WESTPA Documentation

Release 1.0b1

Matthew C. Zwier and Lillian T. Chong

Jan 15, 2021

CONTENTS

CHAPTER

ONE

OVERVIEW

WESTPA is a package for constructing and running stochastic simulations using the "weighted ensemble" approach of Huber and Kim (1996) (see overview).

For use of WESTPA please cite the following:

Zwier, M.C., Adelman, J.L., Kaus, J.W., Pratt, A.J., Wong, K.F., Rego, N.B., Suarez, E., Lettieri, S., Wang, D. W., Grabe, M., Zuckerman, D. M., and Chong, L. T. "WESTPA: An Interoperable, Highly Scalable Software Package For Weighted Ensemble Simulation and Analysis," J. Chem. Theory Comput., 11: 800809 (2015).

To help us fund development and improve WESTPA please fill out a one-minute survey and consider contributing documentation or code to the WESTPA community.

WESTPA is free software, licensed under the terms of the GNU General Public License, Version 3. See the file `COPYING` for more information.

CHAPTER

TWO

OBTAINING AND INSTALLING WESTPA

WESTPA is developed and tested on Unix-like operating systems, including Linux and Mac OS X.

Before installing WESTPA, you will need to first install the Python 2.7 version provided by the latest free Anaconda Python distribution. After installing the Anaconda Python distribution, either add the Python executable to your $PATH or set the environment variable WEST_PYTHON:

```
export WEST_PYTHON=/opt/anaconda/bin/python3
```

We recommend obtaining the latest release of WESTPA by downloading the corresponding tar.gz file from the releases page. After downloading the file, unpack the file and install WESTPA by executing the following:

```
tar xvzf westpa-master.tar.gz
cd westpa
./setup.sh
```

A westpa.sh script is created during installation, and will set the following environment variables:

```
WEST_ROOT
WEST_BIN
WEST_PYTHON
```

These environment variables must be set in order to run WESTPA on your computing cluster.

To define environment variables post-installation, simply source the `westpa.sh` script in the `westpa` directory from the command line or your setup scripts.

CHAPTER

THREE

GETTING STARTED

A Quickstart guide and tutorials are provided here.

CHAPTER

FOUR

GETTING HELP

4.1 FAQ

Responses to frequently asked questions (FAQ) can be found in the following page:

- Frequently Asked Questions (FAQ)

A mailing list for WESTPA is available, at which one can ask questions (or see if a question one has was previously addressed). This is the preferred means for obtaining help and support. See http://groups.google.com/group/westpa-users to sign up or search archived messages.

Further, all WESTPA command-line tools (located in `westpa/bin`) provide detailed help when given the -h/–help option.

Finally, while WESTPA is a powerful tool that enables expert simulators to access much longer timescales than is practical with standard simulations, there can be a steep learning curve to figuring out how to effectively run the simulations on your computing resource of choice. For serious users who have completed the online tutorials and are ready for production simulations of their system, we invite you to contact Lillian Chong (ltchong AT pitt DOT edu) about spending a few days with her lab and/or setting up video conferencing sessions to help you get your simulations off the ground.

CHAPTER

FIVE

COPYRIGHT, LICENSE, AND WARRANTY INFORMATION

5.1 For WESTPA

The WESTPA package is copyright (c) 2013, Matthew C. Zwier and Lillian T. Chong. (Individual contributions noted in each source file.)

WESTPA is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

WESTPA is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program (see the included file `COPYING`). If not, see <http://www.gnu.org/licenses/>.

Unless otherwise noted, source files included in this distribution and lacking a more specific attribution are subject to the above copyright, terms, and conditions.

5.2 For included software

Distributions of WESTPA include a number of components without modification, each of which is subject to its own individual terms and conditions. Please see each package's documentation for the most up-to-date possible information on authorship and licensing. Such packages include:

h5py See lib/h5py/docs/source/licenses.rst

blessings See lib/blessings/LICENSE

In addition, the `wwmgr` work manager is derived from the `concurrent.futures` module (as included in Python 3.2) by Brian Quinlan and copyright 2011 the Python Software Foundation. See http://docs.python.org/3/license.html for more information.

CHAPTER

SIX

ADVANCED REFERENCES

6.1 WEST

6.1.1 Setup

Defining and Calculating Progress Coordinates

Binning

The Weighted Ensemble method enhances sampling by partitioning the space defined by the progress coordinates into non-overlapping bins. WESTPA provides a number of pre-defined types of bins that the user must parameterize within the system.py file, which are detailed below.

Users are also free to implement their own mappers. A bin mapper must implement, at least, an `assign(coords, mask=None, output=None)` method, which is responsible for mapping each of the vector of coordinate tuples `coords` to an integer (`numpy.uint16`) indicating what bin that coordinate tuple falls into. The optional `mask` (a numpy bool array) specifies that some coordinates are to be skipped; this is used, for instance, by the recursive (nested) bin mapper to minimize the number of calculations required to definitively assign a coordinate tuple to a bin. Similarly, the optional `output` must be an integer (`uint16`) array of the same length as `coords`, into which assignments are written. The `assign()` function must return a reference to `output`. (This is used to avoid allocating many temporary output arrays in complex binning scenarios.)

A user-defined bin mapper must also make an `nbins` property available, containing the total number of bins within the mapper.

RectilinearBinMapper

Creates an N-dimensional grid of bins. The Rectilinear bin mapper is initialized by defining a set of bin boundaries:

```
self.bin_mapper = RectilinearBinMapper(boundaries)
```

where `boundaries` is a list or other iterable containing the bin boundaries along each dimension. The bin boundaries must be monotonically increasing along each dimension. It is important to note that a one-dimensional bin space must still be represented as a list of lists as in the following example::

```
bounds = [-float('inf'), 0.0, 1.0, 2.0, 3.0, float('inf')]
self.bin_mapper = RectilinearBinMapper([bounds])
```

A two-dimensional system might look like::

```
boundaries = [(-1,-0.5,0,0.5,1), (-1,-0.5,0,0.5,1)]
self.bin_mapper = RectilinearBinMapper(boundaries)
```

where the first tuple in the list defines the boundaries along the first progress coordinate, and the second tuple defines the boundaries along the second. Of course a list of arbitrary dimensions can be defined to create an N-dimensional grid discretizing the progress coordinate space.

VoronoiBinMapper

A one-dimensional mapper which assigns a multidimensional progress coordinate to the closest center based on a distance metric. The Voronoi bin mapper is initialized with the following signature within the `WESTSystem.initialize`::

```
self.bin_mapper = VoronoiBinMapper(dfunc, centers, dfargs=None, dfkwargs=None)
```

- `centers` is a `(n_centers, pcoord_ndim)` shaped numpy array defining the generators of the Voronoi cells
- `dfunc` is a method written in Python that returns an `(n_centers, )` shaped array containing the distance between a single set of progress coordinates for a segment and all of the centers defining the Voronoi tessellation. It takes the general form::

  ```
  def dfunc(p, centers, *dfargs, **dfkwargs):
      ...
      return d
  ```

where `p` is the progress coordinates of a single segment at one time slice of shape `(pcoord_ndim,)`, `centers` is the full set of centers, `dfargs` is a tuple or list of positional arguments and `dfwargs` is a dictionary of keyword arguments. The bin mapper's `assign` method then assigns the progress coordinates to the closest bin (minimum distance). It is the responsibility of the user to ensure that the distance is calculated using the appropriate metric.

- `dfargs` is an optional list or tuple of positional arguments to pass into `dfunc`.
- `dfkwargs` is an optional dict of keyword arguments to pass into `dfunc`.

FuncBinMapper

A bin mapper that employs a set of user-defined function, which directly calculate bin assignments for a number of coordinate values. The function is responsible for iterating over the entire coordinate set. This is best used with C/Cython/Numba methods, or intellegently-tuned numpy-based Python functions.

The `FuncBinMapper` is initialized as::

```
self.bin_mapper = FuncBinMapper(func, nbins, args=None, kwargs=None)
```

where `func` is the user-defined method to assign coordinates to bins, `nbins` is the number of bins in the partitioning space, and `args` and `kwargs` are optional positional and keyword arguments, respectively, that are passed into `func` when it is called.

The user-defined function should have the following form::

```
def func(coords, mask, output, *args, **kwargs)
    ....
```

where the assignments returned in the `output` array, which is modified in-place.

As a contrived example, the following function would assign all segments to bin 0 if the sum of the first two progress coordinates was less than `s*0.5`, and to bin 1 otherwise, where `s=1.5`::

```
def func(coords, mask, output, s):
    output[coords[:,0] + coords[:,1] < s*0.5] = 0
    output[coords[:,0] + coords[:,1] >= s*0.5] = 1

....

self.bin_mapper = FuncBinMapper(func, 2, args=(1.5,))
```

VectorizingFuncBinMapper

Like the `FuncBinMapper`, the `VectorizingFuncBinMapper` uses a user-defined method to calculate bin assignments. They differ, however, in that while the user-defined method passed to an instance of the `FuncBinMapper` is responsible for iterating over all coordinate sets passed to it, the function associated with the `VectorizingFuncBinMapper` is evaluated once for each unmasked coordinate tuple provided. It is not responsible explicitly for iterating over multiple progress coordinate sets.

The `VectorizingFuncBinMapper` is initialized as::

```
self.bin_mapper = VectorizingFuncBinMapper(func, nbins, args=None, kwargs=None)
```

where `func` is the user-defined method to assign coordinates to bins, `nbins` is the number of bins in the partitioning space, and `args` and `kwargs` are optional positional and keyword arguments, respectively, that are passed into `func` when it is called.

The user-defined function should have the following form::

```
def func(coords, *args, **kwargs)
    ....
```

Mirroring the simple example shown for the `FuncBinMapper`, the following should result in the same result for a given set of coordinates. Here segments would be assigned to bin 0 if the sum of the first two progress coordinates was less than `s*0.5`, and to bin 1 otherwise, where `s=1.5`::

```
def func(coords, s):
    if coords[0] + coords[1] < s*0.5:
        return 0
    else:
        return 1
....

self.bin_mapper = VectorizingFuncBinMapper(func, 2, args=(1.5,))
```

PiecewiseBinMapper

RecursiveBinMapper

The `RecursiveBinMapper` is used for assembling more complex bin spaces from simpler components and nesting one set of bins within another. It is initialized as::

```
self.bin_mapper = RecursiveBinMapper(base_mapper, start_index=0)
```

The `base_mapper` is an instance of one of the other bin mappers, and `start_index` is an (optional) offset for indexing the bins. Starting with the `base_mapper`, additional bins can be nested into it using the `add_mapper(mapper, replaces_bin_at)`. This method will replace the bin containing the coordinate tuple `replaces_bin_at` with the mapper specified by `mapper`.

As a simple example consider a bin space in which the `base_mapper` assigns a segment with progress coordinate with values <1 into one bin and >= 1 into another. Within the former bin, we will nest a second mapper which partitions progress coordinate space into one bin for progress coordinate values <0.5 and another for progress coordinates with values >=0.5. The bin space would look like the following with corresponding code::

```
'''
             0                             1                       2
             +-----------------------------+-----------------------+
             |             0.5             |                       |
             | +-----------+-------------+ |                       |
             | |           |             | |                       |
             | |     1     |      2      | |           0           |
             | |           |             | |                       |
             | |           |             | |                       |
             | +-----------+-------------+ |                       |prettyprint
             +---------------------------------------------------+
'''

def fn1(coords, mask, output):
    test = coords[:,0] < 1
    output[mask & test] = 0
    output[mask & ~test] = 1

def fn2(coords, mask, output):
    test = coords[:,0] < 0.5
    output[mask & test] = 0
    output[mask & ~test] = 1

outer_mapper = FuncBinMapper(fn1,2)
inner_mapper = FuncBinMapper(fn2,2)
rmapper = RecursiveBinMapper(outer_mapper)
rmapper.add_mapper(inner_mapper, [0.5])
```

Examples of more complicated nesting schemes can be found in the tests for the WESTPA binning apparatus.

Initial/Basis States

A WESTPA simulation is initialized using `w_init` with an initial distribution of replicas generated from a set of basis states. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation or due to recycling. Basis states are specified when running `w_init` either in a file specified with `--bstates-from`, or by one or more `--bstate` arguments. If neither `--bstates-from` nor at least one `--bstate` argument is provided, then a default basis state of probability one identified by the state ID zero and label "basis" will be created (a warning will be printed in this case, to remind you of this behavior, in case it is not what you wanted).

When using a file passed to `w_init` using `--bstates-from`, each line in that file defines a state, and contains a label, the probability, and optionally a data reference, separated by whitespace, as in::

```
unbound     1.0
```

or:

```
unbound_0     0.6         state0.pdb
unbound_1     0.4         state1.pdb
```

Basis states can also be supplied at the command line using one or more `--bstate` flags, where the argument matches the format used in the state file above. The total probability summed over all basis states should equal unity, however WESTPA will renormalize the distribution if this condition is not met.

Initial states are the generated from the basis states by optionally applying some perturbation or modification to the basis state. For example if WESTPA was being used to simulate ligand binding, one might want to have a basis state where the ligand was some set distance from the binding partner, and initial states are generated by randomly orienting the ligand at that distance. When using the executable propagator, this is done using the script specified under the `gen_istate` section of the `executable` configuration. Otherwise, if defining a custom propagator, the user must override the `gen_istate` method of `WESTPropagator`.

When using the executable propagator, the the script specified by `gen_istate` should take the data supplied by the environmental variable `$WEST_BSTATE_DATA_REF` and return the generated initial state to `$WEST_ISTATE_DATA_REF`. If no transform need be performed, the user may simply copy the data directly without modification. This data will then be available via `$WEST_PARENT_DATA_REF` if `$WEST_CURRENT_SEG_INITPOINT_TYPE` is `SEG_INITPOINT_NEWTRAJ`.

Target States

WESTPA can be run in a recycling mode in which replicas reaching a target state are removed from the simulation and their weights are assigned to new replicas created from one of the initial states. This mode creates a non-equilibrium steady-state that isolates members of the trajectory ensemble originating in the set of initial states and transitioning to the target states. The flux of probability into the target state is then inversely proportional to the mean first passage time (MFPT) of the transition.

Target states are defined when initializing a WESTPA simulation when calling `w_init`. Target states are specified either in a file specified with `--tstates-from`, or by one or more `--tstate` arguments. If neither `--tstates-from` nor at least one `--tstate` argument is provided, then an equilibrium simulation (without any sinks) will be performed.

Target states can be defined using a text file, where each line defines a state, and contains a label followed by a representative progress coordinate value, separated by whitespace, as in::

```
bound     0.02
```

for a single target and one-dimensional progress coordinates or::

```
bound    2.7    0.0
drift    100    50.0
```

for two targets and a two-dimensional progress coordinate.

The argument associated with `--tstate` is a string of the form `'label, pcoord0 [,pcoord1[,...]]'`, similar to a line in the example target state definition file above. This argument may be specified more than once, in which case the given states are appended to the list of target states for the simulation in the order they appear on the command line, after those that are specified by `--tstates-from`, if any.

WESTPA uses the representative progress coordinate of a target-state and converts the **entire** bin containing that progress coordinate into a recycling sink.

Propagators

The Executable Propagator

Writing custom propagators

While most users will use the Executable propagator to run dynamics by calling out to an external piece of software, it is possible to write custom propagators that can be used to generate sampling directly through the python interface. This is particularly useful when simulating simple systems, where the overhead of starting up an external program is large compared to the actual cost of computing the trajectory segment. Other use cases might include running sampling with software that has a Python API (e.g. OpenMM).

In order to create a custom propagator, users must define a class that inherits from `WESTPropagator` and implement three methods:

- `get_pcoord(self, state)`: Get the progress coordinate of the given basis or initial state.
- `gen_istate(self, basis_state, initial_state)`: Generate a new initial state from the given basis state. This method is optional if `gen_istates` is set to `False` in the propagation section of the configuration file, which is the default setting.
- `propagate(self, segments)`: Propagate one or more segments, including any necessary per-iteration setup and teardown for this propagator.

There are also two stubs that that, if overridden, provide a mechanism for modifying the simulation before or after the iteration:

- `prepare_iteration(self, n_iter, segments)`: Perform any necessary per-iteration preparation. This is run by the work manager.
- `finalize_iteration(self, n_iter, segments)`: Perform any necessary post-iteration cleanup. This is run by the work manager.

Several examples of custom propagators are available:

- 1D Over-damped Langevin dynamics
- 2D Langevin dynamics
- Langevin dynamics - CA atom Elastic Network Model

Configuration File

The configuration of a WESTPA simulation is specified using a plain text file written in YAML. This file specifies, among many other things, the length of the simulation, which modules should be loaded for specifying the system, how external data should be organized on the file system, and which plugins should used. YAML is a hierarchical format and WESTPA organizes the configuration settings into blocks for each component. While below, the configuration file will be referred to as **west.cfg**, the user is free to name the configuration file something else. Most of the scripts and tools that WESTPA provides, however, require that the name of the configuration file be specified if the default name is not used.

The top most heading in *west.cfg* should be specified as::

```
---
west:
    ...
```

with all sub-section specified below it. A complete example can be found for the NaCl example: https://github.com/westpa/westpa/blob/master/lib/examples/nacl_gmx/west.cfg

In the following section, the specifications for each section of the file can be found, along with default parameters and descriptions. Required parameters are indicated as REQUIRED.:

```
---
west:
    ...
    system:
        driver: REQUIRED
        module_path: []
```

The `driver` parameter must be set to a subclass of `WESTSystem`, and given in the form *module.class*. The `module_path` parameter is appended to the system path and indicates where the class is defined.:

```
---
west:
    ...
    we:
        adjust_counts: True
        weight_split_threshold: 2.0
        weight_merge_cutoff: 1.0
```

The `we` section section specifies parameters related to the Huber and Kim resampling algorithm. WESTPA implements a variation of the method, in which setting `adust_counts` to `True` strictly enforces that the number of replicas per bin is exactly `system.bin_target_counts`. Otherwise, the number of replicas per is allowed to fluctuate as in the original implementation of the algorithm. Adjusting the counts can improve load balancing for parallel simulations. Replicas with weights greater than `weight_split_threshold` times the ideal weight per bin are tagged as candidates for splitting. Replicas with weights less than `weight_merge_cutoff` times the ideal weight per bin are candidates for merging.:

```
---
west:
    ...
    propagation:
        gen_istates: False
        block_size: 1
        save_transition_matrices: False
        max_run_wallclock: None
        max_total_iterations: None
```

- `gen_istates`: Boolean specifying whether to generate initial states from the basis states. The executable propagator defines a specific configuration block (*add internal link to other section*), and custom propagators should override the `WESTPropagator.gen_istate()` method.
- `block_size`: An integer defining how many segments should be passed to a worker at a time. When using the serial work manager, this value should be set to the maximum number of segments per iteration to avoid significant overhead incurred by the locking mechanism in the WMFutures framework. Parallel work managers might benefit from setting this value greater than one in some instances to decrease network communication load.
- `save_transition_matrices`:
- `max_run_wallclock`: A time in dd:hh:mm:ss or hh:mm:ss specifying the maximum wallclock time of a particular WESTPA run. If running on a batch queuing system, this time should be set to less than the job allocation time to ensure that WESTPA shuts down cleanly.
- `max_total_iterations`: An integer value specifying the number of iterations to run. This parameter is checked against the last completed iteration stored in the HDF5 file, not the number of iterations completed for a specific run. The default value of `None` only stops upon external termination of the code.:

```
---
west:
    ...
    data:
        west_data_file: REQUIRED
        aux_compression_threshold: 1048576
        iter_prec: 8
        datasets:
            -name: REQUIRED
             h5path:
             store: True
             load: False
             dtype:
             scaleoffset: None
             compression: None
             chunks: None
        data_refs:
            segment:
            basis_state:
            initial_state:
```

- `west_data_file`: The name of the main HDF5 data storage file for the WESTPA simulation.
- `aux_compression_threshold`: The threshold in bytes for compressing the auxiliary data in a dataset on an iteration-by-iteration basis.
- `iter_prec`: The length of the iteration index with zero-padding. For the default value, iteration 1 would be specified as iter_00000001.
- `datasets`:
- `data_refs`:
- plugins
- executable

Environmental Variables

There are a number of environmental variables that can be set by the user in order to configure a WESTPA simulation:

- WEST_ROOT: path to the base directory containing the WESTPA install
- WEST_SIM_ROOT: path to the base directory of the WESTPA simulation
- WEST_PYTHON: path to python executable to run the WESTPA simulation
- WEST_PYTHONPATH: path to any additional modules that WESTPA will require to run the simulation
- WEST_KERNPROF: path to `kernprof.py` script to perform line-by-line profiling of a WESTPA simulation (see python line_profiler). This is only required for users who need to profile specific methods in a running WESTPA simulation.

Work manager related environmental variables:

- WM_WORK_MANAGER
- WM_N_WORKERS

WESTPA makes available to any script executed by it (e.g. **runseg.sh**), a number of environmental variables that are set dynamically by the executable propagator from the running simulation.

Programs executed for an iteration

The following environment variables are passed to programs executed on a per-iteration basis, notably pre-iteration and post-iteration scripts.

Variable	Possible values	Function
WEST_CURRENT_ITER	Integer >=1	Current iteration number

Programs executed for a segment

The following environment variables are passed to programs executed on a per-segment basis, notably dynamics propagation.

Variable	Possible values	Function
WEST_CURRENT_ITER	Integer >=1	Current iteration number
WEST_CURRENT_SEG_ID	Integer >=0	Current segment ID
WEST_CURRENT_SEG_DATA_REF	String	General-purpose reference, based on current segment information, configured in west.cfg. Usually used for storage paths
WEST_CURRENT_SEG_INITPOINT_TYPE	Enumeration: SEG_INITPOINT_CONTINUES, SEG_INITPOINT_NEWTRAJ	Whether this segment continues a previous trajectory or initiates a new one.
WEST_PARENT_ID	Integer	Segment ID of parent segment. Negative for initial points.
WEST_PARENT_DATA_REF	String	General purpose reference, based on parent segment information, configured in west.cfg. Usually used for storage paths
WEST_PCOORD_RETURN	Filename	Where progress coordinate data must be stored
WEST_RAND16	Integer	16-bit random integer
WEST_RAND32	Integer	32-bit random integer
WEST_RAND64	Integer	64-bit random integer
WEST_RAND128	Integer	128-bit random integer
WEST_RANDFLOAT	Floating-point	Random number in [0,1).

Additionally for any additional datasets specified in the configuration file, WESTPA automatically provides `WEST_X_RETURN`, where `X` is the uppercase name of the dataset. For example if the configuration file contains the following:

```
data:
    ...
    datasets: # dataset storage options
      - name: energy
```

WESTPA would make `WEST_ENERGY_RETURN` available.

Programs executed for a single point

Programs used for creating initial states from basis states (`gen_istate.sh`) or extracting progress coordinates from structures (e.g. `get_pcoord.sh`) are provided the following environment variables:

Variable	Available for	Possible values	Function
WEST_STRUCT_DATA_REF	All single-point calculations	String	General-purpose reference, usually a pathname, associated with the basis/initial state.
WEST_BSTATE_ID	get_pcoord for basis state, gen_istate	Integer >= 0	Basis state ID
WEST_BSTATE_DATA_REF	get_pcoord for basis state, gen_istate	String	Basis state data reference
WEST_ISTATE_ID	get_pcoord for initial state, gen_istate	Integer >= 0	Inital state ID
WEST_ISTATE_DATA_REF	get_pcoord for initial state, gen_istate	String	Initial state data references, usually a pathname
WEST_PCOORD_RETURN	get_pcoord for basis or initial state	Pathname	Where progress coordinate data is expected to be found after execution

Plugins

WESTPA has a extensible plugin architecture that allows the user to manipulate the simulation at specified points during an iteration.

- Activating plugins in the config file
- Plugin execution order/priority

Weighted Ensemble Algorithm (Resampling)

6.1.2 Running

Overview

The **w_run** command is used to run weighted ensemble simulations *configured* <*setup*> with **w_init**.

Setting simulation limits

Running a simulation

Running on a single node

Running on multiple nodes with MPI

Running on multiple nodes with ZeroMQ

Managing data

Recovering from errors

By default, information about simulation progress is stored in **west-JOBID.log** (where JOBID refers to the job ID given by the submission engine); any errors will be logged here.

- The error "could not read pcoord from 'tempfile': progress coordinate has incorrect shape" may come about from multiple causes; it is possible that the progress coordinate length is incorrectly specified in system.py (**self.pcoord_len**), or that GROMACS (or whatever simulation package you are using) had an error during the simulation.
- The first case will be obvious by what comes after the message: (XX, YY) (where XX is non-zero), expected (ZZ, GG) (whatever is in system.py). This can be corrected by adjusting system.py.
- In the second case, the progress coordinate length is 0; this indicates that no progress coordinate data exists (null string), which implies that the simulation software did not complete successfully. By default, the simulation package (GROMACS or otherwise) terminal output is stored in a log file inside of seg_logs. Any error that occurred during the actual simulation will be logged here, and can be corrected as needed.

6.1.3 Analysis

Gauging simulation progress and convergence

Progress coordinate distribution (w_pcpdist)

w_pcpdist and plothist

Kinetics for source/sink simulations

w_fluxanl

Kinetics for arbitrary state definitions

In order to calculate rate constants, it is necessary to run three different tools:

```
- :ref:`w_assign`
- :ref:`w_kinetics`
- :ref:`w_kinavg`
```

The w_assign tool assigns trajectories to states (states which correspond to a target bin) at a sub-tau resolution. This allows w_kinetics to properly trace the trajectories and prepare the data for further analysis.

Although the bin and state definitions can be pulled from the system, it is frequently more convenient to specify custom bin boundaries and states; this eliminates the need to know what constitutes a state prior to starting the simulation. Both files must be in the YAML format, of which there are numerous examples of online. A quick example for each file follows:

```
States:
---
states:
  - label: unbound
    coords:
      - [25,0]
  - label: boun
    coords:
      - [1.5,33.0]

Bins:
---
bins:
  type: RectilinearBinMapper
  boundaries: [[0.0,1.57,25.0,10000],[0.0,33.0,10000]]
```

This system has a two dimensional progress coordinate, and two definite states, as defined by the PMF. The binning used during the simulation was significantly more complex; defining a smaller progress coordinate (in which we have three regions: bound, unbound, and in between) is simply a matter of convenience. Note that these custom bins do not change the simulation in any fashion; you can adjust state definitions and bin boundaries at will without altering the way the simulation runs.

The help definition, included by running:

```
w_assign --help
```

usually contains the most up-to-date help information, and so more information about command line options can be obtained from there. To run with the above YAML files, assuming they are named STATES and BINS, you would run the following command:

```
w_assign --states-from-file STATES --bins-from-file BINS
```

By default, this produces a .h5 file (named assign.h5); this can be changed via the command line.

The w_kinetics tool uses the information generated from w_assign to trace through trajectories and calculate flux with included color information. There are two main methods to run w_kinetics:

```
w_kinetics trace
w_kinetics matrix
```

The matrix method is still in development; at this time, trace is the recommended method.

Once the w_kinetics analysis is complete, you can check for convergence of the rate constants. WESTPA includes two tools to help you do this: w_kinavg and ploterr. First, begin by running the following command (keep in mind that w_kinavg has the same type of analysis as w_kinetics does; whatever method you chose (trace or matrix) in the w_kinetics step should be used here, as well):

```
w_kinavg trace -e cumulative
```

This instructs w_kinavg to produce a .h5 file with the cumulative rate information; by then using ploterr, you can determine whether the rates have stopped changing:

```
ploterr kinavg
```

By default, this produces a set of .pdf files, containing cumulative rate and flux information for each state-to-state transition as a function of the WESTPA iteration. Determine at which iteration the rate stops changing; then, rerun w_kinavg with the following systems:

```
w_kinavg trace --first-iter ITER
```

where ITER is the beginning of the unchanging region. This will then output information much like the following:

```
fluxes into macrostates:
unbound: mean=1.712580005863456e-02 CI=(1.596595628304422e-02, 1.808249529394858e-
↪02) * tau^-1
bound  : mean=5.944989301935855e-04 CI=(4.153556214886056e-04, 7.789568983584020e-
↪04) * tau^-1

fluxes from state to state:
unbound -> bound  : mean=5.944989301935855e-04 CI=(4.253003401668849e-04, 7.
↪720997503648696e-04) * tau^-1
bound   -> unbound: mean=1.712580005863456e-02 CI=(1.590547796439216e-02, 1.
↪808154616175579e-02) * tau^-1

rates from state to state:
unbound -> bound  : mean=9.972502012305491e-03 CI=(7.165030136921814e-03, 1.
↪313767180582492e-02) * tau^-1
bound   -> unbound: mean=1.819520888349874e-02 CI=(1.704608273094848e-02, 1.
↪926165865735958e-02) * tau^-1
```

Divide by tau to calculate your rate constant.

6.2 WEST Tools

The command line tools included with the WESTPA software package are broadly separable into two categories: **Tools for initializing a simulation** and **tools for analyzing results**.

Command function can be user defined and modified. The particular parameters of different command line tools are specified, in order of precedence, by:

- User specified command line arguments
- User defined environmental variables
- Package defaults

This page focuses on outlining the general functionality of the command line tools and providing an overview of command line arguments that are shared by multiple tools. See the *index of command-line tools* for a more comprehensive overview of each tool.

6.2.1 Overview

All tools are located in the `$WEST_ROOT/bin` directory, where the shell variable `WEST_ROOT` points to the path where the WESTPA package is located on your machine.

You may wish to set this variable automatically by adding the following to your `~/.bashrc` or `~/.profile` file:

```
export WEST_ROOT="$HOME/westpa"
```

where the path to the westpa suite is modified accordingly.

Tools for setting up and running a simulation

Use the following commands to initialize, configure, and run a weighted ensemble simulation. Command line arguments or environmental variables can be set to specify the work managers for running the simulation, where configuration data is read from, and the *HDF5* file in which results are stored.

Command	Function
w_init	Initializes simulation configuration files and environment. Always run this command before starting a new simulation.
w_bins	Set up binning, progress coordinate
w_run	Launches a simulation. Command arguments/environmental variables can be included to specify the work managers and simulation parameters
w_truncate	Truncates the weighted ensemble simulation from a given iteration.

Tools for analyzing simulation results

The following command line tools are provided for analysis after running a weighted ensemble simulation (and collecting the results in an HDF5 file).

With the exception of the plotting tool `plothist`, all analysis tools read from and write to *HDF5* type files.

Command	Function
w_assign	Assign walkers to bins and macrostates (using simulation output as input). Must be done before some other analysis tools (e.g. *w_kinetics*, *w_kinavg*)
w_trace	Trace the path of a given walker segment over a user-specified number of simulation iterations.
w_fluxanl	Calculate average probability flux into user-defined 'target' state with relevant statistics.
w_pdist	Construct a probability distribution of results (e.g. progress coordinate membership) for subsequent plotting with *plothist*.
plothist	Tool to plot output from other analysis tools (e.g. *w_pdist*).

6.2.2 General Command Line Options

The following arguments are shared by all command line tools:

```
-r config file, --rcfile config file
  Use config file as the configuration file (Default: File named west.cfg)
--quiet, --verbose, --debug
  Specify command tool output verbosity (Default: 'quiet' mode)
--version
  Print WESTPA version number and exit
-h, --help
  Output the help information for this command line tool and exit
```

A note on specifying a configuration file

A *configuration file*, which should be stored in your simulation root directory, is read by all command line tools. The *configuration file* specifies parameters for general simulation setup, as well as the *hdf5* file name where simulation data is stored and read by analysis tools.

If not specified, the **default configuration file** is assumed to be named **west.cfg**.

You can override this to use configuration file *file* by either:

- Setting the environmental variable `WESTRC` equal to *file*:

```
export WESTRC=/path/to/westrcfile
```

- Including the command line argument `-r /path/to/westrcfile`

6.2.3 Work Manager Options

Note: See *wwmgr overview* for a more detailed explanation of the work manager framework.

Work managers a used by a number of command-line tools to process more complex tasks, especially in setting up and running simulations (i.e. *w_init* and *w_run*) - in general, work managers are involved in tasks that require multiprocessing and/or tasks distributed over multiple nodes in a cluster.

Overview

The following command-line tools make use of work managers:

- *w_init*
- *w_run*

General work manager options

The following are general options used for specifying the type of work manager and number of cores:

```
--wm-work-manager work_manager
  Specify which type of work manager to use, where the possible choices for
  work_manager are: {processes, gcserial, threads, mpi, or zmq}. See the
  wwmgr overview page <wwmgr>_ for more information on the different types of
  work managers (Default: gcprocesses)
--wm-n-workers n_workers
  Specify the number of cores to use as gcn_workers, if the work manager you
  selected supports this option (work managers that do not will ignore this
  option). If using an gcmpi or zmq work manager, specify gc--wm-n-workers=0
  for a dedicated server (Default: Number of cores available on machine)
```

The `mpi` work manager is generally sufficient for most tasks that make use of multiple nodes on a cluster. The `zmq` work manager is preferable if the `mpi` work manager does not work properly on your cluster or if you prefer to have more explicit control over the distribution of communication tasks on your cluster.

ZeroMQ ('zmq') work manager

The ZeroMQ work manager offers a number of additional options (all of which are optional and have default values). All of these options focus on whether the zmq work manager is set up as a server (i.e. task distributor/ventilator) or client (task processor):

```
--wm-zmq-mode mode
  Options: {server or client}. Specify whether the ZMQ work manager on this
  node will operate as a server or a client (Default: server)

--wm-zmq-info-file info_file
  Specify the name of a temporary file to write (as a server) or read (as a
  client) socket connection endpoints (Default: server_x.json, where x is a
  unique identifier string)

--wm-zmq-task-endpoint task_endpoint
  Explicitly use task_endpoint to bind to (as server) or connect to (as
  client) for task distribution (Default: A randomly determined endpoint that
  is written or read from the specified info_file)

--wm-zmq-result-endpoint result_endpoint
  Explicitly use result_endpoint to bind to (as server) or connect to (as
  client) to distribute and collect task results (Default: A randomly
  determined endpoint that is written to or read from the specified
  info_file)

--wm-zmq-announce-endpoint announce_endpoint
  Explicitly use announce_endpoint to bind to (as server) or connect to (as
  client) to distribute central announcements (Default: A randomly determined
```

(continues on next page)

(continued from previous page)

```
  endpoint that is written to or read from the specified info_file)

--wm-zmq-heartbeat-interval interval
  If a server, send an Im alive ping to connected clients every interval
  seconds; If a client, expect to hear a server ping every approximately
  interval seconds, or else assume the server has crashed and shutdown
  (Default: 600 seconds)

--wm-zmq-task-timeout timeout
  Kill worker processes/jobs after that take longer than timeout seconds to
  complete (Default: no time limit)

--wm-zmq-client-comm-mode mode
  Use the communication mode, mode, (options: {ipc for Unix sockets, or tcp
  for TCP/IP sockets}) to communicate with worker processes (Default: ipc)
```

6.2.4 Initializing/Running Simulations

For a more complete overview of all the files necessary for setting up a simulation, see the *user guide for setting up a simulation*

6.3 WEST Work Manager

6.3.1 Introduction

WWMGR is the parallel task distribution framework originally included as part of the WEMD source. It was extracted to permit independent development, and (more importantly) independent testing. A number of different schemes can be selected at run-time for distributing work across multiple cores/nodes, as follows:

Name	Implementation	Multi-Core	Multi-Node	Appropriate For
serial	None	No	No	Testing, minimizing overhead when dynamics is inexpensive
threads	Python "threading" module	Yes	No	Dynamics propagated by external executables, large amounts of data transferred per segment
processes	Python "multiprocessing" module	Yes	No	Dynamics propagated by Python routines, modest amounts of data transferred per segment
mpi	mpi4py compiled and linked against system MPI	Yes	Yes	Distributing calculations across multiple nodes. Start with this on your cluster of choice.
zmq	ZeroMQ and PyZMQ	Yes	Yes	Distributing calculations across multiple nodes. Use this if MPI does not work properly on your cluster (particularly for spawning child processes).

6.3.2 Environment variables

For controlling task distribution

While the original WEMD work managers were controlled by command-line options and entries in wemd.cfg, the new work manager is controlled using command-line options or environment variables (much like OpenMP). These variables are as follow:

Variable	Applicable to	Default	Meaning
WM_WORK_MANAGER	(none)	processes	Use the given task distribution system: "serial", "threads", "processes", or "zmq"
WM_N_WORKERS	threads, processes, zmq	number of cores in machine	Use this number of workers. In the case of zmq, use this many workers on the current machine only (can be set independently on different nodes).
WM_ZMQ_MODE	zmq	server	Start as a server ("server") or a client ("client"). Servers coordinate a given calculation, and clients execute tasks related to that calculation.
WM_ZMQ_TASK_TIMEOUT	zmq	60	Time (in seconds) after which a worker will be considered hung, terminated, and restarted. This **must** be updated for long-running dynamics segments. Set to zero to disable hang checks entirely.
WM_ZMQ_TASK_ENDPOINT	zmq	Random port	Master distributes tasks at this address
WM_ZMQ_RESULT_ENDPOINT	zmq	Random port	Master receives task results at this address |
WM_ZMQ_ANNOUNCE_ENDPOINT	zmq	Random port	Master publishes announcements (such as "shut down now") at this address
WM_ZMQ_SERVER_INFO	zmq	`west_server_info_PID_ID.json` (where PID is a process ID and ID is a nearly random hex number)	A file describing the above endpoints can be found here (to ease cluster-wide startup)

For passing information to workers

One environment variable is made available by multi-process work managers (processes and ZMQ) to help clients configure themselves (e.g. select an appropriate GPU on a multi-GPU node):

Variable	Applicable to	Meaning
WM_PROCESS_ID	processes, zmq	Contains an integer, 0 based, identifying the process among the set of processes started on a given node.

6.3.3 The ZeroMQ work manager for clusters

The ZeroMQ ("zmq") work manager can be used for both single-machine and cluster-wide communication. Communication occurs over sockets using the ZeroMQ messaging protocol. Within nodes, Unix sockets are used for efficient communication, while between nodes, TCP sockets are used. This also minimizes the number of open sockets on the master node.

The quick and dirty guide to using this on a cluster is as follows:

```
source env.sh
export WM_WORK_MANAGER=zmq
export WM_ZMQ_COMM_MODE=tcp
export WM_ZMQ_SERVER_INFO=$WEST_SIM_ROOT/wemd_server_info.json

w_run &

# manually run w_run on each client node, as appropriate for your batch system
# e.g. qrsh -inherit for Grid Engine, or maybe just simple SSH

for host in $(cat $TMPDIR/machines | sort | uniq); do
   qrsh -inherit -V $host $PWD/node-ltc1.sh &
done
```

6.4 WEST Extensions

6.4.1 Post-Analysis Reweighting

6.4.2 String Method

6.4.3 Weighted Ensemble Equilibrium Dynamics

6.4.4 Weighted Ensemble Steady State

6.5 Command Line Tool Index

6.5.1 w_assign

`w_assign` uses simulation output to assign walkers to user-specified bins and macrostates. These assignments are required for some other simulation tools, namely `w_kinetics` and `w_kinavg`.

`w_assign` supports parallelization (see general work manager options for more on command line options to specify a work manager).

Overview

Usage:

```
w_assign [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [-W WEST_H5FILE] [-o OUTPUT]
                [--bins-from-system | --bins-from-expr BINS_FROM_EXPR | --bins-from-
↪function BINS_FROM_FUNCTION]
                [-p MODULE.FUNCTION]
                [--states STATEDEF [STATEDEF ...] | --states-from-file STATEFILE | --
↪states-from-function STATEFUNC]
                [--wm-work-manager WORK_MANAGER] [--wm-n-workers N_WORKERS]
                [--wm-zmq-mode MODE] [--wm-zmq-info INFO_FILE]
                [--wm-zmq-task-endpoint TASK_ENDPOINT]
                [--wm-zmq-result-endpoint RESULT_ENDPOINT]
                [--wm-zmq-announce-endpoint ANNOUNCE_ENDPOINT]
                [--wm-zmq-listen-endpoint ANNOUNCE_ENDPOINT]
                [--wm-zmq-heartbeat-interval INTERVAL]
                [--wm-zmq-task-timeout TIMEOUT]
                [--wm-zmq-client-comm-mode MODE]
```

Command-Line Options

See the general command-line tool reference for more information on the general options.

Input/output Options

```
-W, --west-data /path/to/file

    Read simulation result data from file *file*. (**Default:** The
    *hdf5* file specified in the configuration file, by default
    **west.h5**)

-o, --output /path/to/file
    Write assignment results to file *outfile*. (**Default:** *hdf5*
    file **assign.h5**)
```

Binning Options

Specify how binning is to be assigned to the dataset.:

```
--bins-from-system
  Use binning scheme specified by the system driver; system driver can be
  found in the west configuration file, by default named **west.cfg**
  (**Default binning**)

--bins-from-expr bin_expr
  Use binning scheme specified in *``bin_expr``*, which takes the form a
  Python list of lists, where each inner list corresponds to the binning a
  given dimension. (for example, "[[0,1,2,4,inf],[-inf,0,inf]]" specifies bin
  boundaries for two dimensional progress coordinate. Note that this option
  accepts the special symbol 'inf' for floating point infinity
```

(continues on next page)

(continued from previous page)

```
--bins-from-function bin_func
  Bins specified by calling an external function *``bin_func``*.
  *``bin_func``* should be formatted as '[PATH:]module.function', where the
  function 'function' in module 'module' will be used
```

Macrostate Options

You can optionally specify how to assign user-defined macrostates. Note that macrostates must be assigned for subsequent analysis tools, namely `w_kinetics` and `w_kinavg`.:

```
--states statedef [statedef ...]
  Specify a macrostate for a single bin as *``statedef``*, formatted
  as a coordinate tuple where each coordinate specifies the bin to
  which it belongs, for instance:
  '[1.0, 2.0]' assigns a macrostate corresponding to the bin that
  contains the (two-dimensional) progress coordinates 1.0 and 2.0.
  Note that a macrostate label can optionally by specified, for
  instance: 'bound:[1.0, 2.0]' assigns the corresponding bin
  containing the given coordinates the macrostate named 'bound'. Note
  that multiple assignments can be specified with this command, but
  only one macrostate per bin is possible - if you wish to specify
  multiple bins in a single macrostate, use the
  *``--states-from-file``* option.

--states-from-file statefile
  Read macrostate assignments from *yaml* file *``statefile``*. This
  option allows you to assign multiple bins to a single macrostate.
  The following example shows the contents of *``statefile``* that
  specify two macrostates, bound and unbound, over multiple bins with
  a two-dimensional progress coordinate:

---
states:
  - label: unbound
    coords:
      - [9.0, 1.0]
      - [9.0, 2.0]
  - label: bound
    coords:
      - [0.1, 0.0]
```

Specifying Progress Coordinate

By default, progress coordinate information for each iteration is taken from *pcoord* dataset in the specified input file (which, by default is *west.h5*). Optionally, you can specify a function to construct the progress coordinate for each iteration - this may be useful to consolidate data from several sources or otherwise preprocess the progress coordinate data.:

```
--construct-pcoord module.function, -p module.function
  Use the function *module.function* to construct the progress
  coordinate for each iteration. This will be called once per
  iteration as *function(n_iter, iter_group)* and should return an
  array indexable as [seg_id][timepoint][dimension]. The
```

(continues on next page)

(continued from previous page)

```
  **default** function returns the 'pcoord' dataset for that iteration
  (i.e. the function executes return iter_group['pcoord'][...])
```

Examples

6.5.2 w_bins

`w_bins` deals with binning modification and statistics

Overview

Usage:

```
$WEST_ROOT/bin/w_bins [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              [-W WEST_H5FILE]
              {info,rebin} ...
```

Display information and statistics about binning in a WEST simulation, or modify the binning for the current iteration of a WEST simulation.

Command-Line Options

See the general command-line tool reference for more information on the general options.

Options Under 'info'

Usage:

```
$WEST_ROOT/bin/w_bins info [-h] [-n N_ITER] [--detail]
                  [--bins-from-system | --bins-from-expr BINS_FROM_EXPR | --bins-from-
↪function BINS_FROM_FUNCTION | --bins-from-file]
```

Positional options:

```
info
  Display information about binning.
```

Options for 'info':

```
-n N_ITER, --n-iter N_ITER
  Consider initial points of segment N_ITER (default: current
  iteration).

--detail
  Display detailed per-bin information in addition to summary
  information.
```

Binning options for 'info':

```
--bins-from-system
  Bins are constructed by the system driver specified in the WEST
  configuration file (default where stored bin definitions not
  available).

--bins-from-expr BINS_FROM_EXPR, --binbounds BINS_FROM_EXPR
  Construct bins on a rectilinear grid according to the given BINEXPR.
  This must be a list of lists of bin boundaries (one list of bin
  boundaries for each dimension of the progress coordinate), formatted
  as a Python expression. E.g. "[[0,1,2,4,inf],[-inf,0,inf]]". The
  numpy module and the special symbol "inf" (for floating-point
  infinity) are available for use within BINEXPR.

--bins-from-function BINS_FROM_FUNCTION, --binfunc BINS_FROM_FUNCTION
  Supply an external function which, when called, returns a properly
  constructed bin mapper which will then be used for bin assignments.
  This should be formatted as "[PATH:]MODULE.FUNC", where the function
  FUNC in module MODULE will be used; the optional PATH will be
  prepended to the module search path when loading MODULE.

--bins-from-file
  Load bin specification from the data file being examined (default
  where stored bin definitions available).
```

Options Under 'rebin'

Usage:

```
$WEST_ROOT/bin/w_bins rebin [-h] [--confirm] [--detail]
                    [--bins-from-system | --bins-from-expr BINS_FROM_EXPR | --bins-
↪from-function BINS_FROM_FUNCTION]
                    [--target-counts TARGET_COUNTS | --target-counts-from FILENAME]
```

Positional option:

```
rebin
  Rebuild current iteration with new binning.
```

Options for 'rebin':

```
--confirm
  Commit the revised iteration to HDF5; without this option, the
  effects of the new binning are only calculated and printed.

--detail
  Display detailed per-bin information in addition to summary
  information.
```

Binning options for 'rebin';

Same as the binning options for 'info'.

Bin target count options for 'rebin';:

```
--target-counts TARGET_COUNTS
  Use TARGET_COUNTS instead of stored or system driver target counts.
```

(continues on next page)

(continued from previous page)

```
  TARGET_COUNTS is a comma-separated list of integers. As a special
  case, a single integer is acceptable, in which case the same target
  count is used for all bins.

--target-counts-from FILENAME
  Read target counts from the text file FILENAME instead of using
  stored or system driver target counts. FILENAME must contain a list
  of integers, separated by arbitrary whitespace (including newlines).
```

Input Options

```
-W WEST_H5FILE, --west_data WEST_H5FILE
  Take WEST data from WEST_H5FILE (default: read from the HDF5 file
  specified in west.cfg).
```

Examples

(TODO: Write up an example)

6.5.3 w_crawl

6.5.4 w_eddist

6.5.5 w_fluxanl

`w_fluxanl` calculates the probability flux of a weighted ensemble simulation based on a pre-defined target state. Also calculates confidence interval of average flux. Monte Carlo bootstrapping techniques are used to account for autocorrelation between fluxes and/or errors that are not normally distributed.

Overview

usage:

```
$WEST_ROOT/bin/w_fluxanl [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                         [-W WEST_H5FILE] [-o OUTPUT]
                         [--first-iter N_ITER] [--last-iter N_ITER]
                         [-a ALPHA] [--autocorrel-alpha ACALPHA] [-N NSETS] [--
↪evol] [--evol-step ESTEP]
```

Note: All command line arguments are optional for `w_fluxanl`.

Command-Line Options

See the general command-line tool reference for more information on the general options.

Input/output options

These arguments allow the user to specify where to read input simulation result data and where to output calculated progress coordinate probability distribution data.

Both input and output files are *hdf5* format.:

```
-W, --west-data file
  Read simulation result data from file *file*. (**Default:** The
  *hdf5* file specified in the configuration file)

-o, --output file
  Store this tool's output in *file*. (**Default:** The *hdf5* file
  **pcpdist.h5**)
```

Iteration range options

Specify the range of iterations over which to construct the progress coordinate probability distribution.:

```
--first-iter n_iter
  Construct probability distribution starting with iteration *n_iter*
  (**Default:** 1)

--last-iter n_iter
  Construct probability distribution's time evolution up to (and
  including) iteration *n_iter* (**Default:** Last completed
  iteration)
```

Confidence interval and bootstrapping options

Specify alpha values of constructed confidence intervals.:

```
-a alpha
  Calculate a (1 - *alpha*) confidence interval for the mean flux
  (**Default:** 0.05)

--autocorrel-alpha ACalpha
  Identify autocorrelation of fluxes at *ACalpha* significance level.
  Note: Specifying an *ACalpha* level that is too small may result in
  failure to find autocorrelation in noisy flux signals (**Default:**
  Same level as *alpha*)

-N n_sets, --nsets n_sets
  Use *n_sets* samples for bootstrapping (**Default:** Chosen based
  on *alpha*)

--evol
  Calculate the time evolution of flux confidence intervals
  (**Warning:** computationally expensive calculation)
```

(continues on next page)

(continued from previous page)

```
--evol-step estep
  (if ``'--evol'`` specified) Calculate the time evolution of flux
  confidence intervals for every *estep* iterations (**Default:** 1)
```

Examples

Calculate the time evolution flux every 5 iterations:

```
$WEST_ROOT/bin/w_fluxanl --evol --evol-step 5
```

Calculate mean flux confidence intervals at 0.01 signicance level and calculate autocorrelations at 0.05 significance:

```
$WEST_ROOT/bin/w_fluxanl --alpha 0.01 --autocorrel-alpha 0.05
```

Calculate the mean flux confidence intervals using a custom bootstrap sample size of 500:

```
$WEST_ROOT/bin/w_fluxanl --n-sets 500
```

6.5.6 w_fork

6.5.7 w_init

`w_init` initializes the weighted ensemble simulation, creates the main HDF5 file and prepares the first iteration.

Overview

Usage:

```
$WEST_ROOT/bin/w_init [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              [--force] [--bstate-file BSTATE_FILE] [--bstate BSTATES]
              [--tstate-file TSTATE_FILE] [--tstate TSTATES]
              [--segs-per-state N] [--no-we] [--wm-work-manager WORK_MANAGER]
              [--wm-n-workers N_WORKERS] [--wm-zmq-mode MODE]
              [--wm-zmq-info INFO_FILE] [--wm-zmq-task-endpoint TASK_ENDPOINT]
              [--wm-zmq-result-endpoint RESULT_ENDPOINT]
              [--wm-zmq-announce-endpoint ANNOUNCE_ENDPOINT]
              [--wm-zmq-heartbeat-interval INTERVAL]
              [--wm-zmq-task-timeout TIMEOUT] [--wm-zmq-client-comm-mode MODE]
```

Initialize a new WEST simulation, creating the WEST HDF5 file and preparing the first iteration's segments. Initial states are generated from one or more "basis states" which are specified either in a file specified with `--bstates-from`, or by one or more `--bstate` arguments. If neither `--bstates-from` nor at least one `--bstate` argument is provided, then a default basis state of probability one identified by the state ID zero and label "basis" will be created (a warning will be printed in this case, to remind you of this behavior, in case it is not what you wanted). Target states for (non- equilibrium) steady-state simulations are specified either in a file specified with `--tstates-from`, or by one or more `--tstate` arguments. If neither `--tstates-from` nor at least one `--tstate` argument is provided, then an equilibrium simulation (without any sinks) will be performed.

Command-Line Options

See the general command-line tool reference for more information on the general options.

State Options

```
--force
  Overwrites any existing simulation data

--bstate BSTATES
  Add the given basis state (specified as a string
  'label,probability[,auxref]') to the list of basis states (after
  those specified in --bstates-from, if any). This argument may be
  specified more than once, in which case the given states are
  appended in the order they are given on the command line.

--bstate-file BSTATE_FILE, --bstates-from BSTATE_FILE
  Read basis state names, probabilities, and (optionally) data
  references from BSTATE_FILE.

--tstate TSTATES
  Add the given target state (specified as a string
  'label,pcoord0[,pcoord1[,...]]') to the list of target states (after
  those specified in the file given by --tstates-from, if any). This
  argument may be specified more than once, in which case the given
  states are appended in the order they appear on the command line.

--tstate-file TSTATE_FILE, --tstates-from TSTATE_FILE
  Read target state names and representative progress coordinates from
  TSTATE_FILE.

--segs-per-state N
  Initialize N segments from each basis state (default: 1).

--no-we, --shotgun
  Do not run the weighted ensemble bin/split/merge algorithm on
  newly-created segments.
```

Examples

(TODO: write 3 examples; Setting up the basis states, explanation of bstates and istates. Setting up an equilibrium simulation, w/o target(s) for recycling. Setting up a simulation with one/multiple target states.)

6.5.8 w_kinavg

6.5.9 w_kinetics

6.5.10 w_ntop

6.5.11 w_pdist

`w_pcpdist` constructs and calculates the progress coordinate probability distribution's evolution over a user-specified number of simulation iterations. `w_pcpdist` supports progress coordinates with dimensionality 1.

The resulting distribution can be viewed with the *plothist* tool.

Overview

Usage:

```
$WEST_ROOT/bin/w_pdist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                       [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER]
                       [-b BINEXPR] [-o OUTPUT]
                       [--construct-dataset CONSTRUCT_DATASET | --
↪dsspecs DSSPEC [DSSPEC ...]]
                       [--serial | --parallel | --work-manager WORK_MANAGER]
                       [--n-workers N_WORKERS] [--zmq-mode MODE]
                       [--zmq-info INFO_FILE] [--zmq-task-endpoint TASK_ENDPOINT]
                       [--zmq-result-endpoint RESULT_ENDPOINT]
                       [--zmq-announce-endpoint ANNOUNCE_ENDPOINT]
                       [--zmq-listen-endpoint ANNOUNCE_ENDPOINT]
                       [--zmq-heartbeat-interval INTERVAL]
                       [--zmq-task-timeout TIMEOUT] [--zmq-client-comm-mode MODE]
```

Note: This tool supports parallelization, which may be more efficient for especially large datasets.

Command-Line Options

See the general command-line tool reference for more information on the general options.

Input/output options

These arguments allow the user to specify where to read input simulation result data and where to output calculated progress coordinate probability distribution data.

Both input and output files are *hdf5* format:

```
-W, --WEST_H5FILE file
  Read simulation result data from file *file*. (**Default:** The
  *hdf5* file specified in the configuration file (default config file
  is *west.h5*))

-o, --output file
  Store this tool's output in *file*. (**Default:** The *hdf5* file
  **pcpdist.h5**)
```

Iteration range options

Specify the range of iterations over which to construct the progress coordinate probability distribution.:

```
--first-iter n_iter
  Construct probability distribution starting with iteration *n_iter*
  (**Default:** 1)

--last-iter n_iter
  Construct probability distribution's time evolution up to (and
```

(continues on next page)

(continued from previous page)

```
  including) iteration *n_iter* (**Default:** Last completed
  iteration)
```

Probability distribution binning options

Specify the number of bins to use when constructing the progress coordinate probability distribution. If using a multidimensional progress coordinate, different binning schemes can be used for the probability distribution for each progress coordinate.:

```
-b binexpr
  *binexpr* specifies the number and formatting of the bins. Its
  format can be as follows:

      1. an integer, in which case all distributions have that many
      equal sized bins
      2. a python-style list of integers, of length corresponding to
      the number of dimensions of the progress coordinate, in which
      case each progress coordinate's probability distribution has the
      corresponding number of bins
      3. a python-style list of lists of scalars, where the list at
      each index corresponds to each dimension of the progress
      coordinate and specifies specific bin boundaries for that
      progress coordinate's probability distribution.

  (**Default:** 100 bins for all progress coordinates)
```

Examples

Assuming simulation results are stored in *west.h5* (which is specified in the configuration file named *west.cfg*), for a simulation with a 1-dimensional progress coordinate:

Calculate a probability distribution histogram using all default options (output file: *pdist.h5*; histogram binning: 100 equal sized bins; probability distribution over the lowest reached progress coordinate to the largest; work is parallelized over all available local cores using the 'processes' work manager):

```
$WEST_ROOT/bin/w_pdist
```

Same as above, except using the serial work manager (which may be more efficient for smaller datasets):

```
$WEST_ROOT/bin/w_pdist --serial
```

6.5.12 w_run

`w_run` starts or continues a weighted ensemble simualtion.

Overview

Usage:

```
$WEST_ROOT/bin/w_run [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
             [--oneseg ] [--wm-work-manager WORK_MANAGER]
             [--wm-n-workers N_WORKERS] [--wm-zmq-mode MODE]
             [--wm-zmq-info INFO_FILE] [--wm-zmq-task-endpoint TASK_ENDPOINT]
             [--wm-zmq-result-endpoint RESULT_ENDPOINT]
             [--wm-zmq-announce-endpoint ANNOUNCE_ENDPOINT]
             [--wm-zmq-heartbeat-interval INTERVAL]
             [--wm-zmq-task-timeout TIMEOUT] [--wm-zmq-client-comm-mode MODE]
```

Command-Line Options

See the *command-line tool index* for more information on the general options.

Segment Options

::

--oneseg Only propagate one segment (useful for debugging propagators)

Example

A simple example for using w_run (mostly taken from odld example that is available in the main WESTPA distribution):

```
$WEST_ROOT/bin/w_run &> west.log
```

This commands starts up a serial weighted ensemble run and pipes the results into the west.log file. As a side note `--debug` option is very useful for debugging the code if something goes wrong.

6.5.13 w_select

6.5.14 w_stateprobs

6.5.15 w_states

6.5.16 w_succ

6.5.17 w_trace

6.5.18 w_truncate

`w_truncate` removes all iterations after a certain point

Overview

Usage:

```
$WEST_ROOT/bin/w_truncate [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                  [-n N_ITER]
```

Remove all iterations after a certain point in a

Command-Line Options

See the *command-line tool index* <*command_line_tool_index*> for more information on the general options.

Iteration Options

```
-n N_ITER, --iter N_ITER
  Truncate this iteration and those following.
```

Examples

(TODO: Write up an example)

6.5.19 ploterr

6.5.20 plothist

Use the `plothist` tool to plot the results of *w_pdist*. This tool uses an *hdf5* file as its input (i.e. the output of another analysis tool), and outputs a *pdf* image.

The `plothist` tool operates in one of three (mutually exclusive) plotting modes:

- ``**evolution**``: Plots the relevant data as a time evolution over specified number of simulation iterations
- ``**average**``: Plots the relevant data as a time average over a specified number of iterations
- ``**instant**``: Plots the relevant data for a single specified iteration

Overview

The basic usage, independent of plotting mode, is as follows:

usage:

```
$WEST_ROOT/bin/
plothist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
``            {instant,average,evolution} input ...``
```

Note that the user must specify a plotting mode (i.e. ‘`instant`‘, ‘`average`‘, or ‘`evolution`‘) and an input file, `input`.

Therefore, this tool is always called as:

```
$WEST_ROOT/bin/plothist mode input_file [``other`` ``options``]
```

‘instant‘ mode

usage:

```
$WEST_ROOT/bin/plothist instant [-h] input [-o PLOT_OUTPUT]
``                  [–hdf5-output HDF5_OUTPUT] [–text-output TEXT_OUTPUT]``
``                  [–title TITLE] [–range RANGE] [–linear | –energy | –log10]``
``                  [–iter N_ITER] ``
``                  [DIMENSION] [ADDTLDIM]``
```

‘average‘ mode

usage:

```
$WEST_ROOT/bin/plothist average [-h] input [-o PLOT_OUTPUT]
``                  [–hdf5-output HDF5_OUTPUT] [–text-output TEXT_OUTPUT]``
``                  [–title TITLE] [–range RANGE] [–linear | –energy | –log10]``
``                  [–first-iter N_ITER] [–last-iter N_ITER]                 ``
``                  [DIMENSION] [ADDTLDIM]``
```

‘evolution‘ mode

usage:

```
$WEST_ROOT/bin/plothist evolution [-h] input [-o PLOT_OUTPUT]
``                   [–hdf5-output HDF5_OUTPUT]``
``                   [–title TITLE] [–range RANGE] [–linear | –energy | –log10]``
``                   [–first-iter N_ITER] [–last-iter N_ITER]``
``                   [–step-iter STEP]                       ``
``                   [DIMENSION]``
```

Command-Line Options

See the *command-line tool index* for more information on the general options.

Unless specified (as a **Note** in the command-line option description), the command-line options below are shared for all three plotting modes

Input/output options

No matter the mode, an input *hdf5* file must be specified. There are three possible outputs that are mode or user-specified: A text file, an *hdf5* file, and a pdf image.

Specifying input file

``input`` Specify the input *hdf5* file ‘’`input`. This is the output file from a previous analysis tool (e.g. ‘pcpdist.h5’)

Output plot pdf file

``-o ‘’plot_output”, –plot_output ‘’plot_output”`` Specify the name of the pdf plot image output (**Default:** ‘hist.pdf’). **Note:** You can suppress plotting entirely by specifying an empty string as *plot_output* (i.e. `-o ''` or `--plot_output ''`)

Additional output options

Note: `plothist` provides additional, optional arguments to output the data points used to construct the plot:

``–hdf5-output ‘’hdf5_output”`` Output plot data *hdf5* file `'hdf5_output'` (**Default:** No *hdf5* output file)

``–text-output ‘’text_output”`` Output plot data as a text file named `'text_output'` (**Default:** No text output file) **Note:** This option is only available for 1 dimensional histogram plots (that is, `'average'` and `'instant'` modes only)

Plotting options

The following options allow the user to specify a plot title, the type of plot (i.e. energy or probability distribution), whether to apply a log transformation to the data, and the range of data values to include.

``–title ‘’title”`` Optionally specify a title, ``*title*``, for the plot (**Default:** No title)

``–range ‘’<nowiki>’</nowiki>LB, UB<nowiki>’</nowiki>”`` Optionally specify the data range to be plotted as “`LB, UB`” (e.g. `' --range "-1, 10" '` - note that the quotation marks are necessary if specifying a negative bound). For 1 dimensional histograms, the range affects the y axis. For 2 dimensional plots (e.g. evolution plot with 1 dimensional progress coordinate), it corresponds to the range of the color bar

Mutually exclusive plotting options

The following three options determine how the plotted data is represented (**Default:** `'--energy'`)

``**–energy** `` Plots the probability distribution on an inverted natural log scale (i.e. -ln[P(x)]), corresponding to the free energy (**Default**)

``**–linear** `` Plots the probability distribution function as a linear scale

``**–log10** `` Plots the (base-10) logarithm of the probability distribution

Iteration selection options

Depending on plotting mode, you can select either a range or a single iteration to plot.

``**'instant'**`` mode only:

``**–iter ''n_iter"** `` Plot the distribution for iteration `''n_iter''` (**Default:** Last completed iteration)

``**'average'**`` and ``**'evolution'**`` modes only:

``**–first-iter ''first_iter"** `` Begin averaging or plotting at iteration ``*first_iter*`` (**Default:** 1)

``**–last-iter ''last_iter"** `` Average or plot up to and including ``*last_iter*`` (**Default:** Last completed iteration)

``**'evolution'**`` mode only:

``**–iter_step ''n_step"** `` Average every ``*n_step*`` iterations together when plotting in `'evolution'` mode (**Default:** 1 - i.e. plot each iteration)

Specifying progress coordinate dimension

For progress coordinates with dimensions greater than 1, you can specify the dimension of the progress coordinate to use, the of progress coordinate values to include, and the progress coordinate axis label with a single positional argument:

``**dimension** `` Specify `'dimension'` as 'int[:[LB,UB]:label]', where 'int' specifies the dimension (starting at 0), and, optionally, 'LB,UB' specifies the lower and upper range bounds, and/or 'label' specifies the axis label (**Default:** `int` = 0, full range, default label is 'dimension `int`'; e.g 'dimension 0')

For `'average'` and `'instant'` modes, you can plot two dimensions at once using a color map if this positional argument is specified:

``**addtl_dimension** `` Specify the other dimension to include as `'addtl_dimension'`

Examples

These examples assume the input file is created using w_pcpdist and is named 'pcpdist.h5'

Basic plotting

Plot the energy (-ln(P(x))) for the last iteration

```
$WEST_ROOT/bin/plothist instant pcpdist.h5
```

Plot the evolution of the log10 of the probability distribution over all iterations

``$WEST_ROOT/bin/plothist evolution pcpdist.h5 –log10 ``

Plot the average linear probability distribution over all iterations

```
$WEST_ROOT/bin/plothist average pcpdist.h5 --linear
```

Specifying progress coordinate

Plot the average probability distribution as the energy, label the x-axis 'pcoord', over the entire range of the progress coordinate

```
$WEST_ROOT/bin/plothist average pcpdist.h5 0::pcoord
```

Same as above, but only plot the energies for with progress coordinate between 0 and 10

```
$WEST_ROOT/bin/plothist average pcpdist.h5 '0:0,10:pcoord'
```

(Note: the quotes are needed if specifying a range that includes a negative bound)

(For a simulation that uses at least 2 progress coordinates) plot the probability distribution for the 5th iteration, representing the first two progress coordinates as a heatmap

```
$WEST_ROOT/bin/plothist instant pcpdist.h5 0 1 --iter 5 --linear
```

6.6 HDF5 File Schema

WESTPA stores all of its simulation data in the cross-platform, self-describing HDF5 file format. This file format can be read and written by a variety of languages and toolkits, including C/C++, Fortran, Python, Java, and Matlab so that analysis of weighted ensemble simulations is not tied to using the WESTPA framework. HDF5 files are organized like a filesystem, where arbitrarily-nested groups (i.e. directories) are used to organize datasets (i.e. files). The excellent HDFView program may be used to explore WEST data files.

The canonical file format reference for a given version of the WEST code is described in src/west/data_manager.py.

6.6.1 Overall structure

```
/
    #ibstates/
        index
        naming
            bstate_index
            bstate_pcoord
            istate_index
            istate_pcoord
    #tstates/
        index
    bin_topologies/
        index
```

(continues on next page)

(continued from previous page)

```
        pickles
    iterations/
        iter_XXXXXXXX/\|iter_XXXXXXXX/
            auxdata/
            bin_target_counts
            ibstates/
                bstate_index
                bstate_pcoord
                istate_index
                istate_pcoord
            pcoord
            seg_index
            wtgraph
        ...
    summary
```

6.6.2 The root group (/)

The root of the WEST HDF5 file contains the following entries (where a trailing "/" denotes a group):

Name	Type	Description
ibstates/	Group	Initial and basis states for this simulation
tstates/	Group	Target (recycling) states for this simulation; may be empty
bin_topologies/	Group	Data pertaining to the binning scheme used in each iteration
iterations/	Group	Iteration data
summary	Dataset (1-dimensional, compound)	Summary data by iteration

The iteration summary table (/summary)

Field	Description
n_particles	the total number of walkers in this iteration
norm	total probability, for stability monitoring
min_bin_prob	smallest probability contained in a bin
max_bin_prob	largest probability contained in a bin
min_seg_prob	smallest probability carried by a walker
max_seg_prob	largest probability carried by a walker
cputime	total CPU time (in seconds) spent on propagation for this iteration
walltime	total wallclock time (in seconds) spent on this iteration
binhash	a hex string identifying the binning used in this iteration

6.6.3 Per iteration data (/iterations/iter_XXXXXXXX)

Data for each iteration is stored in its own group, named according to the iteration number and zero-padded out to 8 digits, as in `/iterations/iter_00000001` for iteration 1. This is done solely for convenience in dealing with the data in external utilities that sort output by group name lexicographically. The field width is in fact configurable via the `iter_prec` configuration entry under `data` section of the WESTPA configuration file.

The HDF5 group for each iteration contains the following elements:

Name	Type	Description
auxdata/	Group	All user-defined auxiliary data0 sets
bin_target_counts	Dataset (1-dimensional)	The per-bin target count for the iteration
ibstates/	Group	Initial and basis state data for the iteration
pcoord	Dataset (3-dimensional)	Progress coordinate data for the iteration stored as a (num of segments, pcoord_len, pcoord_ndim) array
seg_index	Dataset (1-dimensional, compound)	Summary data for each segment
wtgraph	Dataset (1-dimensional)	

The segment summary table (/iterations/iter_XXXXXXXX/seg_index)

Field	Description
weight	Segment weight
parent_id	Index of parent
wtg_n_parents	
wtg_offset	
cputime	Total cpu time required to run the segment
walltime	Total walltime required to run the segment
endpoint_type	
status	

6.6.4 Bin Topologies group (/bin_topologies)

Bin topologies used during a WE simulation are stored as a unique hash identifier and a serialized `BinMapper` object in python pickle format. This group contains two datasets:

- `index`: Compound array containing the bin hash and pickle length
- `pickle`: The pickled `BinMapper` objects for each unique mapper stored in a (num unique mappers, max pickled size) array

CHAPTER

SEVEN

FOR DEVELOPERS

7.1 Overview

7.2 Style Guide

7.2.1 Preface

The WESTPA documentation should help the user to understand how WESTPA works and how to use it. To aid in effective communication, a number of guidelines appear below.

When writing in the WESTPA documentation, please be:

- Correct
- Clear
- Consistent
- Concise

Articles in this documentation should follow the guidelines on this page. However, there may be cases when following these guidelines will make an article confusing: when in doubt, use your best judgment and ask for the opinions of those around you.

7.2.2 Style and Usage

Acronyms and abbreviations

- Software documentation often involves extensive use of acronyms and abbreviations.

  Acronym: A word formed from the initial letter or letters of each or most of the parts of a compound term

  Abbreviation: A shortened form of a written word or name that is used in place of the full word or name

- Define non-standard acronyms and abbreviations on their first use by using the full-length term, followed by the acronym or abbreviation in parentheses.

  A potential of mean force (PMF) diagram may aid the user in visuallizing the energy landscape of the simulation.

- Only use acronyms and abbreviations when they make an idea more clear than spelling out the full term. Consider clarity from the point of view of a new user who is intelligent but may have little experience with computers.

Correct: The WESTPA wiki supports HyperText Markup Language (HTML). For example, the user may use HTML tags to give text special formatting. However, be sure to test that the HTML tag gives the desired effect by previewing edits before saving.

Avoid: The WESTPA wiki supports HyperText Markup Language. For example, the user may use HyperText Markup Language tags to give text special formatting. However, be sure to test that the HyperText Markup Language tag gives the desired effect by previewing edits before saving.

Avoid: For each iter, make sure to return the pcoord and any auxdata.

- Use all capital letters for abbreviating file types. File extensions should be lowercase.

  HDF5, PNG, MP4, GRO, XTC

  west.h5, bound.png, unfolding.mp4, protein.gro, segment.xtc

- Provide pronunciations for acronyms that may be difficult to sound out.
- Do not use periods in acronyms and abbreviations except where it is customary:

  Correct: HTML, U.S.

  Avoid: H.T.M.L., US

Capitalization

- Capitalize at the beginning of each sentence.
- Do not capitalize after a semicolon.
- Do not capitalize after a colon, unless multiple sentences follow the colon.
- In this case, capitalize each sentence.
- Preserve the capitalization of computer language elements (commands,
- utilities, variables, modules, classes, and arguments).
- Capitilize generic Python variables according to the
- PEP 0008 Python Style Guide. For example, generic class names should follow the *CapWords* convention, such as `GenericClass`.

Contractions

- Do not use contractions. Contractions are a shortened version of word characterized by the omission of internal letters.

  Avoid: can't, don't, shouldn't

- Possessive nouns are not contractions. Use possessive nouns freely.

Internationalization

- Use short sentences (less than 25 words). Although we do not maintain WESTPA documentation in languages other than English, some users may use automatic translation programs. These programs function best with short sentences.
- Do not use technical terms where a common term would be equally or more clear.
- Use multiple simple sentences in place of a single complicated sentence.

Italics

- Use italics (surround the word with * * on each side) to highlight words that are not part of a sentence's normal grammer.

  Correct: The word *istates* refers to the initial states that WESTPA uses to begin trajectories.

Non-English words

- Avoid Latin words and abbreviations.

  Avoid: etc., et cetera, e.g., i.e.

Specially formatted characters

- Never begin a sentence with a specially formatted character. This includes abbreviations, variable names, and anything else this guide instructs to use with special tags. Sentences may begin with *WESTPA*.

  Correct: The program `ls` allows the user to see the contents of a directory.

  Avoid: `ls` allows the user to see the contents of a directory.
- Use the word *and* rather than an `&` ampersand .
- When a special character has a unique meaning to a program, first use the character surrounded by `` tags and then spell it out.

  Correct: Append an `&` ampersand to a command to let it run in the background.

  Avoid: Append an "&" to a command... Append an `&` to a command... Append an ampersand to a command...
- There are many names for the # hash mark, including hash tag, number sign, pound sign, and octothorpe. Refer to this symbol as a "hash mark".

Subject

- Refer to the end WESTPA user as *the user* in software documentation.

  Correct: The user should use the `processes` work manager to run segments in parallel on a single node.
- Refer to the end WESTPA user as *you* in tutorials (you is the implied subject of commands). It is also acceptable to use personal pronouns such as *we* and *our*. Be consistent within the tutorial.

  Correct: You should have two files in this directory, named `system.py` and `west.cfg`.

Tense

- Use *should* to specify proper usage.

  Correct: The user should run `w_truncate -n <var>iter</var>` to remove iterations after and including iter from the HDF5 file specified in the WESTPA configuration file.

- Use *will* to specify expected results and output.

  Correct: WESTPA will create a HDF5 file when the user runs `w_init`.

Voice

- Use active voice. Passive voice can obscure a sentence and add unnecessary words.

  Correct: WESTPA will return an error if the sum of the weights of segments does not equal one.

  Avoid: An error will be returned if the sum of the weights of segments does not equal one.

Weighted ensemble

- Refer to weighted ensemble in all lowercase, unless at the beginning of a sentence. Do not hyphenate.

  Correct: WESTPA is an implementation of the weighted ensemble algorithm.

  Avoid: WESTPA is an implementation of the weighted-ensemble algorithm.

  Avoid: WESTPA is an implementation of the Weighted Ensemble algorithm.

WESTPA

- Refer to WESTPA in all capitals. Do not use bold, italics, or other special formatting except when another guideline from this style guide applies.

  Correct: Install the WESTPA software package.

- The word *WESTPA* may refer to the software package or a entity of running software.

  Correct: WESTPA includes a number of analysis utilities.

  Correct: WESTPA will return an error if the user does not supply a configuration file.

7.2.3 Computer Language Elements

Classes, modules, and libraries

- Display class names in fixed-width font using the `` tag.

  Correct: `WESTPropagator`

  Correct: The `numpy` library provides access to various low-level mathematical and scientific calculation routines.

- Generic class names should be relevant to the properties of the class; do not use *foo* or *bar*

```
class UserDefinedBinMapper(RectilinearBinMapper)
```

Methods and commands

- Refer to a method by its name without parentheses, and without prepending the name of its class. Display methods in fixed-width font using the `` tag.

  Correct: the `arange` method of the `numpy` library

  Avoid: the `arange()` method of the `numpy` library

  Avoid: the `numpy.arange` method

- When referring to the arguments that a method expects, mention the method without arguments first, and then use the method's name followed by parenthesis and arguments.

  Correct: WESTPA calls the `assign` method as assign(coords, mask=None, output=None)

- Never use a method or command as a verb.

  Correct: Run `cd` to change the current working directory.

  Avoid: `cd` into the main simulation directory.

Programming languages

- Some programming languages are both a language and a command. When referring to the language, capitalize the word and use standard font. When referring to the command, preserve capitalization as it would appear in a terminal and use the `` tag.

  Using WESTPA requires some knowledge of Python.

  Run `python` to launch an interactive session.

  The Bash shell provides some handy capabilities, such as wildcard matching.

  Use `bash` to run `example.sh`.

Scripts

- Use the `.. code-block::` directive for short scripts. Options are available for some languages, such as `.. code-block:: bash` and `.. code-block:: python`.

```
#!/bin/bash
# This is a generic Bash script.

BASHVAR="Hello, world!"
echo $BASHVAR
```

```
#!/usr/bin/env python
# This is a generic Python script.

def main():
    pythonstr = "Hello, world!"
    print(pythonstr)
    return
if __name__ == "__main__":
    main()
```

- Begin a code snippet with a `#!` *shebang* (yes, this is the real term), followed by the usual path to a program. The line after the shebang should be an ellipsis, followed by lines of code. Use `#!/bin/bash` for Bash scripts,

`#!/bin/sh` for generic shell scripts, and `#!/usr/bin/env python` for Python scripts. For Python code snippets that are not a stand-alone script, place any import commands between the shebang line and ellipsis.

```
#!/usr/bin/env python
import numpy
...
def some_function(generic_vals):
    return 1 + numpy.mean(generic_vals)
```

- Follow the PEP 0008 Python Style Guide for Python scripts.
  - Indents are four spaces.
  - For comments, use the # hash mark followed by a single space, and then the comment's text.
  - Break lines after 80 characters.
- For Bash scripts, consider following Google's Shell Style Guide
- Indents are two spaces.
- Use blank lines to improve readability
- Use `; do` and `; then` on the same line as `while`, `for`, and `if`.
- Break lines after 80 characters.
- For other languages, consider following a logical style guide. At minimum, be consistent.

Variables

- Use the fixed-width `` ` ` `` tag when referring to a variable.

  the `ndim` attribute
- When explicitly referring to an attribute as well as its class, refer to an attribute as: the `attr` attribute of `GenericClass`, rather than `GenericClass.attr`
- Use the `$` dollar sign before Bash variables.

  WESTPA makes the variable `$WEST_BSTATE_DATA_REF` available to new trajectories.

7.3 Source Code Management

7.4 Documentation Practices

Documentation for WESTPA is maintained using Sphinx Docstrings are formatted in the Numpy style, which are converted to ReStructuredText using Sphinx' Napoleon plugin, which is included with Sphinx 1.3.

The documentation may be built by navigating to the `doc` folder, and running:

```
make html
```

to prepare an html version or:

```
make latexpdf
```

To prepare a pdf. The latter requires `latex` to be available.

A quick command to update the documentation in gh-pages repo is also available:

```
make ghpages
```

This command will run Sphinx html command and change the htmls to fit with the gh-pages format, it also runs:

```
git checkout gh-pages
git commit -a
git push
```

for you. Also note that this will change the current branch you are at to gh-pages branch. It also leaves behind a doc/_build folder that is no longer useful. Once you run ghpages command I suggest going up a folder and removing the unnecessary doc folder that is there by:

```
cd ../
rm -r doc
```

Remeber to make sure you are indeed in gh-pages branch, this branch is not supposed to have a folder named doc. Sometimes if you are not careful git checkout fails and you might end up removing the folder you were working with if you are not careful.

7.5 WESTPA Modules API

7.5.1 Binning

7.5.2 YAMLCFG

7.5.3 RC

7.6 WESTPA Tools