupling with respect to bead position is defined in the function `dV_couple`. Following download, `dV_couple` should look like the code below:

```
inline double dV_couple(const double &Q, int state1, int state2)
{
     return 0;
}
```

Mathematically, `dV_couple` corresponds to $\frac{\partial V_{n,m}(Q_\alpha)}{\partial Q_\alpha}$, where $n \neq m$. This function takes three arguments, which are the same as arguments as `V_couple`: `Q`, `state1`, and `state2`. The layout of `dV_couple` is identical to `V_couple`, except now we are dealing with derivatives. In Model I, which uses constant coupling, `dV_couple` returns 0.

It is important to recognize that `dV_couple` is returning a number and not a *valarray*. For reasons that are not important to using MAVARIC, it is easier to work with the derivative of the potential energy coupling with respect to a single bead rather than the entire gradient.

5.9 Complicated Model

Let's walk through one more example using a significantly more complicated model. We will use a three state model with non-constant coupling. Our state independent potential is,

$$V_0(Q_\alpha) = c_1 Q_\alpha^4 + c_2 Q_\alpha^2. \tag{14}$$

The three state dependent terms are,

$$V_{11}(Q_\alpha) = D_1\left(1 - e^{-B_1(Q_\alpha - R_1)}\right)^2 \tag{15}$$
$$V_{22}(Q_\alpha) = D_2\left(1 - e^{-B_2(Q_\alpha - R_2)}\right)^2 \tag{16}$$
$$V_{33}(Q_\alpha) = D_3\left(1 - e^{-B_3(Q_\alpha - R_3)}\right)^2 \tag{17}$$
$$\tag{18}$$

Finally, the coupling terms are

$$V_{12}(Q_\alpha) = V_{21}(Q_\alpha) = A_{12} e^{-g_{12}(Q_\alpha - Q_{12})^2} \tag{19}$$
$$V_{13}(Q_\alpha) = V_{31}(Q_\alpha) = A_{13} e^{-g_{13}(Q_\alpha - Q_{13})^2} \tag{20}$$
$$V_{23}(Q_\alpha) = V_{32}(Q_\alpha) = A_{23} e^{-g_{23}(Q_\alpha - Q_{23})^2} \tag{21}$$
$$\tag{22}$$

Before touching any code, we need to compute the necessary derivatives. The derivative of the state independent potential is,

$$\frac{\partial}{\partial Q_\alpha} V_0(Q_\alpha) = 4c_1 Q_\alpha^3 + 2c_2 Q_\alpha. \tag{23}$$

The derivatives of the state dependent terms are,

$$\frac{\partial}{\partial Q_\alpha} V_{11}(Q_\alpha) = -2B_1 D_1\, e^{B_1(R_1 - Q_\alpha)}\left(e^{B_1(R_1 - Q)} - 1\right) \tag{24}$$
$$\frac{\partial}{\partial Q_\alpha} V_{22}(Q_\alpha) = -2B_2 D_2\, e^{B_2(R_2 - Q_\alpha)}\left(e^{B_2(R_2 - Q)} - 1\right) \tag{25}$$
$$\frac{\partial}{\partial Q_\alpha} V_{33}(Q_\alpha) = -2B_3 D_3\, e^{B_3(R_3 - Q_\alpha)}\left(e^{B_3(R_3 - Q)} - 1\right) \tag{26}$$

Finally, the derivatives of the coupling terms are,

$$\frac{\partial}{\partial Q_\alpha} V_{12}(Q_\alpha) = \frac{\partial}{\partial Q_\alpha} V_{21}(Q_\alpha) = -2g_{12} A_{12}(Q_\alpha - Q_{12}) e^{-g_{12}(Q_\alpha - Q_{12})^2} \tag{27}$$
$$\frac{\partial}{\partial Q_\alpha} V_{13}(Q_\alpha) = \frac{\partial}{\partial Q_\alpha} V_{31}(Q_\alpha) = -2g_{13} A_{13}(Q_\alpha - Q_{13}) e^{-g_{13}(Q_\alpha - Q_{13})^2} \tag{28}$$
$$\frac{\partial}{\partial Q_\alpha} V_{23}(Q_\alpha) = \frac{\partial}{\partial Q_\alpha} V_{32}(Q_\alpha) = -2g_{23} A_{23}(Q_\alpha - Q_{23}) e^{-g_{23}(Q_\alpha - Q_{23})^2} \tag{29}$$

A good place to get started coding up our model is to define all our macros - there are a lot in this model.

```
#define C1 1.0
#define C2 2.0

#define D1 0.5
#define D2 0.3
#define D3 0.2

#define B1 10.0
#define B2 20.0
#define B3 30.0

#define R1 1.0
#define R2 1.0
#define R3 1.0

#define A12 17.0
#define A13 20.0
#define A23 23.0

#define G12 1.0
#define G13 1.0
#define G23 1.0

#define Q12 3.0
#define Q13 3.0
#define Q23 3.0
```

Remember, these need to be defined before the line `struct Potentials{` . Please note that the values we've assigned to the macros above are completely arbitrary and only for the sake of demonstration.

To code the state independent potential energy surface, we could do the following:

```
inline double V0(const std::valarray<double> Q){
     return B1*(Q*Q*Q*Q).sum() + B2*(Q*Q).sum();
}
```

For the gradient of the state independent potential energy surface, we would write,

```
inline std::valarray<double> dV0_dQ
  (const std::valarray<double> Q){
     return 4.0*C1*(Q*Q*Q) + 2.0*C2*(Q);
}
```

Now we will concentrate on the state dependent potential energy surfaces. The code should look like the following:

```
inline void Velec(const std::valarray<double> &Q,
   std::valarray<double> &Vout){

    //state 1
    Vout[std::slice(0,num_beads,1)] =
       D1*pow(1.0 - exp(-B1*(Q - R1)) , 2);

    //state 2
    Vout[std::slice(num_beads,num_beads,1)] =
       D2*pow(1.0 - exp(-B2*(Q - R2) , 2);

    //state 3
    Vout[std::slice(2*num_beads,num_beads,1)] =
      D3*pow(1.0 - exp(-B3*(Q - R3) , 2);

}
```

The gradient of the state dependent potential energy surfaces will be very similar to the code we just wrote:

```
inline void dVelec(const std::valarray<double> &Q,
   std::valarray<double> &Vout){

    //state 1
    Vout[std::slice(0,num_beads,1)] =
       -2.0*B1*D1*(exp(B1*(R1-Q)))*(exp(B1*(R1 - Q)) - 1.0);

    //state 2
    Vout[std::slice(num_beads,num_beads,1)] =
       -2.0*B2*D2*(exp(B2*(R2-Q)))*(exp(B2*(R2 - Q)) - 1.0);

    //state 3
    Vout[std::slice(2*num_beads,num_beads,1)] =
       -2.0*B3*D3*(exp(B3*(R3-Q)))*(exp(B3*(R3 - Q)) - 1.0);

}
```

Finally, we'll code up the coupling terms. Now that we have non-constant coupling, we need to use if statements to decide which two states we are evaluating the coupling between. Here is an example of one way we can achieve this.

```
inline double V_couple(const double &Q, int state1, int state2)
{
    if((state1==0  state2==1) || (state1==1  state2==0)){
      return A12*exp(-G12*pow(Q - Q12 , 2));
   }
    else if((state1==0  state2==2) || (state1==2  state2==0)){
      return A13*exp(-G13*pow(Q - Q13 , 2));
   }
    else {
      return A23*exp(-G23*pow(Q - Q23 , 2));
   }
}
```

For the derivative of the coupling terms, we follow the same format as in `V_couple`, but substitute in the derivatives.

```
inline double dV_couple(const double &Q, int state1, int state2)
{
     if((state1==0  state2==1) || (state1==1  state2==0)){
       return -2.0*G12*A12*(Q - Q12)*exp(-G12*pow(Q - Q12 , 2));
    }
     else if((state1==0  state2==2) || (state1==2  state2==0)){
       return -2.0*G13*A13*(Q - Q13)*exp(-G13*pow(Q - Q13 , 2));
    }
     else {
       return -2.0*G23*A23*(Q - Q23)*exp(-G23*pow(Q - Q23 , 2));
    }
}
```

Remember that after you make any changes to `Potentials.h`, you need to recompile MAVARIC. Please note that the code demonstrations above were written for the sake of demonstration and are not well optimized. A more efficient implementation would be to define a single macro any time a series of constants are multiplied in a function.

6

SECTION

References

[1] N. Ananth, *J. Chem. Phys.*, 2013, **139**.

[2] *Numerical Recipes Third Edition*, Chapter 17; W.H. Press, S. A. Teukolsky, W.T. Vetterling, B.P. Flannery, 2007

[3] *Statistical Mechanics: Theory and Molecular Simulation*, M. Tuckerman, 2009