atesa Documentation

atesa

May 10, 2022

For Users:

A Python program for automating transition path sampling with aimless shooting, suitable for experts and novices alike.

ATESA automates a particular Transition Path Sampling (TPS) workflow that uses the flexible-length aimless shooting algorithm of Mullen et al. 2015. ATESA interacts directly with a batch system or job manager to dynamically submit, track, and interpret various simulation and analysis jobs based on one or more initial structures provided to it. The flexible-length implementation periodically checks simulations for commitment to user-defined reactant and product states in order to maximize the acceptance ratio and minimize wasted computational resources.

ATESA implements automation for obtaining a suitable initial transition state, flexible-length aimless shooting, inertial likelihood maximization, committor analysis, umbrella sampling (and analysis with the Multistate Bennett Acceptance Ratio), and equilibrium path sampling. These components constitute a near-complete automation of the workflow between identifying the reaction of interest, and obtaining, validating, and analyzing the energy profile along an unbiased and *bona fide* reaction coordinate that describes it.

The batch systems and molecular simulations packages currently supported by ATESA (please raise an issue with the "enhancement" label on our GitHub page if you'd like to see something added to this list!):

Batch Systems

- Slurm
- PBS/TORQUE

Simulations Packages

- Amber

CHAPTER 1

Theory and Definitions

This page is intended to briefly introduce readers who may not be familiar with the theory of rare event sampling to some of the theory and vocabulary of transition path sampling (TPS) with aimless shooting, as well as to how ATESA implements them. It is intended for prospective users who may not yet be sure whether TPS or ATESA is right for their application. A thorough explanation of TPS as a technique and its relationship to other methods is beyond the scope of this document; for this the reader is instead directed to Beckham and Peters, 2010.

1.1 Why Do Rare Event Sampling?

Molecular simulations are a powerful tool for investigating the workings of chemical systems at the extremely small scale. However, due to technical limitations, simulations are necessarily quite limited in scope and cannot replicate the time- and length-scales relevant in laboratory studies. This can be particularly troublesome when the important feature of a system is a chemical reaction or transformation with a significant activation barrier; although such an event may take place very quickly in the eyes of an experimentalist, it could take years of computer time before the same event might be expected to occur just once in a simulation. This is what is meant when we call certain reactions or transformations "rare events".

In order to apply simulations to the study of rare events, we must make use of advanced sampling methods. These methods take advantage of knowledge about the system or about chemistry in general to modify the behavior of simulations and narrow their focus to a particular event or events and allow them to be simulated on tractable timescales. In particular, ATESA automates a transition path sampling workflow using aimless shooting.

1.2 What is Transition Path Sampling?

Transition path sampling refers to any method of rare event sampling that aims to characterize the ensemble of "transition paths" (that is, trajectories through phase space that connect one discrete state to another) as a means of characterizing the rare event as a whole. You should look towards transition path sampling methods when you want to characterize *how* a rare event occurs from beginning to end with minimal bias. For instance, you might use transition path sampling methods to discover the key parameters that describe a change in the configuration of a structure, or to determine the chemical mechanism of a reaction.

1.3 What is Aimless Shooting?

Aimless shooting is a transition path sampling method for performing efficient, unbiased sampling of the region(s) of phase space corresponding to the ensemble of transition states. Because by definition the transition state is a local maximum in energy along at least one dimension, this region is difficult to sample using conventional simulations – that is, a transition is a rare event. The aimless shooting approach is to leverage one or more putative or "guess" transition state structures (which are obtained by other methods as a prerequisite to beginning aimless shooting, though ATESA is equipped with a tool to help do so), which will be "aimlessly" "shot" through phase space using unbiased initial velocities chosen from the appropriate Boltzmann distribution. The resulting trajectory is at the same time also simulated in reverse (using initial velocities of opposite direction and equal magnitude), and if after simulations the two trajectories converge to different pre-defined energetic basins (*e.g.*, one "products", one "reactants"), then the reactive trajectory connecting them is considered a success. New starting points are daisy-chained from older successful ones by taking an early frame from the reactive trajectory as the initial coordinates, and in this way it is ensured that sampling remains nearby the transition state separatrix (that is, the surface in phase space that divides the products from the reactants with equal commitment probability in either direction).

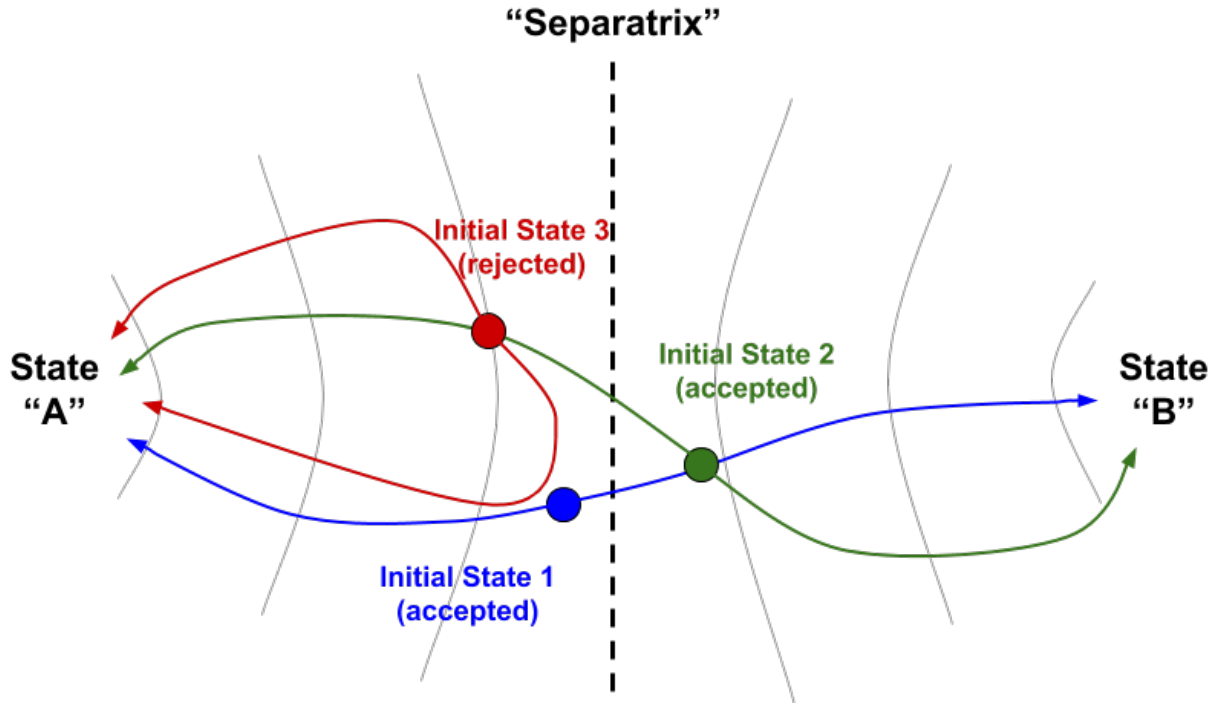


Fig. 1: An example of three aimless shooting moves in a hypothetical 2-D state space. Each shooting move consists of an initial coordinate (colored circle) from which two trajectories begin in opposite directions (colored lines). If the two trajectories go to opposite basins ("A" and "B"), then the move is accepted and new initial coordinates for the next step are chosen from an early part of the accepted trajectory (as move 2 (green) begins along the pathway from move 1 (blue)). If a move is not accepted (move 3 (red)), then the next step would begin from a different point and/or with different initial velocities from the previous accepted move (not shown).

1.4 When is Aimless Shooting the Right Tool?

Of course, aimless shooting is not always to best tool for the job. Efficiency aside, the primary advantage of aimless shooting compared to other path sampling tools is that it is completely unbiased; the only thing you need to get started is a way to distinguish one state from the other, and a guess about what lies in between. If you're unsure whether

aimless shooting is the best option for your application, consider the following:

- Aimless shooting is designed to focus sampling around transition states, rather than at the stable states that they connect. If you are more interested in comparing properties of stable states than in understanding how one transitions to another, aimless shooting is not the right tool.
- Aimless shooting is best used to discover or describe a mechanism when one is unknown. If you have a reaction coordinate or collective variable that describes the transformation already in mind and just want to characterize it, you may look to a pathway free energy method like umbrella sampling or equilibrium path sampling (the latter of which is implemented in ATESA), or a path sampling method that makes use of a known collective variable like transition interface sampling.
- Like all transition path sampling methods, aimless shooting is at its best where the energy barrier is high (and therefore transitions are rare). If your transition occurs quickly enough to reasonably observe many times over the course of an unbiased molecular dynamics or quantum mechanics simulation, aimless shooting may be overkill.
- Aimless shooting is the best tool when you are interested in efficiently arriving at an accurate description of the transition state of a rare event without specifying a mechanism *a priori*.

1.5 What is ATESA, and Why Should I Use It?

ATESA is a Python program that implements aimless shooting and several attendant setup and/or analysis methods, with the intent of expediting the full workflow and making it readily accessible to non-experts without requiring them to write (or read!) code. It automates the aimless shooting process with a system of independent "threads" representing one particular path in the search through phase space. A thread has a given set of initial coordinates, which it repeatedly "shoots" until it finds a successful reactive trajectory, at which point it picks a new shooting point on the reactive trajectory and continues. Because threads run entirely in parallel and ATESA supports multiprocessing, aimless shooting with ATESA scales almost perfectly so long as sufficient computational resources are available.

ATESA also features a suite of analysis and utility tools that run in much the same fashion. It can be used to obtain an initial transition state guess to seed aimless shooting based on definitions of the stable states, and once aimless shooting has been completed (see *On Termination Criteria*), ATESA can be used to automate likelihood maximization to "mine" the data for a reaction coordinate that describes the transition path, committor analysis to verify that reaction coordinate, and equilibrium path or umbrella sampling to obtain the free energy profile along it.

By design, ATESA is suitable for researchers who are familiar with the basic tenants of molecular simulation, but may not be experts in Python or in rare event sampling. By providing tools and guidance for a ready-made workflow from start to finish, ATESA aims to take the guesswork out of adding transition path sampling to your toolkit. For details on implementing this workflow, see *Basic Workflow*.

1.6 What is Likelihood Maximization?

The output of aimless shooting is a large set of combined variable (CV) values paired with corresponding commitment basins (products or reactants). In order to convert this information into a usable form, the method of likelihood maximization can be used to select a model that describes the reaction progress in terms of relatively few parameters. ATESA supports the inertial likelihood maximization procedure first published in Peters 2012, in addition to the original non-inertial procedure. For details on ATESA's implementation of likelihood maximization, see *lmax.py: Likelihood Maximization*.

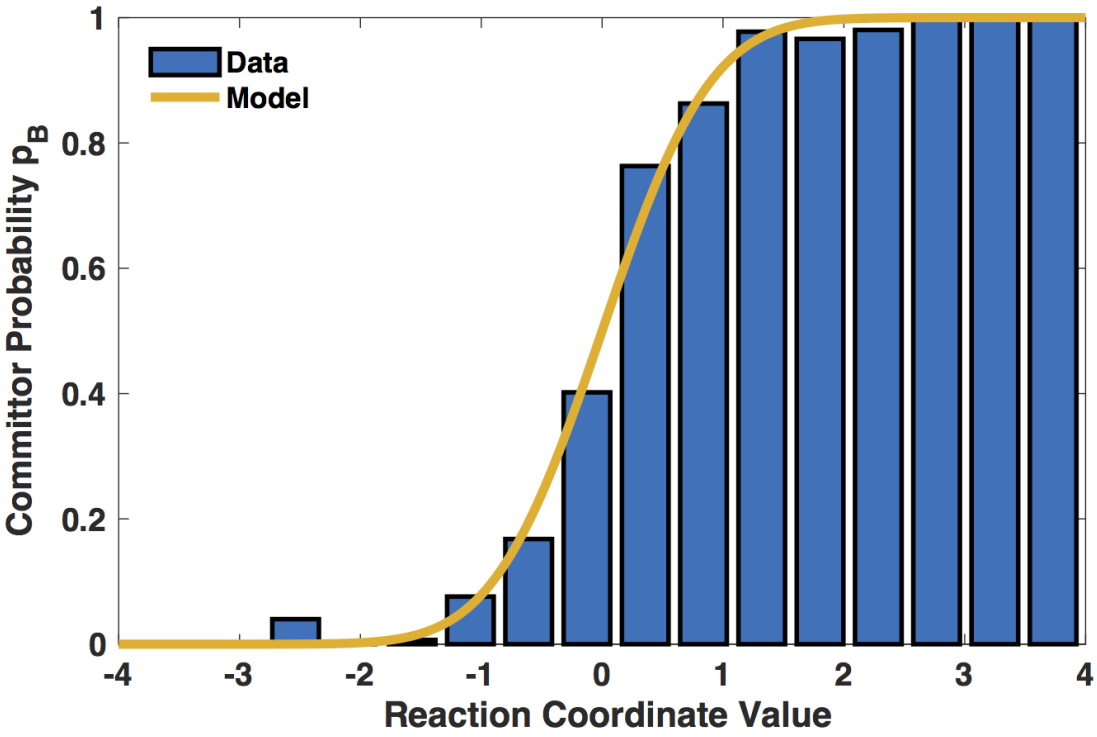


Fig. 2: An example depicting the fitting of a reaction coordinate model (yellow line) to aimless shooting data (blue histogram). Good fit between the histogram and the model is a necessary-but-not-sufficient condition for a good reaction coordinate.

1.7 What is Committor Analysis?

Once a reaction coordinate has been obtained, it should be verified using new, unbiased simulations that were not included in the model training dataset. The method of committor analysis is to simply select a large number (hundreds) of initial coordinates with reaction coordinate values very close to zero (the predicted transition state) and run several unbiased simulations starting from each of them to verify that they are as likely on average to proceed towards the reactants as towards the products. The extent to which this likelihood is clustered around 50% probability of either outcome is a measure of the effectiveness of the reaction coordinate in describing the transition state.

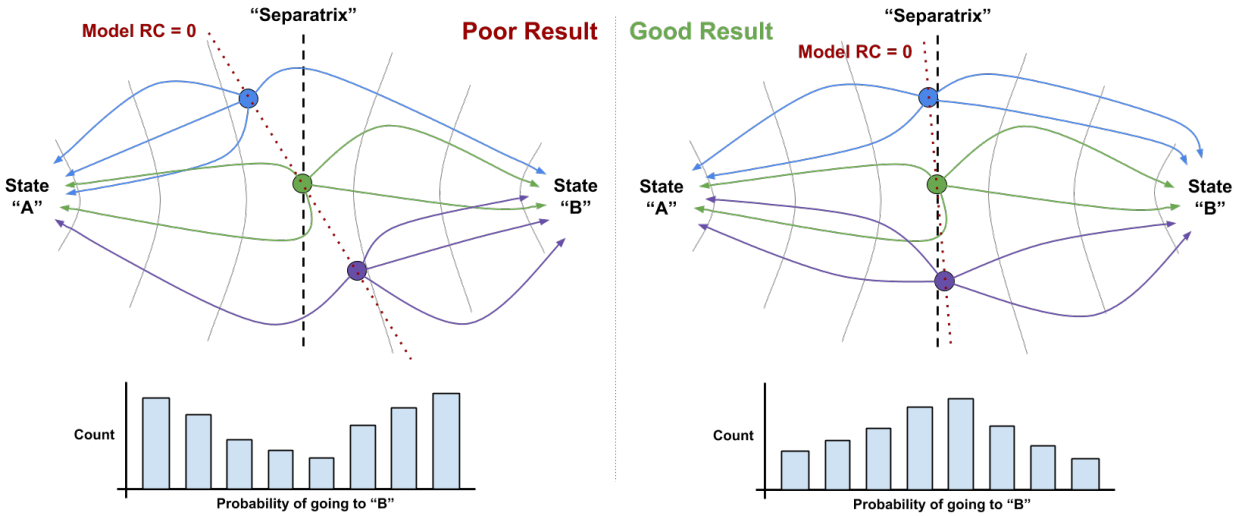


Fig. 3: An pair of examples of committor analysis. At left, a "poor" model misjudges the reaction coordinate (RC) and the resulting committor analysis distribution (at bottom) is bimodal at either end. At right, a much better model closely matches its predicted separatrix (RC = 0) with the "real" separatrix, resulting in a unimodal distribution centered near 1/2.

1.8 What is Umbrella Sampling?

ATESA's preferred method for obtaining a free energy profile along a determined reaction coordinate is umbrella sampling. This is a fairly simple method where many simulations beginning along different portions of the reaction coordinate are restrained to that portion using a harmonic bias. The shape of the resulting distribution of reaction coordinate values sampled over the course of the simulations can be interpreted to measure the underlying free energy profile by "subtracting" the influence of the known harmonic restraints using any of a number of algorithms (one of which, the Multistate Bennett Acceptance Ratio, or "MBAR", is automated in ATESA using the pymbar package).

Umbrella sampling is a very efficient free energy method, but its primary limitation is the requirement that a restraint can be defined along the desired reaction coordinate. ATESA automatically handles these restraints using PLUMED or Amber's irxncor module, if available.

1.9 What is Pathway-Restrained Umbrella Sampling?

In addition to automating traditional umbrella sampling, ATESA features a novel method for constraining the sampling within the known transition pathway ensemble. Although committor analysis can be used to confirm that the chosen reaction coordinate contains all of the key CVs to describe the transition state ensemble, one of the weaknesses of this analysis is that there is no guarantee that the appropriate set of CVs to describe the transition pathway ensemble

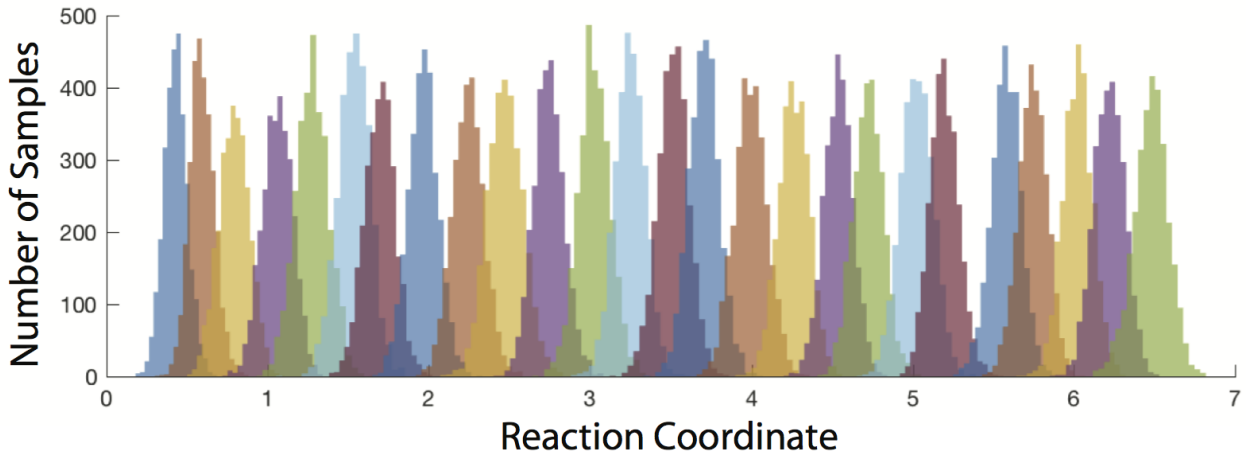


Fig. 4: An example of the raw sampling data from an umbrella sampling job. Colors alternate to help distinguish different simulations, with adjacent simulations overlapping in sampling to avoid gaps. This data can be directly interpreted using MBAR to obtain a free energy profile.

remains the same along the full path from one stable state to the other. This can pose a problem when attempting to apply umbrella sampling along the reaction coordinate: if there exist any dimensions along which the transition pathway ought to be restrained for some portion of it, but those dimensions ought *not* to be restrained at the transition state (where the reaction coordinate was defined), then relaxation along those dimensions will result in misjudging the shape of the free energy profile along those portions during umbrella sampling.

Fortunately, sampling data from aimless shooting can be leveraged to address this issue. To the extent that aimless shooting explored the ensemble of transition pathways, the regions of state space represented among its accepted trajectories describe the boundaries of the transition pathway in every dimension (not just those that contribute to the reaction coordinate). Our approach, which we call "pathway-restrained" umbrella sampling, is to apply additional restraints during umbrella sampling simulations to every dimension that was recorded during aimless shooting. The restraints have flat (zero) weight in the range of values observed during frames of accepted aimless shooting trajectories with reaction coordinate values closest to the umbrella sampling window in question, and steeply increasing weight outside that range. As a result, each umbrella sampling simulation is only able to explore the same regions of state space that were already explored along the corresponding point along the transition pathway during aimless shooting.

Warning: pathway restraints have the potential to impart additional error during umbrella sampling, and so should only be used when necessary, not as a first resort. See *Umbrella Sampling* for a discussion on when pathway restraints are warranted.

Pathway-restrained umbrella sampling is implemented in ATESA by: first, obtaining an "as_full_cvs.out" file by running aimless shooting with the *full_cvs* = *True* option, or resampling a completed aimless shooting job that was run with *cleanup* = *False* with the options *resample* = *True* and *full_cvs* = *True*; and then running umbrella sampling with the *us_pathway_restraints_file* option pointing to that "as_full_cvs.out" file.

1.10 What is Equilibrium Path Sampling?

Although it is efficient, umbrella sampling is not always suitable for every reaction coordinate. The most general pathway free energy method is equilibrium path sampling, wherein the reaction coordinate is divided into bins and the unbiased distribution of reaction coordinate values sampled within those bins is converted directly into free energy. This method requires no restraints or external packages, and so supports competely arbitrary reaction coordinates. ATESA automates collection of equilibrium path sampling data from an arbitrary array of initial coordinates, filling in gaps automatically using the tails of simulations from adjacent windows. Note that the tradeoff for the generality of

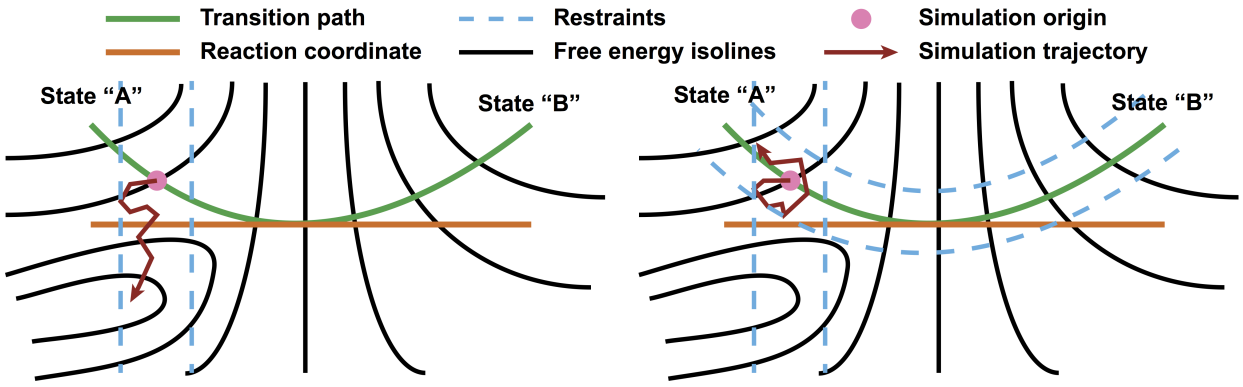


Fig. 5: Schema depicting a simple free energy surface on which umbrella sampling is being performed with and without pathway restraints derived from aimless shooting data. Likelihood maximization optimizes the reaction coordinate only at the separatrix, where the orange and green lines intersect. The lack of pathway restraints in the scheme at left leaves the depicted simulation trajectory free to relax into the off-pathway free energy basin, which would cause errors in measuring the free energy along the pathway. At right, pathway restraints are added to prevent this. Note that in practice umbrella sampling restraints are harmonic, not rigid walls as depicted here for clarity.

this method is that it can be highly resource-inefficient compared to umbrella sampling, especially for rare events with high activation energies.

CHAPTER 2

Getting Started with ATESA

2.1 Installation

ATESA should be installed directly on the high-performance computing (HPC) resource that you intend to use. The simplest way to do so is with pip:

```
pip install atesa
```

You may need to append the `--user` option depending on how your HPC resource is configured. ATESA requires Python 3, and the above pip command may fail if your active python environment defaults to Python 2.

Alternatively, you can download or clone ATESA directly from GitHub and install using the `setup.py` script in its root directory directly:

```
python setup.py install --user
```

If you're using a custom python environment, remember to activate it before installing!

Once you've installed ATESA, you may want to head over to the *Example Study* page to take a look at what ATESA can do and get a feel for its basic operation.

If you want to perform umbrella sampling simulations with ATESA, you need to make sure that PLUMED is available. Versions of Amber released since 2015 have native PLUMED support, so once it has been installed (see this link for instructions), all you need to do is set the PLUMED_KERNEL environment variable to the appropriate path for `libplumedKernel.so`, like so:

```
export PLUMED_KERNEL=/path/to/libplumedKernel.so
```

2.2 Usage

ATESA is designed to dynamically handle jobs on a PBS/Torque or Slurm batch system on a high-performance computing cluster. It is invoked as:

```
atesa <config_file> [<working_directory>]
```

The config_file parameter is required and supports absolute or relative paths. The working_directory parameter is optional, but if it is provided it overrides the value in the configuration file. This option is made available for use on systems that only allocate working space for jobs after they have been initialized. For detailed documentation on the contents of the configuration file, see *The Configuration File* (but it's easier to get started by modifying the examples in the *Example Study*).

Although ATESA can be run directly from the command line, because its process continues for the entirety of the job it is usually best to submit it as its own batch job, or to run it on an interactive resource allocation. A single core and a modest allocation of memory should be sufficient to run ATESA on most platforms (it is typically neither memory- nor processor-intensive, although many features involve a significant amount of I/O). ATESA supports multiprocessing during aimless shooting on UNIX-based systems; this may improve performance in some cases, depending on ATESA settings, simulation speed, and the batch queue. Simply allocate the desired number of cores and ATESA will use them as efficiently as it can.

Note that almost all ATESA jobs will produce a large amount of data, at least transiently. For this reason, the working directory should probably be set to a path inside the "scratch" filesystem or equivalent on your HPC cluster. If you aren't sure what this means, consult the documentation or support staff for your specific resource.

2.3 Setting Up Simulation Files

Beyond the installation of ATESA itself and the creation of the configuration file, the only other step required before performing simulations is to provide the program with the necessary input files and batch script templates for those simulations. ATESA looks for two directories:

- 'input_files', which contains input files for various types of Amber simulations; and
- 'templates', which contains files formatted for use as templates, mostly batch scripts

ATESA comes packaged with examples of both of these directories, located inside the 'atesa/data' directory in the ATESA installation folder. The contents of these directories will be used by ATESA by default, although the user can specify other directories instead (see *File Path Settings*). Note that the default scripts are certainly **not** appropriate for any given user's needs; they are provided as examples only. It is suggested that the user take a look at these files and the following documentation on this page before beginning to construct their own versions, in order to better understand what each file is for.

2.3.1 Input Files

The 'input_files' directory contains input files for molecular simulations. Each file should be named according to the scheme:

```
<job_type>_<step_type>_<md_engine>.in
```

Each of the bracketed values should be replaced by the appropriate name. The "job_type" is identical to the chosen value of the option of the same name in the configuration file (see *Core Settings*). The step type is either "init" or "prod": "prod" is used in every job type and is the primary simulation step, while "init" is used only in aimless shooting and equilibrium path sampling. Finally, "md_engine" should be the name of the simulations engine to be used (at this time, only "amber" is supported). For example, the file for the "prod" step of an "aimless_shooting" job performed with Amber would be:

```
aimless_shooting_prod_amber.in
```

Every input file must be appropriate for your specific molecular model, and also for the specific job and step types. As Amber is the only simulation software currently supported, prospective users who are not familiar with Amber are directed to the relevant tutorials. Following are details for each job type describing what each input file is used for in that job, and any necessary characteristics of that input file. This may seem daunting at first, but once you are comfortable with the basic usage of Amber it should be quite straightforward to construct input files appropriate for your model!

`aimless_shooting`

Aimless shooting input files for the following step types are required for jobs with job_type "aimless_shooting" or "find_ts":

- **aimless_shooting_init_amber.in**: Aimless shooting "init" steps are extremely short simulations whose only purpose is to obtain a fresh set of initial velocities for the following "prod" step. To this end, the "init" input file should be configured to generate new initial velocities from the Boltzmann distribution (as opposed to using velocities from the input coordinate file), and to only perform a single simulation step with an extremely small time step. In Amber, the following settings should be specified in the &cntrl namelist, in addition to any other model-specific settings ("!" denotes a comment in Amber input files):

```
ntx=1,          ! read coordinates but not velocities from input coordinate file
ntxo=1,         ! ASCII-formatted restart file (required for ATESA)
nstlim=1,       ! one simulation step total
dt=0.00001,     ! extremely short time step (too small can cause velocity␣
↪overflow errors)
tempi=300.0,    ! or whatever temperature (same as temp0)
temp0=300.0,    ! or whatever temperature (same as tempi)
ig=-1           ! random initial velocities; should be default but may not be in␣
↪some versions/patches of Amber
```

- **aimless_shooting_prod_amber.in**: Aimless shooting "prod" steps are the primary simulation steps for each shooting move. They take the initial coordinates and velocities from an "init" step (with velocities reversed in the case of backward trajectories) and run until the simulation commits to one of the stable states defined in the configuration file. Therefore, the time step and number of simulation steps should be much larger than in an "init" simulation. In Amber, the following settings should be specified in the &cntrl namelist, instead of the above "init" settings and in addition to any other model-specific settings:

```
ntx=5,          ! read coordinates AND velocities from input coordinate file
ntxo=1,         ! ASCII-formatted restart file (required for ATESA)
nstlim=5000,    ! a large maximum number of steps; will probably be terminated␣
↪early
dt=0.001,       ! or whatever desired simulation time step
irest=1,        ! restart simulation from preceding "init" step
temp0=300.0,    ! or whatever temperature
ntwx=1,         ! or whatever trajectory write frequency, but not only at the end␣
↪of the simulation
ntwv=-1,        ! include velocities in trajectory files (required if the option
↪"include_qdot" is True (which is default)
```

`committor_analysis`

Only a "prod" committor analysis input file is required for jobs with job_type "committor_analysis":

- **committor_analysis_prod_amber.in**: These jobs can use exactly the same settings as aimless shooting "prod" steps, except that each simulation should obtain new initial velocities, as in an aimless shooting "init" steps. In Amber, that means that these three options should be set as follows:

```
ntx=1,          ! read coordinates but not velocities from input coordinate file
tempi=300.0,    ! or whatever temperature (same as temp0)
irest=0,        ! do not restart, use new velocities (this is the default)
```

`umbrella_sampling`

Only a "prod" umbrella sampling input file is required for jobs with job_type "umbrella_sampling":

- **umbrella_sampling_prod_amber.in**: By default, umbrella sampling restraints are applied using PLUMED. The umbrella sampling input file can be almost identical to a committor analysis "prod" file, with the following mandatory additions:

```
plumed=1,               ! enable plumed backend
plumedfile={{ plumedfile}},             ! template slot for declaring plumed file
```

  ATESA will write the appropriate plumed file automatically and insert a reference to it into the input file as needed.

`equilibrium_path_sampling`

Equilibrium path sampling input files for the following step types are required for jobs with job_type "equilibrium_path_sampling":

- **equilibrium_sampling_init_amber.in**: Equilibrium path sampling "init" steps are functionally identical to aimless shooting "init" steps and can use an identical input file.
- **equilibrium_sampling_prod_amber.in**: Equilibrium path sampling "prod" steps are the only type of job currently supported by ATESA that does *not* take its input file from the "input_files" directory. Instead, the input file is constructed from the file in the "templates" directory named as:

```
<md_engine>_eps_in.tpl
```

  This input file can be functionally identical to an aimless shooting "prod" input file, with two key exceptions: the number of simulation steps must be replaced with the exact string `{{ nstlim }}` and the frequency of writes to the output trajectory must be replaced with the exact string `{{ ntwx }}`. In Amber:

```
nstlim={{ nstlim }},
ntwx={{ ntwx }},
```

`find_ts`

In addition to making sure that appropriate input files for aimless shooting are available, jobs with job_type "find_ts" require their own "prod" input file:

- **find_ts_prod_amber.in** This file can be mostly identical to the "aimless_shooting" prod input file, with two key additions: there must be a restraint specified using the file "find_ts_restraints.disang", and the weight of the restraint must be set to steadily increase over time (beginning from zero). An example of a working implementation of this in Amber is as follows. Options in the &cntrl namelist that can be the same as in aimless shooting are here replaced by an elipse (...) for brevity, but they must still be explicitly specified in the input file. Other than that, it should be quite safe to copy the rest of this exactly into your Amber "find_ts" input file, or customize it as you see fit:

```
 &cntrl
  ...
 nmropt=1,              ! turn on restraints
&end
&wt
 type="REST",
 istep1=0,
 istep2=1000,
 value1=0,
 value2=1,
&end
```

(continues on next page)

(continued from previous page)

```
 &wt
  type="REST",
  istep1=1001,
  istep2=2000,
  value1=1,
  value2=1,
 &end
 &wt
  type="END",
 &end
DISANG=find_ts_restraints.disang
```

2.3.2 Templates

The 'templates' directory contains files that ATESA will automatically customize for each individual simulation. It is primarily used for templated batch scripts that will be filled using the *Batch Template Settings* in the configuration file, in addition to several other keywords, described below.

Batch script templates should be named according to the scheme:

```
<md_engine>_<batch_system>.tpl
```

Each of the bracketed values should be replaced by the appropriate name. The "md_engine" should be the name of the simulations engine to be used (at this time, only "amber" is supported). The "batch_system" should be the same as the setting picked for the option of the same name in the configuration file (either "slurm" or "pbs" are currently supported). For example, the Slurm batch template for a simulation with Amber would be named:

```
amber_slurm.tpl
```

In general, the overwrite flag ("-O") should always be present when using Amber with ATESA, or certain features may not work.

Template slots are delimited by double curly braces, as in "{{ example }}". Anything not delimited in this way will be taken as literal. The user should provide batch files that will work for their particular software environment, making use of the templates wherever the batch file might differ between simulations – please refer to the 'atesa/data/templates' directory for examples. In addition to the relevant configuration file settings (again, see *Batch Template Settings*), the following keywords should be included in batch script templates for ATESA to fill out automatically. It is safe to omit any of these keywords if you are sure that a fixed value (or no value at all) is appropriate instead.

`{{ name }}`

The name of the batch job. This will be unique to each step of each thread.

`{{ inp }}`

The input file for this simulation (*e.g.*, one of the files from the 'input_files' directory).

`{{ out }}`

The output/log file for this simulation.

`{{ prmtop }}`

The parameter/topology file for this simulation (the file indicated for the "topology" option in the configuration file).

`{{ inpcrd }}`

The initial coordinate file for this simulation.

`{{ rst }}`

The output coordinate file from this simulation.

`{{ nc }}`

The output trajectory file from this simulation.

As indicated in the preceding section, if you wish to use equilibrium path sampling, the ‘templates’ directory should also include a template file for the equilibrium path sampling “prod” step input file.

CHAPTER 3

Example Study

This page will detail a complete workflow in ATESA for an example chemical reaction. Not every setting or option applied in this workflow will be appropriate for every application of ATESA; the purpose of this page is to illustrate a particular application of the software to help readers better grasp what working with ATESA is actually like. If you're instead looking for an introduction to the theory underlying any of these steps, check out the *Theory and Definitions* page.

Where appropriate, we have included complete examples of input, template, and configuration files as well as key results for each step. They can be found in the 'examples' folder downloaded with ATESA. Using these example files you can skip actually running each step yourself, if desired. **Not all of the contents of these files will be appropriate for your application**. You will almost certainly need to make changes to some of these files to suit your particular model and computational resources, and some advice on what to change has been included below. Also note that because of the non-deterministic nature of these methods, you should not expect to get exactly identical results if you follow along with these steps yourself. Remember that configuration files are used by submitting an ATESA job as follows:

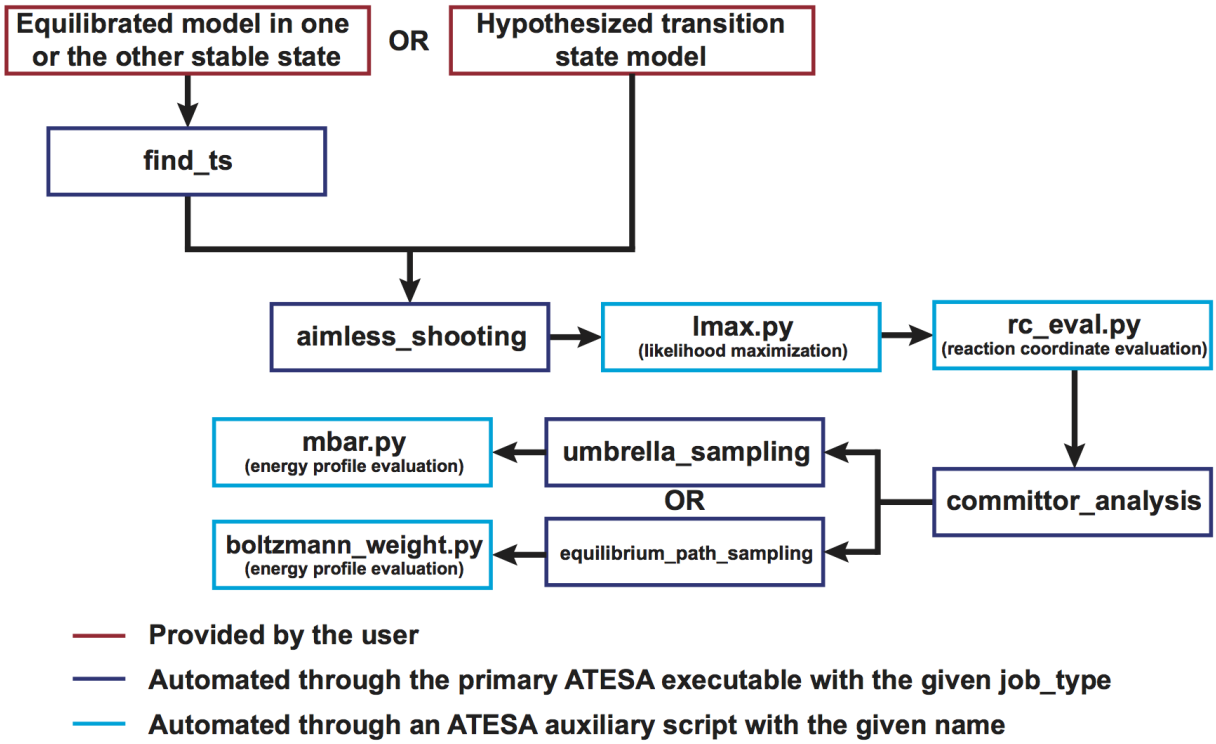
```
atesa example.config
```

Finally, we have included the approximate computational resources consumed during each step. As with all molecular simulations, your results may vary widely based on the details of your simulation and the computational environment, so be cautious in applying these estimates to your own models. In our case, all performance statistics are based on simulations on the San Diego Supercomputer Center's Comet platform, using single cores (except where otherwise specified) on CPU compute nodes with the following characteristics:

```
Intel Haswell Standard Compute Nodes
Clock speed     2.5 GHz
Cores/node      24
DRAM/node       128 GB
SSD memory/node 320 GB
```

3.1 Basic Workflow

The following sections will go through each part of a standard ATESA workflow. Following these steps will allow you to start with only a definition of each stable state and a model of one of them (or with a hypothesized transition state model, if preferred), and end with a validated reaction coordinate and a free energy profile along that coordinate connecting the two basins. The workflow of a complete transition path sampling workflow with ATESA is outlined graphically here.



Specifically, in this example we will be following the branch at top left (using ATESA's `find_ts` jobtype to identify an initial transition state hypothesis) and using umbrella sampling to obtain the free energy profile.

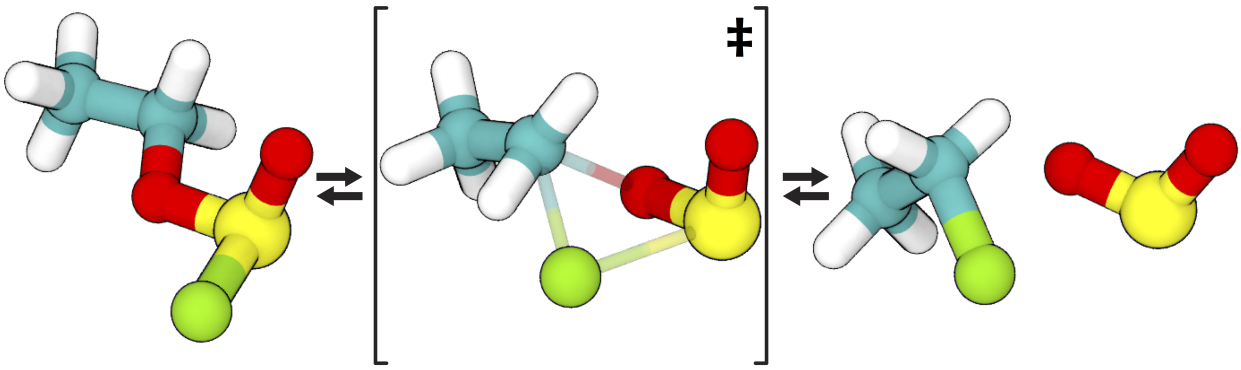
3.2 Initial Setup and the Model

We assume here that ATESA has already been installed on your system. If that is not the case, you should instead start at the *Installation* page.

The model we will be working with here is the gas-phase decomposition of ethyl chlorosulfite into chloroethane and sulfur dioxide. The mechanism via S_Ni reconfiguration has been studied by Schreiner et al. 1995. We chose this reaction because the small number of atoms involved facilitate quick simulations and easy visualizations, but ATESA has also been successfully applied for much larger systems, such as enzyme reaction modeling.

> The reaction pathway via S_Ni reconfiguration for gas phase ethyl chlorosulfite, per Schreiner *et al.* 1995, who demonstrated that this "frontside" attack (where the chlorine bonds to the same side of the carbon as the oxygen departs from) is energetically favorable compared to attacking from the opposite side. Teal: carbon; white: hydrogen; red: oxygen; yellow: sulfur; green: chlorine.

Setup of ATESA for a new system begins with obtaining initial coordinates through whatever means, such as download from a repository like the Protein Databank, or created bespoke using appropriate software. In either case, as with



most molecular simulations, the system should first be minimized, heated to the desired temperature, and equilibrated in the desired relaxed state. Excellent tutorials for these steps (among others) using Amber are available here. Fluency in basic molecular simulations is assumed of users of ATESA, so be sure you're comfortable before moving forward!

In this case, we produced initial coordinates using OpenBabel with the SMILES string "CCOS(=O)Cl", and then equilibrated the system at 300 K over 10 ps in Amber (a very short time, appropriate for such a small system). This equilibration is done with a hybrid QM/MM model, which we will be using to model the chemical reaction. The topology and equilibration input files and the resulting coordinates can be found in examples/model_setup.

3.3 Finding a Transition State

ATESA automates the discovery of suitable initial transition state models using gentle restraints to force rare events to take place, and then identifying the transition state(s) along that forced pathway. The restraints are based only on user-defined definitions of the two stable states that the transition state connects. The complete configuration file used for this job was as follows (remember that many of these options are user- and model-specific!):

```
# examples/find_ts/find_ts.config

job_type = 'find_ts'            # this job is to find a transition state
topology = 'ethyl_chlorosulfite.prmtop'
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_find_ts'
overwrite = False

initial_coordinates = ['ethyl_chlorosulfite.inpcrd']    # initial coordinates in␣
↪reactant state

commit_fwd = ([3,1,1],[5,5,2],[2.2,2.0,3.0],['gt','lt','gt'])    # defines product␣
↪state (see figure below)
commit_bwd = ([3,1,1],[5,5,2],[1.5,3.5,2.4],['lt','gt','lt'])    # defines reactant␣
↪state (see figure below)

max_moves = 5   # number of aimless shooting moves made to confirm transition state

# Paths to simulation/batch template files
path_to_input_files = '/home/tburgin/ethyl_chlorosulfite/input_files'
path_to_templates = '/home/tburgin/ethyl_chlorosulfite/templates'

# Select computational resources to use
prod_walltime = '04:00:00'
```

(continues on next page)

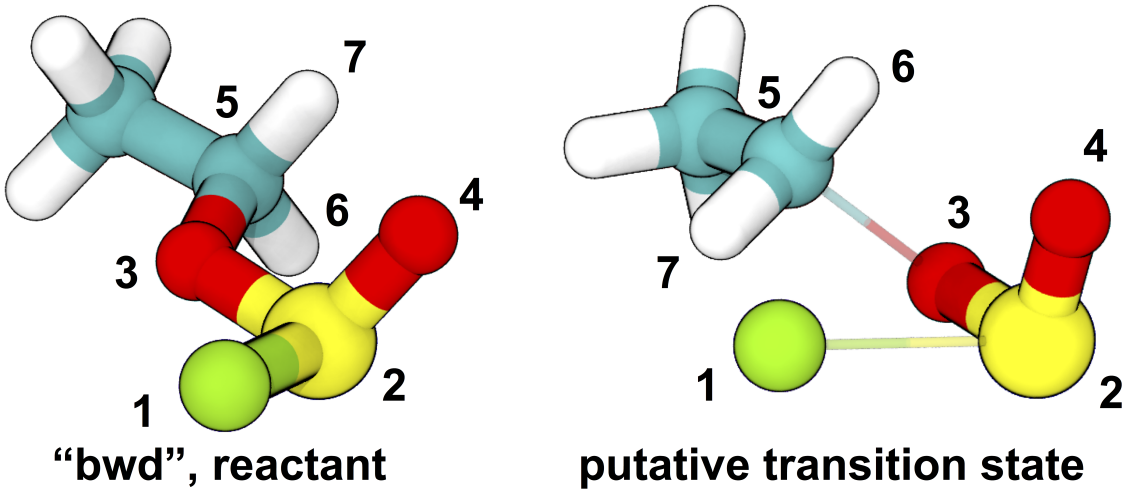
(continued from previous page)

```
prod_ppn = 1
```

If you're adapting this configuration file for your own system, the most important changes to make (besides changing the names of files and directories to match your own) are to the commitment definitions (`commit_fwd` and `commit_bwd`) and the Amber input files contained in the directory pointed to by the `path_to_input_files` line (see *Setting Up Simulation Files*). The commitment definitions need to be selected to uniquely match the two stable states you wish your pathways to connect, and the Amber input files need to be have appropriate settings for your specific model. It will also be important to set the computational resources (at the bottom of this configuration file) to something that is efficient for your particular model.

As for the example study, based on the five aimless shooting moves with which each candidate transition state frame from the forced trajectory was tested, two frames were selected as suitable aimless shooting initial coordinate files, as indicated in ATESA's output following this job. The coordinates for those frames, along with the input and coordinate files, can be found in examples/find_ts. These transition states (they are nearly identical) are very close to the one proposed by Schreiner *et al.* Under our test conditions, this job took 18 minutes to complete.

```
commit_fwd = ([3,1,1],[5,5,2],[2.2,2.0,3.0],['gt','lt','gt'])
commit_bwd = ([3,1,1],[5,5,2],[1.5,3.5,2.4],['lt','gt','lt'])
```



Definitions of stable states and initial and (one of the) final structures from the example transition state search. The stable state definitions are read by inner index; for example, the first element of the definition of the "bwd" state is read as "the distance between atom 3 and atom 5 is less than ('lt') 1.5 A". Based on these definitions, the initial coordinates (at left) occupy the "bwd" state, and restraints are automatically constructed to build a putative transition state (at right) that has significantly non-zero probabilities of relaxing to either state (in this case, meaning that at least one in five aimless shooting moves starting from this state is accepted). The narrow, transparent bonds in the transition state structure show the original topology of the model, for comparison.

3.4 Aimless Shooting

Once a model has been set up near the transition state, aimless shooting can proceed. In this case we used the two transition state models identified in the previous step as the initial coordinates, with 12 copies (`degeneracy = 12`) each to speed up the sampling:

```
# examples/aimless_shooting/aimless_shooting.config

job_type = 'aimless_shooting'   # now we're doing aimless shooting
topology = 'ethyl_chlorosulfite.prmtop'
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_as'   # be sure to use a␣
↪different working directory from find_ts!
overwrite = False

# Use 12 copies each of initial transition state coordinates from our find_ts job
initial_coordinates = ['/scratch/tburgin/ethyl_chlorosulfite_find_ts/as_test/ethyl_
↪chlorosulfite.inpcrd_0_ts_guess_134.rst7', '/scratch/tburgin/ethyl_chlorosulfite_
↪find_ts/as_test/ethyl_chlorosulfite.inpcrd_0_ts_guess_136.rst7']
degeneracy = 12

# Same commitment definitions as in find_ts
commit_fwd = ([3,1,1],[5,5,2],[2.2,2.0,3.0],['gt','lt','gt'])
commit_bwd = ([3,1,1],[5,5,2],[1.5,3.5,2.4],['lt','gt','lt'])

information_error_freq = 2500   # how often to check termination criterion

path_to_input_files = '/home/tburgin/ethyl_chlorosulfite/input_files'
path_to_templates = '/home/tburgin/ethyl_chlorosulfite/templates'

# Computational resources for each simulation during aimless shooting
prod_walltime = '00:30:00'
prod_ppn = 1

cleanup = False         # this is important if we want to do pathway-restrained␣
↪umbrella sampling later
```

Note that the we don't define any specific CVs in this file. Instead, we allow ATESA to build CVs automatically based on the atoms indicated in the commitment definitions. In this case, ATESA identified 156 CVs to sample at each shooting point, and printed their definitions to a file named "cvs.txt" in the working directory (this is the default behavior). We set the number of steps between assessments of the information error termination criterion 10 times higher than the default since our model is very small, so we'll accumulate hundreds of simulations very rapidly. A larger system, like a protein, may benefit from more frequent assessments (smaller `information_error_freq`). Similarly, we set a short walltime and allocate only a single core to the production simulations to reflect their low computational requirements. We also set `cleanup = False` so that ATESA does not delete trajectory files for completed moves; this will take up dramatically more storage space in the working directory, but it leaves us the option to use pathway-restrained umbrella sampling later on should we need it (see *What is Pathway-Restrained Umbrella Sampling?* if you don't know what that is).

Note that because the simulations for this job are so short, it is best to take advantage of ATESA's built-in multiprocessing support for this task. The optimal number of cores to allocate will depend greatly on your platform, but using roughly as many cores as you have aimless shooting threads is a reasonable starting point. In this case, we selected 24 aimless shooting threads (12 for each initial coordinate file) to make optimal use of one 24-core node. You should choose the number of threads based on your own available resources.

During our testing, this job collected data at the rate of approximately 410 shooting moves per hour; the bottleneck in this case was waiting for the batch system to allocate resources for individual simulations, but this will not be the case for larger (slower) models. Remember that aimless shooting jobs that end for any reason can be restarted from where they left off by resubmitting the same job with the configuration file setting `restart = True`.

This job collected 15,142 shooting moves before terminating automatically using based on the *Information Error* termination criterion with the default settings. An average acceptance ratio of 31.76% (per "status.txt" in the working

directory) reflects a very healthy level of efficiency (10-30% is about average). ATESA also automatically generates a version of the aimless shooting file that has been decorrelated from the initial state(s) when assessing the information error termination criterion. Both the raw and decorrelated output files (compressed to save space), in addition to the input and configuration files, can be found in examples/aimless_shooting.

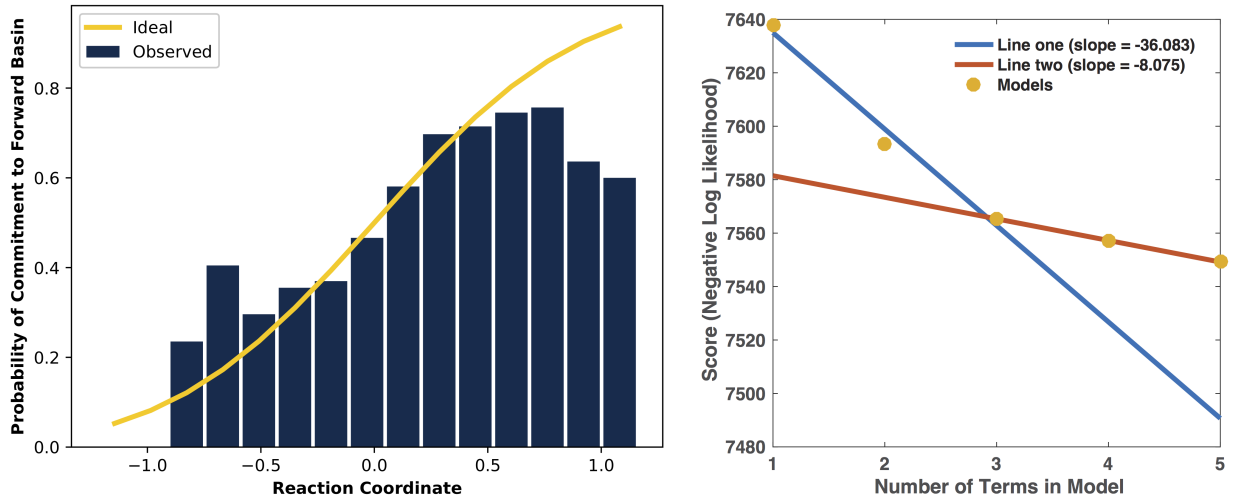
3.5 Likelihood Maximization and Reaction Coordinate Evaluation

After aimless shooting terminates, the results are passed to the auxiliary script *lmax.py: Likelihood Maximization* in order to obtain a model reaction coordinate that describes the probable fate of a simulation beginning from a given set of initial conditions. When using the information error termination criterion (as we did in the last step) this is done automatically every *information_error_freq* aimless shooting steps, and the associated likelihood maximization output files are stored in the working directory.

If we were inclined to repeat this step manually for whatever reason, the command for doing that would be:

```
lmax.py -i /scratch/tburgin/ethyl_chlorosulfite_as/as_decorr_15000.out --two_line_
↪test --plots
```

You should use the largest decorrelated ("decorr") output file available (the one with the biggest number at the end) as the input for lmax.py. After producing the reaction coordinate, the `--plots` option instructs the program to produce the sigmoid committor plot (at left) and, when the `--two_line_test` option is used, the two-line test plot (at right) (see *lmax.py: Likelihood Maximization* for more details):

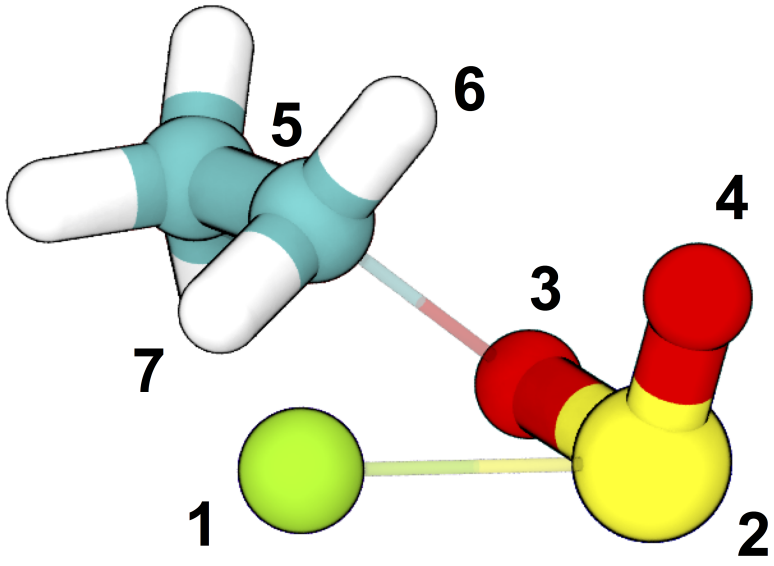


The committor plot in this case is actually of below-average quality, but the fit between the observed and ideal shape is good near the transition state and the overall trend is correct (lower on the left, higher on the right), so as long as we obtain a strong committor analysis result in the next step, we can be satisfied.

The reaction coordinate that ATESA selects contains three CVs (the intersection of the two-line test; see *The two_line_test Option* for more details on this model selection method). This model was:

$$-1.600 + 2.053 * CV156 + 0.576 * CV1 + 0.655 * CV22$$

The identities of these CVs are given in the "cvs.txt" file that ATESA produces in the aimless shooting working directory. In this case, these CVs are as follows:



CV156: difference of distances between atoms [5, 3] and [5, 1]

CV1: distance between atoms [1, 2]

CV22: angle between atoms [1, 5, 7]

ATESA has discovered that these three CVs produce a good description of reaction progress without even being told what to consider! After selecting a reaction coordinate, to set up for the next step we need to assess the reaction coordinate value for each of the aimless shooting moves in our dataset. This is also performed automatically when using the information error termination criterion, but if we want to do it manually, we call the auxiliary script *rc_eval.py: Reaction Coordinate Evaluation*, specifying the aimless shooting working directory we want to analyze, the reaction coordinate, and the decorrelated output file used during likelihood maximization to build that reaction coordinate. This will build a file named "rc.out" in the working directory:

```
rc_eval.py /scratch/tburgin/ethyl_chlorosulfite_as -1.600+2.053*CV156+0.576*CV1+0.
↪655*CV22 /scratch/tburgin/ethyl_chlorosulfite_as/as_decorr_15000.out
```

The files "rc.out", "15000_lmax.out", and "cvs.txt" can all be found in examples/lmax.

3.6 Committor Analysis

Having obtained what appears to be a suitable reaction coordinate, the final step in validating it before using it to analyze the energy profile is to perform committor analysis. By performing new simulations (*i.e.*, simulations whose results were not included in the likelihood maximization training data) from various initial configurations with reaction coordinate values of approximately zero, we can confirm that the reaction coordinate is an accurate descriptor of the transition state (at least within the context of our particular simulation conditions).

Committor analysis is again called through the main ATESA script. Our complete configuration file for this job is as follows:

```
# examples/committor_analysis/committor_analysis.config

job_type = 'committor_analysis'          # specify committor analysis
topology = 'ethyl_chlorosulfite.prmtop'
```

(continues on next page)

(continued from previous page)

```
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_as/committor_analysis'
↪  # a new directory within the parent aimless shooting directory
overwrite = False

# Inherit settings from the associated aimless shooting job
as_settings_file = '/scratch/tburgin/ethyl_chlorosulfite_as/settings.pkl'

# Committor analysis settings
committor_analysis_use_rc_out = True    # select initial coordinates automatically
path_to_rc_out = '/scratch/tburgin/ethyl_chlorosulfite_as/rc.out'       # see␣
↪previous section
rc_threshold = 0.0025   # distance from RC = 0 to permit
committor_analysis_n = 20       # simulations per initial coordinates (at least 10 is␣
↪good)

path_to_input_files = '/home/tburgin/ethyl_chlorosulfite/input_files'
path_to_templates = '/home/tburgin/ethyl_chlorosulfite/templates'

prod_walltime = '01:00:00'
prod_ppn = 1
```

The use of `as_settings_file` to point to the `settings.pkl` file produced during aimless shooting ensures that the same commitment basin and CV definitions are used. The next block of options specifies how committor analysis will be carried out: each of the shooting points identified in `/scratch/tburgin/ethyl_chlorosulfite_as/rc.out` (the file produced just before by `rc_eval.py`) as having a reaction coordinate absolute value of less than or equal to the threshold value of 0.0025 will be used to seed 20 individual committor analysis simulations. The threshold was chosen manually by inspecting the specified "rc.out" file so as to include approximately 200 separate coordinate files, which provides a good amount of statistical power, so you may want to choose a different threshold based on your particular data. The resulting output file and the input and configuration files are available in examples/committor_analysis. During our testing, this job completed in 1 hour and 12 minutes.

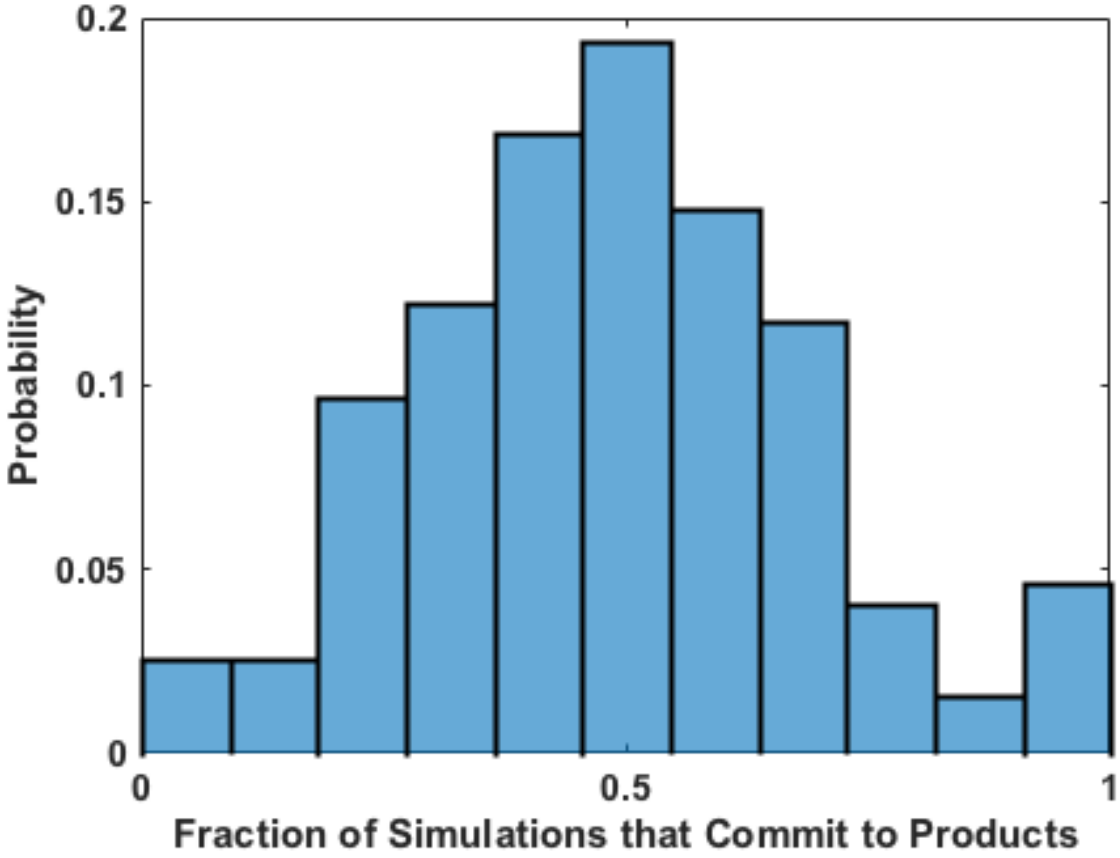
Plotting the contents of the output file produced by this job (`/scratch/tburgin/ethyl_chlorosulfite_as/committor_analysis/committor_analysis.out`) as a histogram, we see that it is very even and centered at 0.5, which affirms that our reaction coordinate is a strong model. **You should always plot the committor analysis data before moving forward!** If you're unsure what constitutes an acceptable committor analysis result, see the Troubleshooting section on *Committor Analysis*.

3.7 Umbrella Sampling

Finally, we're ready to evaluate the energy profile along our reaction coordinate. ATESA features two separate job types for this purpose: equilibrium path sampling, and umbrella sampling. Usually the latter is strongly preferable, so we'll focus on that here.

First, we need to identify the appropriate minimum and maximum RC values to sample over. ATESA's built-in script `rc_eval.py` has a feature to facilitate this (see *rc_eval.py: Reaction Coordinate Evaluation* for more details):

```
rc_eval.py /scratch/tburgin/ethyl_chlorosulfite_as/ -1.600+2.053*CV156+0.576*CV1+0.
↪655*CV22 /scratch/tburgin/ethyl_chlorosulfite_as_/as_decorr_15000.out True
```


0.2
0.15
0.1
0.05
0
Probability
0
0.5
1
Fraction of Simulations that Commit to Products

This completes in a matter of seconds, and simply returns the ending RC values observed at the ends of both trajectories in an accepted shooting move. In other words, it approximates the RC values of the stable states. In our case, it returned:

```
Shooting move name: ethyl_chlorosulfite.inpcrd_0_ts_guess_134.rst7_0_663_init.rst7
extrema: [8.1966, -3.9343]
```

The first line indicates the shooting move that was selected, and the second indicates the RC extrema. To be sure to include the full stable state energy basins, we suggest extending umbrella sampling past these values by about 10%.

Finally, we need to select appropriate spacing (`us_rc_step`, the space from one window to the next) and restraint weights for our umbrella sampling windows. Since the applied restraints are harmonic, the expected width of the sampled distribution is approximately proportional to the inverse square root of the restraint weight. In practice the appropriate restraint weight and spacing is something you'll have to come to through some trial and error depending on your specific reaction coordinate and energy profile, but ATESA's defaults (50 kcal/mol, spaced every 0.5 units along the RC) are usually a reasonable starting point. If you're unsure of how to choose restraint weights and spacing for your system, it is usually wise to run a pilot study with only two or three windows located just a bit to either side of the transition state to measure the approximate width of the sampling histogram for your particular settings (in general each window will be approximately even in width, though they may be shifted from their centers somewhat). It's no problem if your windows overlap too much (other than being an inefficient use of resources), but if there are any gaps, the analysis could be badly incorrect.

In the case of our example, we already know from other studies that the reaction we're looking at has a fairly high activation energy (about 50 kcal/mol), so we'll err on the side of tighter restraints spaced more closely together:

```
# examples/umbrella_sampling/unrestrained_umbrella_sampling.config

job_type = 'umbrella_sampling'  # this is an umbrella sampling job
topology = 'ethyl_chlorosulfite.prmtop'
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_as/umbrella_sampling'
overwrite = False

# Automatically select initial coordinates from aimless shooting
us_auto_coords_directory = '/scratch/tburgin/ethyl_chlorosulfite_as'

# Reaction coordinate definition from likelihood maximization
rc_definition = '-1.600 + 2.053*CV156 + 0.576*CV1 + 0.655*CV22'

# Inherit aimless shooting results and settings
as_out_file = '/scratch/tburgin/ethyl_chlorosulfite_as/as_decorr_15000.out'
as_settings_file = '/scratch/tburgin/ethyl_chlorosulfite_as/settings.pkl'

# Umbrella sampling settings
us_rc_step = 0.1        # distance between sampling window centers
us_restraint = 100      # harmonic restraint weight in kcal/mol
us_rc_min = -4.2        # left boundary of sampling windows
us_rc_max = 9           # right boundary of sampling windows

path_to_input_files = '/home/tburgin/ethyl_chlorosulfite/input_files'
path_to_templates = '/home/tburgin/ethyl_chlorosulfite/templates'

prod_walltime = '04:00:00'
prod_ppn = 1
```

The input and configuration files for this job can be found in examples/umbrella_sampling, tagged with the prefix "unrestrained". One significant change in the input file we used for umbrella sampling is that we have changed the

quantum mechanics model (*qm_theory* option in Amber) from the semi-empirical PM3 to the density functional tight binding model DFTB3, in order to improve the accuracy of the energy calculations at the expense of some speed. More commonly, you should use the same QM model (if any) throughout your study.

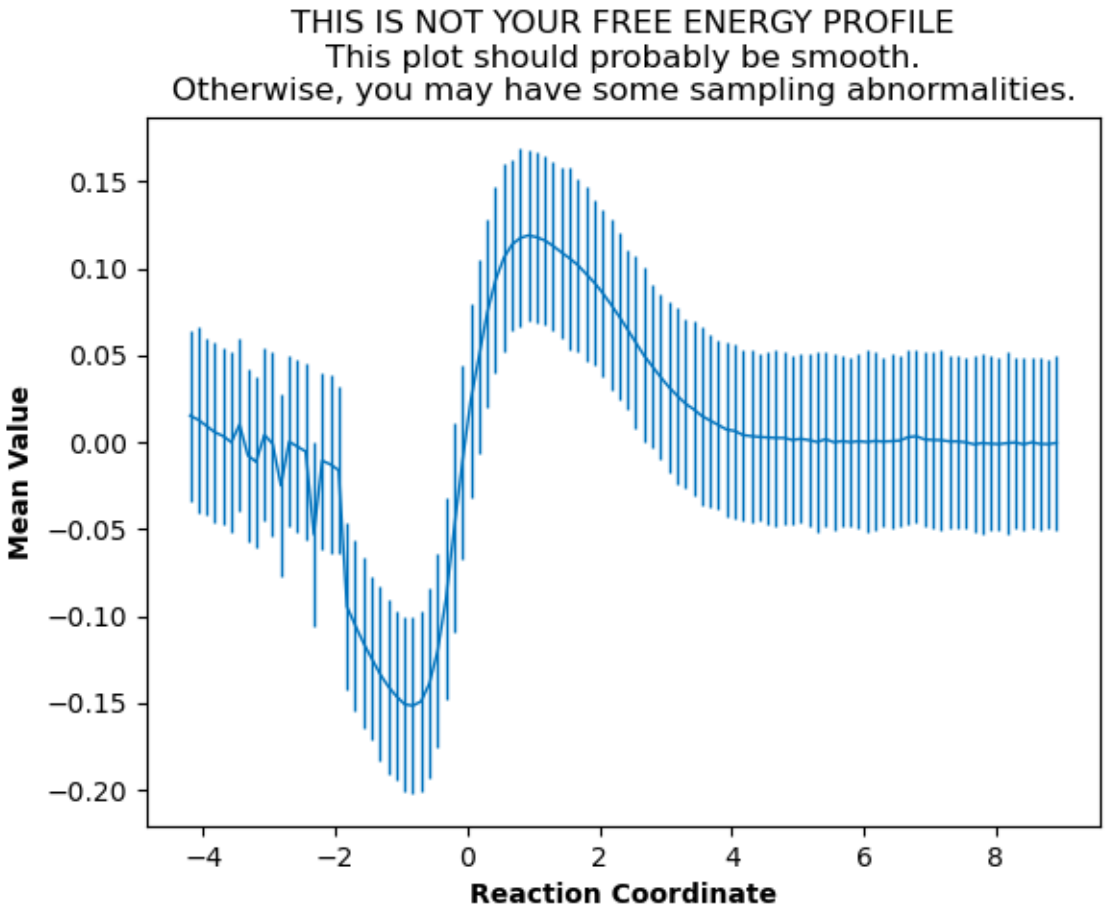
During our testing, this job took 1 hour and 14 minutes to complete. This job produces a large number of data files named with the suffix "_us.dat" in the working directory, each of which represents the umbrella sampling data from one simulation. When the job is finished, we can call the auxiliary script `mbar.py` to analyze the data in each of those files together:

```
mbar.py --decorr -k 100 -i /scratch/tburgin/ethyl_chlorosulfite_as/umbrella_sampling
```

Here we use the `--decorr` flag to specify that we have not checked the data for decorrelation or equilibration, so pyMBAR will do that work for us. We also set `-k 100` to indicate that the umbrella sampling restraint weight was 100 kcal/mol. Finally, the `-i` flag specifies the directory that contains the data files. Note that this script can take some time to complete (minutes to hours if you have a huge amount of data), so you will probably want to run it in a batch job or interactive resource allocation, not a login node.

If your shell supports it, this script will produce a few plots for you, but otherwise it will print raw data to the output file ("mbar.out" by default, but you can rename it with the "-o" flag) for you to plot yourself. These plots are: the mean value plot, a sampling histogram, and finally the free energy profile. Please see *Umbrella Sampling* for a discussion of how the first two of these plots can be used to assess the quality of umbrella sampling data.

In our case, the mean value plot produced by the above job looked like this:



Of particular note in this figure is that there is a stark discontinuity in the slope on the left side, around the reaction coordinate value of -2. This indicates that the simulations are sampling from discontiguous parts of the free energy surface, only one of which is likely to be occupied by the "real" transition path ensemble that we sampled during aimless shooting. For further discussion of this theory, see *What is Pathway-Restrained Umbrella Sampling?*.

In order to correct this, we'll try to apply pathway restraints. Because we specified `cleanup = False` in our aimless shooting configuration file, the aimless shooting working directory still contains all of the simulation trajectory files. The first step in pathway-restrained umbrella sampling is to run another aimless shooting job with `resample = True` and `full_cvs = True` in that same directory, in order to obtain a new file called "as_full_cvs.out":

```
# examples/umbrella_sampling/resample.config

job_type = 'aimless_shooting'   # aimless shooting resampling job
topology = 'ethyl_chlorosulfite.prmtop'
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_as'
overwrite = False       # don't forget to set this to False!!

# Just the same initial coordinates as the original aimless shooting job we ran
initial_coordinates = ['/scratch/tburgin/ethyl_chlorosulfite_find_ts/as_test/ethyl_
↪chlorosulfite.inpcrd_0_ts_guess_134.rst7', '/scratch/tburgin/ethyl_chlorosulfite_
↪find_ts/as_test/ethyl_chlorosulfite.inpcrd_0_ts_guess_136.rst7']

resample = True         # don't run aimless shooting again, just resample from␣
↪existing simulations
full_cvs = True         # produce input for pathway-restrained umbrella sampling

# Same commitment definitions we've been using all along
commit_fwd = ([3,1,1],[5,5,2],[2.2,2.0,3.0],['gt','lt','gt'])
commit_bwd = ([3,1,1],[5,5,2],[1.5,3.5,2.4],['lt','gt','lt'])

information_error_freq = 2500
```

This file is just the same as the aimless shooting configuration file (with some extraneous options removed for clarity, though leaving them in would not cause errors), but with the addition of the `resample` and `full_cvs` options. This job won't actually perform aimless shooting; it will just reanalyze the existing aimless shooting data in the specified working directory. Be careful to specify `overwrite = False` to ensure that your aimless shooting data is not deleted! This configuration file can also be found in examples/umbrella_sampling.

Because there's a lot of data to analyze, we suggest making use of ATESA's multiprocessing support when resampling with `full_cvs = True`. In this case we allocated 24 cores, and this job finished after 1 hour and 19 minutes.

Having finished that, we're ready to try umbrella sampling again, using our freshly resampled data to construct pathway restraints. The configuration file for this job is just the same as the previous umbrella sampling file, with a single addition, the `us_pathway_restraints_file` option:

```
# examples/umbrella_sampling/restrained_umbrella_sampling.config

job_type = 'umbrella_sampling'
topology = 'ethyl_chlorosulfite.prmtop'
batch_system = 'slurm'
restart = False
working_directory = '/scratch/tburgin/ethyl_chlorosulfite_as/umbrella_sampling_
↪pathway_restrained'
overwrite = False

us_auto_coords_directory = '/scratch/tburgin/ethyl_chlorosulfite_as'
```

(continues on next page)

(continued from previous page)

```
# Add this line to do pathway-restrained umbrella sampling
us_pathway_restraints_file = '/scratch/tburgin/ethyl_chlorosulfite_as/as_full_cvs.out'

rc_definition = '-1.600 + 2.053*CV156 + 0.576*CV1 + 0.655*CV22'

as_out_file = '/scratch/tburgin/ethyl_chlorosulfite_as/as_decorr_15000.out'
as_settings_file = '/scratch/tburgin/ethyl_chlorosulfite_as/settings.pkl'

us_rc_step = 0.1
us_restraint = 100
us_rc_min = -4.2
us_rc_max = 9

path_to_input_files = '/home/tburgin/ethyl_chlorosulfite/input_files'
path_to_templates = '/home/tburgin/ethyl_chlorosulfite/templates'

prod_walltime = '04:00:00'
prod_ppn = 1
```

Once again, the files for this job are available in examples/umbrella_sampling, this time with the prefix "restrained". ATESA will automatically interpret the specified "as_full_cvs.out" file to build and apply restraints to each umbrella sampling simulation to require that it remain within the known range of all CV values for the observed transition path ensemble. This takes a little extra time; when we tested it, this job completed after 1 hour and 55 minutes. Running `mbar.py` to analyze the resulting data, we see that the discontinuity we observed before has been largely corrected:

A couple of the sampling windows have abnormally large error bars, but their means are reasonable, and this is an overall better mean value plot compared to the one we obtained without pathway restraints. Proceeding with the analysis, our script also produces a series of histograms that shows good overlap between windows with no gaps:

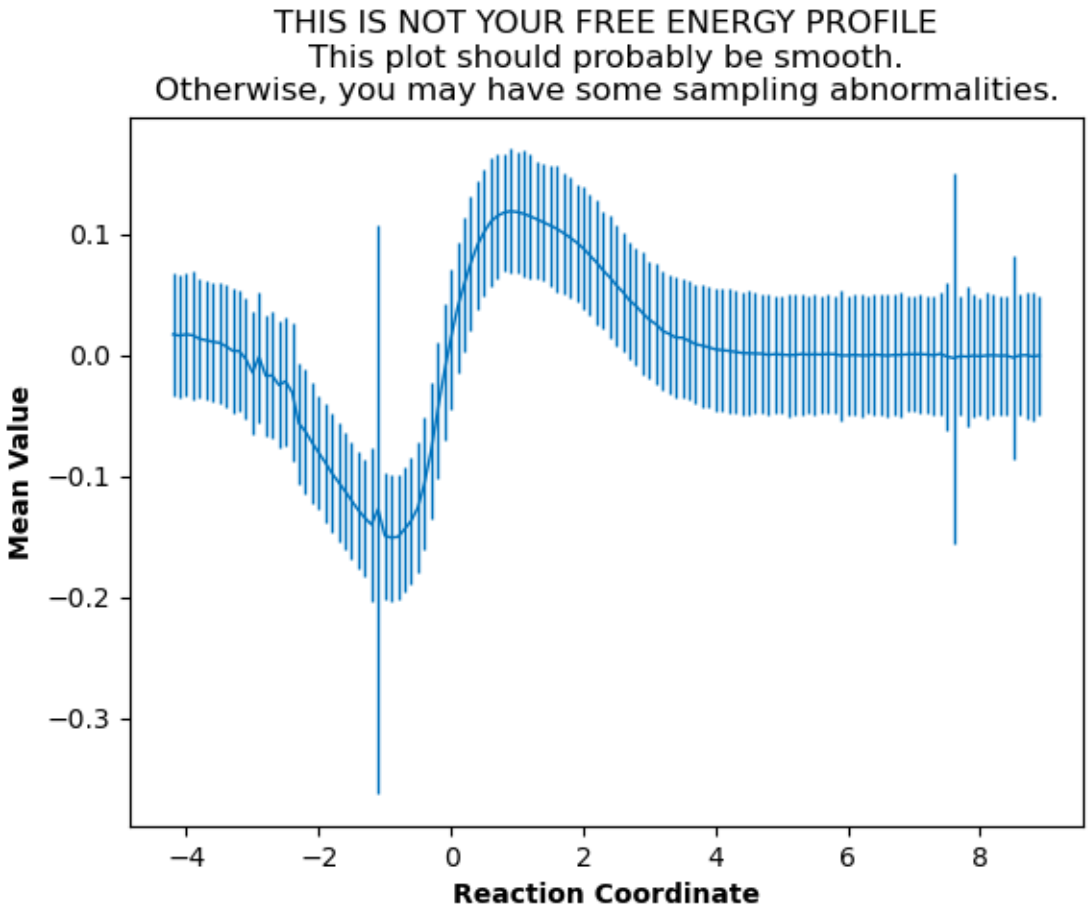
And finally, produces a smooth free energy profile:

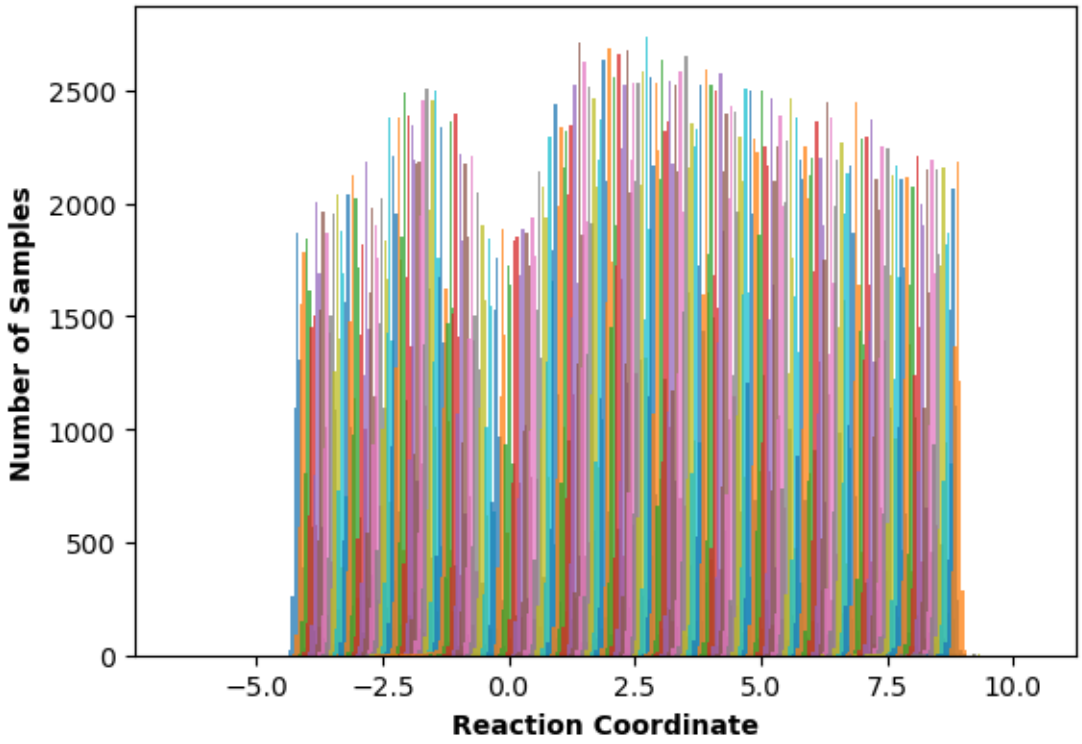
The reactants state occupies the left-hand minimum and the products state, the right-hand minimum. This analysis results in an activation energy of about 51.3 kcal/mol, which is reasonably close to the 48.1 kcal/mol estimated by Schreiner et al. (especially since we used a different QM basis set).

3.8 Conclusion

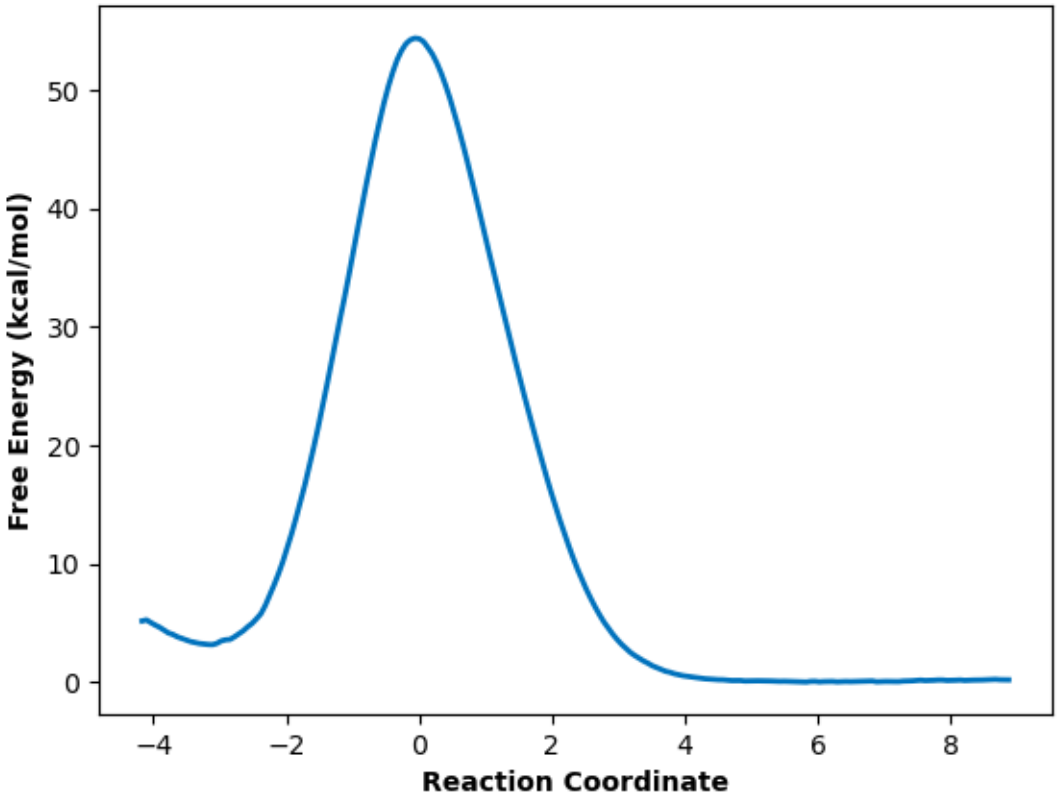
We have illustrated a full workflow with ATESA, beginning with only a SMILES string and definitions for two stable states, and ending with a validated reaction mechanism and full free energy profile. This same workflow can be adapted with minimal changes for almost any rare event you may want to study. To summarize what was described above, the primary changes that would have to be made are simply:

- Replacing the Amber input files in the specified `path_to_input_files` directory with ones appropriate for the desired simulations;
- Identifying appropriate definitions for the desired stable states to use for `commit_fwd` and `commit_bwd`; and
- Modifying the configuration file's *Batch Template Settings* to make efficient use of available computational resources when running the desired simulations.


THIS IS NOT YOUR FREE ENERGY PROFILE
This plot should probably be smooth.
Otherwise, you may have some sampling abnormalities.
Mean Value
0.1
0.0
−0.1
−0.2
−0.3
−4
−2
0
2
4
6
8
Reaction Coordinate



Also note that although we studied a single-step reaction here, ATESA can also be used for multi-step reactions. Simply set `commit_bwd` to identify the reactant state and `commit_fwd` to the first stable intermediate, and perform the workflow as above; then repeat with `commit_bwd` set to the first stable intermediate and *commit_fwd* set to the second stable intermediate (if any) and repeat; and so on, until the products state is reached. The resulting free energy profiles from each step can be stitched together to obtain the full reaction energy pathway.


Free Energy (kcal/mol)
50
40
30
20
10
0
−4
−2
0
2
4
6
8
Reaction Coordinate

CHAPTER 4

The Configuration File

The configuration file is the primary means of controlling the behavior of ATESA. In order to support the wide array of functionality that any given user may need, the configuration file supports many options and can be quite long; however, in most cases a relatively short configuration file will be sufficient. This page provides some recommendations for building the configuration file for a handful of common use cases, and provides detailed documentation for each setting. **The easiest way to get started is to modify the relevant example config files described on the :ref:'ExampleStudy' page!**

The contents of the configuration file are read line-by-line into ATESA as literal python code, which enables invocation of python built-in functions as well as methods of pytraj and numpy (and anything else you may wish to import). This means comments can be included in-line or on their own lines preceded by a '#' character, and blank lines are simply ignored. **Warning**: This input is not sanitized in any way. For this reason among others, "shutil.rmtree('/')" makes for a poor working directory!!

4.1 Core Settings

Certain settings should be given for every job. The following settings do not have valid default values and should be set in every configuration file:

```
job_type                # Valid options: 'find_ts', 'aimless_shooting', 'committor_
↪analysis', 'equilibrium_path_sampling'
batch_system            # Valid options: 'slurm', 'pbs'
restart                 # Valid options: True, False
overwrite               # Valid options: True, False
topology                # Valid options: Absolute or relative path as a string
working_directory       # Valid options: Absolute or relative path as a string
```

`job_type`

The type of simulations to perform. Details on each type are given below in the *Basic Workflow* section.

`batch_system`

Indicates the type of batch system on which ATESA is running. Supported options are 'slurm' and 'pbs' (the latter is also known as TORQUE).

`restart`

Indicates whether this is a new job (False) or a continuation of an old one *in the same working directory* (True).

`overwrite`

Indicates whether to delete the existing working directory (if one exists) and create a new one *if and only if* restart = False (has no effect otherwise). If restart = False, overwrite = True will *always* delete the working directory if it exists; conversely, overwrite = False will *never* delete it (regardless of the "restart" setting).

`topology`

An absolute or relative path given as a string and pointing to the simulation topology file.

`working_directory`

An absolute or relative path given as a string and pointing to the desired working directory (this can be omitted if the working directory is set in the command line). This is the directory in which all of the simulations will be performed. It will be created if it does not exist.

4.1.1 Find Transition State

If you want to perform aimless shooting but don't have a candidate transition state structure from which to begin (or just want to obtain more), ATESA can automatically build them from a given product or reactant state structure. If you do already have a transition state candidate to begin with, you can skip this step. In addition to the *Core Settings*, such a job should be set up as:

```
job_type = 'find_ts'
initial_coordinates = [<coord_file>]
commit_fwd = [...]
commit_bwd = [...]
max_moves = 5
```

In this case, <coord_file> should represent a structure in either the "fwd" or "bwd" commitment basin. See *Commitment Basin Definitions* for details on the *commit_fwd* and *commit_bwd* options.

The *find_ts* job type works by applying a modest and steadily increasing restraint to the atoms that make up the target commitment basin definition in order to force the desired rare event to take place, and then automatically performing a small amount of aimless shooting from structures near the middle to identify suitable transition states. Working transition state structures are identified in the file "status.txt" in the subdirectory "as_test" created in the find_ts working directory. The aimless shooting portion can be safely interrupted if a suitable transition state (*i.e.*, one with a non-zero acceptance ratio) has been identified. This works better for some systems than others, and depending on the basin definitions may result in a transition state that technically falls along the separatrix, but is in fact far from the minimum energy transition path. The user should carefully sanity-check the resulting structure(s).

If find_ts gives you trouble, one alternative option for obtaining a good initial transition state structure is to run unbiased simulations at a high temperature to accelerate the transition, though this runs the risk of pushing the system into otherwise inaccessible configurations, and is beyond the scope of this documentation.

4.1.2 Aimless Shooting

The most central function of ATESA is aimless shooting. If you have one or more initial transition structure guesses and want to begin transition path sampling, you should call ATESA with the following settings (in addition to the *Core*

Settings above):

```
job_type = 'aimless_shooting'
initial_coordinates = [<coord_file_1>, <coord_file_2>, ...]
cvs = [<cv1>, <cv2>, ...]
commit_fwd = [...]
commit_bwd = [...]
```

See *Commitment Basin Definitions* for details on the *commit_fwd* and *commit_bwd* options and *CV Settings* for details on the *cvs* option (the *cvs* option can be omitted if the *auto_cvs_radius* option is sufficient).

These settings will automatically use *Information Error* as a termination criterion with the default settings. These can be modified as described in *Aimless Shooting Settings*.

While aimless shooting is running, you can check on its progress using the "status.txt" file found in the working directory. You should look for the average acceptance ratio across all threads to be in the range of at least 10-30%; lower values may indicate a poor initial transition state guess (see *Troubleshooting*).

4.1.3 Committor Analysis

After completing aimless shooting, the next step is to obtain a reaction coordinate and verify it with committor analysis. Before running committor analysis, the user should call likelihood maximization with their preferred settings (see *lmax.py: Likelihood Maximization*) and then reaction coordinate evaluation using the aimless shooting working directory and the resulting reaction coordinate (see *rc_eval.py: Reaction Coordinate Evaluation*).

Then, committor analysis is performed through the main ATESA executable with the following settings (in addition to the *Core Settings* above):

```
job_type = 'committor_analysis'
as_settings_file = <as_settings_file>
path_to_rc_out = <rc_out_file>
rc_threshold = <threshold>
```

The *as_settings_file* should point to the the "settings.pkl" file in the previous aimless shooting working directory; see the last entry in *CV Settings*. The *path_to_rc_out* option should point the output file of `rc_eval.py`, which by default is named "rc.out" and located in the aimless shooting working directory. The value of *rc_threshold* should be selected by manually inspecting the file indicated by *path_to_rc_out* and finding the appropriate cutoff value to include the desired number of initial coordinates in committor analysis (a good choice is approximately 200). For example, if the absolute value of the 200th line of rc.out is 0.09, you might choose *rc_threshold* = 0.1. Note that larger values of rc_threshold make the result less powerful, as structures further from the transition state separatrix are included. If you feel you have to choose a large value of *rc_threshold* in order to include enough structures for committor analysis, it may indicate that you need to perform more aimless shooting.

The working directory here should NOT be the same as the aimless shooting directory containing the data to perform committor analysis with (although it can be a subdirectory). The aimless shooting directory will be identified as needed using the *path_to_rc_out* setting (which should be inside the aimless shooting working directory). The working directory for a committor analysis job should be a new directory, though it can be a subdirectory of the aimless shooting directory if desired.

The results of a committor analysis job are written to the file "committor_analysis.out" in the working directory. Each line in this file gives the ratio of jobs that committed to the "forward" basin to the total number of jobs that committed to either basin. For more details on interpreting these results, see *What is Committor Analysis?*.

4.1.4 Umbrella Sampling

The final analysis step after a satisfactory committor analysis run is to obtain the free energy profile along the reaction coordinate. The preferred method for this is umbrella sampling, which is automated in ATESA using PLUMED by default.

The conditions under which equilibrium path sampling should be used instead are as follows:

- PLUMED is strictly not available; or
- The desired reaction coordinate contains unusual CV types. The supported CV types are: distances, angles, dihedrals, and differences of distances. ATESA's automatically generated CVs are only ever of these types; or
- Umbrella Sampling (even when pathway-restrained, see *What is Pathway-Restrained Umbrella Sampling?*) is unable to produce a reasonable free energy profile.

Umbrella Sampling can be called in ATESA through the main executable using the following settings (in addition to the *Core Settings* above):

```
job_type = 'umbrella_sampling'
rc_definition = <rc_definition>
as_settings_file = <as_settings_file>
as_out_file = <as_output_file>
us_auto_coords_directory = <as_directory>
```

The *as_settings_file* should point to the the "settings.pkl" file in the previous aimless shooting working directory; see the last entry in *CV Settings*. See *Reaction Coordinate Definition* for details on the *rc_definition* option. The same *rc_definition* should be used for both committor analysis and umbrella sampling (or, if the *path_to_rc_out* option was used for committor analysis, then the same RC should be used for umbrella sampling and the run of `rc_eval.py` that generated the "rc.out" file for committor analysis). The *as_out_file* option should point to the same file that was used as the input for the `lmax.py` run that generated the reaction coordinate being used.

The `us_auto_coords_directory` option can be used to allow ATESA to automatically select initial coordinates from accepted aimless shooting trajectories. This option should point to the working directory of the aimless shooting run that produced the data under analysis. Alternatively, if the *initial_coordinates* option is used, the provided files can (and probably should) be *trajectories*, not just single-frame coordinate files. These trajectories should represent at least one full transition path, from one basin to another. The easiest choice is to take the most recent accepted aimless shooting move and provide both its 'fwd' and 'bwd' trajectory files. ATESA will look through each frame of these trajectories to find the best initial coordinates for each sampling window.

Beyond these settings, the user will probably want to set the lower and upper bounds of the reaction coordinate to sample over. See *Umbrella Sampling Settings* for details. After an umbrella sampling run, the data can be converted into a free energy profile using *mbar.py: Energy Profiles from US*.

4.1.5 Equilibrium Path Sampling

As a more generalized but less efficient alternative to umbrella sampling, ATESA supports equilibrium path sampling (EPS) to obtain this profile through the main executable, using the following settings (in addition to the *Core Settings* above):

```
job_type = 'equilibrium_path_sampling'
rc_definition = <rc_definition>
as_settings_file = <as_settings_file>
as_out_file = <as_output_file>
initial_coordinates = [<coord_file_1>, <coord_file_2>, ...]
```

The *as_settings_file* should point to the the "settings.pkl" file in the previous aimless shooting working directory; see the last entry in *CV Settings*. See *Reaction Coordinate Definition* for details on the *rc_definition* option. The same

rc_definition should be used for both committor analysis and equilibrium path sampling (or, if the *path_to_rc_option* was used for committor analysis, then the same for equilibrium path sampling and `rc_eval.py`). The *as_out_file* option should point to the same file that was used as the input for the `lmax.py` run that generated the reaction coordinate being used. Finally, *initial_coordinates* can be used to select any number of coordinate files, usually taken from shooting points (coordinate files whose names end with "_init.rst7") from aimless shooting. By default, EPS windows that have fewer than 20 initial coordinate files are filled up by the endpoints of simulations in adjacent windows, but it is still advisable to provide at least a handful of unique starting structures.

Beyond these settings, the user will probably want to set the lower and upper bounds of the reaction coordinate to sample over. See *Equilibrium Path Sampling Settings* for details.

EPS is a highly generalized free energy method that does not rely on restraints or biases of any kind. The cost of this benefit is that it is also among the least efficient free energy methods available, requiring a relatively large amount of simulation to acquire comparable sampling coverage to, for example, umbrella sampling. For this reason, EPS is recommended for use only in cases where other methods are unsuitable or unavailable.

CAUTION: Because equilibrium path sampling measures the full energy profile instead of merely assessing the endpoints of simulations (as in aimless shooting and committor analysis), it is very sensitive to errors in the evaluation of the energy of any given state. For this reason, it is completely possible to have obtained reasonable aimless shooting and committor analysis results with a system or simulation parameters that are not suitable for equilibrium path sampling, for example owing to poor SCF convergence in QM calculations along portions of the RC. ATESA can NOT identify such errors on its own, and may produce EPS results that are not correct (but may appear reasonable at first glance)! It is the responsibility of the user to ensure that the EPS simulations are well-behaved and do not suffer from severe energetic or undersampling errors. Please direct any questions to our GitHub page as an issue with the "question" label.

The raw output data from an EPS run is stored in the working directory as "eps.out". This data can be converted into an energy profile using boltzmann_weight.py (see *boltzmann_weight.py: Energy Profiles from EPS*), which calculates the relative probabilities of states within each bin and converts these into relative free energies.

4.2 Full Configuration Options

Here, the full list of valid configuration file options are documented (excluding the *Core Settings*, documented above.) **NOTE:** most options below will not need to be touched by most users. Exceptions (*i.e.*, options that basic users may or should be interested to set or change from their defaults) are denoted with a "‡".

4.2.1 Batch Template Settings

These settings control how batch file templates are filled out. In general, 'init' simulations are very short (one step) and only used to prepare something for a longer simulation, whereas 'prod' simulations are where the primary data collection of a job takes place. When and whether 'init' or 'prod' variables are used is controlled by the job type.

These settings are used to fill in the template slots in the user-provided template files. If you do not wish for ATESA to use an option, you can simply omit its template slot from the appropriate file and leave it unset in the configuration file.

`init_nodes`

The number of compute nodes to request for 'init' simulations, given as an integer. Default = 1

`init_ppn`

The number of cores or processes to request per node (ppn: "processes per node") for 'init' simulations, given as an integer. Default = 1

`init_mem`

The amount of RAM to request for 'init' simulations, given as a string of appropriate format for the batch system. Depending on the batch system, this may be interpreted as total memory, or as memory per core. Default = '4000mb'

`init_walltime`

The amount of walltime (real time limit for the batch job) to request for 'init' simulations, given as a string of appropriate format for the batch system. 'init' simulations are very quick, but if one does not produce the necessary results in this time and is cancelled early, it will be resubmitted; err on the side of more time. Default = '00:30:00'

`init_solver` ‡

The name of the executable to use to perform 'init' simulations, given as a string. Default = 'sander' (which is specific to Amber)

`init_extra`

An additional template slot for 'init' simulations to be used however the user sees fit. This option is provided in case a user has an unforseen need to template something other than the above options. Default = '' (an empty string)

`prod_nodes` ‡

The number of compute nodes to request for 'prod' simulations, given as an integer. Default = 1

`prod_ppn` ‡

The number of cores or processes to request per node (ppn: "processes per node") for 'prod' simulations, given as an integer. Default = 8

`prod_mem` ‡

The amount of RAM to request for 'prod' simulations, given as a string of appropriate format for the batch system. Depending on the batch system, this may be interpreted as total memory, or as memory per core. Default = '4000mb'

`prod_walltime`

The amount of walltime (real time limit for the batch job) to request for 'prod' simulations, given as a string of appropriate format for the batch system. Err on the side of more time. Default = '02:00:00'

`prod_solver` ‡

The name of the executable to use to perform 'prod' simulations, given as a string. Default = 'sander' (which is specific to Amber)

`prod_extra`

An additional template slot for 'prod' simulations to be used however the user sees fit. This option is provided in case a user has an unforseen need to template something other than the above options. Default = '' (an empty string)

4.2.2 File Path Settings

These settings define the paths where ATESA will search for user-defined input files and template files. They are used in all job types.

`path_to_input_files` ‡

Absolute path (as a string enclosed in quotes) to the directory containing the input files. The default is the directory 'data/input_files' located inside the ATESA installation directory.

`path_to_templates` ‡

Absolute path (as a string enclosed in quotes) to the directory containing the template files. The default is the directory 'data/templates' located inside the ATESA installation directory.

4.2.3 CV Settings

These settings define the combined variables (CVs) for the job. In aimless shooting, these are the values that are written to the output file for interpretation by likelihood maximization in building the reaction coordinate (RC). In umbrella sampling, equilibrium path sampling, and committor analysis, they are used to evaluate the RC (see *Reaction Coordinate Definition*). They are not used in `find_transition_state` jobs.

`cvs` ‡

A list of CV definitions, given as strings, as in: [<cv1>, <cv2>, ... <cvN>] (where the contents of each pair of angled braces is a string). Each item is interpreted as raw python code (caution: unsanitized) that returns the desired CV value (in a format that can cast to a float). In addition to built-in python functions, calls to pytraj, mdtraj, and numpy (as both 'numpy' and 'np') are supported. In support of pytraj and mdtraj calls, the following variables are available for use in CV definitions:

traj: the coordinate file being evaluated, as a pytraj.iterload object (for use with pytraj function calls). Note that pytraj returns units of angstroms and degrees for distances and angles, respectively.

mtraj: the coordinate file being evaluated, as an mdtraj.load object (for use with mdtraj function calls). Note that mdtraj returns units of nanometers and radians for distances and angles, respectively.

traj_name: the name of the coordinate file, as a string

settings.topology: the name of the topology file, as a string

ATESA will handle making sure that the CV definitions are only ever used to evaluate a single frame at a time, but it is up to the user to ensure that each item in *cvs* returns exactly the desired CV value as a float or integer (and not, for example, a one-length list containing that value). For example, the following value of *cvs* would interpret the interatomic distance between atoms 1 and 2 as CV1, the difference between the distances 3-to-4 and 5-to-6 as CV2, and the angle formed by atoms 7-8-9 as CV3 (all on one line):

```
cvs = ['pytraj.distance(traj, \'@1 @2\')[0]',
       'pytraj.distance(traj, \'@3 @4\')[0] - pytraj.distance(traj, \'@5 @6\
↪')[0]',
       'pytraj.angle(traj, \'@7 @8 @9\')[0]']
```

Notice in particular the usage of "traj" as the first argument in these pytraj function calls, the escaped single-quote characters within each function call, and the specification of the zero'th indexed item returned by each function call (as these pytraj functions return one-length lists). Keep in mind that atoms referenced in pytraj are 1-indexed while in mdtraj they are 0-indexed. See the respective documentation pages for pytraj and mdtraj for more information about using these packages. Default = ['']

`auto_cvs_radius` ‡

Alternatively or in addition to defining explicit CVs with the *cvs* option, this option can be used to automatically obtain and use CVs representing every bond, angle, and dihedral, respectively, consisting of contiguously bonded atoms within *auto_cvs_radius* angstroms of any of the atoms present in the the *commit_fwd* or *commit_bwd* options (see *Commitment Basin Definitions*). In addition, difference-of-distance CVs are produced wherever an atom appears in two or more elements in the commitment definitions. For example, if the following combination of settings is provided:

```
commit_fwd = ([101, 102, 102], [103, 104, 105], [1.5, 2.0, 1.7], ['lt', 'gt',
↪ 'lt'])
commit_bwd = ([101, 102, 102], [103, 104, 105], [2.0, 1.5, 2.5], ['gt', 'lt',
↪ 'gt'])
auto_cvs_radius = 5
```

Then every bond, angle, and dihedral consisting of atoms within at least 5 angstroms of atoms 101, 102, 103, 104, or 105 (indexed from 1) would be included as a CV, in addition to the difference of distances 102-to-104 and 102-to-105. The index, description, and code used to evaluate each CV derived in this manner is printed to the file "cvs.txt" in the working directory. Automatic CVs can be disabled by setting *auto_cvs_radius* to 0. If *auto_cvs_radius* is greater than 0 and CVs are also defined manually using the *cvs* option, then the manually defined CVs are appended to the end of the list of automatically generated CVs.

Using *auto_cvs* treats all of the atoms in *commit_fwd* and *commit_bwd* as bonded to one another for the purposes of determining CVs, so there is no need to define these CVs manually. Examples of CVs that *should* be defined manually if desired include any other distances, angles, or dihedrals defined using atoms that are not contiguously bonded to one another (*e.g.*, a distance between nearby charged particles), or any other custom CVs of unique relevance to the rare event of interest.

The number of CVs that are created using this option can grow very large very quickly when large radii are chosen, which in extreme cases can cause significant I/O overhead and slow down calls to *lmax.py: Likelihood Maximization* (which is used internally when evaluating the information error termination criterion). Furthermore, when using *auto_cvs* even with relatively small radii, it is unlikely that `lmax.py` runs to optimize models of three or more dimensions will be feasible without using either the *-r* or *–two_line_test* flags. Default = 5

`auto_cvs_exclude_water`

A boolean. If False, the behavior where *auto_cvs_radius* is greater than zero includes any atoms belonging to water molecules within the given radius of the commitment-defining atoms. Otherwise, water molecules are excluded. Atoms that are part of a commitment definition are never excluded in this way and water atoms may still be included in CVs defined using the *cvs* option if desired. Default = True

`auto_cvs_exclude_hydrogen`

A boolean. If False, the behavior where *auto_cvs_radius* is greater than zero includes any hydrogen atoms within the given radius of the commitment-defining atoms. Otherwise, hydrogens are excluded. Atoms that are part of a commitment definition are never excluded in this way and hydrogen atoms may still be included in CVs defined using the *cvs* option if desired. Default = True

`auto_cvs_type`

A string, either 'pytraj' or 'mdtraj'. This controls the package used to define the CVs built by auto_cvs. Note that you are free to use both pytraj and mdtraj with the *cvs* option regardless of this setting; it only controls the CVs built automatically. WARNING: there appears to be a memory leak associated with using pytraj, so mdtraj is recommended for now. Default = 'mdtraj'

`include_qdot`

A boolean indicating whether the rate of change of each CV defined in *cvs* should also be counted as a CV. These values can be used in inertial likelihood maximization to favor reaction coordinates with high transmission coefficients (see Peters 2012 and/or *lmax.py: Likelihood Maximization*). Rate of change CVs come after the rest of the CVs in the aimless shooting output file, as in:

```
<basin> <- <CV1> <CV2> ... <CVn> <qdot1> <qdot2> ... <qdotn>
```

Default = True

`as_settings_file` ‡

A string pointing to a previously generated "settings.pkl" file. These files are produced in the working directory whenever ATESA is run. By pointing a new ATESA job to the settings.pkl file of a previous one using this option, the CVs and commitment basin definitions from that run are loaded. The loaded settings take priority, and the following configuration file options will be ignored if an *as_settings_file* is indicated:

```
cvs
commit_fwd
commit_bwd
include_qdot
auto_cvs_radius (and associated options)
```

Use of this option is recommended in particular for committor analysis, umbrella sampling, and equilibrium path sampling jobs following an aimless shooting job. In this case, *as_settings_file* should point to the settings.pkl file from that aimless shooting job. Default = ''

`sigfigs`

An integer specifying the number of digits after the decimal place to include when reporting CV values in the output file. Larger numbers mean larger files, which means more I/O overhead and memory consumption. In general you should only need to change this option if you're using a custom CV that requires more digits. Default = 3

4.2.4 Initial Coordinates

`initial_coordinates` ‡

The initial coordinates for the job, used as appropriate for the given job type, given as a list of strings. All jobs require initial coordinates (although in committor analysis they may be identified using the *path_to_rc_out* option instead, if *committor_analysis_use_rc_out* is True (which is not the default), and in umbrella sampling they may be identified automatically if *us_auto_coords_directory* is set). All the coordinate files identified in this list are used to spawn independent threads. The same file can be given more than once to spawn multiple threads with the same initial coordinates. Default = ['']

4.2.5 Commitment Basin Definitions

These options define the geometric boundaries past which simulations are considered to have committed to the specified energetic basin (usually either "products" or "reactants"). In general, the basin definitions should be chosen so as to be mutually exclusive and deep enough into the state that they represent to ensure that simulations are not terminated before they have truly "committed" that a given basin. When in doubt, choose a more conservative basin definition.

Commitment basin definitions are required in aimless shooting, committor analysis, and find transition state, but are not used for equilibrium path sampling.

`commit_fwd` ‡

The definition of the "forward" basin (usually products). The syntax for this option is as follows:

```
commit_fwd = ([index_1a, index_1b, ... index_1n], [index_2a, index_2b, ...␣
↪index_2n],
[dist_1, dist_2, ... dist_n], ['gt'/'lt', 'gt'/'lt', ... 'gt'/'lt']
```

This is interpreted as: the distance between the atoms with indices *index_1a* and *index_2a* must be either greater than (*'gt'*) or less than (*'lt'*) the distance dist_1 (in Å), and so on. Atom indices are strictly 1-indexed; if the model itself is 0-indexed, you need to increment each index by one when including it in commitment basin definitions. Each item in each list sharing the same subindex (*e.g.*, index_1a, index_2a,

dist_1, and the first 'gt'/'lt') is treated as a single criterion for commitment to the basin, and only once ALL the criteria are met simultaneously is the simulation considered committed. No default value.

`commit_bwd` ‡

This options behaves just like *commit_fwd*, but for the "backward" basin (usually reactants). No default value.

4.2.6 Reaction Coordinate Definition

The options define the reaction coordinate (RC) in terms of the CVs in the *cvs* options (see *CV Settings*). These options are only used in committor analysis and equilibrium path sampling.

`rc_definition` ‡

The RC definition itself as a string. This string is interpreted as raw python code, with each CV term replaced by the appropriate value. For example:

```
rc_definition = '-0.52 + CV4*1.23 - CV2/CV25'
```

would be interpreted as -0.52 plus the value of CV4 (the fourth item of *cvs*, one-indexed), minus the ratio of the values of CV2 and CV25. No default.

`rc_reduced_cvs`

A boolean indicating whether the CVs in *rc_definition* should be normalized to between 0 and 1 using the minimum and maximum values observed (True), or if the raw values should be used (False). Default = True

`as_out_file` ‡

Path to an aimless shooting output file. The minimum and maximum values of each CV contained herein are used to reduce the CV values in the RC if *rc_reduced_cvs* = *True* (otherwise this option is unused). The CVs must be given in the same order in this file as they are in the *cvs* option (as is the case if the same *cvs* option was used in the aimless shooting run that produced this file). This option must be set explicitly if the aimless shooting output file is not present in the working directory, as is usually the case by default in committor analysis and equilibrium path sampling. Default = 'as_raw.out'

4.2.7 Aimless Shooting Settings

These settings are specific to aimless shooting runs only.

`min_dt`

When a shooting move is accepted, the next move begins from an early time step of the accepted trajectory. This option sets the minimum number of simulation time steps away from the initial coordinates that the initial coordinates of the next move may be (in either direction). Higher options will explore the configuration space more quickly, but will decrease the move acceptance ratio. Default = 1

`max_dt`

As *min_dt*, but sets the maximum number of steps. The number of steps can be fixed by setting both options to the same value. Higher options will explore the configuration space more quickly, but will decrease the move acceptance ratio. Default = 10

`always_new`

A boolean. If True, failed shooting moves pick a new frame in the range of *min_dt* to *max_dt* from the last accepted trajectory in the thread as their initial coordinates for the next attempt. If False, the same coordinates are used repeatedly with only the velocities randomized at each step. Default = True

`resample`

A boolean. If True, aimless shooting will NOT be performed, and instead the existing aimless shooting data found in *working_directory* are used to produce new output files based on the settings of the current ATESA job. The primary usage of this option is to add additional CVs to the output files without needing to repeat any simulations. This option can also be used with committor analysis (see below). Default = False

`full_cvs`

A boolean. If *True*, resampling and aimless shooting also produce the output file "as_full_cvs.out", which is used as an input for the umbrella sampling option "us_pathway_restraints_file". When this option is set to *True* for resampling, this file is produced using every accepted trajectory in the working directory, which will only be the most recently accepted trajectories in each thread unless *cleanup = False* was set during aimless shooting (which is **not** the default.) When this option is set to *True* during an aimless shooting job, the file is written to after each successful aimless shooting move, regardless of the *cleanup* setting, although be warned that this can slow down aimless shooting somewhat, especially when processing and/or I/O performed directly by ATESA is a significant bottleneck for your particular resources and simulations. Default = False

`degeneracy` ‡

An integer number of "degenerate" threads to begin from each given initial coordinate file. This option is merely a shortcut for increasing the degree of parallelization in the sampling process, but keep in mind that higher values sample less efficiently, especially for systems that decorrelate slowly and/or when *min_dt* and *max_dt* are small. Default = 1 (no degeneracy, only one thread per initial coordinate file)

`cleanup` ‡

A boolean. If True, trajectory files for each shooting move are deleted after they no longer represent the most recent accepted trajectory in a thread. The initial coordinate files from each shooting move are always retained for resampling. This option is useful to greatly reduce the amount of storage space consumed in the course of aimless shooting; however, note that when *cleanup = True*, you will not be able to resample the full set of aimless shooting trajectories to produce a new "as_full_cvs.out" file when *resample* and *full_cvs* are both *True* (although it will still be produced as usual during aimless shooting if *full_cvs = True* is declared in the aimless shooting configuration file). The "as_full_cvs.out" file is only used for the optional "pathway restraints" setting in umbrella sampling, so if you don't plan to use umbrella sampling or if you're sure you don't need that option, you can safely leave this on True. Default = True

`max_moves`

The maximum number of moves that an aimless shooting thread can make before terminating. This can be useful for limiting the amount of resource consumption for each thread. If a negative value is given, there is no limit. Default = -1

`max_consecutive_fails`

The maximum number of consecutive failed simulations permitted in any single thread before the entire aimless shooting run terminates. In this case, "failed" simulations refers to simulations that do not produce valid output files – it has nothing to do with whether a shooting move is "accepted" or "rejected". Sometimes simulations occasionally fail for innocuous reasons, but if many failures happen in a row it usually indicates some systemic issue that warrants investigation. If a negative value is given, there is no limit. Default = 10

The following options concern the information error convergence criterion in aimless shooting, which is turned on by default. See *Information Error* for a general description of this method.

`information_error_checking`

A boolean. If True, the information error convergence criterion is used to terminate aimless shooting once the mean parameter standard error falls below the *information_error_threshold*. If False, none of the other options beginning with "information_error" are used. Default = True

`information_error_threshold`

The threshold of mean parameter standard error below which the information error termination criterion is satisfied and aimless shooting ends. Error generally diminishes as roughly the square root of the number of samples, so an order of magnitude lower threshold requires approximately a hundred times more sampling (assuming that the CVs constituting the maximum likelihood model remain the same throughout). It is a good practice to manually inspect the *info_err.out* file in the working directory after termination to ensure that it appears to be converged to within the user's desired tolerance (*i.e.*, that it did not terminate for some other reason). Default = 0.1

`information_error_freq`

An integer number of shooting moves between each check of the information error convergence criterion. Too-small values can cause some significant overhead in the main ATESA process, but this usually does not significantly affect aimless shooting sampling efficiency. Aimless shooting runs with *resample* = *True* or *restart* = *True* will reassess information error if this option has changed since the last run. Default = 250

`information_error_lmax_string`

Part of the command string passed to lmax.py when assessing information error. Information error is based on the reaction coordinate model produced using `lmax.py` on the decorrelated aimless shooting output files, with options for the *-o*, *-i*, and *-q* flags set automatically. All other options (most importantly, *-k*, *-f*, *-r*, *–two_line_test*, and *–two_line_threshold*) can be set using this option in the configuration file. See *lmax.py: Likelihood Maximization* for documentation on `lmax.py`. You should use this option if you have a preference for tying your aimless shooting termination criterion to a particular model building strategy other than the default, but keep in mind that certain settings (especially those depending on the *-k* option) may take a very long time to assess. Aimless shooting runs with *resample* = *True* or *restart* = *True* will reassess information error if this option has changed since the last run. Default = '–two_line_test'

`information_error_max_dims`

The maximum number of dimensions that may be included in the "two_line_test" reaction coordinate assessed during information error checking (if the –two_line_test option is in use; see *The two_line_test Option*). If this value is set, the algorithm will move forward with a model of this dimensionality if no smaller model meets the two_line_test –two_line_threshold criterion. Larger values are more robust but may result in an undesirably high-dimensional RC being used for the convergence criterion, which in turn could lead to slower convergence of the information error. If a value less than zero is given, there is no maximum. Aimless shooting runs with *resample* = *True* or *restart* = *True* will reassess information error if this option has changed since the last run. Default = 6

`information_error_override`

A boolean. The default likelihood maximization procedure with two_line_test raises an exception if even an RC with every single candidate CV included is unable to pass the two_line_test convergence criteria. This generally indicates that one or more important CVs are missing from consideration (in the *cvs* option). If this option is set to True, the exception is bypassed and the largest-dimensional RC available is used. Use with caution. Default = False

`two_line_threshold`

A float defining the maximum ratio of the slopes of the lines constituting one of the likelihood maximization "two_line_test" algorithm's convergence criteria. See *The two_line_test Option* for details. Default = 0.5

4.2.8 Committor Analysis Settings

These options are specific to committor analysis runs only.

`committor_analysis_n`

The number of independent simulations to run *for each initial structure* during committor analysis. This number should be large enough to distinguish confidently between structures that make good and poor transition states. The default choice is fairly safe, but increasing it may smooth out the result in some cases. It should probably never be decreased. Default = 10.

`committor_analysis_use_rc_out` ‡

A boolean. If True, the choice of initial coordinates is based not on the *initial_coordinates* option in the configuration file, but on the contents of an RC output file, based on the following two options. Default = False

`path_to_rc_out` ‡

The path to the RC output file, given as a string. This file should contain the name of each shooting move file (aimless shooting files ending in '*_init_fwd.rst7') followed by a colon, a space, and then the value of the reaction coordinate at that point. For example:

```
initial_coords_1_1_init_fwd.rst7: -0.0913
initial_coords_1_2_init_fwd.rst7: -0.1125
```

Files of this sort can be automatically generated using the auxiliary script *rc_eval.py: Reaction Coordinate Evaluation*. No default.

`rc_threshold` ‡

The threshold of *absolute value* of reaction coordinate below which shooting moves in the indicated *path_to_rc_out* file will be included in committor analysis. For example, if the above example contents of such a file were used and *rc_threshold* were set to 0.1, only the first of the two files (initial_coords_1_1_init_fwd.rst7) would be used for committor analysis. The user is encouraged to check the RC output file manually before using this option to ensure that they will have enough unique initial coordinate files to produce a useful committor analysis result (roughly 200 files is a good target). Default = 0.05

`resample`

A boolean. If True, committor analysis will NOT be performed, and instead the existing committor analysis trajectories found in *working_directory* are used to produce a new output file based on the settings of the current ATESA job. The primary usage of this option is to regenerate results from completed simulations when ATESA has failed to produce them in the first place (*e.g.*, due to a crash) or when some other mistake has been made. In other words, this option is supported "just in case". Please note that in general committor analysis simulations performed to test one RC are not applicable to testing any other RC, meaning this option cannot be used to simply recycle simulations to test a newly selected RC. This option can also be used with aimless shooting, where it will be more common (see above). Default = False

4.2.9 Umbrella Sampling Settings

These options are specific to umbrella sampling (US) runs only. They mostly pertain to defining the restraints. It may be useful to run a test run first with only a few windows along a small subset of the full range of reaction coordinate values to verify these settings before continuing.

Keep in mind that if you perform a large amount of sampling and then discover that there are gaps in your sampling, you can always submit a new umbrella sampling job (in a new working directory) with the same settings but with

windows centered in the gaps, and then copy the resulting output data files (named with "_us.dat") to the original working directory to include them all in the same analysis.

`us_rc_min` ‡

The minimum value of the reaction coordinate to sample during US. It can be useful to manually evaluate the RC of an equilibrated reactant state structure, and then add some small fraction of that number again (say, 10%) to select the value for this option. Default = -12

`us_rc_max` ‡

The maximum value of the reaction coordinate to sample during US. It can be useful to manually evaluate the RC of an equilibrated product state structure, and then add some small fraction of that number again (say, 10%) to select the value for this option. Default = 12

`us_rc_step` ‡

The step size between the centers of adjacent windows during US. Window boundaries are assigned from the left, such that the first window always begins exactly at *us_rc_min*, and then steps of size *us_rc_step* delineate further windows until the next step would be equal to or greater than *us_rc_max*. Default = 0.25

`us_restraint` ‡

The weight of the energetic restraint applied in each US window, measured in kcal/(mol-units^2), where "units" is the dimension of the reaction coordinate. Restraints are applied according to the equation:

$$U = (us\_restraint) * (RC - RC_0)^2$$

where U is the restraint energy and RC_0 is the window center. Note that PLUMED calculates the restraint internally with an additional factor of 1/2, so *us_restraint* is doubled when building PLUMED input files to compensate. Too-low values of *us_restraint* sample inefficiently and may leave gaps in the sampling, but too-high values necessitate more windows along the RC and may prevent sampling from the full ensemble of appropriate configurations. The best values to choose will depend on the underlying free energy profile and the simulation parameters. The default values are a good starting point for most systems (at 300 K – significantly lower temperatures require weaker restraints and *vice versa*). Default = 50

`us_degeneracy`

The number of independent threads to run *for each window* during umbrella sampling. Setting it to 1 or less means only one thread per window. This should usually be a small number greater than one. The degenerate threads will all start from the same coordinates. Default = 5

`us_auto_coords_directory` ‡

By default, umbrella sampling uses initial coordinates generated from the files designated in the *initial_coordinates* option. Alternatively, and more conveniently, this option can be used to specify an aimless shooting working directory from which to automatically identify suitable initial coordinates. The most recent accepted trajectory from a random thread will be selected. If this option is used, the contents of *initial_coordinates* will be ignored. Default = ''

`us_pathway_restraints_file`

Implements pathway restrained umbrella sampling. This is a string giving the path to the full CVs output file (default name "as_full_cvs.out", produced during aimless shooting or resampling (*resample = True*) when *full_cvs = True*) in the desired aimless shooting working directory.

By default, the restraints applied during umbrella sampling are only along the reaction coordinate, with all other degrees of freedom left unrestrained. In some cases, this can result in errant sampling of regions of state space that technically have the desired reaction coordinate value, but do not actually fall within the transition path ensemble. Such errors are usually visible in umbrella sampling output data as discontinuities or unsmoothness in the mean value plot produced by the mbar.py analysis script; see the *Umbrella Sampling* section of the *Troubleshooting* page for more information. One possible way to

fix this is by rerunning umbrella sampling with this option set, which will apply restraints to every CV included in the aimless shooting output files. These restraints are flat and equal to zero within the range of values observed during any accepted aimless shooting trajectory, and then increase in energy steeply outside this range. In this way, and to the extent that aimless shooting explored the relevant phase space of each CV and that the included CVs cover the relevant dimensions, this option requires the umbrella sampling simulations to remain within the reaction pathway ensemble, producing much better results.

Because of the risk that important regions of state space are errantly gated off, this option should only be used as necessary, not as a first-resort. **Important:** If this option is set, you must include 'nmropt=1,' in the &cntrl namelist of the 'umbrella_sampling_prod_amber.in' file in the input_files directory. Default = ''

`us_implementation`

A string, supported options are "plumed" or "amber_rxncor".

If "plumed" is selected, umbrella sampling restraints are automatically generated and applied using PLUMED, which must be installed and available for use with Amber. Also requires that both "plumed=1" and "plumedfile={{ plumedfile }}" are declared in the umbrella sampling input file.

If "amber_rxncor" is selected, umbrella sampling restraints are implemented directly in Amber using the "irxncor" option. The version of Amber being used must support it (at time of writing, no publicly available version of Amber yet supports this option). Also requires that both "irxncor=1" and "nmropt=1" are declared in the umbrella sampling input file.

Default = 'plumed'

4.2.10 Equilibrium Path Sampling Settings

These options are specific to equilibrium path sampling (EPS) runs only.

`eps_rc_min` ‡

The minimum value of the reaction coordinate to sample during EPS. It can be useful to manually evaluate the RC of an equilibrated reactant state structure, and then add some small fraction of that number again (say, 10%) to select the value for this option. Default = -12

`eps_rc_max` ‡

The maximum value of the reaction coordinate to sample during EPS. It can be useful to manually evaluate the RC of an equilibrated product state structure, and then add some small fraction of that number again (say, 10%) to select the value for this option. Default = 12

`eps_rc_step`

The step size between the centers of adjacent windows during EPS. Window boundaries are assigned from the left, such that the first window always begins exactly at *eps_rc_min*, and then steps of size *eps_rc_step* delineate further windows until the next step would be equal to or greater than *eps_rc_max*. Default = 1

`eps_rc_overlap`

The amount of overlap between adjacent windows. This is in addition to the boundaries calculated with the above three options and can extend past the edges; for example, if the windows without overlap had been bounded as:

```
[[-2, -1], [-1, 0], [0, 1], [1, 2]]
```

Then with overlap of 0.1, the new boundaries would be:

```
[[-2.1, -0.9], [-1.1, 0.1], [-0.1, 1.1], [0.9, 2.1]]
```

Default = 0.1

`eps_n_steps`

The number of simulation timesteps to include in each EPS trajectory. Larger numbers explore more of the energy profile per simulation and may converge more quickly, but are more likely to exit their assigned window and not be counted (in whole or in part) (especially when the energy profile is very steep) and so could be less computationally efficient. Default = 6

`eps_out_freq`

The number of steps in each trajectory between writes to the EPS output file. The *eps_n_steps* option must be divisible by this number. Default = 1

`eps_dynamic_seed`

An option for automatically building new EPS threads from the tails of simulations in adjacent windows. This option can either be supplied as an integer or as a list of integers of length equal to the number of EPS windows. If just an integer is supplied, it is treated as a list of that length where each item is that integer. The value in each index in this list indicates the number of independent threads each window should support. As long as a window has fewer than this number of threads, every time a simulation in an adjacent window contains frames inside the window in question a new independent thread will begin from the earliest such frame. Only one new thread can begin from any given trajectory. Using this option it is possible to obtain a full energy profile along the RC using only a single coordinate file located at the transition state; however, use of this option can slow down convergence of the sampled distribution towards the equilibrium distribution. It is the responsibility of the user to ensure that their final dataset is equilibrium-distributed. Default = 20

`samples_per_window`

A termination criterion for EPS that terminates the sampling in a given EPS window after this number of samples within it have been written to the EPS output file. Negative values mean no limit; sampling will continue indefinitely (until ATESA is terminated by other means). Default = -1

4.2.11 Other Settings

The following settings don't belong anywhere else in particular.

`restart_terminated_threads`

A boolean. If True and if the *restart* option is also True, threads that had already terminated for any reason will be restarted along with the rest of the threads. If whatever termination criterion that caused a thread to terminate is still met after a single additional step, it will be terminated again. Default = False

CHAPTER 5

Auxiliary Scripts

This page documents the various other python scripts packaged with ATESA that can be called directly (i.e., not through the primary `atesa` executable).

5.1 lmax.py: Likelihood Maximization

In addition to the core job types performed through calls directly to `atesa`, ATESA comes packaged with a separate likelihood maximization (LMax) script for obtaining reaction coordinates (RC's) from aimless shooting output.

The product of aimless shooting is a (large) set of combined variable (CV) values paired with corresponding commitment basins (products or reactants). In order to convert this information into a usable form, likelihood maximization selects a model that describes the reaction progress in terms of relatively few parameters. ATESA supports the intertial likelihood maximization procedure first published in Peters 2012, in addition to the original non-inertial procedure.

Likelihood maximization is invoked from the command line as:

```
lmax.py -i input_file [-k dimensions | -r dimensions | --two_line_test [--two_line_
↪threshold ratio]] [-f fixed_cvs] [-s skip] [-q qdot_setting] [-o output_file] [--
↪plots] [--quiet]
```

-i input_file The only strictly required argument for lmax.py, *input_file* should point to the aimless shooting output file of interest. Usually, this should be the longest "decorrelated" aimless shooting output file in the target working directory (named as "as_decorr_<length>.out", where <length> is the number of shooting points included before decorrelating). Decorrelated output files include only the shooting points after the point where all of the CVs have no correlation with their initial values for that thread with at least 95% confidence, or in other words when the autocorrelation across all CVs is less than or equal to 1.96 / sqrt(n) for n shooting moves in the thread. These files are produced automatically when using the information error convergence criterion criterion with aimless shooting, but otherwise they can be produced by running a new aimless shooting job with *resample* = True.

-k dimensions This option mutually exclusive with *-r* and *–two_line_test*. The *dimensions* argument should be an integer number of dimensions to include in the final reaction coordinate. This options will always return a reaction coordinate with "k" dimensions (assuming there are at least "k" CVs in the input file), and will arrive at that number of dimensions by comparing every possible k-dimensional combination of CVs. Note that this

can take a prohibitively long time when there are a large number of CVs and/or shooting moves to consider, in which case use of either the *-r* or *–two_line_test* option is encouraged instead. When using this option, the order that the CVs appear in the final RC is arbitrary.

-r dimensions This option mutually exclusive with *-k* and *–two_line_test*. The *dimensions* argument should be an integer number of dimensions to include in the final reaction coordinate. This options will always return a reaction coordinate with "r" dimensions (assuming there are at least "r" CVs in the input file), but unlike the *-k* setting, arrives at an r-dimensional result incrementally by finding the best 1-dimensional RC, then the best 2-dimensional RC that includes the coordinate from the best 1-dimensional RC, and so on. This procedure is much faster for high-dimensional data but may not result in the best possible RC, especially in cases where some particular *combination* of CVs is highly predictive of the simulation outcome while the individual CVs are not. The order that the CVs appear in the final RC is indicative of the order in which they were added (leftward terms first).

–two_line_test This option mutually exclusive with *-k* and *-r*. If this argument is given, the dimensionality of the result is determined by assessing the relative slopes of best-fit lines on different portions of the scoring data for successively higher-dimensional RCs. The order that the CVs appear in the final RC is indicative of the order in which they were added (leftward terms first). The exact operation of this algorithm is described in more detail in the *The two_line_test Option* section below.

–two_line_threshold ratio Sets the threshold of the ratio of slopes below which the two_line_test two-line test may pass. See *The two_line_test Option* for details. Default = 0.5

-f fixed_cvs This option specifies one or more CVs that are required in the final RC. For example, *-k 4 -f 12 31* would return the best four-dimensional RC that contains both CV12 and CV31. The number of required CVs specified with this option must be less than or equal to the number given for *-k* or *-r* if those options are given.

-s skip Similar to *-f*, but this option specifies CVs NOT to include in the RC. This can be useful when your dataset includes dimensions that appear to be well-correlated with the commitment basin but that you have some reason to believe should not be included in the model. Can also be used to obtain runners up to the best models produced with the *–two_line_test* and *-r* settings, by running *lmax.py* again and skipping the last dimension added to the model at hand (the right-most term). Obviously, the same CV cannot be specified with both *-f* and *-s* simultaneously.

-q qdot_setting Specifies the inertial behavior of LMax. Valid options are: *present*, *absent*, and *ignore*. Default is *present*. *present* specifies that for each CV in the input file, there is a corresponding rate-of-change (or "qdot") value, such that the input file is formatted as:

```
<basin> <- <CV1> <CV2> ... <CVn> <qdot1> <qdot2> ... <qdotn>
```

This format is the one created by aimless shooting runs in ATESA when called with the *include_qdot = True* option (which is the default, as is this setting). *absent* specifies that the input file does NOT contain rate-of-change values for its CVs; that is, every number in every column of the input file is a separate CV. *ignore* specifies that the input file DOES contain rate-of-change values, but that they should be ignored.

-p prefilter An option to "filter out" the apparently worst terms from the model first in order to reduce the search space during optimization and get results faster. A fraction between 0 and 1 should be given, and then a one-term RC is optimized for every CV in the input file and only that fraction of the CVs that perform best in this test are considered going forward. This step is skipped (as it would have no effect) when *-p* is 1 (which is the default). This option can save huge amounts of time in some conditions, especially when the input file contains large numbers of irrelevant CVs, but it is not strictly "safe" in that it may result in a worse RC than would have been produced otherwise in the case that it filters out CVs that might have been useful when considered in the context of other CVs.

-o output_file If this option is given, a new file named *output_file* is written containing the results of the optimization. If this option is not given, the results are instead written directly to the terminal. This option will overwrite existing files.

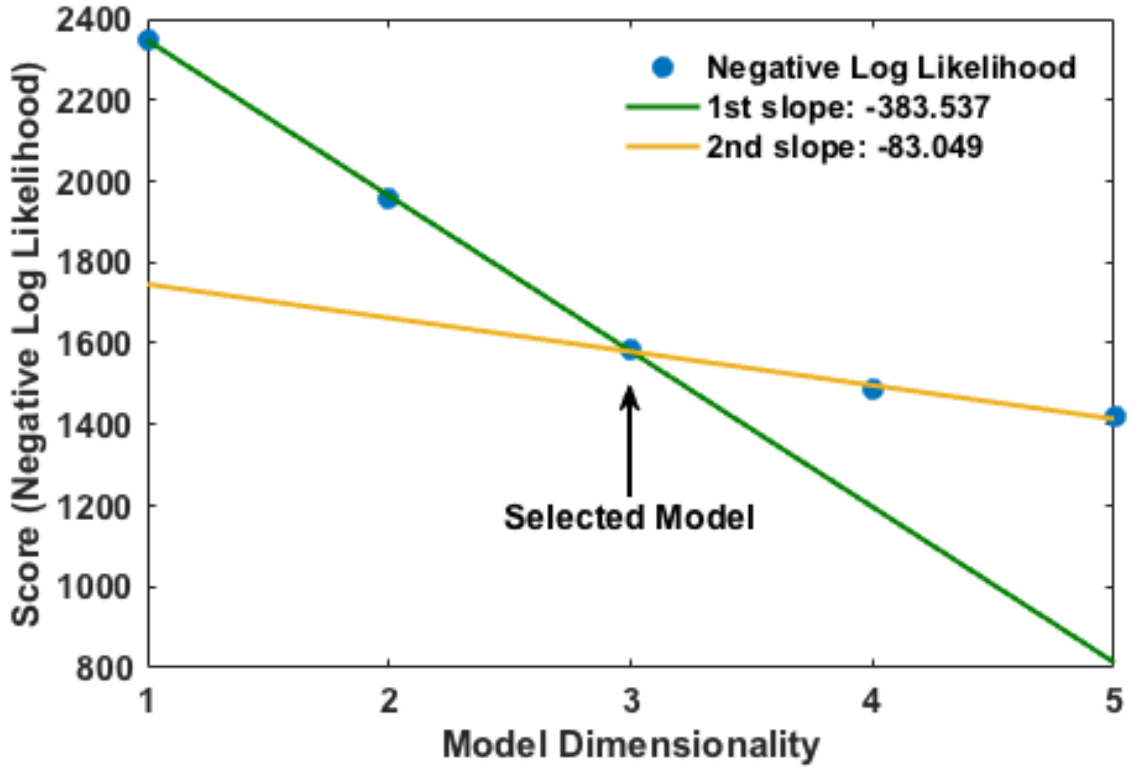
–plots Produces a matplotlib figure for the committor sigmoid (likelihood of committing to the forward basin as a function of the reaction coordinate value) for the final model as a histogram, compared against the theoretical ideal as a line. If *–two_line_test* is provided, also produces ASCII-style plots in the terminal for each step of two_line_test above four dimensions, if Python package gnuplot is available; see *The two_line_test Option* for details.

–quiet By default, `lmax.py` writes one or more progress bars (one for each optimization step) to the terminal. If this option is given, these progress bars are suppressed. Note that the output will still be written to the terminal if no output file is specified with the *-o* option.

5.1.1 The two_line_test Option

A common model selection problem when attempting to find a suitable reaction coordinate for a given dataset is that the appropriate number of CVs to include in the final RC (that is, its mathematical dimensionality) cannot be identified in advance. There are several approaches to comparing the "information content" of various models aimed towards penalizing each successive parameter such that only significant improvements are permitted, such as the Bayesian and Akaike information criteria. However, these methods are designed to find the "best" model for a given process, regardless of how many parameters that model contains, whereas for practical reasons useful RCs are usually low-dimensional. That is, while an RC selected using even a highly selective Bayesian information criterion may contain many (*e.g.*, upwards of ten) parameters, a much simpler model (say, of three or four dimensions) is usually sufficient and more useful. This is the motivation behind the "two_line_test" algorithm. In short, whether two_line_test is appropriate for your use-case depends on whether you would rather sacrifice a modicum of model accuracy in exchange for a fast and fully automated approach to obtaining a reaction coordinate.

two_line_test attempts to include only the most important parameters in the final RC, as defined by the change in model score for each successive parameter. To accomplish this, the algorithm first uses the *-r* approach to model optimization as described above to obtain one- through five-dimensional RCs; then, it fits two lines onto contiguous subsections of the data [1, M] and [M, N] (where N is the dimensionality of the highest-dimensional model yet derived and 2 M N - 1). The resulting RC is the one containing M CVs, if and only if the two lines intersect closer to the M'th point than any other point *and* the ratio of slopes s[M, N]/s[1, M] is at least 0.5 (that is, the slope of the second line is at most 50% that of the first line; this threshold can be overridden using the *two_line_threshold* command line argument or by running lmax.py in a directory containing a settings.pkl object (created by ATESA based on the configuration file) that specifies the option "two_line_threshold"). An example meeting these two criteria is shown here:



This plot (in ASCII form) would be outputted to the terminal at the end of the optimization if the *–plots* option were supplied. If the criteria cannot be met, an additional model of dimensionality N+1 is obtained and the process is repeated. If enough dimensions are available, this algorithm will always converge eventually. This approach is very efficient for arriving at a *good* reaction coordinate (though it is by no means guaranteed to be the "best" possible one), though it suffers from two shortcomings:

1. One-dimensional models can never be selected; and
2. The cutoff ratio of slopes is arbitrary (that is, it reflects an arbitrary judgement of what constitutes a sufficient drop in the rate of change of model scores)

5.2 rc_eval.py: Reaction Coordinate Evaluation

ATESA also comes with a separate script for evaluating reaction coordinates for each shooting point coordinate file in a given directory. This script should be given an aimless shooting working directory, where it will produce a new file *rc.out* containing the reaction coordinate values of each point, sorted by ascending absolute value (such that points closest to the supposed transition state come first).

Alternatively, when *extrema* = *True*, the script skips creating *rc.out* and simply returns the RC values of a the final forward and backward frames of a single accepted trajectory in the working directory. This is useful when preparing for equilibrium path sampling or umbrella sampling jobs, which require the user to specify the range of RC values to sample over.

The syntax is as follows:

```
rc_eval.py working_directory rc_definition as_out_file [extrema]
```

working_directory Specifies the aimless shooting working directory in which to operate

rc_definition Defines the reaction coordinate to evaluate for each shooting point. The format is the same as in the *rc_definition* configuration file setting (see *Reaction Coordinate Definition*), except that here there must be no whitespace (' ') characters. The identities of CVs are determined from the settings.pkl object stored in the working directory.

as_out_file The path to the aimless shooting output file used to build the reaction coordinate (the *-i* argument for `lmax.py`. Usually this should be the largest "decorrelated" output file in the aimless shooting working directory.

extrema A boolean, either "True" or "False". If "True", the script skips creating *rc.out* and simply returns the RC values of a the final forward and backward frames of a single accepted trajectory in the working directory. This is useful when preparing for equilibrium path sampling or umbrella sampling jobs, which require the user to specify the range of RC values to sample over. This is the only option with a default value; if it is omitted, it will be set to False.

The produced output file *rc.out* is (optionally) used as input for a committor analysis run (see *What is Committor Analysis?*). Note that running this script with *extrema = False* can take a long time if there is a large number of shooting moves in the indicated working directory.

5.3 mbar.py: Energy Profiles from US

The output files from an umbrella sampling (US) run can be converted into a free energy profile by any number of methods, but one of the most ideal is the Multistate Bennett Acceptance Ratio", or "MBAR". ATESA comes with a suitable implementation of MBAR using the pymbar package available from the Chodera lab. If you publish work making use of this script, be sure to cite the appropriate papers described on that page; at minimum, you should cite:

> Shirts M. R. and Chodera, J. D. Statistically optimal analysis of samples from multiple equilibrium states. J. Chem. Phys. 129:124105 (2008). DOI: 10.1063/1.2978177

The basic task in interpreting umbrella sampling data is to "subtract" the effect of the known harmonic restraints on the sampling, leaving only the underlying free energy profile. For a discussion of the exact workings of MBAR, the reader is directed to the original paper.

If supported by the local python environment, mbar.py produces several plots: first, a "mean value" plot that shows the derivation from the window center in each data file. This is a diagnostic tool to help identify any problematic regions; if there is no issue, the plot should be a smooth waveform passing through 0 near the middle. Then, it produces a histogram to show the coverage of sampling over the range of the reaction coordinate. There should be no gaps in this plot, or else additional data must be collected to cover the gaps. Finally, it plots the free energy profile itself. All of the data for these plots is also printed the the output file (see the *-o* option below) regardless of whether the plots are shown. In cases where the data exists on a remote server, it may be convenient to copy the necessary files (see following paragraph) to a local directory before running `mbar.py` in order to produce these plots automatically.

mbar.py looks for and uses all data files in the present directory whose names begin with "rcwin_" and end with "_us.dat". This matches the output files produced by umbrella sampling with ATESA. The script is called directly in the command line from within the desired working directory as follows:

```
mbar.py [-k kconst] [-t temp] [-o output] [--min_data min] [--ignore threshold] [--
↪decorr] [--rc_min min] [--rc_max max] [--quiet]
```

-k kconst

> The harmonic restraint weight used during umbrella sampling in kcal/mol, according to the equation:
>
> $$U = k(RC - RC_0)^2$$
>
> Where RC_0 is the position of the restraint along the reaction coordinate. This particular implementation of MBAR requires that all of the restraints have the same weight. The default is equal to the default setting

during an ATESA umbrella sampling job, so if you didn't change it there, don't change it here. Default = 50

-t temp

The temperature at which to perform the analysis, in K. This implementation of MBAR requires that the temperatures across all the samples be identical. Default = 300

-o output

The name of the output file produced by the script. It will be overwritten if it exists. Default = mbar.out

–min_data min

The minimum number of samples that must be present in a given data file for it to be included in the analysis. This can be useful to exclude results from simulations that did not finish for some reason, but should be used with care. Default = 0

–ignore threshold

The number of samples from the beginning of each data file to ignore during analysis. This is useful for manually specifying a decorrelation time from the initial coordinates in each window, if desired. Probably should not be used in combination with *–decorr* unless you know what you're doing. Default = 1

–decorr

If this option is given, then the built-in pymbar.timeseries.detectEquilibration and pymbar.timeseries.subsampleCorrelatedData functions are used to attempt to automatically pare down the data in each data file to only equilibrated and decorrelated samples. The decorrelation is performed before the program produces any plots or results. If you don't know what this means, you probably *should* use it. If you publish work that makes use of this option, you must cite (in addition to the aforementioned MBAR paper):

```
Chodera, J. D. A simple method for automated equilibration detection in␣
↪molecular simulations. J. Chem. Theor. Comput. 12:1799, 2016. DOI: 10.1021/
↪acs.jctc.5b00784
```

–rc_min min

The smallest value of the reaction coordinate to include in the final energy profile. If this option isn't specified, then the smallest window center is used instead (which is usually safe).

–rc_max max

The largest value of the reaction coordinate to include in the final energy profile. If this option isn't specified, then the largest window center is used instead (which is usually safe).

–quiet

If this option is given, all the output to the terminal and the display of plots is suppressed, and the only result is the output file.

5.4 boltzmann_weight.py: Energy Profiles from EPS

The output file from an equilibrium path sampling (EPS) run can be converted into a free energy profile by simply weighting the observed probability of each state (that is, a certain discretized range of RC values) according to the Boltzmann distribution:

$$G = -k_B T ln(p)$$

Where G is the relative free energy, k_BT is the Boltzmann constant times the absolute temperature T, and p is the relative probability of the state in question.

boltzmann_weight.py is a utility script that automates this calculation for data in the format of an ATESA equilibrium path sampling output file, and stitches together the free energy profiles of adjacent windows to construct the overall free energy profile. It also automates subsampling of the data for bootstrapping in order to obtain error bars. It is called as follows:

```
boltzmann_weight.py -i input_file [-o output_file] [-t temp] [-n nbins] [-c␣
↪bootstrapCyc] [-b bootstrapN] [--noplot]
```

-i input_file Path to the EPS output file containing the data to analyze. This file should be formatted in three columns separated by whitespace:

[EPS window lower boundary] [EPS window upper boundary] [sampled RC value]

Samples from each window do not need to be in contiguous groups of lines, but the first two columns of samples from the same EPS window do need to be identical when rounded to three decimal places in order to be counted as belonging to the same window.

-o output_file Name of the output file to produce, containing the final free energy profile and bootstrapped error if applicable. Default is 'fep.out'.

-t temp The temperature in Kelvin at which to evaluate the free energy profile (that is, T in k_BT). Default is 300.

-n nbins The number of bins into which each EPS window is divided. Must be an integer. Larger values allow for higher resolution, but also require more data in order to remain smooth. Too-low values of n may provide misleading results, while too-high values will add considerable noise. The user is advised to try a few different values of n before settling on one, in order to get a feel for how it affects the result. Default is 5.

-c bootstrapCyc The number of bootstrapping iterations to perform. Must be an integer. A value of zero turns off bootstrapping. Each iteration subsamples the data in each window to get a new estimate of the free energy profile, and then the standard deviation of the distribution of energy values from across the iterations is provided in the final result. Default is 100.

-b bootstrapN The number of samples to include in each window when bootstrapping. Must be an integer. Default is 25.

–noplot By default, boltzmann_weight.py produces a histogram of the binned data in each window to help assess good overlap between adjacent windows, as well as a plot of the resulting free energy profile using matplotlib, if supported by the interpreter. Providing this option suppresses this behavior.

Note that if *–noplot* is not provided and a histogram is shown, the plot window must be manually closed before the remainder of the calculation will take place. Similarly, the program will not terminate until the free energy profile plot window is closed.

CHAPTER 6

On Termination Criteria

Along with ATESA, I have introduced a novel termination criterion for aimless shooting that I believe should be used in all cases. In summary, the idea is to estimate the error in the reaction coordinate produced with likelihood maximization using the diagonal terms of the Godambe Information matrix. Use of this termination criterion prevents wasted simulations and adds confidence to the final result. For more details and context, see the rest of this page.

6.1 The Sampling Problem

Most any molecular system that is interesting enough to bother simulating operates within an extremely multidimensional state space. In the most general case, a complete exploration of this state space requires not only that the full range of each dimension be explored individually, but also the full range of combinations of dimensions; that is, the complexity of state spaces scales exponentially with the number of dimensions. Thus, perfect sampling of even small sections of state space is most often totally impractical.

If we know that our simulation will never fully explore state space, how do we know whether we've sampled enough to trust the results? This issue has been aptly dubbed "the sampling problem".

The only "solution" to the sampling problem is to circumvent it by relying on free energy to "weight" regions of state space by their relative importance in determining relevant properties of the system. That is, regions of state space with relatively lower free energy play a proportionally larger role in the system's behavior, so distributing sampling according to free energy is the most efficient means of arriving at a good approximation of the total behavior of the system. Of course, this is not really a solution at all – that is, it does not address the central issue of identifying when our sampling is sufficient, or even whether we're sampling all of the important regions of state space in the first place. This can only be done *post hoc*, by verifying the putative results of simulations in some other way (*e.g.*, experimentally), and then retrospectively affirming that the sampling had been sufficient.

6.2 Relevance to Aimless Shooting

The process of aimless shooting can be thought of as constrained sampling – specifically, in the region of state space near enough to the reaction separatrix that both the product and reactant states are accessible. The aimless shooting algorithm distributes its sampling according to the acceptance probability (that is, the likelihood of forming a complete

reactive trajectory from a given initial set of coordinates shot "forward" and "backward") and the free energy (simulations are more likely to proceed towards regions of lower energy, which in aggregate causes successive shooting moves to become centered at local energy minima along the separatrix). This sort of sampling does favor the most important transition states; however, this is no remedy to the sampling problem, as there remains no means of determining when sufficient sampling has been performed. But we can't go on sampling forever, so how do we choose when to stop?

6.3 Choosing When to Stop with Committor Analysis

To date, the most common method of determining when to stop aimless shooting has been committor analysis (*What is Committor Analysis?*). This requires that the aimless shooting data be mined for a reaction coordinate that describes the reaction progress (usually via likelihood maximization (*What is Likelihood Maximization?*)), and then that that reaction coordinate be verified by showing that unbiased simulations beginning from its center are (approximately) equally as likely to proceed to the reactants as to the products. This method is invaluable and should certainly be used to affirm the correctness of the final reaction coordinate, but as a termination criterion it suffers from two major issues:

1. It requires an extensive amount of additional simulation, which is expensive and time consuming; and
2. It provides no feedback about whether further sampling should be expected to improve the result.

In practice, this means that the researcher performing aimless shooting must either perform rounds of committor analysis many times throughout the study, or else collect such an excess of data that any unsatisfactory result in committor analysis is assumed to be due to issues elsewhere. Either option wastes time, both computational and real.

6.4 Information Error

ATESA introduces a new method of addressing the two shortcomings identified above, based on an assessment of the error in the likelihood maximization procedure. Specifically, ATESA measures the mean value of the parametric standard errors from the Godambe information matrix for a given model as a function of the amount of data collected, and by default uses a threshold on this parameter as its termination criterion during aimless shooting.

This "information error" is a property of likelihood maximization derived from the first and second derivatives of the log likelihood function evaluated at the model optimization solution (the "maximum likelihood estimator"). It is a measure of the "information" about the optimization parameters that is stored in the dataset, as described by the sensitivity of the optimization result to changes in the values of the parameters. As the information error decreases, the confidence that the optimized model is the "best" possible one for the given sets of data/observations and included dimensions increases.

In the context of aimless shooting, the information error in the model can be interpreted as a metric of convergence of sampling *within the explored regions of state space*. That is, it does not resolve *The Sampling Problem* in that it remains impossible to determine (except through retrospection) whether the regions of state space being sampled are sufficient to describe the system. It does, however, address issue (2) above by assessing convergence of the sampling *that does occur*. In other words, to the extent that information error is converged, one can be assured that further sampling from the same distribution will not improve the result appreciably (to within the chosen tolerance). Furthermore, because information error is derived from likelihood maximization and therefore does not involve any simulation, it is comparatively much faster and more resource-efficient than committor analysis and can be assessed quite frequently without wasting significant resources, addressing issue (1).

To reiterate, information error should not *supplant* committor analysis but should *augment* it. Aimless shooting should be performed until information error has converged to within some reasonable tolerance, and only then should the more expensive step of committor analysis be attempted. Besides increasing efficiency, the addition of information error to the aimless shooting workflow adds information to the committor analysis result: if a positive (desirable) result is obtained with low information error, it is probably also a good approximation of the best possible committor analysis result for the given region(s) of sampling and candidate reaction coordinate parameters. Conversely, if a

negative (undesirable) result is obtained, it is likely due to omission of one or more key parameters from the reaction coordinate, or else a fundamental flaw in the sampling, but *not* due to insufficient sampling of the sampled region(s).

CHAPTER 7

Troubleshooting

7.1 Technical Issues

Although every effort has been made to provide helpful error messages to correspond to every foreseeable issue, unfortunately not every issue is, in fact, foreseeable. This page is designed to help users troubleshoot should they run into issues not accompanied by a sufficiently helpful error message. Just because your transition path sampling shooting is aimless, doesn't mean your troubleshooting has to be!

Issues running ATESA will fall into one of three categories, based on where in the pipeline the issue arises:

1. The local shell or the batch system
2. Amber
3. ATESA itself

A description of how to identify the category of error being encountered, and suggestions to resolve the issue, are provided. Further sections are provided for selected job types to facilitate troubleshooting in cases where the software appears to be functioning correctly, but the result is undesirable. If this page does not help you resolve your issue, please raise it on our GitHub page.

7.1.1 Issues with the local shell or the batch system

Errors encountered by these systems will appear in one of two places. If the shell encounters an error, it will be reported directly in the command line after ATESA is called. This likely indicates an issue with the installation of the software, or of Python itself. Command line errors will also arise if the batch system does not recognize the formatting of the batch files submitted by ATESA; double-check that your batch templates are of the correct format. Also, note that if ATESA is called from inside a batch job (as is recommended when performing many simulations), the command line output will be piped to the batch output file for that job.

Errors encountered by the batch system may also appear in batch output files for individual simulations, and in this case they will appear in the individual batch job output files in the working directory. These errors are likely to do with the formatting of the batch template files used by ATESA to create its job batch files. For example, if the user has removed the line in the batch template indicating the nodes to be assigned to the job on a Slurm system, ATESA

will not complain (as omitting variables in the templates is supported in general), but the batch system will (because this particular variable is mandatory in Slurm). Other batch system errors include those not specific to ATESA, such as insufficient funds on the user account; to check whether a given error is of this type, try submitting a job that does not involve ATESA.

7.1.2 Issues with Amber

Amber has its own output file for each job, usually given by the suffix ".out" or ".mdout" in the working directory (note: ATESA may produce output files named with the ".out" suffix in this directory, as well.) If an error arises here, it likely indicates an issue with the formatting of the input coordinate or topology files, or else with the input file passed to Amber. Another common source of error when using certain QM/MM models is the absence of the necessary quantum mechanics files for the chosen level of theory. If these are missing they may need to be installed for you by a user with root access. ATESA has been tested in at least a limited capacity with Amber 12, 14, 16, and 18, but other versions are not guaranteed to be compatible.

7.1.3 Issues with ATESA

Sometimes, especially during aimless shooting, ATESA may hang, neither crashing nor apparently doing anything for a long time. In this case the easiest option is usually to restart the job with "restart = True" as appropriate.

Errors encountered by ATESA itself will be outputted into the command line in the event that the program has been called directly, or else into the batch output file in the event that it has been called in a batch job. Hopefully, most errors of this type will be self-explanatory. There may, however, be unforeseen errors that are not handled in a useful way. Errors of this sort could have to do with improperly installed dependencies, especially pytraj. One possible troubleshooting option is to retry the job with as many default settings as possible to see if the error persists — if it does not, add on custom settings one-by-one (starting with those you are least confident in) until the culprit is found.

If ATESA outputs an error message that confuses you, especially if it happens very early on in the job, double-check that the simulations that have been run so far (if there are any) appear to have behaved properly. For example, the error:

```
ValueError: must provie a filename or list of filenames or file pattern
```

is often the result of a failed simulation, usually because of a systemic issue in the way the model is set up. It may also arise when a necessary file has been errantly deleted. Regardless of the particular error, if everything seemed fine and then ATESA suddenly crashed, often simply resubmitting the job (with restart = True, as appropriate) will fix it.

Another common issue is:

```
EOFError: Ran out of input
```

while trying to open the restart pickle file ("restart.pkl"). This occurs because the restart file has been errantly emptied due to an interrupted write attempt. ATESA automatically generates a backup file called "restart.pkl.bak" in the working directory, so if you get this error, simply copy the backup file over the empty original.

Repeated "errors" with messages like these while running simulations are normal and not an issue so long as ATESA continues to run:

```
Internal Error: Global index 0 is out of range.
Error: NetCDF file has no frames.
Error: Could not set up '<trajectory file>' for reading.
```

Although the user is of course permitted under the license to attempt to debug the code themselves (and to use the modified code in whatever manner they see fit), no one finds it easy to read someone else's code. If you encounter an

issue with ATESA that you cannot resolve, I encourage you to raise an issue on our GitHub page. And if you encounter an error that you did resolve by modifying the code, please submit a pull request so that everyone can benefit!

7.2 Job Type-Specific Issues

This section concerns issues that may arise in individual job types when the code is apparently functioning properly, but the results are in some way not satisfactory. Not every possible issue can be mentioned here, so if you come across something that's stumped you and isn't addressed on this page, please raise an issue on our GitHub page with the "question" and/or "help wanted" tags.

7.2.1 Aimless Shooting

Barring technical issues, the only problem you might experience during aimless shooting is very low acceptance ratios. The acceptance ratio for each thread represents the ratio of shooting moves that have been "accepted" (recall that this means that the combined foreward and backward trajectories connect one stable state to the other) to the total number of shooting moves in that thread so far. The acceptance ratio is sensitive to many factors, including the initial coordinates, the *min_dt* and *max_dt* configuration file settings, and the underlying energetic landscape, but in general a "good" acceptance ratio is somewhere in the range of 10-to-40%. The efficiency of aimless shooting in sampling the state space around the separatrix is directly tied to the acceptance ratio, so in general it is important to be getting an acceptable value. While higher acceptance ratio values are not a problem, they may indicate that your sampling is too conservative (differences between adjacent shooting points are too small), which also negatively impacts sampling efficiency.

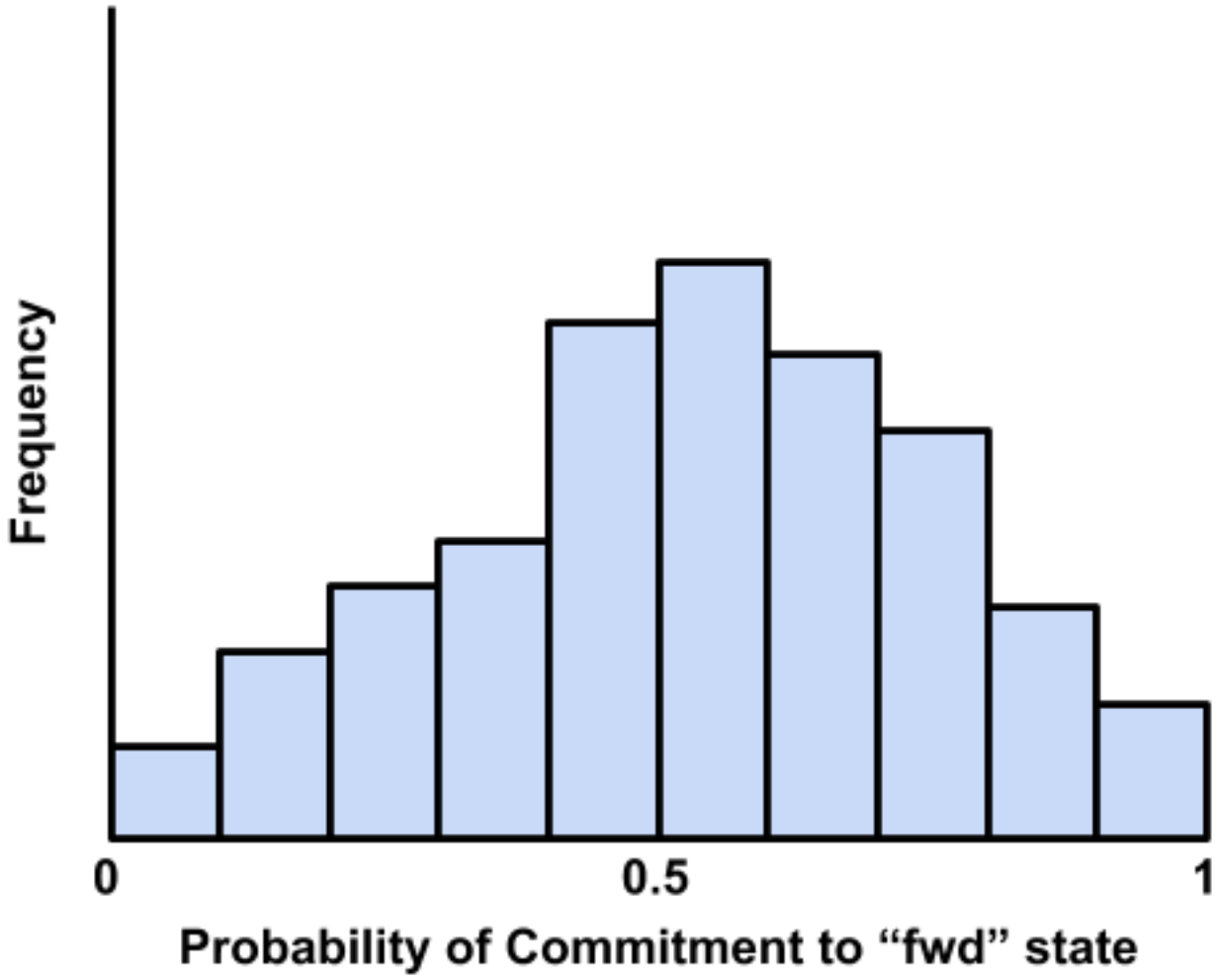
A chronically low acceptance ratio during aimless shooting, even across many degenerate threads or slightly different input coordinate files, probably indicates that your initial coordinates are not as close to the reaction separatrix as you might have hoped. Even if they are very close, an extremely steep energetic landscape makes aimless shooting difficult, and depending on the context of your study may indicate an incorrect putative reaction mechanism. You should also consider making the *max_dt* and/or *min_dt* settings smaller; although larger values can explore phase space more efficiently, it also increases the risk of a thread straying too far from the separatrix and being unable to climb back up, resulting in poorer acceptance ratios after that step. This effect can be mitigated by setting *always_new* to "True" (which is the default).

If you are getting *zero* acceptance despite the simulations themselves looks reasonable, you should interpret it to mean that your initial coordinates are too far from the separatrix to be acceptable. If you obtained your initial coordinates through some means other than ATESA's *jobtype* = *find_ts* option, you should give that a try, as it will only ever provide coordinates with non-zero acceptance ratios (and provide custom advice if it is unable to do so). Otherwise, you'll have to look to whatever means you're using to obtain your initial coordinates.

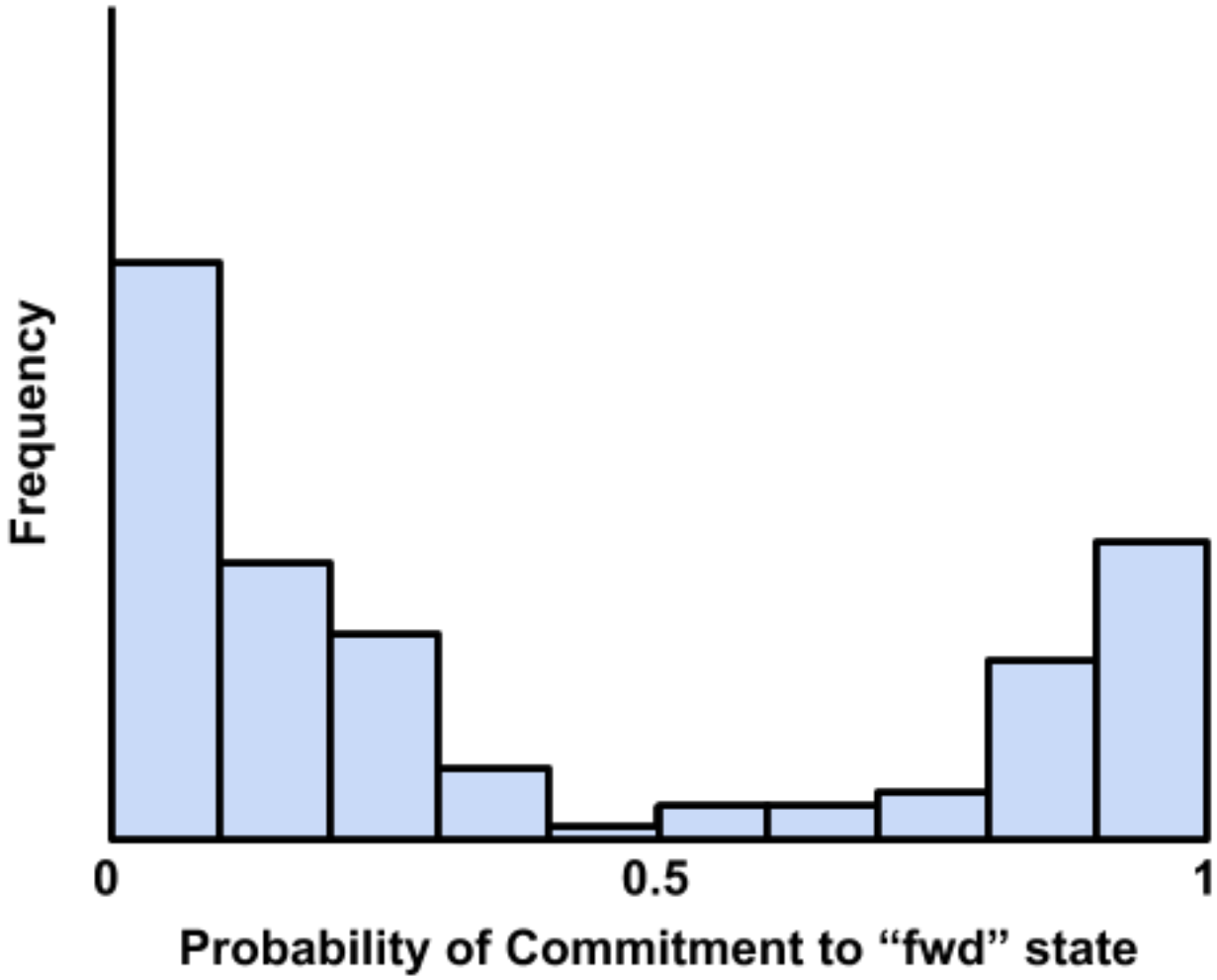
Finally, if simulations seem to be going fine but are simply taking a very long time, the issue is probably with the setup of individual jobs. As always when running a new model on a high performance cluster, you should first run a series of short jobs to assess how your simulation speed scales with the resources allocated. Keep in mind that certain settings are much more computationally expensive (and thus slow), such as large quantum mechanics regions. Also ensure that you have allocated sufficient memory for each job and for ATESA itself; at least a few gigabytes is safe.

7.2.2 Committor Analysis

The "ideal" committor analysis result is a perfectly narrow peak of exactly 50% probability of going to each stable state. In practice however, the best result we can hope for is a roughly gaussian distribution peaked somewhere close to 50%, and a roughly flat distribution is also generally acceptable. The rest of this section will be organized in terms of other possible distributions with advice about how to interpret them, and then ending with some examples of real published committor analysis results.


Frequency
0
0.5
1
Probability of Commitment to “fwd” state

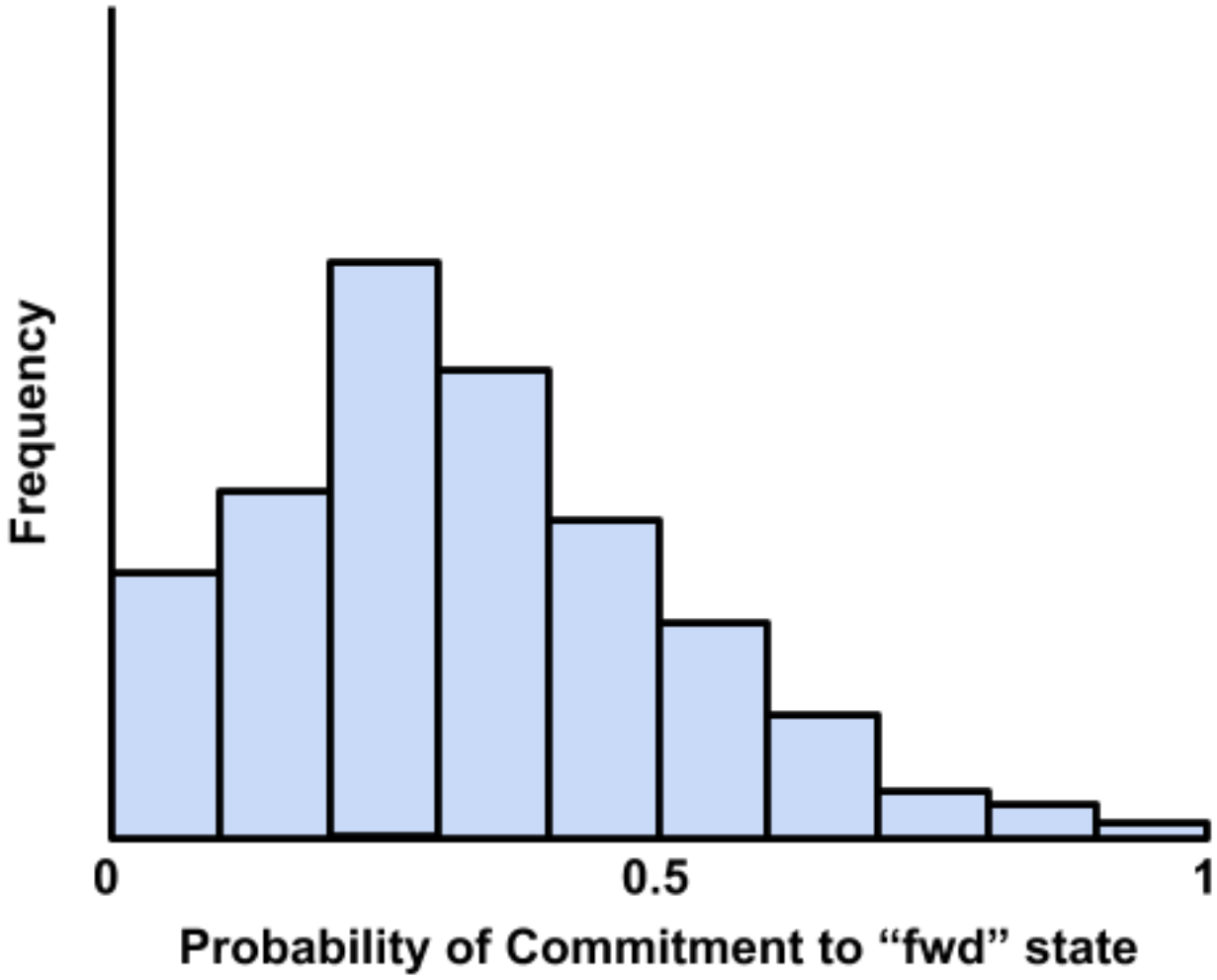
This is an example of an excellent committor analysis result. The model used to arrive at this result appears to be very strong. That the peak is not *quite* at 0.5 is of little consequence, and in fact to be expected when attempting to describe very high-dimensional systems with a relatively low-dimensional model.

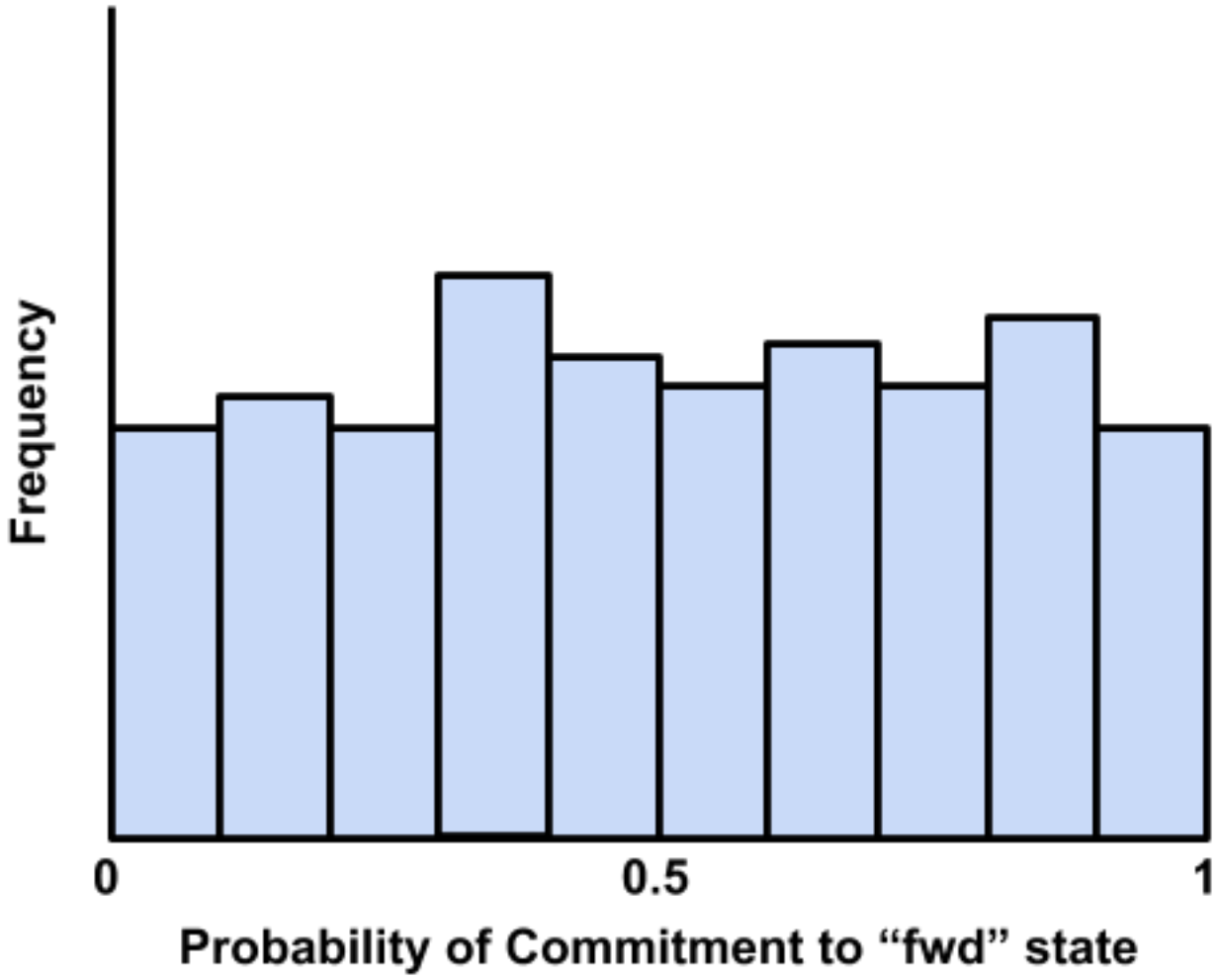


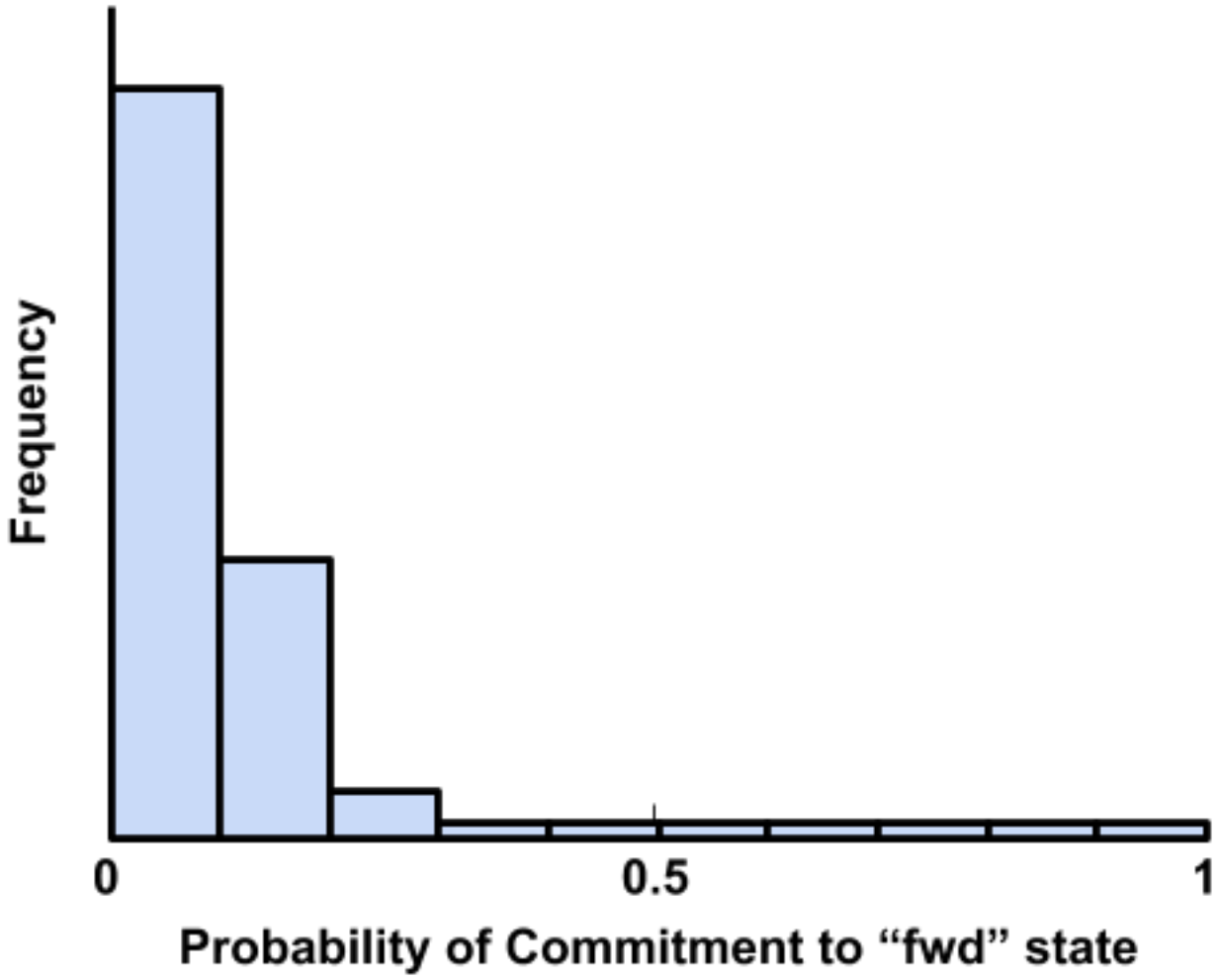
A common poor committor analysis result, the distribution is bimodal at or near the edges. This happens when the model was built along a dimensional projection that causes shooting points on opposite sides of the actual separatrix to look close together. Usually it means that one or more key dimensions has been omitted from the list of candidate CVs, so add as many as you can imagine might be important and run aimless shooting with *resample* = *True* to resample the shooting points with your new CVs before attempting likelihood maximization and committor analysis again.

The distribution is roughly gaussian, but centered far from 50%. This is another common result that arises when there's simply not enough data from aimless shooting to arrive at a strong model through likelihood maximization. If the peak isn't right along an edge (0 or 1) then this result is still fairly strong, but if you want to improve it, simply collecting more data or using a higher-dimensional reaction coordinate may help.

A roughly flat distribution, this result can arise either from insufficient sampling during aimless shooting or committor analysis, missing candidate dimensions, or the use of a lower-dimensional model than is truly appropriate for the system. Similarly to the previous example, this is still a reasonably strong result and may indicate a strong enough model, depending on your purposes.

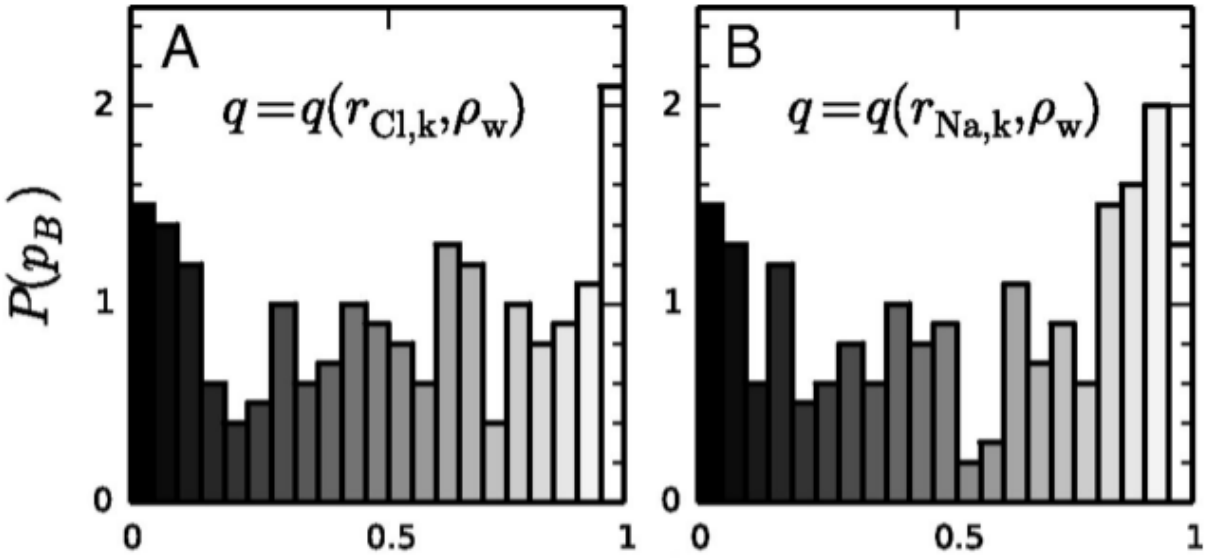

Frequency
0
0.5
1
Probability of Commitment to “fwd” state


Frequency
0
0.5
1
Probability of Commitment to “fwd” state


Frequency
0
0.5
1
Probability of Commitment to “fwd” state

All or nearly all of the simulations are grouped along one edge (either one). This should be a rare result, and is the only one here that represents a fundamental failure somewhere in the workflow. The underlying cause is either: (a) that the settings or other important features of the simulations or ATESA have changed significantly between aimless shooting and committor analysis (for example, a different quantum mechanics model, or a change in the definition of the commitment basins); or (b) that the aimless shooting data has been misinterpreted in some way, due to some unnoticed error. If after carefully verifying that the settings have not changed (remember to check the simulation input files, batch file templates, and ATESA configuration files) you still cannot find the source of this error, please raise an issue on our GitHub page with the "bug" label. Please also be sure to include a thorough description of your problem and attach the files "settings.pkl" and "restart.pkl" from the aimless shooting working directory.

Here are some examples of real committor analysis results in published manuscripts. Though none are perfect, all of these results were deemed acceptable within the context of their work by their authors and passed peer review.



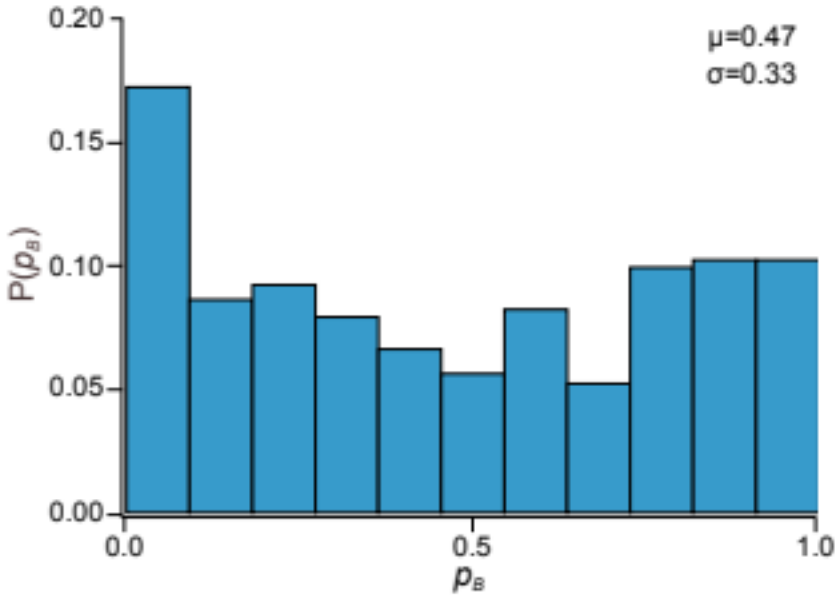
Joswiak, M. N., Doherty, M. F., & Peters, B. (2018). Ion dissolution mechanism and kinetics at kink sites on NaCl surfaces. Proceedings of the National Academy of Sciences, 115(4), 656–661. https://doi.org/10.1073/pnas.1713452115

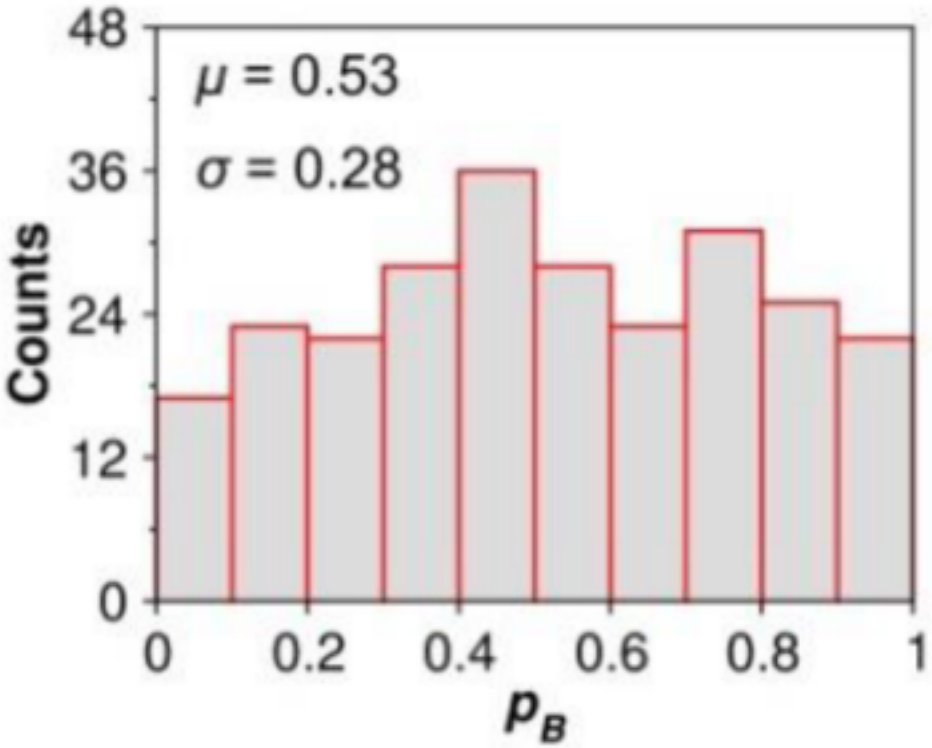
Mayes, H. B., Knott, B. C., Crowley, M. F., Broadbelt, L. J., Ståhlberg, J., & Beckham, G. T. (2016). Who's on base? Revealing the catalytic mechanism of inverting Family 6 glycoside hydrolase. Chemical Science, 5955–5968. https://doi.org/10.1039/C6SC00571C

Silveira, R. L., Knott, B. C., Pereira, C. S., Crowley, M. F., Skaf, M. S., & Beckham, G. T. (2021). Transition Path Sampling Study of the Feruloyl Esterase Mechanism. Journal of Physical Chemistry B, 125(8), 2018–2030. https://doi.org/10.1021/acs.jpcb.0c09725

7.2.3 Umbrella Sampling

Umbrella sampling is a powerful tool for efficiently evaluating the free energy profile along a chosen reaction coordinate. However, as with all restrained simulations methods the simulations may not behave as expected, leading to errant results. In this section we will describe a few types of errors commonly encountered during umbrella sampling and suggest solutions. Note that this section assumes that the simulations and code are running without error, and that the issue is instead with the data itself.


μ=0.47
σ=0.33
0.20
0.15
0.10
0.05
0.00
P(p_B)
0.0
0.5
1.0
p_B


μ = 0.53
σ = 0.28
48
36
24
12
0
Counts
0
0.2
0.4
0.6
0.8
1
p_B

The standard workflow when analyzing umbrella sampling data with ATESA is to run `mbar.py` with the `-i` flag pointing to the umbrella sampling working directory. Before analyzing the data, this script returns two "diagnostic" plots to help the user ensure that the data is sound (these plots are returned numerically instead of graphically in the output file (default name *mbar.out*) if the shell does not support producing graphs directly, in which case you can plot them yourself). The first is a histogram and the second is a "mean value" plot.

- The Histogram

  The histogram is actually composed of many individual histogram plots, one for each unique window center in the data. The purpose of the histogram is to visually ensure that there are no gaps in the data (that is, that there are no large regions between histograms where no sampling has occurred) and that the sampling is roughly even (that is, that all of the peaks are roughly at the same height, though there will be some natural variation).

  If there are gaps, the solution may simply be to run additional simulations with the same restraint weight centered in the under-sampled region(s). Keep in mind that there is no need for the sampling windows to be evenly spaced. Alternatively, if the gaps are caused by restrained simulations in those regions falling away from them, you should either redo umbrella sampling with increased the restraint weights (which may necessitate more tightly packed windows) and/or with *What is Pathway-Restrained Umbrella Sampling?* enabled.

  If there are under-sampled regions, you should investigate the root cause by looking to the simulations in those regions themselves. One potential source of this issue in reaction models is poor quantum mechanical convergence. Resolving this issue is highly system-specific and lies outside the scope of this document, but note that in some cases it may be alleviated by adding a small electronic temperature to the simulations.

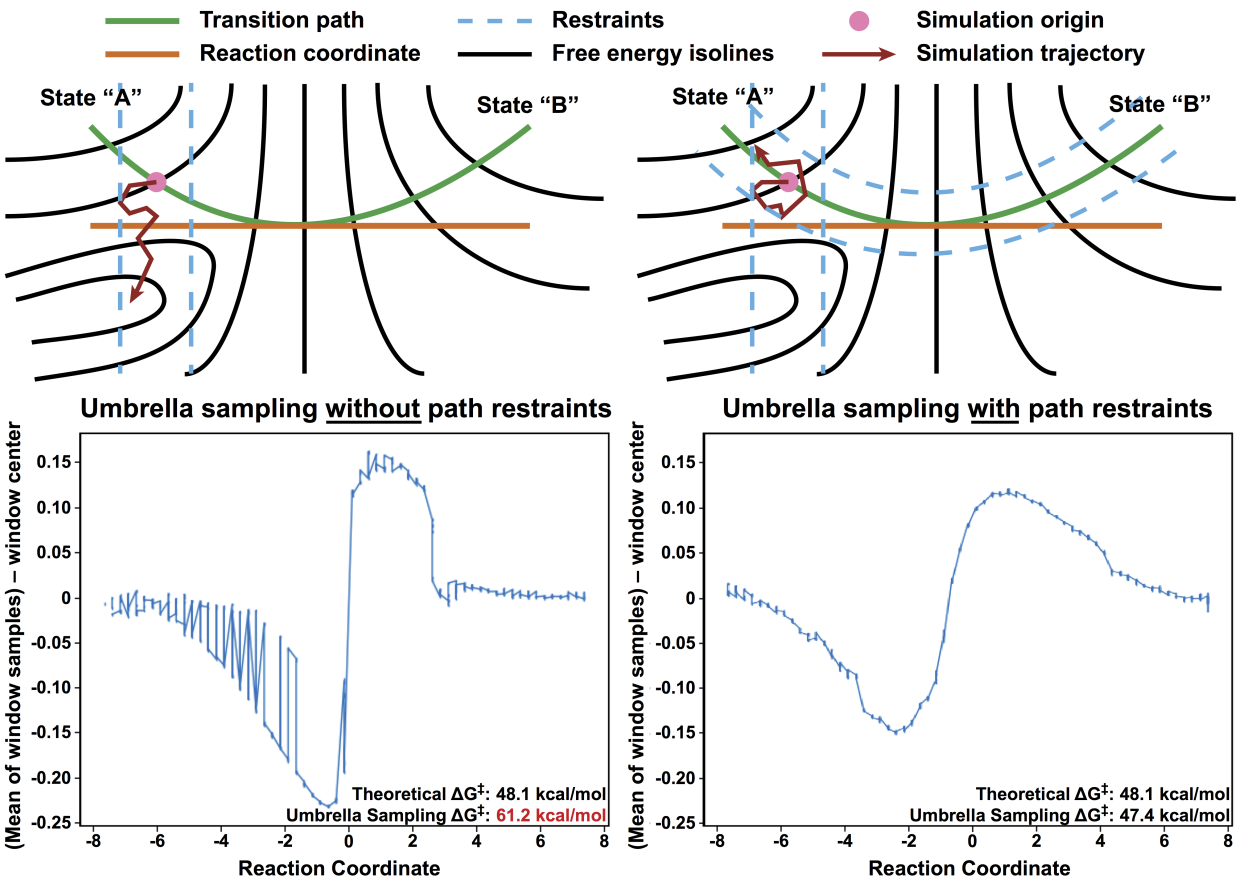
- The Mean Value Plot

  The second plot is a line plot depicting the difference between the mean value of the sampling data in each window and that window's restraint center on the vertical axis, versus the window restraint center on the horizontal axis. If there are multiple simulations located at the same window center (and there ought to be; by default there are five), these will be averaged together.

  The ideal mean value plot should be a smooth sinusoid passing through the value of zero on the vertical axis at three points: near the leftward extreme, near the middle, and near the rightward extreme. These correspond to the regions of the free energy profile with zero slope at one stable state, the transition state, and the other stable state, respectively. If either of the extrema do not pass through zero, further umbrella sampling windows should be added on the corresponding end until zero (and ideally, a little bit beyond) is reached.

  The other issue visible on this plot is unsmoothness. Unsmoothness includes both incongruently large error bars compared to other windows, or an abrupt discontinuity between adjacent points on the plot. This is usually caused by sampling of two or more significantly different regions of state space with similar reaction coordinate values. Depending on the underlying cause of this issue, it may be solvable using ATESA's pathway-restrained umbrella sampling feature (see *What is Pathway-Restrained Umbrella Sampling?* for details). It can also be improved in many cases by using an alternate reaction coordinate, especially a higher-dimensional one where any further dimensions are largely orthogonal to those already included.

  An example of the sort of error that can necessitate pathway-restrained umbrella sampling. (a) Two energetically distinct structures with identical reaction coordinate values for the example system (see *Example Study*). This is the sort of error that causes unsmoothness within a single window. (b) Examples of mean value plots. The mean value plot on the left is unsmooth, but application of pathway restraints results in the much-improved plot on the right.


Transition path
Reaction coordinate
Restraints
Free energy isolines
Simulation origin
Simulation trajectory
State "A"
State "B"
State "A"
State "B"
Umbrella sampling without path restraints
Umbrella sampling with path restraints
(Mean of window samples) – window center
Reaction Coordinate
Theoretical ΔG‡: 48.1 kcal/mol
Umbrella Sampling ΔG‡: 61.2 kcal/mol
Theoretical ΔG‡: 48.1 kcal/mol
Umbrella Sampling ΔG‡: 47.4 kcal/mol

CHAPTER 8

atesa package

8.1 Subpackages

8.1.1 atesa.tests package

Submodules

atesa.tests.test_information_error module

Unit and regression test for information_error.py.

class atesa.tests.test_information_error.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of class

test_main()
Tests information_error.main using sham shooting points in test_data

atesa.tests.test_jobtype module

Unit and regression test for jobtype.py.

class atesa.tests.test_jobtype.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of each method

test_algorithm_aimless_shooting_init()
Tests algorithm with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_algorithm_aimless_shooting_prod_accepted()
Tests algorithm with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod'] for an accepted move

test_algorithm_aimless_shooting_prod_always_new_not_accepted()
Tests algorithm with job_type = 'aimless_shooting', always_new = True and thread.current_type = ['prod', 'prod'] for a move that isn't accepted

test_algorithm_aimless_shooting_prod_not_always_new_not_accepted()
Tests algorithm with job_type = 'aimless_shooting', always_new = False and thread.current_type = ['prod', 'prod'] for a move that isn't accepted

test_algorithm_equilibrium_path_sampling_init()
Tests algorithm with job_type = 'equilibrium_path_sampling' and thread.current_type = ['init']

test_algorithm_equilibrium_path_sampling_prod_accepted()
Tests algorithm with job_type = 'equilibrium_path_sampling' and thread.current_type = ['prod', 'prod'] for an accepted move

test_algorithm_equilibrium_path_sampling_prod_not_accepted()
Tests algorithm with job_type = 'equilibrium_path_sampling' and thread.current_type = ['prod', 'prod'] for a move that isn't accepted

test_algorithm_equilibrium_path_sampling_prod_not_accepted_no_accepted_yet()
Tests algorithm with job_type = 'equilibrium_path_sampling' and thread.current_type = ['prod', 'prod'] for a move that isn't accepted and with no accepted moves in the thread's history

test_check_for_successful_step_aimless_shooting_init()
Tests check_for_successful_step with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_check_for_successful_step_aimless_shooting_prod()
Tests check_for_successful_step with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod']

test_check_for_successful_step_committor_analysis()
Tests check_for_successful_step with job_type = 'committor_analysis'

test_check_for_successful_step_equilibrium_path_sampling()
Tests check_for_successful_step with job_type = 'equilibrium_path_sampling'

test_check_termination_aimless_shooting_init()
Tests check_termination with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_check_termination_aimless_shooting_prod()
Tests check_termination with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod']

test_check_termination_committor_analysis()
Tests check_termination with job_type = 'committor_analysis'

test_check_termination_equilibrium_path_sampling_global()
Tests check_termination global criterion with job_type = 'equilibrium_path_sampling'

test_check_termination_equilibrium_path_sampling_thread()
Tests check_termination thread criterion with job_type = 'equilibrium_path_sampling'

test_gatekeep_aimless_shooting()
Tests gatekeep with job_type = 'aimless_shooting'

test_gatekeep_committor_analysis()
Tests gatekeep with job_type = 'committor_analysis'

test_gatekeep_equilibrium_path_sampling()
Tests gatekeep with job_type = 'committor_analysis'

test_get_batch_file_aimless_shooting_amber()
Tests thread.get_batch_template with job_type = 'aimless_shooting' and md_engine = 'amber'

test_get_batch_file_aimless_shooting_broken()
Tests thread.get_batch_template with job_type = 'aimless_shooting' and invalid type = 'fwdd'

test_get_batch_file_equilibrium_path_sampling_amber()
Tests thread.get_batch_template with job_type = 'equilibrium_path_sampling' and md_engine = 'amber'

test_get_batch_file_equilibrium_path_sampling_broken()
Tests thread.get_batch_template with job_type = 'equilibrium_path_sampling' and invalid type = 'fwdd'

test_get_batch_template_committor_analysis()
Tests get_batch_template with job_type = 'committor_analysis'

test_get_initial_coordinates_aimless_shooting()
Tests get_initial_coordinates with job_type = 'aimless_shooting'

test_get_initial_coordinates_committor_analysis_rc_out()
Tests get_initial_coordinates with job_type = 'committor_analysis' using an RC out file

test_get_initial_coordinates_committor_analysis_rc_out_does_not_exist()
Tests get_initial_coordinates with job_type = 'committor_analysis' using an RC out file that does not exist

test_get_initial_coordinates_committor_analysis_rc_out_no_shooting_points()
Tests get_initial_coordinates with job_type = 'committor_analysis' using an RC out file containing no shooting points within the threshold

test_get_initial_coordinates_equilibrium_path_sampling()
Tests get_initial_coordinates with job_type = 'equilibrium_path_sampling'

test_get_inpcrd_aimless_shooting_init()
Tests get_inpcrd with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_get_inpcrd_aimless_shooting_prod()
Tests get_inpcrd with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod']

test_get_inpcrd_committor_analysis()
Tests get_inpcrd with job_type = 'committor_analysis'

test_get_inpcrd_equilibrium_path_sampling()
Tests get_inpcrd with job_type = 'equilibrium_path_sampling'

test_get_input_file_aimless_shooting()
Tests get_input_file with job_type = 'aimless_shooting'

test_get_input_file_committor_analysis()
Tests get_input_file with job_type = 'committor_analysis'

test_get_input_file_equilibrium_path_sampling()
Tests get_input_file with job_type = 'equilibrium_path_sampling'

test_get_input_file_rxncor_umbrella_sampling()
Tests get_input_file with job_type = 'umbrella_sampling'

test_get_next_step_aimless_shooting_init()
Tests thread.get_next_step with job_type = 'aimless_shooting' and type = ['init']

test_get_next_step_aimless_shooting_prod()
Tests thread.get_next_step with job_type = 'aimless_shooting' and type = ['fwd','bwd']

test_get_next_step_committor_analysis()
Tests get_next_step with job_type = 'committor_analysis'

test_get_next_step_equilibrium_path_sampling_first()
Tests get_next_step with job_type = 'equilibrium_path_sampling' and an empty current_type

test_get_next_step_equilibrium_path_sampling_init()
Tests get_next_step with job_type = 'equilibrium_path_sampling' and current_type = ['init']

test_get_next_step_equilibrium_path_sampling_prod()
Tests get_next_step with job_type = 'equilibrium_path_sampling' and current_type = ['prod', 'prod']

test_update_history_aimless_shooting_init()
Tests update_history with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_update_history_aimless_shooting_prod()
Tests update_history with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod']

test_update_history_committor_analysis()
Tests update_history with job_type = 'committor_analysis'

test_update_history_equilibrium_path_sampling_init()
Tests update_history with job_type = 'equilibrium_path_sampling' with current_type = ['init']

test_update_history_equilibrium_path_sampling_out_of_bounds()
Tests update_history with job_type = 'equilibrium_path_sampling' with an initially out-of-bounds RC value

test_update_history_equilibrium_path_sampling_prod()
Tests update_history with job_type = 'equilibrium_path_sampling' with current_type = ['prod', 'prod']

test_update_results_aimless_shooting_init()
Tests update_results with job_type = 'aimless_shooting' and thread.current_type = ['init']

test_update_results_aimless_shooting_prod()
Tests update_results with job_type = 'aimless_shooting' and thread.current_type = ['prod', 'prod']

test_update_results_committor_analysis()
Tests update_results with job_type = 'committor_analysis'

test_update_results_equilibrium_path_sampling()
Tests update_results with job_type = 'equilibrium_path_sampling'

atesa.tests.test_jobtype.**config_equilibrium_path_sampling**()
Sets up configuration settings for equilibrium path sampling tests, to be overwritten as needed

atesa.tests.test_lmax module

Unit and regression test for lmax.py.

class atesa.tests.test_lmax.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of class

test_main_improper_input()
Tests main using an input file that does exist, but is improperly formatted

test_main_k_and_f()
Tests main using k and f

test_main_k_f_and_p()
Tests main using k and f

test_main_k_f_and_q()
Tests main using k, f, and q

test_main_non_existent_input()
Tests main using an input file name that does not exist

test_main_running()
Tests main using running

test_main_two_line_test()
Tests main using two_line_test

test_objective_function()
Tests objective_function using dummy data

test_two_line_test()
Tests two_line_test using dummy data

atesa.tests.test_lmax_temp module

atesa.tests.test_main module

Unit and regression test for the atesa_v2 package.

class atesa.tests.test_main.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of each method

test_atesa_v2_imported()
Sample test, will always pass so long as import statement worked

test_configure_broken()
Tests configure.py with a non-existent file

test_configure_directory()
Tests configure.py behavior in correcting an improperly formatted directory

test_import_cvs()
Tests importing CVs and commitment definitions from a given settings.pkl file

test_init_threads_new()
Tests successful initialization of new threads

test_init_threads_restart()
Tests successful initialization of restarted threads

test_main()
Tests main.main with DEBUG = True to skip actual job submission/monitoring

test_thread_get_frame_amber()
Tests thread.get_frame method with md_engine = 'amber' and frame = -1

atesa.tests.test_process module

Unit and regression test for process.py.

class atesa.tests.test_process.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of class

test_process_to_be_terminated()
Tests process.py for a thread with terminated = True

test_process_to_submit()
Tests process.py for a thread that should be submitted

atesa.tests.test_rc_eval module

Unit and regression test for utilities.py.

class atesa.tests.test_rc_eval.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of class

test_main()
Tests rc_eval.main using sham shooting points in test_data

atesa.tests.test_utilities module

Unit and regression test for utilities.py.

class atesa.tests.test_utilities.**Tests**
Bases: object

setup_method(*test_method*)

classmethod teardown_method(*method*)
Runs at end of class

test_check_commit_bwd()
Tests check_commit using a dummy coordinate file and commitments defined to get result 'bwd'

test_check_commit_fwd()
Tests check_commit using a dummy coordinate file and commitments defined to get result 'fwd'

test_check_commit_none()
Tests check_commit using a dummy coordinate file and commitments defined to get result ''

test_check_commit_traj()
Tests check_commit using a dummy trajectory file and commitments defined to get result 'fwd'

test_check_commit_value_errors()
Tests check_commit using a dummy coordinate file and commitments defined to get ValueErrors

test_evaluate_rc()
Tests evaluate_rc

test_get_cvs_no_qdot()
Tests get_cvs with a dummy coordinate file and include_qdot = False

test_get_cvs_no_qdot_reduce()
Tests get_cvs with a dummy coordinate file, include_qdot = False, and reduce = True

test_get_cvs_with_qdot()
Tests get_cvs with a dummy coordinate file, include_qdot = True, and a coordinate file with velocities

test_get_cvs_with_qdot_broken()
Tests get_cvs with a dummy coordinate file, include_qdot = True, and a coordinate file without velocities

test_interpret_cv()
Test interpret_cv for a distance, angle, dihedral, and difference of distances for both pytraj and mdtraj

test_resample()
Tests resample

test_rev_vels()
Tests rev_vels by comparing to a known-correct file

Module contents

This is the docstring for an empty init file, to support testing packages besides PyTest such as Nose that may look for such a file

8.2 Submodules

8.3 atesa.batchsystem module

Interface for BatchSystem objects. New BatchSystems can be implemented by constructing a new class that inherits from BatchSystem and implements its abstract methods.

class atesa.batchsystem.**AdaptPBS**
Bases: *atesa.batchsystem.BatchSystem*

Adapter class for PBS/Torque BatchSystem.

cancel_job(*jobid*, *settings*)
Cancel the job given by jobid

Parameters

- **jobid** (*str*) – The jobid to cancel
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **output** – Raw output string from batch system, if any

Return type str

get_status(*jobid*, *settings*)
Query batch system for a status string for the given job.

Parameters

- **jobid** (*str*) – The jobid to query

- **settings** (*argparse.Namespace*) – Settings namespace object

Returns status – A one-character status string. Options are: 'R'unning, 'Q'ueued, and 'C'omplete/'C'anceled.

Return type str

get_submit_command()
Return the appropriate terminal command for submitting a batch job, with '{file}' where the file to submit should be indicated.

Parameters None

Returns output – Appropriate command including '{file}' substring

Return type str

class atesa.batchsystem.**AdaptSlurm**
Bases: *atesa.batchsystem.BatchSystem*

Adapter class for Slurm BatchSystem.

cancel_job(*jobid*, *settings*)
Cancel the job given by jobid

Parameters

- **jobid** (*str*) – The jobid to cancel
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns output – Raw output string from batch system, if any

Return type str

get_status(*jobid*, *settings*)
Query batch system for a status string for the given job.

Parameters

- **jobid** (*str*) – The jobid to query
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns status – A one-character status string. Options are: 'R'unning, 'Q'ueued, and 'C'omplete/'C'anceled.

Return type str

get_submit_command()
Return the appropriate terminal command for submitting a batch job, with '{file}' where the file to submit should be indicated.

Parameters None

Returns output – Appropriate command including '{file}' substring

Return type str

class atesa.batchsystem.**BatchSystem**
Bases: abc.ABC

Abstract base class for HPC cluster batch systems.

Implements methods for all of the batch system-specific tasks that ATESA might need.

cancel_job(*jobid*, *settings*)
Cancel the job given by jobid

Parameters

- **jobid** (*str*) – The jobid to cancel
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **output** – Raw output string from batch system, if any

Return type str

get_status (*jobid*, *settings*)
Query batch system for a status string for the given job.

Parameters

- **jobid** (*str*) – The jobid to query
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – A one-character status string. Options are: 'R'unning, 'Q'ueued, and 'C'omplete/'C'anceled.

Return type str

get_submit_command ()
Return the appropriate terminal command for submitting a batch job, with '{file}' where the file to submit should be indicated.

Parameters **None**

Returns **output** – Appropriate command including '{file}' substring

Return type str

8.4 atesa.boltzmann_weight module

Standalone script for converting equilibrium path sampling output files into free energy profiles via Boltzmann weighting.

atesa.boltzmann_weight.**main** (***kwargs*)
Process equilibrium path sampling output file kwargs['i'] to obtain free energy profile.

This is the main function of boltzmann_weight.py, which converts each EPS window into a stretch of the energy profile and stitches these stretches together into a continuous plot. Simply put, this function descretizes and reweights the data according to the Boltzmann weight (E = kT*ln(p), where p is the probability of a given state).

Parameters **kwargs** (*dict*) – Dictionary object containing arguments passed in from command line in "if __name__ == '__main__'" section.

Returns

Return type None

atesa.boltzmann_weight.**objective_function** (*slope*, *probs*, *RC_values*, *kT*)

atesa.boltzmann_weight.**update_progress** (*progress*, *message='Progress'*)
Print a dynamic progress bar to stdout.

Credit to Brian Khuu from stackoverflow, https://stackoverflow.com/questions/3160699/python-progress-bar

Parameters

- **progress** (*float*) – A number between 0 and 1 indicating the fractional completeness of the bar. A value under 0 represents a 'halt'. A value at 1 or bigger represents 100%.

- **message** (*str*) – The string to precede the progress bar (so as to indicate what is progressing)

Returns

Return type None

8.5 atesa.configure module

configure.py Takes user input file and returns settings namespace object

atesa.configure.**configure**(*input_file*, *user_working_directory=''*)

Configure the settings namespace based on the config file.

Parameters

- **input_file** (*str*) – Name of the configuration file to read
- **user_working_directory** (*str*) – User override for working directory (overrides value in input_file), ignored if set to ''

Returns **settings** – Settings namespace object

Return type argparse.Namespace

8.6 atesa.factory module

Factory script for obtaining the desired interfaces from the various interface scripts.

atesa.factory.**batchsystem_factory**(*batchsystem_toolkit*)

Factory function for BatchSystems.

Parameters **batchsystem_toolkit** (*str*) – Name of the BatchSystem to invoke

Returns **batchsystem** – Instance of a BatchSystem adapter

Return type *BatchSystem*

atesa.factory.**jobtype_factory**(*jobtype_toolkit*)

Factory function for JobTypes.

Parameters **jobtype_toolkit** (*str*) – Name of the JobType to invoke

Returns **jobtype** – Instance of a JobType adapter

Return type *JobType*

atesa.factory.**mdengine_factory**(*mdengine_toolkit*)

Factory function for MDEngines.

Parameters **mdengine_toolkit** (*str*) – Name of the MDEngine to invoke

Returns **mdengine** – Instance of an MDEngine adapter

Return type *MDEngine*

atesa.factory.**taskmanager_factory**(*taskmanager_toolkit*)

Factory function for TaskManagers.

Parameters **taskmanager_toolkit** (*str*) – Name of the TaskManager to invoke

Returns **taskmanager** – Instance of a TaskManager adapter

Return type *TaskManager*

8.7 atesa.information_error module

Script to evaluate and store information error in support of the information error convergence criterion in aimless shooting. Implemented as a separate script to facilitate multiprocessing.

atesa.information_error.**main**()

Evaluate the information error during aimless shooting and output results to info_err.out.

Reads decorrelated aimless shooting output files from the present directory to evaluate the mean parametric variance based on the Godambe information error of the aimless shooting process.

This function depends on lmax.py for evaluations of the information error.

Parameters None

Returns

Return type None

8.8 atesa.interpret module

This portion of the program is responsible for handling update of the results, checking global termination criteria, and implementing the calls to JobType methods to control the value of the thread.coordinates attribute for the next step.

atesa.interpret.**interpret**(*thread*, *allthreads*, *running*, *settings*)

The main function of interpret.py. Makes calls to JobType methods to update results, check termination criteria, and update thread.coordinates

Parameters

- **thread** (*Thread*) – The Thread object on which to act
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – True if a global termination criterion has been met; False otherwise

Return type bool

8.9 atesa.jobtype module

Interface for JobType objects. New JobTypes can be implemented by constructing a new class that inherits from JobType and implements its abstract methods.

class atesa.jobtype.**AimlessShooting**

Bases: *atesa.jobtype.JobType*

Adapter class for aimless shooting

algorithm(*thread*, *allthreads*, *running*, *settings*)

Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)

Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination(*thread*, *allthreads*, *settings*)

Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup(*settings*)

Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper(*thread*, *settings*)

Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template (*thread*, *type*, *settings*)
Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates (*settings*)
Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd (*thread*)
Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file (*thread*, *job_index*, *settings*)
Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step (*thread*, *settings*)
Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns type – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history (*thread*, *settings*, ***kwargs*)
Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results (*thread*, *allthreads*, *settings*)
Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify (*arg*, *argtype*)
Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

class atesa.jobtype.**CommittorAnalysis**
Bases: *atesa.jobtype.JobType*

Adapter class for committor analysis

algorithm(*thread*, *allthreads*, *running*, *settings*)
Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)
Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination(*thread*, *allthreads*, *settings*)
Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup (*settings*)
Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper (*thread*, *settings*)
Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template (*thread*, *type*, *settings*)
Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates (*settings*)
Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd (*thread*)
Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file (*thread*, *job_index*, *settings*)
Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step(*thread*, *settings*)

Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **type** – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history(*thread*, *settings*, ***kwargs*)

Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results(*thread*, *allthreads*, *settings*)

Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify(*arg*, *argtype*)
Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

class atesa.jobtype.**EquilibriumPathSampling**
Bases: *atesa.jobtype.JobType*

Adapter class for equilibrium path sampling

algorithm(*thread*, *allthreads*, *running*, *settings*)
Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)
Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination(*thread*, *allthreads*, *settings*)
Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup(*settings*)

Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper(*thread*, *settings*)

Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template(*thread*, *type*, *settings*)

Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates(*settings*)

Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd(*thread*)

Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file (*thread*, *job_index*, *settings*)
Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step (*thread*, *settings*)
Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **type** – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history (*thread*, *settings*, ***kwargs*)
Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results (*thread*, *allthreads*, *settings*)
Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify(*arg*, *argtype*)

Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

class atesa.jobtype.**FindTS**

Bases: *atesa.jobtype.JobType*

Adapter class for finding transition state (find TS)

algorithm(*thread*, *allthreads*, *running*, *settings*)

Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)

Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination(*thread*, *allthreads*, *settings*)
Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup(*settings*)
Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper(*thread*, *settings*)
Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template(*thread*, *type*, *settings*)
Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates(*settings*)
Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd(*thread*)
Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file(*thread*, *job_index*, *settings*)
Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step(*thread*, *settings*)
Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **type** – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history(*thread*, *settings*, ***kwargs*)
Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results(*thread*, *allthreads*, *settings*)
Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify(*arg*, *argtype*)
Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

class atesa.jobtype.**JobType**
Bases: abc.ABC

Abstract base class for job types.

Implements methods for all of the job type-specific tasks that ATESA might need.

algorithm(*thread*, *allthreads*, *running*, *settings*)
Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)
Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an

unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination (*thread*, *allthreads*, *settings*)

Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup (*settings*)

Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper (*thread*, *settings*)

Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template (*thread*, *type*, *settings*)

Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates(*settings*)
Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd(*thread*)
Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file(*thread*, *job_index*, *settings*)
Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step(*thread*, *settings*)
Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **type** – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history(*thread*, *settings*, ***kwargs*)
Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results(*thread*, *allthreads*, *settings*)
Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify(*arg*, *argtype*)
Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

class atesa.jobtype.**UmbrellaSampling**
Bases: *atesa.jobtype.JobType*

Adapter class for umbrella sampling

add_pathway_restraints(*thread*, *settings*)
Build appropriate pathway restraints DISANG file for the window in the specified thread

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

algorithm(*thread*, *allthreads*, *running*, *settings*)
Update thread attributes to prepare for next move.

This is where the core logical algorithm of a method is implemented. For example, in aimless shooting, it encodes the logic of when and how to select a new shooting move.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **running** (*list*) – The list of all currently running Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The list of all currently running Thread objects

Return type list

check_for_successful_step(*thread*, *settings*)

Check whether a just-completed step was successful, as defined by whether the update_results and check_termination methods should be run.

This method returns True if the previous step appeared successful (distinct from 'accepted' as the term refers to for example aimless shooting) and False otherwise. The implementation of what to DO with an unsuccessful step should appear in the corresponding algorithm method, which should be run regardless of the output from this method.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **result** – True if successful; False otherwise

Return type bool

check_termination(*thread*, *allthreads*, *settings*)

Check termination criteria for the particular thread at hand as well as for the entire process.

These methods should update thread.status and thread.terminated in order to communicate the results of the check for the inidividual thread, and should return a boolean to communicate the results of the check for the entire run.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **termination** – Global termination criterion for entire process (True means terminate)

Return type bool

cleanup(*settings*)

Perform any last tasks between the end of the main loop and the program exiting.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

gatekeeper(*thread*, *settings*)

Return boolean indicating whether job is ready for next interpretation step.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **status** – If True, ready for next interpretation step; otherwise, False

Return type bool

get_batch_template(*thread*, *type*, *settings*)

Return name of batch template file for the type of job indicated.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **type** (*str*) – Name of the type of job desired, corresponding to the template file
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **name** – Name of the batch file template requested

Return type str

get_initial_coordinates(*settings*)

Obtain list of initial coordinate files and copy them to the working directory.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **initial_coordinates** – List of strings naming the applicable initial coordinate files that were copied to the working directory

Return type list

get_inpcrd(*thread*)

Return a list (possibly of length one) containing the names of the appropriate inpcrd files for the next step in the thread given by thread.

Parameters **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects

Returns **inpcrd** – List of strings containing desired file names

Return type list

get_input_file(*thread*, *job_index*, *settings*)

Obtain appropriate input file for next job.

At its most simple, implementations of this method can simply return settings.path_to_input_files + '/' + settings.job_type + '_' + thread.current_type[job_index] + '_' + settings.md_engine + '.in'

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **job_index** (*int*) – 0-indexed integer identifying which job within thread.current_type to return the input file for
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **input_file** – Name of the applicable input file

Return type str

get_next_step (*thread*, *settings*)
Return name of next type of simulation to run in the itinerary of this job type.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns type – List containing strings for name(s) of next type(s) of simulation(s)

Return type list

update_history (*thread*, *settings*, ***kwargs*)
Update or initialize the history namespace for this job type.

This namespace is used to store the full history of a threads coordinate and trajectory files, as well as their results if necessary.

If update_history is called with a kwargs containing {'initialize': True}, it simply prepares a blank history namespace and returns it. Otherwise, it adds the values of the desired keywords (which are desired depends on the implementation) to the corresponding history attributes in the index given by thread.suffix.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object
- **kwargs** (*dict*) – Dictionary of arguments that might be used to update the history object

Returns

Return type None

update_results (*thread*, *allthreads*, *settings*)
Update appropriate results file(s) as needed.

These methods are designed to be called at the end of every step in a job, even if writing to an output file is not necessary after that step; for that reason, they also encode the logic to decide when writing is needed.

Parameters

- **thread** (*Thread*) – Methods in the JobType abstract base class are intended to operate on Thread objects
- **allthreads** (*list*) – The list of all extant Thread objects
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

verify (*arg*, *argtype*)
Confirm or deny that the object arg is a valid object of type type.

Parameters

- **arg** (*any_type*) – Object to verify
- **argtype** (*str*) – Name of type of object

Returns

Return type None

8.10 atesa.lmax module

Likelihood maximization script. This program is designed to be entirely separable from ATESA in that it can be called manually to perform likelihood maximization to user specifications and with arbitrary input files; however, it is required by ATESA's aimless shooting information error convergence criterion.

atesa.lmax.**eval_rc**(*params*, *obs*)

atesa.lmax.**main**(***kwargs*)
Main runtime function of lmax.py.

Assembles lists of models to optimize in the form of lists of CVs, passes them to optimize, interprets results, and repeats or terminates in accordance with argument-dependent termination criteria.

Parameters **kwargs** (*dict*) – Dictionary object containing arguments

Returns

Return type None

atesa.lmax.**objective_function**(*params*, *A_data*, *B_data*)
Evaluate the negative log likelihood function for the given parameters and lists of observations.

This function evaluates the goodness of fit of the given parameters and data to an error function ansatz, as described in Peters, 2012. Chem. Phys. Lett. 554: 248.

Designed to be called by an optimization routine to obtain the best fitting params.

Parameters

- **params** (*list*) – Parameters for the current model to be tested
- **A_data** (*list*) – List of observations from aimless shooting that committed to basin "A" (usually the reactants)
- **B_data** (*list*) – List of observations from aimless shooting that committed to basin "B" (usually the products)

Returns **negative_log_likelihood** – The negative log likelihood of the fit to the ansatz for the given parameters and observations

Return type float

atesa.lmax.**two_line_test_func**(*results*, *plots*, *two_line_threshold=0.5*)
Perform a double linear regression on intersecting subsets of the data in results to determine whether to terminate and how many dimensions to return in the RC during two_line_test.

Can only be called with len(results) >= 5.

Parameters

- **results** (*list*) – List of dictionary objects indexed by step of two_line_test, each possessing attribute 'fun' giving the optimization score for that step
- **plots** (*bool*) – If True, plot lines using gnuplot
- **two_line_threshold** (*float*) – Ratio of second slope to first slope (as a fraction) below which the two-line test can pass

Returns **out** – Index of selected 'best' RC from two-line test; or, -1 if no best RC could be determined

Return type int

atesa.lmax.**update_progress**(*progress*, *message='Progress'*, *eta=0*, *quiet=False*)
Print a dynamic progress bar to stdout.

Credit to Brian Khuu from stackoverflow, https://stackoverflow.com/questions/3160699/python-progress-bar

Parameters

- **progress** (*float*) – A number between 0 and 1 indicating the fractional completeness of the bar. A value under 0 represents a 'halt'. A value at 1 or bigger represents 100%.
- **message** (*str*) – The string to precede the progress bar (so as to indicate what is progressing)
- **eta** (*int*) – Number of seconds to display as estimated completion time (converted into HH:MM:SS)
- **quiet** (*bool*) – If True, suppresses output entirely

Returns

Return type None

8.11 atesa.lmax_temp module

8.12 atesa.main module

main.py Version 2 of Aimless Transition Ensemble Sampling and Analysis refactors the code to make it portable, extensible, and flexible.

This script handles the primary loop of building and submitting jobs in independent Threads, using the methods thereof to execute various interfaced/abstracted commands.

class atesa.main.**Thread**
Bases: object

Object representing a series of simulations and containing the relevant information to define its current state.

Threads represent the level on which ATESA is parallelized. This flexible object is used for every type of job performed by ATESA.

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

cancel_job(*job_index*, *settings*)

gatekeeper(*settings*)

get_batch_template(*type*, *settings*)

get_frame(*traj*, *frame*, *settings*)

get_next_step(*settings*)

get_status(*job_index*, *settings*)

interpret(*allthreads*, *running*, *settings*)

process(*running*, *settings*)

atesa.main.**handle_loop_exception**(*attempted_rescue*, *running*, *settings*)
Handle attempted rescue of main loop after encountering an exception, or cancellation of jobs if rescue fails.

Parameters

- **attempted_rescue** (*bool*) – True if rescue has already been attempted and this function is being called again. Skips attempting rescue again and simply cancels all running jobs.
- **running** (*list*) – List of Thread objects that are currently running. These are the threads that will be canceled if the ATESA run cannot be rescued.
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

atesa.main.**init_threads**(*settings*)

Initialize all the Thread objects called for by the user input file.

In the case where settings.restart == True, this involves unpickling restart.pkl; otherwise, brand new objects are produced in accordance with settings.job_type (aimless_shooting, committor_analysis, equilibrium_path_sampling, or isee).

Parameters **settings** (*argparse.Namespace*) – Settings namespace object

Returns **allthreads** – List of all Thread objects, including those for which no further tasks are scheduled.

Return type list

atesa.main.**main**(*settings*, *rescue_running=[]*)

Perform the primary loop of building, submitting, monitoring, and analyzing jobs.

This function works via a loop of calls to thread.process and thread.interpret for each thread that hasn't terminated, until either the global termination criterion is met or all the individual threads have completed.

Parameters

- **settings** (*argparse.Namespace*) – Settings namespace object
- **rescue_running** (*list*) – List of threads passed in from handle_loop_exception, containing running threads. If given, setup is skipped and the function proceeds directly to the main loop.

Returns **exit_message** – A message indicating the status of ATESA at the end of main

Return type str

atesa.main.**main_loop**(*settings*, *allthreads*, *running*)

atesa.main.**run_main**()

8.13 atesa.mdengine module

Interface for MDEngine objects. New MDEngines can be implemented by constructing a new class that inherits from MDEngine and implements its abstract methods.

class atesa.mdengine.**AdaptAmber**

Bases: *atesa.mdengine.MDEngine*

Adapter class for Amber MDEngine.

get_frame(*trajectory*, *frame*, *settings*)

Return a new file containing just the frame'th frame of a trajectory in Amber .rst7 format

Parameters

- **trajectory** (*str*) – Name of trajectory file to obtain last frame from
- **frame** (*int*) – Index of frame to return; 1-indexed, -1 gives last frame, 0 is invalid
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **last_frame** – Name of .rst7 format coordinate file corresponding to desired frame of trajectory, if it exists; an empty string otherwise

Return type str

write_find_ts_restraint (*basin*, *inp_file*)

Return the input file for a restrained simulation that performs barrier crossing in find_ts

Parameters

- **basin** (*tuple*) – Either settings.commit_fwd or settings.commit_bwd, defining the basin to restrain towards
- **inp_file** (*str*) – Name of original input file without restraint

Returns **input_file** – Name of the simulation input file

Return type str

class atesa.mdengine.**MDEngine**

Bases: abc.ABC

Abstract base class for molecular dynamics engines.

Implements methods for all of the engine-specific tasks that ATESA might need.

get_frame (*trajectory*, *frame*, *settings*)

Return a new file containing just the frame'th frame of a trajectory in Amber .rst7 format

Parameters

- **trajectory** (*str*) – Name of trajectory file to obtain last frame from
- **frame** (*int*) – Index of frame to return; 1-indexed, -1 gives last frame, 0 is invalid
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **last_frame** – Name of .rst7 format coordinate file corresponding to desired frame of trajectory, if it exists; an empty string otherwise

Return type str

write_find_ts_restraint (*basin*, *inp_file*)

Return the input file for a restrained simulation that performs barrier crossing in find_ts

Parameters

- **basin** (*tuple*) – Either settings.commit_fwd or settings.commit_bwd, defining the basin to restrain towards
- **inp_file** (*str*) – Name of original input file without restraint

Returns **input_file** – Name of the simulation input file

Return type str

8.14 atesa.process module

This portion of the program is responsible for handling setup of the appropriate batch script(s) for the next step in a Thread, passing them to a task manager to submit them, and updating the list of currently running threads accordingly.

atesa.process.**process**(*thread*, *running*, *settings*)

The main function of process.py. Reads the thread to identify the next step, then builds and submits the batch file(s) as needed.

Parameters

- **thread** (*Thread()*) – The Thread object on which to act
- **running** (*list*) – The list of currently running threads, which will be updated as needed
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **running** – The updated list of currently running threads after this process step

Return type list

8.15 atesa.rc_eval module

rc_eval.py Standalone script to evaluate RC values given an aimless shooting working directory and reaction coordinate definition

atesa.rc_eval.**main**(*working_directory*, *rc_definition*, *as_out_file*, *extrema=False*)

The main function of rc_eval.py. Accepts an aimless shooting working directory and a reaction coordinate definition, producing in that directory a new file named 'rc.out' (overwriting if one already exists) identifying each shooting point (files in the directory whose names end in "_init.rst7") and its corresponding reaction coordinate value in a sorted list.

If extrema == True, skips producing rc.out and just returns the minimum and maximum RC value for a single accepted shooting move, which is useful when preparing umbrella sampling simulations.

Parameters

- **working_directory** (*str*) – The path to the aimless shooting working directory in which to act
- **rc_definition** (*str*) – A reaction coordinate definition formatted as a string of python-readable code with "CV[X]" standing in for the Xth CV value (one-indexed); this RC definition should be in terms of reduced variables (values between 0 and 1)
- **as_out_file** (*str*) – Path to the aimless shooting output file used to build the reaction coordinate. Usually this should be a decorrelated file (named with "decorr").
- **extrema** (*bool*) – If True, skips producing rc.out and just returns the minimum and maximum RC value for a single accepted shooting move, which is useful when preparing umbrella sampling simulations.

Returns

Return type None

atesa.rc_eval.**update_progress**(*progress*, *message='Progress'*, *eta=0*, *quiet=False*)

Print a dynamic progress bar to stdout.

Credit to Brian Khuu from stackoverflow, https://stackoverflow.com/questions/3160699/python-progress-bar

Parameters

- **progress** (*float*) – A number between 0 and 1 indicating the fractional completeness of the bar. A value under 0 represents a 'halt'. A value at 1 or bigger represents 100%.
- **message** (*str*) – The string to precede the progress bar (so as to indicate what is progressing)
- **eta** (*int*) – Number of seconds to display as estimated completion time (converted into HH:MM:SS)
- **quiet** (*bool*) – If True, suppresses output entirely

Returns

Return type None

8.16 atesa.resample_and_inferr module

Helper file to call utilities.resample and then information_error.main in sequence in a single process.

`atesa.resample_and_inferr.main()`

8.17 atesa.taskmanager module

Interface for TaskManager objects. New TaskManagers can be implemented by constructing a new class that inherits from TaskManager and implements its abstract methods.

class atesa.taskmanager.**AdaptSimple**

Bases: *atesa.taskmanager.TaskManager*

Adapter class for my simple, no-frills task manager.

Just interfaces directly with the batch system through the terminal.

submit_batch(*filename*, *settings*)

Submit a batch file to the task manager.

Parameters

- **filename** (*str*) – Name of batch file to submit
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **jobid** – Identification number for this task, such that it can be cancelled by referring to this string

Return type str

class atesa.taskmanager.**TaskManager**

Bases: abc.ABC

Abstract base class for task managers.

Implements methods for all of the task manager-specific tasks that ATESA might need.

submit_batch(*filename*, *settings*)

Submit a batch file to the task manager.

Parameters

- **filename** (*str*) – Name of batch file to submit

- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **jobid** – Identification number for this task, such that it can be cancelled by referring to this string

Return type str

8.18 atesa.utilities module

Utility functions that don't properly fit anywhere else. These are operations that aren't specific to any particular interface or script.

atesa.utilities.**check_commit**(*filename*, *settings*)

Check commitment of coordinate file to basins defined by settings.commit_fwd and settings.commit_bwd.

Raises a RuntimeError if the supplied file is not suitable for checking for commitment.

Parameters

- **filename** (*str*) – Name of coordinate or trajectory file to be checked. If the file has more than one frame, only the last frame will be loaded.
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns **commit_flag** – Either 'fwd' or 'bwd' if the coordinates are in the corresponding basin, or '' if in neither

Return type str

atesa.utilities.**evaluate_rc**(*rc_definition*, *cv_list*)

Evaluate the RC value given by RC definition for the given list of CV values given by cv_list.

Parameters

- **rc_definition** (*str*) – A reaction coordinate definition formatted as a string of python-readable code with "CV[X]" standing in for the Xth CV value (zero-indexed); e.g., "CV2" has value "4" in the cv_list [1, -2, 4, 6]
- **cv_list** (*list*) – A list of CV values whose indices correspond to the desired values in rc_definition

Returns **rc_value** – The value of the reaction coordinate given the values in cv_list

Return type float

atesa.utilities.**get_cvs**(*filename*, *settings*, *reduce=False*, *frame=0*)

Get CV values for a coordinate file given by filename, as well as rates of change if settings.include_qdot = True.

If reduce = True, the returned CVs will be reduced to between 0 and 1 based on the minimum and maximum values of that CV in as.out, which is assumed to exist at settings.as_out_file.

If frame is > 0 or 'all', filename will be interpreted as a trajectory file instead of a .rst7 coordinate file. This option is incompatible with settings.include_qdot = True.

Parameters

- **filename** (*str*) – Name of .rst7-formatted coordinate file to be checked (or trajectory file if frame > 0 or frame == 'all')
- **settings** (*argparse.Namespace*) – Settings namespace object
- **reduce** (*bool*) – Boolean determining whether to reduce the CV values for use in evaluating an RC value that uses reduced values

- **frame** (*int or str*) – Frame of trajectory filename to use (1-indexed, negatives not valid, only used if not equal to 0); or, 'all' returns the CVs for each frame of the input trajectory, separated by newlines. 'all' is incompatible with settings.include_qdot = True

Returns **output** – Space-separated list of CV values for the given coordinate file

Return type str

atesa.utilities.**interpret_cv**(*cv_index*, *settings*)

Read the cv_index'th CV from settings.cvs, identify its type (distance, angle, dihedral, or differance-of-distances) and the atom indices the define it (one-indexed) and return these.

This function is designed for use in the umbrella_sampling jobtype only. For this reason, it only supports the aforementioned CV types. If none of these types appears to fit, this function raises a RuntimeError.

Parameters

- **cv_index** (*int*) – The index for the CV to use; e.g., 6 corresponds to CV6. Must be in the range [1, len(settings.cvs)].
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

- **atoms** (*list*) – A list of 1-indexed atom indices as strings that define the given CV
- **optype** (*str*) – A string (either 'distance', 'angle', 'dihedral', or 'diffdistance') corresponding to the type for this CV.
- **nat** (*int*) – The number of atoms constituting the given CV

atesa.utilities.**partial_full_cvs**(*thread*, *filename*, *settings*)

Write the full CVs list to the file given by filename for each accepted move in each thread in threads.

A helper function for parallelizing get_cvs (has to be defined at top-level for multiprocessing)

Parameters

- **thread** (*Thread*) – Thread object to evaluate
- **filename** (*str*) – Name of the file to write CVs to
- **settings** (*argparse.Namespace*) – Settings namespace object

Returns

Return type None

atesa.utilities.**resample**(*settings*, *partial=False*, *full_cvs=False*)

Resample each shooting point in each thread with different CV definitions to produce new output files with extant aimless shooting data.

This function also assesses decorrelation times and produces one or more decorrelated output files. If and only if settings.information_error_checking == True, decorrelated files are produced at each settings.information_error_freq increment. In this case, if partial == True, decorrelation will only be assessed for data lengths absent from the info_err.out file in the working directory.

Parameters

- **settings** (*argparse.Namespace*) – Settings namespace object
- **partial** (*bool*) – If True, reads the info_err.out file and only builds new decorrelated output files where the corresponding lines are missing from that file. If partial == False, decorrelation is assessed for every valid data length. Has no effect if not settings.information_error_checking.

- **full_cvs** (*bool*) – If True, also resamples as_full_cvs.out using every prod trajectory in the working directory.

Returns

Return type None

atesa.utilities.**rev_vels**(*restart_file*)

Reverse all the velocity terms in a restart file and return the name of the new, 'reversed' file.

Parameters **restart_file** (*str*) – Filename of the 'fwd' restart file, in .rst7 format

Returns **reversed_file** – Filename of the newly written 'bwd' restart file, in .rst7 format

Return type str

8.19 Module contents

atesa Python program for automating the "Aimless Transition Ensemble Sampling and Analysis" (ATESA) aimless shooting workflow on PBS/TORQUE or Slurm.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

E

F

G

H

I

J

M

O

P

R

S

T

U

V

W