WESTPA Documentation

Release 2022.08

Matthew C. Zwier and Lillian T. Chong

May 10, 2024

FOR USERS (V2022.XX):

CHAPTER

ONE

OVERVIEW

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

Russo, J. D., Zhang, S., Leung, J.M.G., Bogetti, A.T., Thompson, J.P., DeGrave, A.J., Torrillo, P.A., Pratt, A.J., Wong, K.F., Xia, J., Copperman, J., Adelman, J.L., Zwier, M.C., LeBard, D.N., Zuckerman, D.M., Chong, L.T. WESTPA 2.0: High-Performance Upgrades for Weighted Ensemble Simulations and Analysis of Longer-Timescale Applications. J. Chem. Theory Comput., 18 (2): 638–649 (2022).

See this page and this powerpoint for an overview of weighted ensemble simulation.

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 MIT License. See the file `LICENSE` for more information.

CHAPTER

TWO

REQUIREMENTS

WESTPA is written in Python and requires version 3.7 or later. WESTPA also requires a number of Python scientific software packages. The simplest way to meet these requirements is to download the Anaconda Python distribution from www.anaconda.com (free for all users).

WESTPA currently runs on Unix-like operating systems, including Linux and Mac OS X. It is developed and tested on x86_64 machines running Linux.

CHAPTER

THREE

OBTAINING AND INSTALLING WESTPA

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

Regardless of the chosen method of installation, before installing WESTPA, we recommend you to first install the Python 3 version provided by the latest free Anaconda Python distribution. After installing Anaconda, create a new python environment for the WESTPA install with the following:

```
conda create -n westpa-2.0 python=3.9
conda activate westpa-2.0
```

Then, we recommend installing WESTPA through conda or pip. Execute either of the following:

```
conda install -c conda-forge westpa
```

or:

```
python -m pip install westpa
```

See the install instructions on our wiki for more detailed information.

To install from source (**not recommended**), start 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-main.tar.gz
cd westpa
python -m pip install -e .
```

CHAPTER

FOUR

GETTING STARTED

High-level tutorials of how to use the WESTPA software can be found here. Further, all WESTPA command-line tools 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

GETTING HELP

WESTPA 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.

CHAPTER

SIX

DEVELOPERS

Search archived messages or post to the westpa-devel Google group: https://groups.google.com/group/westpa-devel.

6.1 westpa.cli package

6.1.1 w_init

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

6.1.1.1 Overview

Usage:

```
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.

6.1.1.2 Command-Line Options

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

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

--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.
```

6.1.1.3 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.)

6.1.1.4 westpa.cli.core.w_init module

class westpa.cli.core.w_init.**BasisState**(*label*, *probability*, *pcoord=None*, *auxref=None*, *state_id=None*)

Bases: object

Describes an basis (micro)state. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation (i.e. at w_init) or due to recycling.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **probability** – Probability of this state to be selected when creating a new trajectory.
- **pcoord** – The representative progress coordinate of this state.
- **auxref** – A user-provided (string) reference for locating data associated with this state (usually a filesystem path).

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*)

Read a file defining basis states. Each line 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
```

as_numpy_record()

Return the data for this state as a numpy record array.

class westpa.cli.core.w_init.**TargetState**(*label*, *pcoord*, *state_id=None*)

Bases: object

Describes a target state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **pcoord** – The representative progress coordinate of this state.

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*, *dtype*)

Read a file defining target states. 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.

westpa.cli.core.w_init.**make_work_manager**()

Using cues from the environment, instantiate a pre-configured work manager.

westpa.cli.core.w_init.**entry_point**()

westpa.cli.core.w_init.**initialize**(*tstates*, *tstate_file*, *bstates*, *bstate_file*, *sstates=None*, *sstate_file=None*, *segs_per_state=1*, *shotgun=False*)

Initialize a WESTPA simulation.

tstates : list of str

tstate_file : str

bstates : list of str

bstate_file : str

sstates : list of str

sstate_file : str

segs_per_state : int

shotgun : bool

6.1.2 w_bins

`w_bins` deals with binning modification and statistics

6.1.2.1 Overview

Usage:

```
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.

6.1.2.2 Command-Line Options

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

6.1.2.2.1 Options Under 'info'

Usage:

```
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).
```

6.1.2.2.2 Options Under 'rebin'

Usage:

```
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.
  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).
```

6.1.2.3 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).
```

6.1.2.4 Examples

(TODO: Write up an example)

6.1.2.5 westpa.cli.tools.w_bins module

class westpa.cli.tools.w_bins.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.cli.tools.w_bins.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_bins.**BinMappingComponent**

Bases: *WESTToolComponent*

Component for obtaining a bin mapper from one of several places based on command-line arguments. Such locations include an HDF5 file that contains pickled mappers (including the primary WEST HDF5 file), the system object, an external function, or (in the common case of rectilinear bins) a list of lists of bin boundaries.

Some configuration is necessary prior to calling process_args() if loading a mapper from HDF5. Specifically, either set_we_h5file_info() or set_other_h5file_info() must be called to describe where to find the appropriate mapper. In the case of set_we_h5file_info(), the mapper used for WE at the end of a given iteration will be loaded. In the case of set_other_h5file_info(), an arbitrary group and hash value are specified; the mapper corresponding to that hash in the given group will be returned.

In the absence of arguments, the mapper contained in an existing HDF5 file is preferred; if that is not available, the mapper from the system driver is used.

This component adds the following arguments to argument parsers:

--bins-from-system Obtain bins from the system driver

—bins-from-expr=EXPR Construct rectilinear bins by parsing EXPR and calling RectilinearBinMapper() with the result. EXPR must therefore be a list of lists.

–bins-from-function=[PATH:]MODULE.FUNC
Call an external function FUNC in module MODULE (optionally adding PATH to the search path when loading MODULE) which, when called, returns a fully-constructed bin mapper.

—bins-from-file Load bin definitions from a YAML configuration file.

--bins-from-h5file Load bins from the file being considered; this is intended to mean the master WEST HDF5 file or results of other binning calculations, as appropriate.

add_args(*parser*, *description='binning options'*, *suppress=[]*)

Add arguments specific to this component to the given argparse parser.

add_target_count_args(*parser*, *description='bin target count options'*)

Add options to the given parser corresponding to target counts.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

set_we_h5file_info(*n_iter=None*, *data_manager=None*, *required=False*)

Set up to load a bin mapper from the master WEST HDF5 file. The mapper is actually loaded from the file when self.load_bin_mapper() is called, if and only if command line arguments direct this. If `required` is true, then a mapper must be available at iteration `n_iter`, or else an exception will be raised.

set_other_h5file_info(*topology_group*, *hashval*)

Set up to load a bin mapper from (any) open HDF5 file, where bin topologies are stored in `topology_group` (an h5py Group object) and the desired mapper has hash value `hashval`. The mapper itself is loaded when self.load_bin_mapper() is called.

westpa.cli.tools.w_bins.**write_bin_info**(*mapper*, *assignments*, *weights*, *n_target_states*, *outfile=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *detailed=False*)

Write information about binning to `outfile`, given a mapper (`mapper`) and the weights (`weights`) and bin assignments (`assignments`) of a set of segments, along with a target state count (`n_target_states`). If `detailed` is true, then per-bin information is written as well as summary information about all bins.

class westpa.cli.tools.w_bins.**WBinTool**

Bases: *WESTTool*

prog = 'w_bins'

description = 'Display information and statistics about binning in a WEST simulation, or\nmodify the binning for the current iteration of a WEST simulation.\n----------\n'

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

cmd_info()

cmd_rebin()

westpa.cli.tools.w_bins.**entry_point**()

6.1.3 w_run

`w_run` starts or continues a weighted ensemble simualtion.

6.1.3.1 Overview

Usage:

```
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]
```

6.1.3.2 Command-Line Options

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

6.1.3.2.1 Segment Options

::

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

6.1.3.3 Example

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

```
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.

6.1.3.4 westpa.cli.core.w_run module

`westpa.cli.core.w_run.`**`make_work_manager`**`()`

Using cues from the environment, instantiate a pre-configured work manager.

`westpa.cli.core.w_run.`**`entry_point`**`()`

`westpa.cli.core.w_run.`**`run_simulation`**`()`

6.1.4 w_truncate

`w_truncate` removes all iterations after a certain point

6.1.4.1 Overview

Usage:

```
w_truncate [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               [-n N_ITER] [-W WEST_H5FILE]
```

Remove all iterations after a certain point in a

6.1.4.2 Command-Line Options

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

6.1.4.2.1 Iteration Options

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

-W WEST_H5FILE, --west-data WEST_H5FILE
  PATH of H5 file to truncate. By default, it will read from the RCFILE (e.g., west.cfg).
  This option will have override whatever's provided in the RCFILE.
```

6.1.4.3 Examples

Running the following will remove iteration 50 and all iterations after 50 from multi.h5.

```
w_truncate -n 50 -W multi.h5
```

6.1.4.4 westpa.cli.core.w_truncate module

`westpa.cli.core.w_truncate.`**`entry_point`**`()`

6.1.5 w_fork

usage:

```
w_fork [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-i INPUT_H5FILE]
             [-I N_ITER] [-o OUTPUT_H5FILE] [--istate-map ISTATE_MAP] [--no-headers]
```

Prepare a new weighted ensemble simulation from an existing one at a particular point. A new HDF5 file is generated. In the case of executable propagation, it is the user's responsibility to prepare the new simulation directory appropriately, particularly making the old simulation's restart data from the appropriate iteration available as the new simulations initial state data; a mapping of old simulation segment to new simulation initial states is created, both in the new HDF5 file and as a flat text file, to aid in this. Target states and basis states for the new simulation are taken from those in the original simulation.

optional arguments:

```
-h, --help            show this help message and exit
-i INPUT_H5FILE, --input INPUT_H5FILE
                      Create simulation from the given INPUT_H5FILE (default: read from␣
↪configuration
                      file.
-I N_ITER, --iteration N_ITER
                      Take initial distribution for new simulation from iteration N_ITER␣
↪(default:
                      last complete iteration).
-o OUTPUT_H5FILE, --output OUTPUT_H5FILE
                      Save new simulation HDF5 file as OUTPUT (default: forked.h5).
--istate-map ISTATE_MAP
                      Write text file describing mapping of existing segments to new␣
↪initial states
                      in ISTATE_MAP (default: istate_map.txt).
--no-headers          Do not write header to ISTATE_MAP
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

6.1.5.1 westpa.cli.tools.w_fork module

class westpa.cli.core.w_fork.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: `object`

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.cli.core.w_fork.InitialState(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).
- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

westpa.cli.core.w_fork.**n_iter_dtype**

alias of uint32

westpa.cli.core.w_fork.**seg_id_dtype**

alias of int64

westpa.cli.core.w_fork.**entry_point**()

6.1.6 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).

6.1.6.1 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]
```

6.1.6.2 Command-Line Options

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

6.1.6.3 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**)
```

6.1.6.4 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

--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
```

6.1.6.5 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
```

(continues on next page)

(continued from previous page)

```
  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]
```

6.1.6.6 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
  **default** function returns the 'pcoord' dataset for that iteration
  (i.e. the function executes return iter_group['pcoord'][...])
```

6.1.6.7 Examples

6.1.6.8 westpa.cli.tools.w_assign module

westpa.cli.tools.w_assign.**seg_id_dtype**

alias of int64

westpa.cli.tools.w_assign.**weight_dtype**

alias of float64

westpa.cli.tools.w_assign.**index_dtype**

alias of uint16

westpa.cli.tools.w_assign.**assign_and_label**(*nsegs_lb*, *nsegs_ub*, *parent_ids*, *assign*, *nstates*, *state_map*, *last_labels*, *pcoords*, *subsample*)

Assign trajectories to bins and last-visted macrostates for each timepoint.

westpa.cli.tools.w_assign.**accumulate_labeled_populations**(*weights*, *bin_assignments*, *label_assignments*, *labeled_bin_pops*)

For a set of segments in one iteration, calculate the average population in each bin, with separation by last-visited macrostate.

class westpa.cli.tools.w_assign.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_assign.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_assign.**WESTDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

Tool for synthesizing a dataset for analysis from other datasets. This may be done using a custom function, or a list of "data set specifications". It is anticipated that if several source datasets are required, then a tool will have multiple instances of this class.

group_name = 'input dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_assign.**BinMappingComponent**

Bases: *WESTToolComponent*

Component for obtaining a bin mapper from one of several places based on command-line arguments. Such locations include an HDF5 file that contains pickled mappers (including the primary WEST HDF5 file), the system object, an external function, or (in the common case of rectilinear bins) a list of lists of bin boundaries.

Some configuration is necessary prior to calling process_args() if loading a mapper from HDF5. Specifically, either set_we_h5file_info() or set_other_h5file_info() must be called to describe where to find the appropriate mapper. In the case of set_we_h5file_info(), the mapper used for WE at the end of a given iteration will be loaded. In the case of set_other_h5file_info(), an arbitrary group and hash value are specified; the mapper corresponding to that hash in the given group will be returned.

In the absence of arguments, the mapper contained in an existing HDF5 file is preferred; if that is not available, the mapper from the system driver is used.

This component adds the following arguments to argument parsers:

--bins-from-system Obtain bins from the system driver

—bins-from-expr=EXPR Construct rectilinear bins by parsing EXPR and calling RectilinearBinMapper() with the result. EXPR must therefore be a list of lists.

–bins-from-function=[PATH:]MODULE.FUNC
Call an external function FUNC in module MODULE (optionally adding PATH to the search path when loading MODULE) which, when called, returns a fully-constructed bin mapper.

—bins-from-file Load bin definitions from a YAML configuration file.

--bins-from-h5file Load bins from the file being considered; this is intended to mean the master WEST HDF5 file or results of other binning calculations, as appropriate.

add_args(*parser*, *description='binning options'*, *suppress=[]*)

Add arguments specific to this component to the given argparse parser.

add_target_count_args(*parser*, *description='bin target count options'*)

Add options to the given parser corresponding to target counts.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

set_we_h5file_info(*n_iter=None*, *data_manager=None*, *required=False*)

Set up to load a bin mapper from the master WEST HDF5 file. The mapper is actually loaded from the file when self.load_bin_mapper() is called, if and only if command line arguments direct this. If `required` is true, then a mapper must be available at iteration `n_iter`, or else an exception will be raised.

set_other_h5file_info(*topology_group*, *hashval*)

Set up to load a bin mapper from (any) open HDF5 file, where bin topologies are stored in `topology_group` (an h5py Group object) and the desired mapper has hash value `hashval`. The mapper itself is loaded when self.load_bin_mapper() is called.

class westpa.cli.tools.w_assign.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_assign.**WESTPAH5File**(**args*, ***kwargs*)

Bases: `File`

Generalized input/output for WESTPA simulation (or analysis) data.

Create a new file object.

See the h5py user guide for a detailed explanation of the options.

name
Name of the file on disk, or file-like object. Note: for files created with the 'core' driver, HDF5 still requires this be non-empty.

mode
r Readonly, file must exist (default) r+ Read/write, file must exist w Create file, truncate if exists w- or x Create file, fail if exists a Read/write if exists, create otherwise

driver
Name of the driver to use. Legal values are None (default, recommended), 'core', 'sec2', 'direct', 'stdio', 'mpio', 'ros3'.

libver
Library version bounds. Supported values: 'earliest', 'v108', 'v110', 'v112' and 'latest'. The 'v108', 'v110' and 'v112' options can only be specified with the HDF5 1.10.2 library or later.

userblock_size
Desired size of user block. Only allowed when creating a new file (mode w, w- or x).

swmr
Open the file in SWMR read mode. Only used when mode = 'r'.

rdcc_nbytes
Total size of the dataset chunk cache in bytes. The default size is 1024**2 (1 MiB) per dataset. Applies to all datasets unless individually changed.

rdcc_w0
The chunk preemption policy for all datasets. This must be between 0 and 1 inclusive and indicates the weighting according to which chunks which have been fully read or written are penalized when determining which chunks to flush from cache. A value of 0 means fully read or written chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of 1 means fully read or written chunks are always preempted before other chunks. If your application only reads or writes data once, this can be

safely set to 1. Otherwise, this should be set lower depending on how often you re-read or re-write the same data. The default value is 0.75. Applies to all datasets unless individually changed.

rdcc_nslots
The number of chunk slots in the raw data chunk cache for this file. Increasing this value reduces the number of cache collisions, but slightly increases the memory used. Due to the hashing strategy, this value should ideally be a prime number. As a rule of thumb, this value should be at least 10 times the number of chunks that can fit in rdcc_nbytes bytes. For maximum performance, this value should be set approximately 100 times that number of chunks. The default value is 521. Applies to all datasets unless individually changed.

track_order
Track dataset/group/attribute creation order under root group if True. If None use global default h5.get_config().track_order.

fs_strategy
The file space handling strategy to be used. Only allowed when creating a new file (mode w, w- or x). Defined as: "fsm" FSM, Aggregators, VFD "page" Paged FSM, VFD "aggregate" Aggregators, VFD "none" VFD If None use HDF5 defaults.

fs_page_size
File space page size in bytes. Only used when fs_strategy="page". If None use the HDF5 default (4096 bytes).

fs_persist
A boolean value to indicate whether free space should be persistent or not. Only allowed when creating a new file. The default value is False.

fs_threshold
The smallest free-space section size that the free space manager will track. Only allowed when creating a new file. The default value is 1.

page_buf_size
Page buffer size in bytes. Only allowed for HDF5 files created with fs_strategy="page". Must be a power of two value and greater or equal than the file space page size when creating the file. It is not used by default.

min_meta_keep
Minimum percentage of metadata to keep in the page buffer before allowing pages containing metadata to be evicted. Applicable only if page_buf_size is set. Default value is zero.

min_raw_keep
Minimum percentage of raw data to keep in the page buffer before allowing pages containing raw data to be evicted. Applicable only if page_buf_size is set. Default value is zero.

locking
The file locking behavior. Defined as:

- False (or "false") – Disable file locking
- True (or "true") – Enable file locking
- "best-effort" – Enable file locking but ignore some errors
- None – Use HDF5 defaults

> **Warning:** The HDF5_USE_FILE_LOCKING environment variable can override this parameter.

Only available with HDF5 >= 1.12.1 or 1.10.x >= 1.10.7.

alignment_threshold
Together with `alignment_interval`, this property ensures that any file object greater than or equal in

size to the alignment threshold (in bytes) will be aligned on an address which is a multiple of alignment interval.

alignment_interval
This property should be used in conjunction with `alignment_threshold`. See the description above. For more details, see https://portal.hdfgroup.org/display/HDF5/H5P_SET_ALIGNMENT

meta_block_size
Set the current minimum size, in bytes, of new metadata block allocations. See https://portal.hdfgroup.org/display/HDF5/H5P_SET_META_BLOCK_SIZE

Additional keywords
Passed on to the selected file driver.

default_iter_prec = 8

replace_dataset(**args*, ***kwargs*)

iter_object_name(*n_iter*, *prefix=''*, *suffix=''*)

Return a properly-formatted per-iteration name for iteration `n_iter`. (This is used in create/require/get_iter_group, but may also be useful for naming datasets on a per-iteration basis.)

create_iter_group(*n_iter*, *group=None*)

Create a per-iteration data storage group for iteration number `n_iter` in the group `group` (which is '/iterations' by default).

require_iter_group(*n_iter*, *group=None*)

Ensure that a per-iteration data storage group for iteration number `n_iter` is available in the group `group` (which is '/iterations' by default).

get_iter_group(*n_iter*, *group=None*)

Get the per-iteration data group for iteration number `n_iter` from within the group `group` ('/iterations' by default).

westpa.cli.tools.w_assign.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

westpa.cli.tools.w_assign.**parse_pcoord_value**(*pc_str*)

class westpa.cli.tools.w_assign.**WAssign**

Bases: *WESTParallelTool*

prog = 'w_assign'

```
description = 'Assign walkers to bins, producing a file (by default named
"assign.h5")\nwhich can be used in subsequent analysis.\n\nFor consistency in
subsequent analysis operations, the entire dataset\nmust be assigned, even if only a
subset of the data will be used. This\nensures that analyses that rely on tracing
trajectories always know the\noriginating bin of each trajectory.\n\n\
n-----------------------------------------------------------------------------\
nSource
data\n-----------------------------------------------------------------------------\
n\nSource data is provided either by a user-specified
function\n(--construct-dataset) or a list of "data set specifications"
(--dsspecs).\nIf neither is provided, the progress coordinate dataset \'\'pcoord\'\'
is used.\n\nTo use a custom function to extract or calculate data whose
probability\ndistribution will be calculated, specify the function in standard
Python\nMODULE.FUNCTION syntax as the argument to --construct-dataset. This
function\nwill be called as function(n_iter,iter_group), where n_iter is the
iteration\nwhose data are being considered and iter_group is the corresponding
group\nin the main WEST HDF5 file (west.h5). The function must return data which
can\nbe indexed as [segment][timepoint][dimension].\n\nTo use a list of data set
specifications, specify --dsspecs and then list the\ndesired datasets one-by-one
(space-separated in most shells). These data set\nspecifications are formatted as
NAME[,file=FILENAME,slice=SLICE], which will\nuse the dataset called NAME in the
HDF5 file FILENAME (defaulting to the main\nWEST HDF5 file west.h5), and slice it
with the Python slice expression SLICE\n(as in [0:2] to select the first two
elements of the first axis of the\ndataset). The ``slice`` option is most useful for
selecting one column (or\nmore) from a multi-column dataset, such as arises when
using a progress\ncoordinate of multiple dimensions.\n\n\
n-----------------------------------------------------------------------------\
nSpecifying
macrostates\
n-----------------------------------------------------------------------------\n\
nOptionally, kinetic macrostates may be defined in terms of sets of bins.\nEach
trajectory will be labeled with the kinetic macrostate it was most\nrecently in at
each timepoint, for use in subsequent kinetic analysis.\nThis is required for all
kinetics analysis (w_kintrace and w_kinmat).\n\nThere are three ways to specify
macrostates:\n\n 1. States corresponding to single bins may be identified on the
command\n line using the --states option, which takes multiple arguments, one for\n
each state (separated by spaces in most shells). Each state is specified\n as a
coordinate tuple, with an optional label prepended, as in\n ``bound:1.0`` or
``unbound:(2.5,2.5)``. Unlabeled states are named\n ``stateN``, where N is the
(zero-based) position in the list of states\n supplied to --states.\n\n 2. States
corresponding to multiple bins may use a YAML input file specified\n with
--states-from-file. This file defines a list of states, each with a\n name and a
list of coordinate tuples; bins containing these coordinates\n will be mapped to the
containing state. For instance, the following\n file::\n\n ---\n states:\n - label:
unbound\n coords:\n - [9.0, 1.0]\n - [9.0, 2.0]\n - label:  bound\n coords:\n -
[0.1, 0.0]\n\n produces two macrostates:  the first state is called "unbound" and\n
consists of bins containing the (2-dimensional) progress coordinate\n values (9.0,
1.0) and (9.0, 2.0); the second state is called "bound"\n and consists of the single
bin containing the point (0.1, 0.0).\n\n 3. Arbitrary state definitions may be
supplied by a user-defined function,\n specified as
--states-from-function=MODULE.FUNCTION. This function is\n called with the bin
mapper as an argument (``function(mapper)``) and must\n return a list of
dictionaries, one per state. Each dictionary must contain\n a vector of coordinate
tuples with key "coords"; the bins into which each\n of these tuples falls define
the state. An optional name for the state\n (with key "label") may also be provided.
\n\n\n-----------------------------------------------------------------------------\
nOutput
format\
n-----------------------------------------------------------------------------\n\
nThe output file (-o/--output, by default "assign.h5") contains the
```

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

parse_cmdline_states(*state_strings*)

load_config_from_west(*scheme*)

load_state_file(*state_filename*)

states_from_dict(*ystates*)

load_states_from_function(*statefunc*)

assign_iteration(*n_iter*, *nstates*, *nbins*, *state_map*, *last_labels*)

Method to encapsulate the segment slicing (into n_worker slices) and parallel job submission Submits job(s), waits on completion, splices them back together Returns: assignments, trajlabels, pops for this iteration

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_assign.**entry_point**()

6.1.7 w_trace

usage:

```
w_trace [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-W WEST_H5FILE]
          [-d DSNAME] [--output-pattern OUTPUT_PATTERN] [-o OUTPUT]
          N_ITER:SEG_ID [N_ITER:SEG_ID ...]
```

Trace individual WEST trajectories and emit (or calculate) quantities along the trajectory.

Trajectories are specified as N_ITER:SEG_ID pairs. Each segment is traced back to its initial point, and then various quantities (notably n_iter and seg_id) are printed in order from initial point up until the given segment in the given iteration.

Output is stored in several files, all named according to the pattern given by the -o/–output-pattern parameter. The default output pattern is “traj_%d_%d”, where the printf-style format codes are replaced by the iteration number and segment ID of the terminal segment of the trajectory being traced.

Individual datasets can be selected for writing using the `-d/--dataset` option (which may be specified more than once). The simplest form is `-d dsname`, which causes data from dataset `dsname` along the trace to be stored to HDF5. The dataset is assumed to be stored on a per-iteration basis, with the first dimension corresponding to seg_id and the second dimension corresponding to time within the segment. Further options are specified as comma-separated key=value pairs after the data set name, as in:

```
-d dsname,alias=newname,index=idsname,file=otherfile.h5,slice=[100,...]
```

The following options for datasets are supported:

```
alias=newname
    When writing this data to HDF5 or text files, use ``newname``
    instead of ``dsname`` to identify the dataset. This is mostly of
    use in conjunction with the ``slice`` option in order, e.g., to
    retrieve two different slices of a dataset and store then with
    different names for future use.

index=idsname
    The dataset is not stored on a per-iteration basis for all
    segments, but instead is stored as a single dataset whose
    first dimension indexes n_iter/seg_id pairs. The index to
    these n_iter/seg_id pairs is ``idsname``.

file=otherfile.h5
    Instead of reading data from the main WEST HDF5 file (usually
    ``west.h5``), read data from ``otherfile.h5``.

slice=[100,...]
    Retrieve only the given slice from the dataset. This can be
    used to pick a subset of interest to minimize I/O.
```

6.1.7.1 positional arguments

```
N_ITER:SEG_ID         Trace trajectory ending (or at least alive at) N_ITER:SEG_ID.
```

6.1.7.2 optional arguments

```
-h, --help            show this help message and exit
-d DSNAME, --dataset DSNAME
                      Include the dataset named DSNAME in trace output. An extended form␣
↪like
                      DSNAME[,alias=ALIAS][,index=INDEX][,file=FILE][,slice=SLICE] will␣
↪obtain the
                      dataset from the given FILE instead of the main WEST HDF5 file,␣
↪slice it by
                      SLICE, call it ALIAS in output, and/or access per-segment data by a
                      n_iter,seg_id INDEX instead of a seg_id indexed dataset in the␣
↪group for
                      n_iter.
```

6.1.7.3 general options

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

6.1.7.4 WEST input data options

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

6.1.7.5 output options

```
--output-pattern OUTPUT_PATTERN
                      Write per-trajectory data to output files/HDF5 groups whose names␣
↪begin with
                      OUTPUT_PATTERN, which must contain two printf-style format flags␣
↪which will be
                      replaced with the iteration number and segment ID of the terminal␣
↪segment of
                      the trajectory being traced. (Default: traj_%d_%d.)
-o OUTPUT, --output OUTPUT
                      Store intermediate data and analysis results to OUTPUT (default:␣
↪trajs.h5).
```

6.1.7.6 westpa.cli.tools.w_trace module

class westpa.cli.tools.w_trace.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.cli.tools.w_trace.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_trace.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: `object`

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.cli.tools.w_trace.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).

- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

westpa.cli.tools.w_trace.**weight_dtype**

alias of float64

westpa.cli.tools.w_trace.**n_iter_dtype**

alias of uint32

westpa.cli.tools.w_trace.**seg_id_dtype**

alias of int64

westpa.cli.tools.w_trace.**utime_dtype**

alias of float64

class westpa.cli.tools.w_trace.**Trace**(*summary*, *endpoint_type*, *basis_state*, *initial_state*, *data_manager=None*)

Bases: object

A class representing a trace of a certain trajectory segment back to its origin.

classmethod from_data_manager(*n_iter*, *seg_id*, *data_manager=None*)

Construct and return a trajectory trace whose last segment is identified by `seg_id` in the iteration number `n_iter`.

get_segment_data_slice(*datafile*, *dsname*, *n_iter*, *seg_id*, *slice_=None*, *index_data=None*, *iter_prec=None*)

Return the data from the dataset named `dsname` within the given `datafile` (an open h5py.File object) for the given iteration and segment. By default, it is assumed that the dataset is stored in the iteration group for iteration `n_iter`, but if `index_data` is provided, it must be an iterable (preferably a simple array) of (n_iter,seg_id) pairs, and the index in the `index_data` iterable of the matching n_iter/seg_id pair is used as the index of the data to retrieve.

If an optional `slice_` is provided, then the given slicing tuple is appended to that used to retrieve the segment-specific data (i.e. it can be used to pluck a subset of the data that would otherwise be returned).

trace_timepoint_dataset(*dsname*, *slice_=None*, *auxfile=None*, *index_ds=None*)

Return a trace along this trajectory over a dataset which is layed out as [seg_id][timepoint][...]. Overlapping values at segment boundaries are accounted for. Returns (data_trace, weight), where data_trace is a time series of the dataset along this trajectory, and weight is the corresponding trajectory weight at each time point.

If `auxfile` is given, then load the dataset from the given HDF5 file, which must be layed out the same way as the main HDF5 file (e.g. iterations arranged as iterations/iter_*).

If index_ds is given, instead of reading data per-iteration from iter_* groups, then the given index_ds is used as an index of n_iter,seg_id pairs into `dsname`. In this case, the target data set need not exist on a per-iteration basis inside iter_* groups.

If `slice_` is given, then *further* slice the data returned from the HDF5 dataset. This can minimize I/O if it is known (and specified) that only a subset of the data along the trajectory is needed.

class westpa.cli.tools.w_trace.**WTraceTool**

Bases: *WESTTool*

prog = 'w_trace'

description = 'Trace individual WEST trajectories and emit (or calculate) quantities along the\ntrajectory.\n\nTrajectories are specified as N_ITER:SEG_ID pairs. Each segment is traced back\nto its initial point, and then various quantities (notably n_iter and seg_id)\nare printed in order from initial point up until the given segment in the given\niteration.\n\nOutput is stored in several files, all named according to the pattern given by\nthe -o/--output-pattern parameter. The default output pattern is "traj_%d_%d",\nwhere the printf-style format codes are replaced by the iteration number and\nsegment ID of the terminal segment of the trajectory being traced.\n\nIndividual datasets can be selected for writing using the -d/--dataset option\n(which may be specified more than once). The simplest form is ``-d dsname``,\nwhich causes data from dataset ``dsname`` along the trace to be stored to\nHDF5. The dataset is assumed to be stored on a per-iteration basis, with\nthe first dimension corresponding to seg_id and the second dimension\ncorresponding to time within the segment. Further options are specified\nas comma-separated key=value pairs after the data set name, as in\n\n -d dsname,alias=newname,index=idsname,file=otherfile.h5,slice=[100,...]\n\nThe following options for datasets are supported:\n\n alias=newname\n When writing this data to HDF5 or text files, use ``newname``\n instead of ``dsname`` to identify the dataset. This is mostly of\n use in conjunction with the ``slice`` option in order, e.g., to\n retrieve two different slices of a dataset and store then with\n different names for future use.\n\n index=idsname\n The dataset is not stored on a per-iteration basis for all\n segments, but instead is stored as a single dataset whose\n first dimension indexes n_iter/seg_id pairs. The index to\n these n_iter/seg_id pairs is ``idsname``.\n\n file=otherfile.h5\n Instead of reading data from the main WEST HDF5 file (usually\n ``west.h5``), read data from ``otherfile.h5``.\n\n slice=[100,...]\n Retrieve only the given slice from the dataset. This can be\n used to pick a subset of interest to minimize I/O.\n\n\n----------\n'

pcoord_formats = {'f4': '%14.7g', 'f8': '%023.15g', 'i2': '%6d', 'i4': '%11d', 'i8': '%20d', 'u2': '%5d', 'u4': '%10d', 'u8': '%20d'}

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

parse_dataset_string(*dsstr*)

go()

Perform the analysis associated with this tool.

emit_trace_h5(*trace*, *output_group*)

emit_trace_text(*trace*, *output_file*)

Dump summary information about each segment in the given trace to the given output_file, which must be opened for writing in text mode. Output columns are separated by at least one space.

westpa.cli.tools.w_trace.entry_point()

6.1.8 w_ipa

The `w_ipa` is a (beta) WESTPA tool that automates analysis using analysis schemes and enables interactive analysis of WESTPA simulation data. The tool can do a variety of different types of analysis, including the following: * Calculate fluxes and rate constants * Adjust and use alternate state definitions * Trace trajectory segments, including statistical weights, position along the progress coordinate, and other auxiliary data * Plot all of the above in the terminal!

If you are using `w_ipa` for kinetics automated kinetics analysis, keep in mind that `w_ipa` is running `w_assign` and `w_direct` using the scheme designated in your `west.cfg` file. For more diverse kinetics analysis options, consider using `w_assign` and `w_direct` manually. This can be useful if you'd like to use auxiliary coordinates that aren't your progress coordinate, in one or two dimension options.

usage:

```
w_ipa [-h] [-r RCFILE] [--quiet] [--verbose] [--version] [--max-queue-length MAX_QUEUE_
↪LENGTH]
            [-W WEST_H5FILE] [--analysis-only] [--reanalyze] [--ignore-hash] [--debug] [-
↪-terminal]
            [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_WORKERS]
            [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
            [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
            [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
            [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
            [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
            [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

-r RCFILE, --rcfile RCFILE use RCFILE as the WEST run-time configuration file (default: west.cfg)

--quiet emit only essential information

--verbose emit extra information

--version show program's version number and exit

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

WEST input data options:

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

runtime options:

```
--analysis-only, -ao  Use this flag to run the analysis and return to the terminal.
--reanalyze, -ra      Use this flag to delete the existing files and reanalyze.
--ignore-hash, -ih    Ignore hash and don't regenerate files.
--debug, -d           Debug output largely intended for development.
--terminal, -t        Plot output in terminal.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is
↪'processes'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ (“zmq”) work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
```

(continues on next page)

(continued from previous page)

```
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.1.8.1 westpa.cli.tools.w_ipa module

class westpa.cli.tools.w_ipa.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_ipa.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_ipa.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_ipa.**Plotter**(*h5file*, *h5key*, *iteration=-1*, *interface='matplotlib'*)

Bases: object

This is a semi-generic plotting interface that has a built in curses based terminal plotter. It's fairly specific to what we're using it for here, but we could (and maybe should) build it out into a little library that we can use via the command line to plot things. Might be useful for looking at data later. That would also cut the size of this tool down by a good bit.

plot(*i=0*, *j=1*, *tau=1*, *iteration=None*, *dim=0*, *interface=None*)

class westpa.cli.tools.w_ipa.**WIPIDataset**(*raw*, *key*)

Bases: object

keys()

class westpa.cli.tools.w_ipa.**WIPIScheme**(*scheme*, *name*, *parent*, *settings*)

Bases: object

property scheme

property list_schemes

Lists what schemes are configured in west.cfg file. Schemes should be structured as follows, in west.cfg:

west:

system:

analysis:
directory: analysis analysis_schemes:

scheme.1:
enabled: True states:

- label: unbound coords: [[7.0]]
- label: bound coords: [[2.7]]

bins:

- type: RectilinearBinMapper boundaries: [[0.0, 2.80, 7, 10000]]

property iteration

property assign

property direct

The output from w_direct.py from the current scheme.

property state_labels

property bin_labels

property west

property reweight

property current

The current iteration. See help for __get_data_for_iteration__

property past

The previous iteration. See help for __get_data_for_iteration__

class westpa.cli.tools.w_ipa.**WIPI**

Bases: *WESTParallelTool*

Welcome to w_ipa (WESTPA Interactive Python Analysis)! From here, you can run traces, look at weights, progress coordinates, etc. This is considered a 'stateful' tool; that is, the data you are pulling is always pulled from the current analysis scheme and iteration. By default, the first analysis scheme in west.cfg is used, and you are set at iteration 1.

ALL PROPERTIES ARE ACCESSED VIA w or west To see the current iteration, try:
w.iteration OR west.iteration
to set it, simply plug in a new value.
w.iteration = 100
To change/list the current analysis schemes:
w.list_schemes w.scheme = OUTPUT FROM w.list_schemes

To see the states and bins defined in the current analysis scheme:
    w.states w.bin_labels
All information about the current iteration is available in an object called 'current':
    w.current walkers, summary, states, seg_id, weights, parents, kinavg, pcoord, bins, populations, and auxdata, if it exists.
In addition, the function w.trace(seg_id) will run a trace over a seg_id in the current iteration and return a dictionary containing all pertinent information about that seg_id's history. It's best to store this, as the trace can be expensive.

Run help on any function or property for more information!

Happy analyzing!

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

hash_args(*args*, *extra=None*, *path=None*)

Create unique hash stamp to determine if arguments/file is different from before.

stamp_hash(*h5file_name*, *new_hash*)

Loads a file, stamps it, and returns the opened file in read only

analysis_structure()

Run automatically on startup. Parses through the configuration file, and loads up all the data files from the different analysis schematics. If they don't exist, it creates them automatically by hooking in to existing analysis routines and going from there.

It does this by calling in the make_parser_and_process function for w_{assign,reweight,direct} using a custom built list of args. The user can specify everything in the configuration file that would have been specified on the command line.

For instance, were one to call w_direct as follows:
    w_direct –evolution cumulative –step-iter 1 –disable-correl
the west.cfg would look as follows:
west:

analysis:

w_direct:
evolution: cumulative step_iter: 1 extra: ['disable-correl']
Alternatively, if one wishes to use the same options for both w_direct and w_reweight, the key 'w_direct' can be replaced with 'kinetics'.

property assign

property direct

The output from w_kinavg.py from the current scheme.

property state_labels

property bin_labels

property west

property reweight

property scheme

Returns and sets what scheme is currently in use. To see what schemes are available, run:
w.list_schemes

property list_schemes

Lists what schemes are configured in west.cfg file. Schemes should be structured as follows, in west.cfg:
west:

system:

analysis:
directory: analysis analysis_schemes:

scheme.1:
enabled: True states:

- label: unbound coords: [[7.0]]
- label: bound coords: [[2.7]]

bins:

- type: RectilinearBinMapper boundaries: [[0.0, 2.80, 7, 10000]]

property iteration

Returns/sets the current iteration.

property current

The current iteration. See help for __get_data_for_iteration__

property past

The previous iteration. See help for __get_data_for_iteration__

trace(*seg_id*)

Runs a trace on a seg_id within the current iteration, all the way back to the beginning, returning a dictionary containing all interesting information:
seg_id, pcoord, states, bins, weights, iteration, auxdata (optional)
sorted in chronological order.

Call with a seg_id.

property future

Similar to current/past, but keyed differently and returns different datasets. See help for Future.

class Future(*raw*, *key*)

Bases: *WIPIDataset*

go()

Function automatically called by main() when launched via the command line interface. Generally, call main, not this function.

property introduction

Just spits out an introduction, in case someone doesn't call help.

property help

Just a minor function to call help on itself. Only in here to really help someone get help.

westpa.cli.tools.w_ipa.**entry_point**()

6.1.9 w_pdist

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

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

6.1.9.1 Overview

Usage:

```
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.

6.1.9.2 Command-Line Options

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

6.1.9.2.1 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**)
```

6.1.9.2.2 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)
```

6.1.9.2.3 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)
```

6.1.9.3 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):

```
w_pdist
```

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

```
w_pdist --serial
```

6.1.9.4 westpa.cli.tools.w_pdist module

class westpa.cli.tools.w_pdist.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_pdist.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_pdist.**WESTDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

Tool for synthesizing a dataset for analysis from other datasets. This may be done using a custom function, or a list of "data set specifications". It is anticipated that if several source datasets are required, then a tool will have multiple instances of this class.

group_name = 'input dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_pdist.**WESTWDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

group_name = 'weight dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_pdist.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
: The first iteration included in the calculation.

last_iter
: One past the last iteration included in the calculation.

iter_step
: Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.cli.tools.w_pdist.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.cli.tools.w_pdist.**histnd**(*values*, *binbounds*, *weights=1.0*, *out=None*, *binbound_check=True*, *ignore_out_of_range=False*)

Generate an N-dimensional PDF (or contribution to a PDF) from the given values. `binbounds` is a list of arrays of boundary values, with one entry for each dimension (`values` must have as many columns as there are entries in `binbounds`) `weight`, if provided, specifies the weight each value contributes to the histogram; this may be a scalar (for equal weights for all values) or a vector of the same length as `values` (for unequal weights). If `binbound_check` is True, then the boundaries are checked for strict positive monotonicity; set to False to shave a few microseconds if you know your bin boundaries to be monotonically increasing.

westpa.cli.tools.w_pdist.**normhistnd**(*hist*, *binbounds*)

Normalize the N-dimensional histogram `hist` with corresponding bin boundaries `binbounds`. Modifies `hist` in place and returns the normalization factor used.

westpa.cli.tools.w_pdist.**isiterable**(*x*)

class westpa.cli.tools.w_pdist.**WPDist**

Bases: *WESTParallelTool*

prog = 'w_pdist'

```
description = 'Calculate time-resolved, multi-dimensional probability distributions
of WE\ndatasets.\n\n\
n-------------------------------------------------------------------------------\
nSource
data\n-------------------------------------------------------------------------------\
n\nSource data is provided either by a user-specified
function\n(--construct-dataset) or a list of "data set specifications"
(--dsspecs).\nIf neither is provided, the progress coordinate dataset \'\'pcoord\'\'
is used.\n\nTo use a custom function to extract or calculate data whose
probability\ndistribution will be calculated, specify the function in standard
Python\nMODULE.FUNCTION syntax as the argument to --construct-dataset. This
function\nwill be called as function(n_iter,iter_group), where n_iter is the
iteration\nwhose data are being considered and iter_group is the corresponding
group\nin the main WEST HDF5 file (west.h5). The function must return data which
can\nbe indexed as [segment][timepoint][dimension].\n\nTo use a list of data set
specifications, specify --dsspecs and then list the\ndesired datasets one-by-one
(space-separated in most shells). These data set\nspecifications are formatted as
NAME[,file=FILENAME,slice=SLICE], which will\nuse the dataset called NAME in the
HDF5 file FILENAME (defaulting to the main\nWEST HDF5 file west.h5), and slice it
with the Python slice expression SLICE\n(as in [0:2] to select the first two
elements of the first axis of the\ndataset). The ``slice`` option is most useful for
selecting one column (or\nmore) from a multi-column dataset, such as arises when
using a progress\ncoordinate of multiple dimensions.\n\n\
n-------------------------------------------------------------------------------\
nHistogram
binning\
n-------------------------------------------------------------------------------\n\nBy
default, histograms are constructed with 100 bins in each dimension. This\ncan be
overridden by specifying -b/--bins, which accepts a number of different\nkinds of
arguments:\n\n a single integer N\n N uniformly spaced bins will be used in each
dimension.\n\n a sequence of integers N1,N2,... (comma-separated)\n N1 uniformly
spaced bins will be used for the first dimension, N2 for the\n second, and so
on.\n\n a list of lists [[B11, B12, B13, ...], [B21, B22, B23, ...], ...]\n The bin
boundaries B11, B12, B13, ... will be used for the first dimension,\n B21, B22, B23,
... for the second dimension, and so on. These bin\n boundaries need not be
uniformly spaced. These expressions will be\n evaluated with Python\'s ``eval``
construct, with ``np`` available for\n use [e.g. to specify bins using
np.arange()].\n\nThe first two forms (integer, list of integers) will trigger a scan
of all\ndata in each dimension in order to determine the minimum and maximum
values,\nwhich may be very expensive for large datasets. This can be avoided
by\nexplicitly providing bin boundaries using the list-of-lists form.\n\nNote that
these bins are *NOT* at all related to the bins used to drive WE\nsampling.\n\n\
n-------------------------------------------------------------------------------\
nOutput
format\
n-------------------------------------------------------------------------------\n\
nThe output file produced (specified by -o/--output, defaulting to "pdist.h5")\nmay
be fed to plothist to generate plots (or appropriately processed text or\nHDF5
files) from this data. In short, the following datasets are created:\n\n
``histograms``\n Normalized histograms. The first axis corresponds to iteration,
and\n remaining axes correspond to dimensions of the input dataset.\n\n
``/binbounds_0``\n Vector of bin boundaries for the first (index 0) dimension.
Additional\n datasets similarly named (/binbounds_1, /binbounds_2, ...) are
created\n for additional dimensions.\n\n ``/midpoints_0``\n Vector of bin midpoints
for the first (index 0) dimension. Additional\n datasets similarly named are created
for additional dimensions.\n\n ``n_iter``\n Vector of iteration numbers
corresponding to the stored histograms (i.e.\n the first axis of the ``histograms``
dataset).\n\n\
n-------------------------------------------------------------------------------\
nSubsequent
```

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

static parse_binspec(*binspec*)

construct_bins(*bins*)

Construct bins according to `bins`, which may be:
1) A scalar integer (for that number of bins in each dimension)
2) A sequence of integers (specifying number of bins for each dimension)
3) A sequence of sequences of bin boundaries (specifying boundaries for each dimension)

Sets `self.binbounds` to a list of arrays of bin boundaries appropriate for passing to fasthist.histnd, along with `self.midpoints` to the midpoints of the bins.

scan_data_shape()

scan_data_range()

Scan input data for range in each dimension. The number of dimensions is determined from the shape of the progress coordinate as of self.iter_start.

construct_histogram()

Construct a histogram using bins previously constructed with `construct_bins()`. The time series of histogram values is stored in `histograms`. Each histogram in the time series is normalized.

`westpa.cli.tools.w_pdist.`**entry_point**()

6.1.10 w_succ

usage:

```
w_succ [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-A H5FILE] [-W WEST_
↪H5FILE]
             [-o OUTPUT_FILE]
```

List segments which successfully reach a target state.

optional arguments:

```
-h, --help            show this help message and exit
-o OUTPUT_FILE, --output OUTPUT_FILE
                      Store output in OUTPUT_FILE (default: write to standard output).
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
```

(continues on next page)

(continued from previous page)

```
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

general analysis options:

```
-A H5FILE, --analysis-file H5FILE
                      Store intermediate and final results in H5FILE (default: analysis.
↪h5).
```

WEST input data options:

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

6.1.10.1 westpa.cli.core.w_succ module

class westpa.cli.core.w_succ.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: `object`

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.cli.core.w_succ.WESTAnalysisTool

Bases: object

add_args(*parser*, *upcall=True*)

Add arguments to a parser common to all analyses of this type.

process_args(*args*, *upcall=True*)

open_analysis_backing()

close_analysis_backing()

require_analysis_group(*groupname*, *replace=False*)

class westpa.cli.core.w_succ.WESTDataReaderMixin

Bases: *AnalysisMixin*

A mixin for analysis requiring access to the HDF5 files generated during a WEST run.

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

clear_run_cache()

property cache_pcoords

Whether or not to cache progress coordinate data. While caching this data can significantly speed up some analysis operations, this requires copious RAM.

Setting this to False when it was formerly True will release any cached data.

get_summary_table()

get_iter_group(*n_iter*)

Return the HDF5 group corresponding to `n_iter`

get_segments(*n_iter*, *include_pcoords=True*)

Return all segments present in iteration n_iter

get_segments_by_id(*n_iter*, *seg_ids*, *include_pcoords=True*)

Get segments from the data manager, employing caching where possible

get_children(*segment*, *include_pcoords=True*)

get_seg_index(*n_iter*)

get_wtg_parent_array(*n_iter*)

get_parent_array(*n_iter*)

get_pcoord_array(*n_iter*)

get_pcoord_dataset(*n_iter*)

get_pcoords(*n_iter*, *seg_ids*)

get_seg_ids(*n_iter*, *bool_array=None*)

get_created_seg_ids(*n_iter*)

Return a list of seg_ids corresponding to segments which were created for the given iteration (are not continuations).

max_iter_segs_in_range(*first_iter*, *last_iter*)

Return the maximum number of segments present in any iteration in the range selected

total_segs_in_range(*first_iter*, *last_iter*)

Return the total number of segments present in all iterations in the range selected

get_pcoord_len(*n_iter*)

Get the length of the progress coordinate array for the given iteration.

get_total_time(*first_iter=None*, *last_iter=None*, *dt=None*)

Return the total amount of simulation time spanned between first_iter and last_iter (inclusive).

class westpa.cli.core.w_succ.**CommonOutputMixin**

Bases: *AnalysisMixin*

add_common_output_args(*parser_or_group*)

process_common_output_args(*args*)

class westpa.cli.core.w_succ.**WSucc**

Bases: *CommonOutputMixin*, *WESTDataReaderMixin*, *WESTAnalysisTool*

find_successful_trajs()

westpa.cli.core.w_succ.**entry_point**()

6.1.11 w_crawl

usage:

```
w_crawl [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE] [--first-iter N_
↪ITER]
              [--last-iter N_ITER] [-c CRAWLER_INSTANCE]
              [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
              [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
              [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
              [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
              [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
              [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
              [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
              task_callable
```

Crawl a weighted ensemble dataset, executing a function for each iteration. This can be used for postprocessing of trajectories, cleanup of datasets, or anything else that can be expressed as "do X for iteration N, then do something with the result". Tasks are parallelized by iteration, and no guarantees are made about evaluation order.

6.1.11.1 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks
                      that have very large requests/response. Default: no limit.
```

WEST input data options:

```
-W WEST_H5FILE, --west-data WEST_H5FILE
                      Take WEST data from WEST_H5FILE (default: read from the HDF5 file␣
```

(continues on next page)

(continued from previous page)

```
↪specified in
                      west.cfg).
```

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

task options:

```
-c CRAWLER_INSTANCE, --crawler-instance CRAWLER_INSTANCE
                      Use CRAWLER_INSTANCE (specified as module.instance) as an instance␣
↪of
                      WESTPACrawler to coordinate the calculation. Required only if␣
↪initialization,
                      finalization, or task result processing is required.
task_callable         Run TASK_CALLABLE (specified as module.function) on each iteration.
↪ Required.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is
↪'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do␣
↪not support
                      this option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a
                      deprecated synonym for "master" and "client" is a deprecated␣
↪synonym for
                      "node".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g.
                      /tmp); on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
```

(continues on next page)

(continued from previous page)

```
↪instance in
                        INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                        communication of other nodes to choose ports randomly. Downstream␣
↪nodes read
                        this file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                        Read hostname and port information needed to connect to the master␣
↪(or other
                        coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting
                        in coordinating the communication of other nodes to choose ports␣
↪randomly,
                        writing that information with --zmq-write-host-info for this␣
↪instance to read.
--zmq-upstream-rr-endpoint ENDPOINT
                        ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic
                        toward the master.
--zmq-upstream-ann-endpoint ENDPOINT
                        ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                        notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                        ZeroMQ endpoint on which to listen for request/response (task and␣
↪result)
                        traffic from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                        ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                        notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                        Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                        Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                        Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker
                        in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed.␣
↪If a worker
                        doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                        assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                        Amount of time (in seconds) to wait for communication between the␣
↪master and at
                        least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                        computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
```

(continues on next page)

(continued from previous page)

```
                    Amount of time (in seconds) to wait for workers to shut down.
```

6.1.11.2 westpa.cli.tools.w_crawl module

class westpa.cli.tools.w_crawl.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_crawl.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_crawl.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
The first iteration included in the calculation.
last_iter
One past the last iteration included in the calculation.
iter_step
Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.cli.tools.w_crawl.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.cli.tools.w_crawl.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

class westpa.cli.tools.w_crawl.**WESTPACrawler**

Bases: object

Base class for general crawling execution. This class only exists on the master.

initialize(*iter_start*, *iter_stop*)

Initialize this crawling process.

finalize()

Finalize this crawling process.

process_iter_result(*n_iter*, *result*)

Process the result of a per-iteration task.

class westpa.cli.tools.w_crawl.**WCrawl**

Bases: *WESTParallelTool*

prog = 'w_crawl'

description = 'Crawl a weighted ensemble dataset, executing a function for each iteration.\nThis can be used for postprocessing of trajectories, cleanup of datasets,\nor anything else that can be expressed as "do X for iteration N, then do\nsomething with the result". Tasks are parallelized by iteration, and\nno guarantees are made about evaluation order.\n\n\n--\nCommand-line options\n--\n\n'

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_crawl.**entry_point**()

6.1.12 w_direct

usage:

```
w_direct [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--max-queue-length MAX_QUEUE_LENGTH]
                [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
                [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
                [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
                [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
                [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
                [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
                [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
                {help,init,average,kinetics,probs,all} ...
```

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

direct kinetics analysis schemes:

```
{help,init,average,kinetics,probs,all}
  help                print help for this command or individual subcommands
  init                calculate state-to-state kinetics by tracing trajectories
  average             Averages and returns fluxes, rates, and color/state populations.
  kinetics            Generates rate and flux values from a WESTPA simulation via␣
↪tracing.
  probs               Calculates color and state probabilities via tracing.
  all                 Runs the full suite, including the tracing of events.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ (“zmq”) work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
```

(continues on next page)

(continued from previous page)

```
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.1.12.1 westpa.cli.tools.w_direct module

westpa.cli.tools.w_direct.**weight_dtype**

alias of float64

class westpa.cli.tools.w_direct.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_direct.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_direct.**sequence_macro_flux_to_rate**(*dataset*, *pops*, *istate*, *jstate*, *pairwise=True*, *stride=None*)

Convert a sequence of macrostate fluxes and corresponding list of trajectory ensemble populations to a sequence of rate matrices.

If the optional `pairwise` is true (the default), then rates are normalized according to the relative probability of the initial state among the pair of states (initial, final); this is probably what you want, as these rates will then depend only on the definitions of the states involved (and never the remaining states). Otherwise (``pairwise" is false), the rates are normalized according the probability of the initial state among *all* other states.

class westpa.cli.tools.w_direct.**WKinetics**

Bases: `object`

w_kinetics()

class westpa.cli.tools.w_direct.**WESTKineticsBase**(*parent*)

Bases: *WESTSubcommand*

Common argument processing for w_direct/w_reweight subcommands. Mostly limited to handling input and output from w_assign.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_direct.**AverageCommands**(*parent*)

Bases: *WESTKineticsBase*

default_output_file = 'direct.h5'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

stamp_mcbs_info(*dataset*)

open_files()

open_assignments()

print_averages(*dataset*, *header*, *dim=1*)

run_calculation(*pi*, *nstates*, *start_iter*, *stop_iter*, *step_iter*, *dataset*, *eval_block*, *name*, *dim*, *do_averages=False*, ***extra*)

westpa.cli.tools.w_direct.**mcbs_ci_correl**(*estimator_datasets*, *estimator*, *alpha*, *n_sets=None*, *args=None*, *autocorrel_alpha=None*, *autocorrel_n_sets=None*, *subsample=None*, *do_correl=True*, *mcbs_enable=None*, *estimator_kwargs={}*)

Perform a Monte Carlo bootstrap estimate for the (1-alpha) confidence interval on the given dataset with the given estimator. This routine is appropriate for time-correlated data, using the method described in Huber & Kim, "Weighted-ensemble Brownian dynamics simulations for protein association reactions" (1996), doi:10.1016/S0006-3495(96)79552-8 to determine a statistically-significant correlation time and then reducing the dataset by a factor of that correlation time before running a "classic" Monte Carlo bootstrap.

Returns (estimate, ci_lb, ci_ub, correl_time) where estimate is the application of the given estimator to the input dataset, ci_lb and ci_ub are the lower and upper limits, respectively, of the (1-alpha) confidence interval on estimate, and correl_time is the correlation time of the dataset, significant to (1-autocorrel_alpha).

estimator is called as estimator(dataset, *args, **kwargs). Common estimators include:

- np.mean – calculate the confidence interval on the mean of dataset
- np.median – calculate a confidence interval on the median of dataset
- np.std – calculate a confidence interval on the standard deviation of datset.

n_sets is the number of synthetic data sets to generate using the given estimator, which will be chosen using `get_bssize()`_ if n_sets is not given.

autocorrel_alpha (which defaults to alpha) can be used to adjust the significance level of the autocorrelation calculation. Note that too high a significance level (too low an alpha) for evaluating the significance of autocorrelation values can result in a failure to detect correlation if the autocorrelation function is noisy.

The given subsample function is used, if provided, to subsample the dataset prior to running the full Monte Carlo bootstrap. If none is provided, then a random entry from each correlated block is used as the value for that block. Other reasonable choices include np.mean, np.median, (lambda x: x[0]) or (lambda x: x[-1]). In particular, using subsample=np.mean will converge to the block averaged mean and standard error, while accounting for any non-normality in the distribution of the mean.

westpa.cli.tools.w_direct.**accumulate_state_populations_from_labeled**(*labeled_bin_pops*, *state_map*, *state_pops*, *check_state_map=True*)

class westpa.cli.tools.w_direct.**DKinetics**(*parent*)

Bases: *WESTKineticsBase*, *WKinetics*

subcommand = 'init'

default_kinetics_file = 'direct.h5'

default_output_file = 'direct.h5'

help_text = 'calculate state-to-state kinetics by tracing trajectories'

description = 'Calculate state-to-state rates and transition event durations by
tracing\ntrajectories.\n\nA bin assignment file (usually "assign.h5") including
trajectory labeling\nis required (see "w_assign --help" for information on
generating this file).\n\nThis subcommand for w_direct is used as input for all
other w_direct\nsubcommands, which will convert the flux data in the output file
into\naverage rates/fluxes/populations with confidence intervals.\n\
n---\
nOutput
format\
n---\n\
nThe output file (-o/--output, by default "direct.h5") contains the\nfollowing
datasets:\n\n ``/conditional_fluxes`` [iteration][state][state]\n *(Floating-point)*
Macrostate-to-macrostate fluxes. These are **not**\n normalized by the population of
the initial macrostate.\n\n ``/conditional_arrivals`` [iteration][stateA][stateB]\n
(Integer) Number of trajectories arriving at state *stateB* in a given\n
iteration, given that they departed from *stateA*.\n\n ``/total_fluxes``
[iteration][state]\n *(Floating-point)* Total flux into a given macrostate.\n\n
``/arrivals`` [iteration][state]\n *(Integer)* Number of trajectories arriving at a
given state in a given\n iteration, regardless of where they originated.\n\n
``/duration_count`` [iteration]\n *(Integer)* The number of event durations recorded
in each iteration.\n\n ``/durations`` [iteration][event duration]\n *(Structured --
see below)* Event durations for transition events ending\n during a given iteration.
These are stored as follows:\n\n istate\n *(Integer)* Initial state of transition
event.\n fstate\n *(Integer)* Final state of transition event.\n duration\n
(Floating-point) Duration of transition, in units of tau.\n weight\n
(Floating-point) Weight of trajectory at end of transition, **not**\n normalized
by initial state population.\n\nBecause state-to-state fluxes stored in this file
are not normalized by\ninitial macrostate population, they cannot be used as rates
without further\nprocessing. The ``w_direct kinetics`` command is used to perform
this normalization\nwhile taking statistical fluctuation and correlation into
account. See\n``w_direct kinetics --help`` for more information. Target fluxes
(total flux\ninto a given state) require no such normalization.\n\
n---\
nCommand-line
options\
n---\n'

open_files()

go()

class westpa.cli.tools.w_direct.DKinAvg(*parent*)

Bases: *AverageCommands*

subcommand = 'kinetics'

help_text = 'Generates rate and flux values from a WESTPA simulation via tracing.'

```
default_kinetics_file = 'direct.h5'

description = 'Calculate average rates/fluxes and associated errors from weighted
ensemble\ndata. Bin assignments (usually "assign.h5") and kinetics data
(usually\n"direct.h5") data files must have been previously generated
(see\n"w_assign --help" and "w_direct init --help" for information on\ngenerating
these files).\n\nThe evolution of all datasets may be calculated, with or without
confidence\nintervals.\n\
n-------------------------------------------------------------------------------\
nOutput
format\
n-------------------------------------------------------------------------------\n\
nThe output file (-o/--output, usually "direct.h5") contains the
following\ndataset:\n\n /avg_rates [state,state]\n (Structured -- see below)
State-to-state rates based on entire window of\n iterations selected.\n\n
/avg_total_fluxes [state]\n (Structured -- see below) Total fluxes into each state
based on entire\n window of iterations selected.\n\n /avg_conditional_fluxes
[state,state]\n (Structured -- see below) State-to-state fluxes based on entire
window of\n iterations selected.\n\nIf --evolution-mode is specified, then the
following additional datasets are\navailable:\n\n /rate_evolution
[window][state][state]\n (Structured -- see below). State-to-state rates based on
windows of\n iterations of varying width. If --evolution-mode=cumulative, then\n
these windows all begin at the iteration specified with\n --start-iter and grow in
length by --step-iter for each successive\n element. If --evolution-mode=blocked,
then these windows are all of\n width --step-iter (excluding the last, which may be
shorter), the first\n of which begins at iteration --start-iter.\n\n
/target_flux_evolution [window,state]\n (Structured -- see below). Total flux into a
given macro state based on\n windows of iterations of varying width, as in
/rate_evolution.\n\n /conditional_flux_evolution [window,state,state]\n (Structured
-- see below). State-to-state fluxes based on windows of\n varying width, as in
/rate_evolution.\n\nThe structure of these datasets is as follows:\n\n iter_start\n
(Integer) Iteration at which the averaging window begins (inclusive).\n\n
iter_stop\n (Integer) Iteration at which the averaging window ends (exclusive).\n\n
expected\n (Floating-point) Expected (mean) value of the observable as evaluated
within\n this window, in units of inverse tau.\n\n ci_lbound\n (Floating-point)
Lower bound of the confidence interval of the observable\n within this window, in
units of inverse tau.\n\n ci_ubound\n (Floating-point) Upper bound of the confidence
interval of the observable\n within this window, in units of inverse tau.\n\n
stderr\n (Floating-point) The standard error of the mean of the observable\n within
this window, in units of inverse tau.\n\n corr_len\n (Integer) Correlation length of
the observable within this window, in units\n of tau.\n\nEach of these datasets is
also stamped with a number of attributes:\n\n mcbs_alpha\n (Floating-point) Alpha
value of confidence intervals. (For example,\n *alpha=0.05* corresponds to a 95%
confidence interval.)\n\n mcbs_nsets\n (Integer) Number of bootstrap data sets used
in generating confidence\n intervals.\n\n mcbs_acalpha\n (Floating-point) Alpha
value for determining correlation lengths.\n\n\
n-------------------------------------------------------------------------------\
nCommand-line
options\
n-------------------------------------------------------------------------------\n'

w_kinavg()

go()
```

class westpa.cli.tools.w_direct.**DStateProbs**(*parent*)

Bases: *AverageCommands*

subcommand = 'probs'

help_text = 'Calculates color and state probabilities via tracing.'

default_kinetics_file = 'direct.h5'

description = 'Calculate average populations and associated errors in state
populations from\nweighted ensemble data. Bin assignments, including macrostate
definitions,\nare required. (See "w_assign --help" for more information).\n\
n---\
nOutput
format\
n---\n\
nThe output file (-o/--output, usually "direct.h5") contains the
following\ndataset:\n\n /avg_state_probs [state]\n (Structured -- see below)
Population of each state across entire\n range specified.\n\n /avg_color_probs
[state]\n (Structured -- see below) Population of each ensemble across entire\n
range specified.\n\nIf --evolution-mode is specified, then the following additional
datasets are\navailable:\n\n /state_pop_evolution [window][state]\n (Structured --
see below). State populations based on windows of\n iterations of varying width. If
--evolution-mode=cumulative, then\n these windows all begin at the iteration
specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\n /color_prob_evolution [window][state]\n
(Structured -- see below). Ensemble populations based on windows of\n iterations of
varying width. If --evolution-mode=cumulative, then\n these windows all begin at the
iteration specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\nThe structure of these datasets is as
follows:\n\n iter_start\n (Integer) Iteration at which the averaging window begins
(inclusive).\n\n iter_stop\n (Integer) Iteration at which the averaging window ends
(exclusive).\n\n expected\n (Floating-point) Expected (mean) value of the observable
as evaluated within\n this window, in units of inverse tau.\n\n ci_lbound\n
(Floating-point) Lower bound of the confidence interval of the observable\n within
this window, in units of inverse tau.\n\n ci_ubound\n (Floating-point) Upper bound
of the confidence interval of the observable\n within this window, in units of
inverse tau.\n\n stderr\n (Floating-point) The standard error of the mean of the
observable\n within this window, in units of inverse tau.\n\n corr_len\n (Integer)
Correlation length of the observable within this window, in units\n of tau.\n\nEach
of these datasets is also stamped with a number of attributes:\n\n mcbs_alpha\n
(Floating-point) Alpha value of confidence intervals. (For example,\n *alpha=0.05*
corresponds to a 95% confidence interval.)\n\n mcbs_nsets\n (Integer) Number of
bootstrap data sets used in generating confidence\n intervals.\n\n mcbs_acalpha\n
(Floating-point) Alpha value for determining correlation lengths.\n\n\
n---\
nCommand-line
options\
n---\n'

calculate_state_populations(*pops*)

w_stateprobs()

go()

class westpa.cli.tools.w_direct.DAll(*parent*)

Bases: *DStateProbs*, *DKinAvg*, *DKinetics*

subcommand = 'all'

help_text = 'Runs the full suite, including the tracing of events.'

default_kinetics_file = 'direct.h5'

description = 'A convenience function to run init/kinetics/probs. Bin
assignments,\nincluding macrostate definitions, are required. (See\n"w_assign
--help" for more information).\n\nFor more information on the individual subcommands
this subs in for, run\nw_direct {init/kinetics/probs} --help.\n\
n---\
nCommand-line
options\
n---\n'

go()

class westpa.cli.tools.w_direct.DAverage(*parent*)

Bases: *DStateProbs*, *DKinAvg*

subcommand = 'average'

help_text = 'Averages and returns fluxes, rates, and color/state populations.'

default_kinetics_file = 'direct.h5'

description = 'A convenience function to run kinetics/probs. Bin
assignments,\nincluding macrostate definitions, are required. (See\n"w_assign
--help" for more information).\n\nFor more information on the individual subcommands
this subs in for, run\nw_direct {kinetics/probs} --help.\n\
n---\
nCommand-line
options\
n---\n'

go()

class westpa.cli.tools.w_direct.WDirect

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_direct'

subcommands = [<class 'westpa.cli.tools.w_direct.DKinetics'>, <class
'westpa.cli.tools.w_direct.DAverage'>, <class 'westpa.cli.tools.w_direct.DKinAvg'>,
<class 'westpa.cli.tools.w_direct.DStateProbs'>, <class
'westpa.cli.tools.w_direct.DAll'>]

subparsers_title = 'direct kinetics analysis schemes'

westpa.cli.tools.w_direct.entry_point()

6.1.13 w_select

usage:

```
w_select [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE] [--first-iter N_
↪ITER]
                [--last-iter N_ITER] [-p MODULE.FUNCTION] [-v] [-a] [-o OUTPUT]
                [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
                [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
                [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
                [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
                [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
                [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
                [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

Select dynamics segments matching various criteria. This requires a user-provided prediate function. By default, only matching segments are stored. If the -a/–include-ancestors option is given, then matching segments and their ancestors will be stored.

6.1.13.1 Predicate function

Segments are selected based on a predicate function, which must be callable as `predicate(n_iter, iter_group)` and return a collection of segment IDs matching the predicate in that iteration.

The predicate may be inverted by specifying the -v/–invert command-line argument.

6.1.13.2 Output format

The output file (-o/–output, by default “select.h5”) contains the following datasets:

```
``/n_iter`` [iteration]
  *(Integer)* Iteration numbers for each entry in other datasets.

``/n_segs`` [iteration]
  *(Integer)* Number of segment IDs matching the predicate (or inverted
  predicate, if -v/--invert is specified) in the given iteration.

``/seg_ids`` [iteration][segment]
  *(Integer)* Matching segments in each iteration. For an iteration
  ``n_iter``, only the first ``n_iter`` entries are valid. For example,
  the full list of matching seg_ids in the first stored iteration is
  ``seg_ids[0][:n_segs[0]]``.

``/weights`` [iteration][segment]
  *(Floating-point)* Weights for each matching segment in ``/seg_ids``.
```

6.1.13.3 Command-line arguments

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

selection options:

```
-p MODULE.FUNCTION, --predicate-function MODULE.FUNCTION
                      Use the given predicate function to match segments. This function␣
↪should take an
                      iteration number and the HDF5 group corresponding to that␣
↪iteration and return a
                      sequence of seg_ids matching the predicate, as in ``match_
↪predicate(n_iter,
                      iter_group)``.
-v, --invert          Invert the match predicate.
-a, --include-ancestors
                      Include ancestors of matched segments in output.
```

output options:

-o OUTPUT, --output OUTPUT Write output to OUTPUT (default: select.h5).

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ (“zmq”) work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
```

(continues on next page)

(continued from previous page)

```
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.1.13.4 westpa.cli.tools.w_select module

class westpa.cli.tools.w_select.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_select.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_select.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
The first iteration included in the calculation.

last_iter
One past the last iteration included in the calculation.

iter_step
Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.cli.tools.w_select.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.cli.tools.w_select.**seg_id_dtype**

alias of `int64`

westpa.cli.tools.w_select.**n_iter_dtype**

alias of `uint32`

westpa.cli.tools.w_select.**weight_dtype**

alias of `float64`

westpa.cli.tools.w_select.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

class westpa.cli.tools.w_select.**WSelectTool**

Bases: *WESTParallelTool*

prog = 'w_select'

```
description = 'Select dynamics segments matching various criteria. This requires
a\nuser-provided prediate function. By default, only matching segments are\nstored.
If the -a/--include-ancestors option is given, then matching segments\nand their
ancestors will be stored.\n\n\
n-------------------------------------------------------------------------------\
nPredicate
function\
n-------------------------------------------------------------------------------\n\
nSegments are selected based on a predicate function, which must be callable\nas
``predicate(n_iter, iter_group)`` and return a collection of segment IDs\nmatching
the predicate in that iteration.\n\nThe predicate may be inverted by specifying the
-v/--invert command-line\nargument.\n\n\
n-------------------------------------------------------------------------------\
nOutput
format\
n-------------------------------------------------------------------------------\n\
nThe output file (-o/--output, by default "select.h5") contains the
following\ndatasets:\n\n ``/n_iter`` [iteration]\n *(Integer)* Iteration numbers for
each entry in other datasets.\n\n ``/n_segs`` [iteration]\n *(Integer)* Number of
segment IDs matching the predicate (or inverted\n predicate, if -v/--invert is
specified) in the given iteration.\n\n ``/seg_ids`` [iteration][segment]\n
*(Integer)* Matching segments in each iteration. For an iteration\n ``n_iter``, only
the first ``n_iter`` entries are valid. For example,\n the full list of matching
seg_ids in the first stored iteration is\n ``seg_ids[0][:n_segs[0]]``.\n\n
``/weights`` [iteration][segment]\n *(Floating-point)* Weights for each matching
segment in ``/seg_ids``.\n\n\
n-------------------------------------------------------------------------------\
nCommand-line
arguments\
n-------------------------------------------------------------------------------\n'
```

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_select.**entry_point**()

6.1.14 w_states

usage:

```
w_states [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--show | --append | --replace] [--bstate-file BSTATE_FILE] [--bstate␣
↪BSTATES]
                [--tstate-file TSTATE_FILE] [--tstate TSTATES]
                [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
```

(continues on next page)

(continued from previous page)

```
                [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
                [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
                [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
                [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
                [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
                [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

Display or manipulate basis (initial) or target (recycling) states for a WEST simulation. By default, states are displayed (or dumped to files). If `--replace` is specified, all basis/target states are replaced for the next iteration. If `--append` is specified, the given target state(s) are appended to the list for the next iteration. Appending basis states is not permitted, as this would require renormalizing basis state probabilities in ways that may be error-prone. Instead, use `w_states --show --bstate-file=bstates.txt` and then edit the resulting `bstates.txt` file to include the new desired basis states, then use `w_states --replace --bstate-file=bstates.txt` to update the WEST HDF5 file appropriately.

optional arguments:

```
-h, --help            show this help message and exit
--bstate-file BSTATE_FILE
                      Read (--append/--replace) or write (--show) basis state names,␣
↪probabilities, and
                      data references from/to BSTATE_FILE.
--bstate BSTATES      Add the given basis state (specified as a string 'label,
↪probability[,auxref]') to
                      the list of basis states (after those specified in --bstate-file,␣
↪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.
--tstate-file TSTATE_FILE
                      Read (--append/--replace) or write (--show) target state names and␣
↪representative
                      progress coordinates from/to TSTATE_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.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
```

(continues on next page)

(continued from previous page)

```
--version             show program's version number and exit
```

modes of operation:

```
--show                Display current basis/target states (or dump to files).
--append              Append the given basis/target states to those currently in use.
--replace             Replace current basis/target states with those specified.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
```

(continues on next page)

(continued from previous page)

```
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.1.14.1 westpa.cli.core.w_states module

`westpa.cli.core.w_states.`**make_work_manager**`()`

Using cues from the environment, instantiate a pre-configured work manager.

class `westpa.cli.core.w_states.`**Segment**`(`*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*`)`

Bases: `object`

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial

state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.cli.core.w_states.**BasisState**(*label*, *probability*, *pcoord=None*, *auxref=None*, *state_id=None*)

Bases: object

Describes an basis (micro)state. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation (i.e. at w_init) or due to recycling.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **probability** – Probability of this state to be selected when creating a new trajectory.
- **pcoord** – The representative progress coordinate of this state.
- **auxref** – A user-provided (string) reference for locating data associated with this state (usually a filesystem path).

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*)

Read a file defining basis states. Each line 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
```

as_numpy_record()

Return the data for this state as a numpy record array.

class westpa.cli.core.w_states.**TargetState**(*label*, *pcoord*, *state_id=None*)

Bases: object

Describes a target state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **pcoord** – The representative progress coordinate of this state.

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*, *dtype*)

Read a file defining target states. 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.

westpa.cli.core.w_states.**entry_point**()

westpa.cli.core.w_states.**initialize**(*mode*, *bstates*, *_bstate_file*, *tstates*, *_tstate_file*)

6.1.15 w_eddist

usage:

```
w_eddist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--max-queue-length MAX_QUEUE_LENGTH] [-b BINEXPR] [-C] [--loose] --
↪istate ISTATE
                --fstate FSTATE [--first-iter ITER_START] [--last-iter ITER_STOP] [-k␣
↪KINETICS]
                [-o OUTPUT] [--serial | --parallel | --work-manager WORK_MANAGER]
                [--n-workers N_WORKERS] [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE]
                [--zmq-write-host-info INFO_FILE] [--zmq-read-host-info INFO_FILE]
                [--zmq-upstream-rr-endpoint ENDPOINT] [--zmq-upstream-ann-endpoint␣
↪ENDPOINT]
                [--zmq-downstream-rr-endpoint ENDPOINT] [--zmq-downstream-ann-endpoint␣
↪ENDPOINT]
                [--zmq-master-heartbeat MASTER_HEARTBEAT] [--zmq-worker-heartbeat WORKER_
↪HEARTBEAT]
                [--zmq-timeout-factor FACTOR] [--zmq-startup-timeout STARTUP_TIMEOUT]
                [--zmq-shutdown-timeout SHUTDOWN_TIMEOUT]
```

Calculate time-resolved transition-event duration distribution from kinetics results

6.1.15.1 Source data

Source data is collected from the results of 'w_kinetics trace' (see w_kinetics trace –help for more information on generating this dataset).

6.1.15.2 Histogram binning

By default, histograms are constructed with 100 bins in each dimension. This can be overridden by specifying -b/–bins, which accepts a number of different kinds of arguments:

```
a single integer N
  N uniformly spaced bins will be used in each dimension.

a sequence of integers N1,N2,... (comma-separated)
  N1 uniformly spaced bins will be used for the first dimension, N2 for the
  second, and so on.

a list of lists [[B11, B12, B13, ...], [B21, B22, B23, ...], ...]
  The bin boundaries B11, B12, B13, ... will be used for the first dimension,
  B21, B22, B23, ... for the second dimension, and so on. These bin
  boundaries need not be uniformly spaced. These expressions will be
  evaluated with Python's ``eval`` construct, with ``np`` available for
  use [e.g. to specify bins using np.arange()].
```

The first two forms (integer, list of integers) will trigger a scan of all data in each dimension in order to determine the minimum and maximum values, which may be very expensive for large datasets. This can be avoided by explicitly providing bin boundaries using the list-of-lists form.

Note that these bins are *NOT* at all related to the bins used to drive WE sampling.

6.1.15.3 Output format

The output file produced (specified by -o/–output, defaulting to "pdist.h5") may be fed to plothist to generate plots (or appropriately processed text or HDF5 files) from this data. In short, the following datasets are created:

```
``histograms``
  Normalized histograms. The first axis corresponds to iteration, and
  remaining axes correspond to dimensions of the input dataset.

``/binbounds_0``
  Vector of bin boundaries for the first (index 0) dimension. Additional
  datasets similarly named (/binbounds_1, /binbounds_2, ...) are created
  for additional dimensions.

``/midpoints_0``
  Vector of bin midpoints for the first (index 0) dimension. Additional
  datasets similarly named are created for additional dimensions.

``n_iter``
  Vector of iteration numbers corresponding to the stored histograms (i.e.
  the first axis of the ``histograms`` dataset).
```

6.1.15.4 Subsequent processing

The output generated by this program (-o/–output, default "pdist.h5") may be plotted by the `plothist` program. See `plothist --help` for more information.

6.1.15.5 Parallelization

This tool supports parallelized binning, including reading of input data. Parallel processing is the default. For simple cases (reading pre-computed input data, modest numbers of segments), serial processing (–serial) may be more efficient.

6.1.15.6 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
-b BINEXPR, --bins BINEXPR
                      Use BINEXPR for bins. This may be an integer, which will be used␣
↪for each
                      dimension of the progress coordinate; a list of integers␣
↪(formatted as
                      [n1,n2,...]) which will use n1 bins for the first dimension, n2␣
↪for the second
                      dimension, and so on; or a list of lists of boundaries (formatted␣
```

(continues on next page)

(continued from previous page)

```
↪as [[a1, a2,
                      ...], [b1, b2, ...], ... ]), which will use [a1, a2, ...] as bin␣
↪boundaries for
                      the first dimension, [b1, b2, ...] as bin boundaries for the␣
↪second dimension,
                      and so on. (Default: 100 bins in each dimension.)
-C, --compress        Compress histograms. May make storage of higher-dimensional␣
↪histograms more
                      tractable, at the (possible extreme) expense of increased analysis␣
↪time.
                      (Default: no compression.)
--loose               Ignore values that do not fall within bins. (Risky, as this can␣
↪make buggy bin
                      boundaries appear as reasonable data. Only use if you are sure of␣
↪your bin
                      boundary specification.)
--istate ISTATE       Initial state defining transition event
--fstate FSTATE       Final state defining transition event
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

--max-queue-length MAX_QUEUE_LENGTH Maximum number of tasks that can be queued. Useful to limit RAM use for tasks that have very large requests/response. Default: no limit.

iteration range options:

```
--first-iter ITER_START
                      Iteration to begin analysis (default: 1)
--last-iter ITER_STOP
                      Iteration to end analysis
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates (including evolution) are stored␣
↪in KINETICS
                      (default: kintrace.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: eddist.h5).
```

parallelization options:

--serial run in serial mode

--parallel run in parallel mode (using processes)

--work-manager WORK_MANAGER use the given work manager for parallel task distribution. Available work managers are ('serial', 'threads', 'processes', 'zmq'); default is 'processes'

--n-workers N_WORKERS Use up to N_WORKERS on this host, for work managers which support this option. Use 0 for a dedicated server. (Ignored by work managers which do not support this option.)

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a
                      deprecated synonym for "master" and "client" is a deprecated␣
↪synonym for
                      "node".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g.
                      /tmp); on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read
                      this file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting
                      in coordinating the communication of other nodes to choose ports␣
↪randomly,
                      writing that information with --zmq-write-host-info for this␣
↪instance to read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic
                      toward the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result)
                      traffic from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
```

(continues on next page)

(continued from previous page)

```
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker
                      in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed.␣
↪If a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.1.15.7 westpa.cli.tools.w_eddist module

class westpa.cli.tools.w_eddist.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_eddist.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.cli.tools.w_eddist.**histnd**(*values*, *binbounds*, *weights=1.0*, *out=None*, *binbound_check=True*, *ignore_out_of_range=False*)

Generate an N-dimensional PDF (or contribution to a PDF) from the given values. `binbounds` is a list of arrays of boundary values, with one entry for each dimension (`values` must have as many columns as there are entries in `binbounds`) `weight`, if provided, specifies the weight each value contributes to the histogram; this may be a scalar (for equal weights for all values) or a vector of the same length as `values` (for unequal weights). If `binbound_check` is True, then the boundaries are checked for strict positive monotonicity; set to False to shave a few microseconds if you know your bin boundaries to be monotonically increasing.

westpa.cli.tools.w_eddist.**normhistnd**(*hist*, *binbounds*)

Normalize the N-dimensional histogram `hist` with corresponding bin boundaries `binbounds`. Modifies `hist` in place and returns the normalization factor used.

class westpa.cli.tools.w_eddist.**DurationDataset**(*dataset*, *mask*, *iter_start=1*)

Bases: `object`

A facade for the 'dsspec' dataclass that incorporates the mask into get_iter_data method

get_iter_data(*n_iter*)

westpa.cli.tools.w_eddist.**isiterable**(*x*)

class westpa.cli.tools.w_eddist.**WEDDist**

Bases: *WESTParallelTool*

prog = 'w_eddist'

```
    description = 'Calculate time-resolved transition-event duration distribution from
    kinetics results\n\n\
    n-----------------------------------------------------------------------------\
    nSource
    data\n-----------------------------------------------------------------------------\
    n\nSource data is collected from the results of \'w_kinetics trace\' (see w_kinetics
    trace --help for\nmore information on generating this dataset).\n\n\
    n-----------------------------------------------------------------------------\
    nHistogram
    binning\
    n-----------------------------------------------------------------------------\n\nBy
    default, histograms are constructed with 100 bins in each dimension. This\ncan be
    overridden by specifying -b/--bins, which accepts a number of different\nkinds of
    arguments:\n\n a single integer N\n N uniformly spaced bins will be used in each
    dimension.\n\n a sequence of integers N1,N2,... (comma-separated)\n N1 uniformly
    spaced bins will be used for the first dimension, N2 for the\n second, and so
    on.\n\n a list of lists [[B11, B12, B13, ...], [B21, B22, B23, ...], ...]\n The bin
    boundaries B11, B12, B13, ... will be used for the first dimension,\n B21, B22, B23,
    ... for the second dimension, and so on. These bin\n boundaries need not be
    uniformly spaced. These expressions will be\n evaluated with Python\'s ``eval``
    construct, with ``np`` available for\n use [e.g. to specify bins using
    np.arange()].\n\nThe first two forms (integer, list of integers) will trigger a scan
    of all\ndata in each dimension in order to determine the minimum and maximum
    values,\nwhich may be very expensive for large datasets. This can be avoided
    by\nexplicitly providing bin boundaries using the list-of-lists form.\n\nNote that
    these bins are *NOT* at all related to the bins used to drive WE\nsampling.\n\n\
    n-----------------------------------------------------------------------------\
    nOutput
    format\
    n-----------------------------------------------------------------------------\n\
    nThe output file produced (specified by -o/--output, defaulting to "pdist.h5")\nmay
    be fed to plothist to generate plots (or appropriately processed text or\nHDF5
    files) from this data. In short, the following datasets are created:\n\n
    ``histograms``\n Normalized histograms. The first axis corresponds to iteration,
    and\n remaining axes correspond to dimensions of the input dataset.\n\n
    ``/binbounds_0``\n Vector of bin boundaries for the first (index 0) dimension.
    Additional\n datasets similarly named (/binbounds_1, /binbounds_2, ...) are
    created\n for additional dimensions.\n\n ``/midpoints_0``\n Vector of bin midpoints
    for the first (index 0) dimension. Additional\n datasets similarly named are created
    for additional dimensions.\n\n ``n_iter``\n Vector of iteration numbers
    corresponding to the stored histograms (i.e.\n the first axis of the ``histograms``
    dataset).\n\n\
    n-----------------------------------------------------------------------------\
    nSubsequent
    processing\
    n-----------------------------------------------------------------------------\n\
    nThe output generated by this program (-o/--output, default "pdist.h5") may
    be\nplotted by the ``plothist`` program. See ``plothist --help`` for
    more\ninformation.\n\n\
    n-----------------------------------------------------------------------------\
    nParallelization\
    n-----------------------------------------------------------------------------\n\
    nThis tool supports parallelized binning, including reading of input data.\nParallel
    processing is the default. For simple cases (reading pre-computed\ninput data,
    modest numbers of segments), serial processing (--serial) may be\nmore efficient.\n\
    n\n-----------------------------------------------------------------------------\
    nCommand-line
    options\
    n-----------------------------------------------------------------------------\n\n'
```

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

static parse_binspec(*binspec*)

construct_bins(*bins*)

Construct bins according to `bins`, which may be:
1) A scalar integer (for that number of bins in each dimension)
2) A sequence of integers (specifying number of bins for each dimension)
3) A sequence of sequences of bin boundaries (specifying boundaries for each dimension)

Sets `self.binbounds` to a list of arrays of bin boundaries appropriate for passing to fasthist.histnd, along with `self.midpoints` to the midpoints of the bins.

scan_data_shape()

scan_data_range()

Scan input data for range in each dimension. The number of dimensions is determined from the shape of the progress coordinate as of self.iter_start.

construct_histogram()

Construct a histogram using bins previously constructed with `construct_bins()`. The time series of histogram values is stored in `histograms`. Each histogram in the time series is normalized.

`westpa.cli.tools.w_eddist.`**entry_point**()

6.1.16 w_ntop

usage:

```
w_ntop [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-W WEST_H5FILE]
             [--first-iter N_ITER] [--last-iter N_ITER] [-a ASSIGNMENTS] [-n COUNT] [-t␣
↪TIMEPOINT]
             [--highweight | --lowweight | --random] [-o OUTPUT]
```

Select walkers from bins . An assignment file mapping walkers to bins at each timepoint is required (see``w_assign –help`` for further information on generating this file). By default, high-weight walkers are selected (hence the name `w_ntop`: select the N top-weighted walkers from each bin); however, minimum weight walkers and randomly-selected walkers may be selected instead.

6.1.16.1 Output format

The output file (-o/–output, by default "ntop.h5") contains the following datasets:

```
``/n_iter`` [iteration]
  *(Integer)* Iteration numbers for each entry in other datasets.

``/n_segs`` [iteration][bin]
  *(Integer)* Number of segments in each bin/state in the given iteration.
  This will generally be the same as the number requested with
  ``--n/--count`` but may be smaller if the requested number of walkers
  does not exist.

``/seg_ids`` [iteration][bin][segment]
  *(Integer)* Matching segments in each iteration for each bin.
  For an iteration ``n_iter``, only the first ``n_iter`` entries are
  valid. For example, the full list of matching seg_ids in bin 0 in the
  first stored iteration is ``seg_ids[0][0][:n_segs[0]]``.

``/weights`` [iteration][bin][segment]
  *(Floating-point)* Weights for each matching segment in ``/seg_ids``.
```

6.1.16.2 Command-line arguments

optional arguments:

```
-h, --help            show this help message and exit
--highweight          Select COUNT highest-weight walkers from each bin.
--lowweight           Select COUNT lowest-weight walkers from each bin.
--random              Select COUNT walkers randomly from each bin.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

input options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Use assignments from the given ASSIGNMENTS file (default: assign.
↪h5).
```

selection options:

```
-n COUNT, --count COUNT
                      Select COUNT walkers from each iteration for each bin (default: 1).
-t TIMEPOINT, --timepoint TIMEPOINT
                      Base selection on the given TIMEPOINT within each iteration.␣
↪Default (-1)
                      corresponds to the last timepoint.
```

output options:

```
-o OUTPUT, --output OUTPUT
                      Write output to OUTPUT (default: ntop.h5).
```

6.1.16.3 westpa.cli.tools.w_ntop module

class westpa.cli.tools.w_ntop.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.cli.tools.w_ntop.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_ntop.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
The first iteration included in the calculation.

last_iter
One past the last iteration included in the calculation.

iter_step
Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.cli.tools.w_ntop.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.cli.tools.w_ntop.**seg_id_dtype**

alias of `int64`

westpa.cli.tools.w_ntop.**n_iter_dtype**

alias of `uint32`

westpa.cli.tools.w_ntop.**weight_dtype**

alias of `float64`

westpa.cli.tools.w_ntop.**assignments_list_to_table**(*nsegs*, *nbins*, *assignments*)

Convert a list of bin assignments (integers) to a boolean table indicating indicating if a given segment is in a given bin

class westpa.cli.tools.w_ntop.**WNTopTool**

Bases: *WESTTool*

prog = 'w_ntop'

```
description = 'Select walkers from bins . An assignment file mapping walkers
to\nbins at each timepoint is required (see``w_assign --help`` for
further\ninformation on generating this file). By default, high-weight walkers
are\nselected (hence the name ``w_ntop``:  select the N top-weighted walkers
from\neach bin); however, minimum weight walkers and randomly-selected walkers\nmay
be selected instead.\n\n\
n------------------------------------------------------------------------------\
nOutput
format\
n------------------------------------------------------------------------------\n\
nThe output file (-o/--output, by default "ntop.h5") contains the
following\ndatasets:\n\n ``/n_iter`` [iteration]\n *(Integer)* Iteration numbers for
each entry in other datasets.\n\n ``/n_segs`` [iteration][bin]\n *(Integer)* Number
of segments in each bin/state in the given iteration.\n This will generally be the
same as the number requested with\n ``--n/--count`` but may be smaller if the
requested number of walkers\n does not exist.\n\n ``/seg_ids``
[iteration][bin][segment]\n *(Integer)* Matching segments in each iteration for each
bin.\n For an iteration ``n_iter``, only the first ``n_iter`` entries are\n valid.
For example, the full list of matching seg_ids in bin 0 in the\n first stored
iteration is ``seg_ids[0][0][:n_segs[0]]``.\n\n ``/weights``
[iteration][bin][segment]\n *(Floating-point)* Weights for each matching segment in
``/seg_ids``.\n\n\
n------------------------------------------------------------------------------\
nCommand-line
arguments\
n------------------------------------------------------------------------------\n'
```

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_ntop.**entry_point**()

6.1.17 w_multi_west

The `w_multi_west` tool combines multiple WESTPA simulations into a single aggregate simulation to facilitate the analysis of the set of simulations. In particular, the tool creates a single `west.h5` file that contains all of the data from the west.h5 files of the individual simulations. Each iteration x in the new file contains all of the segments from iteration x from each of the set of simulation, all normalized to the total weight.

6.1.17.1 Overview

usage:

```
w_multi_west [-h] [-m master] [-n sims] [--quiet | --verbose | --debug] [--version]
             [-W WEST_H5FILE] [-a aux] [--auxall] [--ibstates]
             [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
             [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
             [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
             [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
             [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
             [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
             [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

optional arguments:

```
-h, --help            show this help message and exit
```

General options::

-m, --master directory Master path of simulations where all the smaller simulations are stored (default: Current Directory)

-n, --sims n Number of simulation directories. Assumes leading zeros. (default: 0)

--quiet emit only essential information

--verbose emit extra information

--version show program's version number and exit

6.1.17.2 Command-Line Options

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

6.1.17.2.1 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, --WEST_H5FILE file
  The name of the main .h5 file inside each simulation directory. (Default: west.h5)

-o, --output file
  Store this tool's output in file. (Default: multi.h5)

-a, --aux auxdata
  Name of additional auxiliary dataset to be combined. Can be called multiple times.
  (Default: None)

-aa, --auxall
  Combine all auxiliary datsets as labeled in ``west.h5`` in folder 01. (Default: False)

-nr, --no-reweight
  Do not perform reweighting. (Default: False)

-ib, --ibstates
  Attempt to combine ``ibstates`` dataset if the basis states are identical across
  all simulations. Needed when tracing with ``westpa.analysis``. (Default: False)
```

6.1.17.3 Examples

If you have five simulations, set up your directory such that you have five directories are named numerically with leading zeroes, and each directory contains a `west.h5` file. For this example, each `west.h5` also contains an auxiliary dataset called `RMSD`. If you run `ls`, you will see the following output:

```
01 02 03 04 05
```

To run the w_multi_west tool, do the following:

```
w_multi_west.py -m . -n 5 --aux=RMSD
```

If you used any custom WESTSystem, include that in the directory where you run the code.

To proceed in analyzing the aggregated simulation data as a single simulation, rename the output file `multi.h5` to `west.h5`.

6.1.17.4 westpa.cli.tools.w_multi_west module

class westpa.cli.tools.w_multi_west.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

westpa.cli.tools.w_multi_west.**n_iter_dtype**

alias of uint32

class westpa.cli.tools.w_multi_west.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_multi_west.**WESTMultiTool**(*wm_env=None*)

Bases: *WESTParallelTool*

Base class for command-line tools which work with multiple simulations. Automatically parses for and gives commands to load multiple files.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

parse_from_yaml(*yamlfilepath*)

Parse options from YAML input file. Command line arguments take precedence over options specified in the YAML hierarchy. TODO: add description on how YAML files should be constructed.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

exception NoSimulationsException

Bases: Exception

generate_file_list(*key_list*)

A convenience function which takes in a list of keys that are filenames, and returns a dictionary which contains all the individual files loaded inside of a dictionary keyed to the filename.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_multi_west.**get_bin_mapper**(*we_h5file*, *hashval*)

Look up the given hash value in the binning table, unpickling and returning the corresponding bin mapper if available, or raising KeyError if not.

westpa.cli.tools.w_multi_west.**create_idtype_array**(*input_array*)

Return a new array with the new istate_dtype while preserving old data.

class westpa.cli.tools.w_multi_west.**WMultiWest**

Bases: *WESTMultiTool*

prog = 'w_multi_west'

description = 'Tool designed to combine multiple WESTPA simulations while accounting for\nreweighting.
n--
nCommand-line
options
n--\n'

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

open_files()

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

total_number_of_walkers()

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_multi_west.**entry_point**()

6.1.18 w_red

usage:

```
w_red [-h] [-r RCFILE] [--quiet] [--verbose] [--version] [--max-queue-length MAX_QUEUE_
↪LENGTH]
            [--debug] [--terminal]
            [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_WORKERS]
            [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
            [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
            [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
            [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
            [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
            [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

-r RCFILE, --rcfile RCFILE use RCFILE as the WEST run-time configuration file (default: west.cfg)

--quiet emit only essential information

--verbose emit extra information

--version show program's version number and exit

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
```

6.1.18.1 westpa.cli.tools.w_red module

westpa.cli.tools.w_red.**H5File**

alias of File

class westpa.cli.tools.w_red.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_red.**DurationCorrector**(*durations*, *weights*, *dtau*, *maxduration=None*)

Bases: object

static from_kinetics_file(*directh5*, *istate*, *fstate*, *dtau*, *n_iters=None*)

property event_duration_histogram

property cumulative_event_duration_histogram

correction(*iters*, *freqs=None*)

Return the correction factor

t=theta tau=t |
| | |
| | ~ |
| | f(tau) dtau dt | * maxduration
| | |
t=0 tau=0 |

|_ _|

where

~` ^ f(tau) is proportional to f(tau)/(theta-tau), and is normalized to
^

integrate to 1, and f(tau) is sum of the weights of walkers with duration time tau.

westpa.cli.tools.w_red.**get_raw_rates**(*directh5*, *istate*, *fstate*, *n_iters=None*)

westpa.cli.tools.w_red.**calc_avg_rate**(*directh5_path*, *istate*, *fstate*, ***kwargs*)

Return the raw or RED-corrected rate constant with the confidence interval.

Parameters

- **nstiter** (*duration of each iteration (number of steps)*)
- **ntpr** (*report inteval (number of steps)*)

westpa.cli.tools.w_red.**calc_rates**(*directh5_path*, *istate*, *fstate*, ***kwargs*)

Return the raw and RED-corrected rate constants vs. iterations. This code is faster than calling calc_rate() iteratively

Parameters

- **nstiter** (*duration of each iteration (number of steps)*)
- **ntpr** (*report inteval (number of steps)*)

class westpa.cli.tools.w_red.**RateCalculator**(*directh5*, *istate*, *fstate*, *assignh5=None*, ***kwargs*)

Bases: object

property conditional_fluxes

property populations

property tau

property dtau

property istate

property fstate

property n_iters

calc_rate(*i_iter=None*, *red=False*, ***kwargs*)

calc_rates(*n_iters=None*, ***kwargs*)

class westpa.cli.tools.w_red.**WRed**

Bases: *WESTParallelTool*

prog = 'w_red'

description = 'Apply the RED scheme to estimate steady-state WE fluxes from\nshorter trajectories.\n
n----------
nSource
data\n----------
n\nSource data is provided as a w_ipa "scheme" which is typically defined\nin the west.cfg file. For instance, if a user wishes to estimate RED\nfluxes for a scheme named "DEFAULT" that argument would be provided\nto w_red and WRed would estimate RED fluxes based off of the data\ncontained in the assign.h5 and direct.h5 files in ANALYSIS/DEFAULT.\n\n'

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_red.**entry_point**()

6.1.19 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

6.1.19.1 Overview

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

usage:

```
| ``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:

`plothist mode input_file [``other`` ``options``]`

6.1.19.1.1 ‘instant‘ mode

usage:

```
| ``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]``
```

6.1.19.1.2 ‘average‘ mode

usage:

```
| ``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]``
```

6.1.19.1.3 ‘evolution‘ mode

usage:

```
| ``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]``
```

6.1.19.2 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

6.1.19.2.1 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. ‘pdist.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)

6.1.19.2.2 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

6.1.19.2.3 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)

6.1.19.2.4 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'`

6.1.19.3 Examples

These examples assume the input file is created using w_pdist and is named ‘pdist.h5’

6.1.19.3.1 Basic plotting

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

```
plothist instant pdist.h5
```

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

``plothist evolution pdist.h5 –log10 ``

Plot the average linear probability distribution over all iterations

```
plothist average pdist.h5 --linear
```

6.1.19.3.2 Specifying progress coordinate

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

```
plothist average pdist.h5 0::pcoord
```

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

```
plothist average pdist.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

```
plothist instant pdist.h5 0 1 --iter 5 --linear
```

6.1.19.4 westpa.cli.tools.plothist module

class westpa.cli.tools.plothist.**NonUniformImage**(*ax*, *, *interpolation='nearest'*, ***kwargs*)

Bases: AxesImage

Parameters

- **ax** (*~matplotlib.axes.Axes*) – The axes the image will belong to.
- **interpolation** (*{'nearest', 'bilinear'}, default: 'nearest'*) – The interpolation scheme used in the resampling.
- ****kwargs** – All other keyword arguments are identical to those of *.AxesImage*.

mouseover = False

make_image(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

Returns

- **image** ((M, N, 4) *numpy.uint8* array) – The RGBA image, resampled unless *unsampled* is True.
- **x, y** (*float*) – The upper left corner where the image should be drawn, in pixel space.
- **trans** (*~matplotlib.transforms.Affine2D*) – The affine transformation from image to pixel space.

set_data(*x*, *y*, *A*)

Set the grid for the pixel centers, and the pixel values.

Parameters

- **x** (*1D array-like*) – Monotonic arrays of shapes (N,) and (M,), respectively, specifying pixel centers.
- **y** (*1D array-like*) – Monotonic arrays of shapes (N,) and (M,), respectively, specifying pixel centers.
- **A** (*array-like*) – (M, N) *~numpy.ndarray* or masked array of values to be colormapped, or (M, N, 3) RGB array, or (M, N, 4) RGBA array.

set_array(**args*)

Retained for backwards compatibility - use set_data instead.

Parameters

A (*array-like*)

set_interpolation(*s*)

Parameters

s (*{'nearest', 'bilinear'} or None*) – If None, use **:rc:`image.interpolation`**.

get_extent()

Return the image extent as tuple (left, right, bottom, top).

set_filternorm(*filternorm*)

Set whether the resize filter normalizes the weights.

See help for *~.Axes.imshow*.

Parameters
filternorm (*bool*)

set_filterrad(*filterrad*)

Set the resize filter radius only applicable to some interpolation schemes – see help for imshow

Parameters
filterrad (*positive float*)

set_norm(*norm*)

Set the normalization instance.

Parameters
norm (*.Normalize* or str or None)

Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

set_cmap(*cmap*)

Set the colormap for luminance data.

Parameters
cmap (*.Colormap* or str or None)

set(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, array=<UNSET>, clim=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, cmap=<UNSET>, data=<UNSET>, extent=<UNSET>, filternorm=<UNSET>, filterrad=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, interpolation=<UNSET>, interpolation_stage=<UNSET>, label=<UNSET>, mouseover=<UNSET>, norm=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, resample=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are
Properties:
agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image alpha: float or 2D array-like or None animated: bool array: unknown clim: (vmin: float, vmax: float) clip_box: *~matplotlib.transforms.BboxBase* or None clip_on: bool clip_path: Patch or (Path, Transform) or None cmap: unknown data: unknown extent: 4-tuple of float figure: *~matplotlib.figure.Figure* filternorm: unknown filterrad: unknown gid: str in_layout: bool interpolation: {'nearest', 'bilinear'} or None interpolation_stage: {'data', 'rgba'} or None label: object mouseover: bool norm: unknown path_effects: list of *.AbstractPathEffect* picker: None or bool or float or callable rasterized: bool resample: bool or None sketch_params: (scale: float, length: float, randomness: float) snap: bool or None transform: *~matplotlib.transforms.Transform* url: str visible: bool zorder: float

class westpa.cli.tools.plothist.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.plothist.**WESTSubcommand**(*parent*)

Bases: *WESTToolComponent*

Base class for command-line tool subcommands. A little sugar for making this more uniform.

subcommand = None

help_text = None

description = None

add_to_subparsers(*subparsers*)

go()

property work_manager

The work manager for this tool. Raises AttributeError if this is not a parallel tool.

westpa.cli.tools.plothist.**normhistnd**(*hist*, *binbounds*)

Normalize the N-dimensional histogram `hist` with corresponding bin boundaries `binbounds`. Modifies `hist` in place and returns the normalization factor used.

westpa.cli.tools.plothist.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

westpa.cli.tools.plothist.**sum_except_along**(*array*, *axes*)

Reduce the given array by addition over all axes except those listed in the scalar or iterable `axes`

class westpa.cli.tools.plothist.**PlotHistBase**(*parent*)

Bases: *WESTSubcommand*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

parse_dimspec(*dimspec*)

parse_range(*rangespec*)

class westpa.cli.tools.plothist.**PlotSupports2D**(*parent*)

Bases: *PlotHistBase*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.plothist.**InstantPlotHist**(*parent*)

Bases: *PlotSupports2D*

subcommand = 'instant'

help_text = 'plot probability distribution for a single WE iteration'

description = 'Plot a probability distribution for a single WE iteration. The probability\ndistribution must have been previously extracted with ``w_pdist`` (or, at\nleast, must be compatible with the output format of ``w_pdist``; see\n``w_pdist --help`` for more information).\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

do_instant_plot_1d()

Plot the histogram for iteration self.n_iter

do_instant_plot_2d()

Plot the histogram for iteration self.n_iter

go()

class westpa.cli.tools.plothist.**AveragePlotHist**(*parent*)

Bases: *PlotSupports2D*

subcommand = 'average'

help_text = 'plot average of a probability distribution over a WE simulation'

description = 'Plot a probability distribution averaged over multiple iterations. The\nprobability distribution must have been previously extracted with ``w_pdist``\n(or, at least, must be compatible with the output format of ``w_pdist``; see\n``w_pdist --help`` for more information).\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

do_average_plot_1d()

Plot the average histogram for iterations self.iter_start to self.iter_stop

do_average_plot_2d()

Plot the histogram for iteration self.n_iter

go()

class westpa.cli.tools.plothist.**EvolutionPlotHist**(*parent*)

Bases: *PlotHistBase*

subcommand = 'evolution'

help_text = 'plot evolution of a probability distribution over the course of a WE simulation'

description = 'Plot a probability distribution as it evolves over iterations. The\nprobability distribution must have been previously extracted with ``w_pdist``\n(or, at least, must be compatible with the output format of ``w_pdist``; see\n``w_pdist --help`` for more information).\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

go()

Plot the evolution of the histogram for iterations self.iter_start to self.iter_stop

class westpa.cli.tools.plothist.**PlotHistTool**

Bases: *WESTMasterCommand*

prog = 'plothist'

subparsers_title = 'plotting modes'

subcommands = [<class 'westpa.cli.tools.plothist.InstantPlotHist'>, <class 'westpa.cli.tools.plothist.AveragePlotHist'>, <class 'westpa.cli.tools.plothist.EvolutionPlotHist'>]

**description = 'Plot probability density functions (histograms) generated by w_pdist or other\nprograms conforming to the same output format. This program operates in one of\nthree modes:\n\n instant\n Plot 1-D and 2-D histograms for an individual iteration. See\n ``plothist instant --help`` for more information.\n\n average\n Plot 1-D and 2-D histograms, averaged over several iterations. See\n ``plothist average --help`` for more information.\n\n evolution\n Plot the time evolution 1-D histograms as waterfall (heat map) plots.\n See ``plothist evolution --help`` for more information.\n\nThis program takes the output of ``w_pdist`` as input (see ``w_pdist --help``\nfor more information), and can generate any kind of graphical output that\nmatplotlib supports.\n\n\
n----------\
nCommand-line
options\
n----------\n'**

westpa.cli.tools.plothist.**entry_point**()

6.1.20 ploterr

usage:

```
ploterr [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              {help,d.kinetics,d.probs,rw.probs,rw.kinetics,generic} ...
```

Plots error ranges for weighted ensemble datasets.

6.1.20.1 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

supported input formats:

```
{help,d.kinetics,d.probs,rw.probs,rw.kinetics,generic}
  help                print help for this command or individual subcommands
  d.kinetics          output of w_direct kinetics
  d.probs             output of w_direct probs
  rw.probs            output of w_reweight probs
  rw.kinetics         output of w_reweight kinetics
  generic             arbitrary HDF5 file and dataset
```

6.1.20.2 westpa.cli.tools.ploterr module

class westpa.cli.tools.ploterr.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.ploterr.**WESTSubcommand**(*parent*)

Bases: *WESTToolComponent*

Base class for command-line tool subcommands. A little sugar for making this more uniform.

subcommand = None

help_text = None

description = None

add_to_subparsers(*subparsers*)

go()

property work_manager

The work manager for this tool. Raises AttributeError if this is not a parallel tool.

class westpa.cli.tools.ploterr.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.ploterr.**Plotter**(*h5file*, *h5key*, *iteration=-1*, *interface='matplotlib'*)

Bases: object

This is a semi-generic plotting interface that has a built in curses based terminal plotter. It's fairly specific to what we're using it for here, but we could (and maybe should) build it out into a little library that we can use via the command line to plot things. Might be useful for looking at data later. That would also cut the size of this tool down by a good bit.

plot(*i=0*, *j=1*, *tau=1*, *iteration=None*, *dim=0*, *interface=None*)

class westpa.cli.tools.ploterr.**CommonPloterrs**(*parent*)

Bases: *WESTSubcommand*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

parse_range(*rangespec*)

do_plot(*data*, *output_filename*, *title=None*, *x_range=None*, *y_range=None*, *x_label=None*, *y_label=None*)

class westpa.cli.tools.ploterr.**GenericIntervalSubcommand**(*parent*)

Bases: *CommonPloterrs*

description = 'Plots generic expectation/CI data. A path to the HDF5 file and the dataset\nwithin it must be provided. This path takes the form **FILENAME/PATH[SLICE]**.\nIf the dataset is not a vector (one dimensional) then a slice must be provided.\nFor example, to access the state 0 to state 1 rate evolution calculated by\n``w_kinavg``, one would use ``kinavg.h5/rate_evolution[:,0,1]``.\n\n\
n----------\
nCommand-line
arguments\
n----------\n'

subcommand = 'generic'

help_text = 'arbitrary HDF5 file and dataset'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

load_and_validate_data()

go()

class westpa.cli.tools.ploterr.DirectKinetics(*parent*)

Bases: *CommonPloterrs*

subcommand = 'd.kinetics'

help_text = 'output of w_direct kinetics'

input_filename = 'direct.h5'

flux_output_filename = 'flux_evolution_d_{state_label}.pdf'

rate_output_filename = 'rate_evolution_d_{istate_label}_{fstate_label}.pdf'

description = 'Plot evolution of state-to-state rates and total flux into states as generated\nby ``w_{direct/reweight} kinetics`` (when used with the ``--evolution-mode``\noption). Plots are generated for all rates/fluxes calculated. Output filenames\nrequire (and plot titles and axis labels support) substitution based on which\nflux/rate is being plotted:\n\n istate_label, fstate_label\n *(String, for rates)* Names of the initial and final states, as originally\n given to ``w_assign``.\n\n istate_index, fstate_index\n *(Integer, for rates)* Indices of initial and final states.\n\n state_label\n *(String, for fluxes)* Name of state\n\n state_index\n *(Integer, for fluxes)* Index of state\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

plot_flux(*istate*)

plot_rate(*istate*, *jstate*)

go()

class westpa.cli.tools.ploterr.**DirectStateprobs**(*parent*)

Bases: *CommonPloterrs*

subcommand = 'd.probs'

help_text = 'output of w_direct probs'

input_filename = 'direct.h5'

pop_output_filename = 'pop_evolution_d_{state_label}.pdf'

color_output_filename = 'color_evolution_d_{state_label}.pdf'

description = 'Plot evolution of macrostate populations and associated uncertainties. Plots\nare generated for all states calculated. Output filenames require (and plot\ntitles and axis labels support) substitution based on which state is being\nplotted:\n\n state_label\n *(String, for fluxes)* Name of state\n\n state_index\n *(Integer, for fluxes)* Index of state\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

plot_pop(*istate*)

plot_color(*istate*)

go()

class westpa.cli.tools.ploterr.**ReweightStateprobs**(*parent*)

Bases: *DirectStateprobs*

subcommand = 'rw.probs'

help_text = 'output of w_reweight probs'

input_filename = 'reweight.h5'

pop_output_filename = 'pop_evolution_rw_{state_label}.pdf'

color_output_filename = 'color_evolution_rw_{state_label}.pdf'

class westpa.cli.tools.ploterr.**ReweightKinetics**(*parent*)

Bases: *DirectKinetics*

subcommand = 'rw.kinetics'

help_text = 'output of w_reweight kinetics'

input_filename = 'reweight.h5'

flux_output_filename = 'flux_evolution_rw_{state_label}.pdf'

rate_output_filename = 'rate_evolution_rw_{istate_label}_{fstate_label}.pdf'

class westpa.cli.tools.ploterr.PloterrsTool

Bases: *WESTMasterCommand*

prog = 'ploterrs'

subcommands = [<class 'westpa.cli.tools.ploterr.DirectKinetics'>, <class 'westpa.cli.tools.ploterr.DirectStateprobs'>, <class 'westpa.cli.tools.ploterr.ReweightStateprobs'>, <class 'westpa.cli.tools.ploterr.ReweightKinetics'>, <class 'westpa.cli.tools.ploterr.GenericIntervalSubcommand'>]

subparsers_title = 'supported input formats'

description = 'Plots error ranges for weighted ensemble datasets.\n\n\
n--\
nCommand-line
options\
n--\n'

westpa.cli.tools.ploterr.entry_point()

6.1.21 westpa.cli package

6.1.21.1 w_kinavg

WARNING: w_kinavg is being deprecated. Please use w_direct instead.

usage:

```
w_kinavg trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER] [--step-
↪iter STEP]
                    [-a ASSIGNMENTS] [-o OUTPUT] [-k KINETICS] [--disable-bootstrap] [--
↪disable-correl]
                    [--alpha ALPHA] [--autocorrel-alpha ACALPHA] [--nsets NSETS]
                    [-e {cumulative,blocked,none}] [--window-frac WINDOW_FRAC] [--
↪disable-averages]
```

Calculate average rates/fluxes and associated errors from weighted ensemble data. Bin assignments (usually "assign.h5") and kinetics data (usually "direct.h5") data files must have been previously generated (see "w_assign –help" and "w_direct init –help" for information on generating these files).

The evolution of all datasets may be calculated, with or without confidence intervals.

6.1.21.1.1 Output format

The output file (-o/–output, usually "direct.h5") contains the following dataset:

```
/avg_rates [state,state]
  (Structured -- see below) State-to-state rates based on entire window of
  iterations selected.

/avg_total_fluxes [state]
```

(continues on next page)

(continued from previous page)

```
  (Structured -- see below) Total fluxes into each state based on entire
  window of iterations selected.

/avg_conditional_fluxes [state,state]
  (Structured -- see below) State-to-state fluxes based on entire window of
  iterations selected.
```

If –evolution-mode is specified, then the following additional datasets are available:

```
/rate_evolution [window][state][state]
  (Structured -- see below). State-to-state rates based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.

/target_flux_evolution [window,state]
  (Structured -- see below). Total flux into a given macro state based on
  windows of iterations of varying width, as in /rate_evolution.

/conditional_flux_evolution [window,state,state]
  (Structured -- see below). State-to-state fluxes based on windows of
  varying width, as in /rate_evolution.
```

The structure of these datasets is as follows:

```
iter_start
  (Integer) Iteration at which the averaging window begins (inclusive).

iter_stop
  (Integer) Iteration at which the averaging window ends (exclusive).

expected
  (Floating-point) Expected (mean) value of the observable as evaluated within
  this window, in units of inverse tau.

ci_lbound
  (Floating-point) Lower bound of the confidence interval of the observable
  within this window, in units of inverse tau.

ci_ubound
  (Floating-point) Upper bound of the confidence interval of the observable
  within this window, in units of inverse tau.

stderr
  (Floating-point) The standard error of the mean of the observable
  within this window, in units of inverse tau.

corr_len
  (Integer) Correlation length of the observable within this window, in units
  of tau.
```

Each of these datasets is also stamped with a number of attributes:

```
mcbs_alpha
  (Floating-point) Alpha value of confidence intervals. (For example,
  *alpha=0.05* corresponds to a 95% confidence interval.)

mcbs_nsets
  (Integer) Number of bootstrap data sets used in generating confidence
  intervals.

mcbs_acalpha
  (Floating-point) Alpha value for determining correlation lengths.
```

6.1.21.1.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: kinavg.h5).
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates are stored in KINETICS (default:␣
↪kintrace.h5).
```

confidence interval calculation options:

```
--disable-bootstrap, -db
                      Enable the use of Monte Carlo Block Bootstrapping.
--disable-correl, -dc
```

(continues on next page)

(continued from previous page)

```
                      Disable the correlation analysis.
--alpha ALPHA         Calculate a (1-ALPHA) confidence interval' (default: 0.05)
--autocorrel-alpha ACALPHA
                      Evaluate autocorrelation to (1-ACALPHA) significance. Note that␣
↪too small an
                      ACALPHA will result in failure to detect autocorrelation in a␣
↪noisy flux signal.
                      (Default: same as ALPHA.)
--nsets NSETS         Use NSETS samples for bootstrapping (default: chosen based on␣
↪ALPHA)
```

calculation options:

```
-e {cumulative,blocked,none}, --evolution-mode {cumulative,blocked,none}
                      How to calculate time evolution of rate estimates. ``cumulative``␣
↪evaluates rates
                      over windows starting with --start-iter and getting progressively␣
↪wider to --stop-
                      iter by steps of --step-iter. ``blocked`` evaluates rates over␣
↪windows of width
                      --step-iter, the first of which begins at --start-iter. ``none``␣
↪(the default)
                      disables calculation of the time evolution of rate estimates.
--window-frac WINDOW_FRAC
                      Fraction of iterations to use in each window when running in␣
↪``cumulative`` mode.
                      The (1 - frac) fraction of iterations will be discarded from the␣
↪start of each
                      window.
```

misc options:

```
--disable-averages, -da
                      Whether or not the averages should be printed to the console (set␣
↪to FALSE if flag
                      is used).
```

6.1.21.1.3 westpa.cli.tools.w_kinavg module

class westpa.cli.tools.w_kinavg.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_kinavg.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_kinavg.**DKinAvg**(*parent*)

Bases: *AverageCommands*

subcommand = 'kinetics'

help_text = 'Generates rate and flux values from a WESTPA simulation via tracing.'

default_kinetics_file = 'direct.h5'

```
    description = 'Calculate average rates/fluxes and associated errors from weighted
    ensemble\ndata. Bin assignments (usually "assign.h5") and kinetics data
    (usually\n"direct.h5") data files must have been previously generated
    (see\n"w_assign --help" and "w_direct init --help" for information on\ngenerating
    these files).\n\nThe evolution of all datasets may be calculated, with or without
    confidence\nintervals.\n\
    n-------------------------------------------------------------------------------\
    nOutput
    format\
    n-------------------------------------------------------------------------------\n\
    nThe output file (-o/--output, usually "direct.h5") contains the
    following\ndataset:\n\n /avg_rates [state,state]\n (Structured -- see below)
    State-to-state rates based on entire window of\n iterations selected.\n\n
    /avg_total_fluxes [state]\n (Structured -- see below) Total fluxes into each state
    based on entire\n window of iterations selected.\n\n /avg_conditional_fluxes
    [state,state]\n (Structured -- see below) State-to-state fluxes based on entire
    window of\n iterations selected.\n\nIf --evolution-mode is specified, then the
    following additional datasets are\navailable:\n\n /rate_evolution
    [window][state][state]\n (Structured -- see below). State-to-state rates based on
    windows of\n iterations of varying width. If --evolution-mode=cumulative, then\n
    these windows all begin at the iteration specified with\n --start-iter and grow in
    length by --step-iter for each successive\n element. If --evolution-mode=blocked,
    then these windows are all of\n width --step-iter (excluding the last, which may be
    shorter), the first\n of which begins at iteration --start-iter.\n\n
    /target_flux_evolution [window,state]\n (Structured -- see below). Total flux into a
    given macro state based on\n windows of iterations of varying width, as in
    /rate_evolution.\n\n /conditional_flux_evolution [window,state,state]\n (Structured
    -- see below). State-to-state fluxes based on windows of\n varying width, as in
    /rate_evolution.\n\nThe structure of these datasets is as follows:\n\n iter_start\n
    (Integer) Iteration at which the averaging window begins (inclusive).\n\n
    iter_stop\n (Integer) Iteration at which the averaging window ends (exclusive).\n\n
    expected\n (Floating-point) Expected (mean) value of the observable as evaluated
    within\n this window, in units of inverse tau.\n\n ci_lbound\n (Floating-point)
    Lower bound of the confidence interval of the observable\n within this window, in
    units of inverse tau.\n\n ci_ubound\n (Floating-point) Upper bound of the confidence
    interval of the observable\n within this window, in units of inverse tau.\n\n
    stderr\n (Floating-point) The standard error of the mean of the observable\n within
    this window, in units of inverse tau.\n\n corr_len\n (Integer) Correlation length of
    the observable within this window, in units\n of tau.\n\nEach of these datasets is
    also stamped with a number of attributes:\n\n mcbs_alpha\n (Floating-point) Alpha
    value of confidence intervals. (For example,\n *alpha=0.05* corresponds to a 95%
    confidence interval.)\n\n mcbs_nsets\n (Integer) Number of bootstrap data sets used
    in generating confidence\n intervals.\n\n mcbs_acalpha\n (Floating-point) Alpha
    value for determining correlation lengths.\n\n\
    n-------------------------------------------------------------------------------\
    nCommand-line
    options\
    n-------------------------------------------------------------------------------\n'

    w_kinavg()

    go()
```

westpa.cli.tools.w_kinavg.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.
category
The Warning category subclass. Defaults to UserWarning.
stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().
source
If supplied, the destroyed object which emitted a ResourceWarning
skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_kinavg.**WKinAvg**(*parent*)

Bases: *DKinAvg*

subcommand = 'trace'

help_text = 'averages and CIs for path-tracing kinetics analysis'

default_kinetics_file = 'kintrace.h5'

default_output_file = 'kinavg.h5'

class westpa.cli.tools.w_kinavg.**WDirect**

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_kinavg'

subcommands = [<class 'westpa.cli.tools.w_kinavg.WKinAvg'>]

subparsers_title = 'direct kinetics analysis schemes'

```
    description = 'Calculate average rates and associated errors from weighted ensemble
    data. Bin\nassignments (usually "assignments.h5") and kinetics data
    (usually\n"kintrace.h5" or "kinmat.h5") data files must have been previously
    generated\n(see "w_assign --help" and "w_kinetics --help" for information on
    generating\nthese files).\n\
    n-------------------------------------------------------------------------------\
    nOutput
    format\
    n-------------------------------------------------------------------------------\n\
    nThe output file (-o/--output, usually "kinavg.h5") contains the
    following\ndataset:\n\n /avg_rates [state,state]\n (Structured -- see below)
    State-to-state rates based on entire window of\n iterations selected.\n\nFor trace
    mode, the following additional datasets are generated:\n\n /avg_total_fluxes
    [state]\n (Structured -- see below) Total fluxes into each state based on entire\n
    window of iterations selected.\n\n /avg_conditional_fluxes [state,state]\n
    (Structured -- see below) State-to-state fluxes based on entire window of\n
    iterations selected.\n\nIf --evolution-mode is specified, then the following
    additional dataset is\navailable:\n\n /rate_evolution [window][state][state]\n
    (Structured -- see below). State-to-state rates based on windows of\n iterations of
    varying width. If --evolution-mode=cumulative, then\n these windows all begin at the
    iteration specified with\n --start-iter and grow in length by --step-iter for each
    successive\n element. If --evolution-mode=blocked, then these windows are all of\n
    width --step-iter (excluding the last, which may be shorter), the first\n of which
    begins at iteration --start-iter.\n\nIf --evolution-mode is specified in trace mode,
    the following additional\ndatasets are available:\n\n /target_flux_evolution
    [window,state]\n (Structured -- see below). Total flux into a given macro state
    based on\n windows of iterations of varying width, as in /rate_evolution.\n\n
    /conditional_flux_evolution [window,state,state]\n (Structured -- see below).
    State-to-state fluxes based on windows of\n varying width, as in
    /rate_evolution.\n\nThe structure of these datasets is as follows:\n\n iter_start\n
    (Integer) Iteration at which the averaging window begins (inclusive).\n\n
    iter_stop\n (Integer) Iteration at which the averaging window ends (exclusive).\n\n
    expected\n (Floating-point) Expected (mean) value of the rate as evaluated within\n
    this window, in units of inverse tau.\n\n ci_lbound\n (Floating-point) Lower bound
    of the confidence interval on the rate\n within this window, in units of inverse
    tau.\n\n ci_ubound\n (Floating-point) Upper bound of the confidence interval on the
    rate\n within this window, in units of inverse tau.\n\n corr_len\n (Integer)
    Correlation length of the rate within this window, in units\n of tau.\n\nEach of
    these datasets is also stamped with a number of attributes:\n\n mcbs_alpha\n
    (Floating-point) Alpha value of confidence intervals. (For example,\n *alpha=0.05*
    corresponds to a 95% confidence interval.)\n\n mcbs_nsets\n (Integer) Number of
    bootstrap data sets used in generating confidence\n intervals.\n\n mcbs_acalpha\n
    (Floating-point) Alpha value for determining correlation lengths.\n\n\
    n-------------------------------------------------------------------------------\
    nCommand-line
    options\
    n-------------------------------------------------------------------------------\n'
```

westpa.cli.tools.w_kinavg.**entry_point**()

6.1.21.2 w_kinetics

WARNING: w_kinetics is being deprecated. Please use w_direct instead.

usage:

```
w_kinetics trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER]
                       [--step-iter STEP] [-a ASSIGNMENTS] [-o OUTPUT]
```

Calculate state-to-state rates and transition event durations by tracing trajectories.

A bin assignment file (usually "assign.h5") including trajectory labeling is required (see "w_assign –help" for information on generating this file).

This subcommand for w_direct is used as input for all other w_direct subcommands, which will convert the flux data in the output file into average rates/fluxes/populations with confidence intervals.

6.1.21.2.1 Output format

The output file (-o/–output, by default "direct.h5") contains the following datasets:

```
``/conditional_fluxes`` [iteration][state][state]
  *(Floating-point)* Macrostate-to-macrostate fluxes. These are **not**
  normalized by the population of the initial macrostate.

``/conditional_arrivals`` [iteration][stateA][stateB]
  *(Integer)* Number of trajectories arriving at state *stateB* in a given
  iteration, given that they departed from *stateA*.

``/total_fluxes`` [iteration][state]
  *(Floating-point)* Total flux into a given macrostate.

``/arrivals`` [iteration][state]
  *(Integer)* Number of trajectories arriving at a given state in a given
  iteration, regardless of where they originated.

``/duration_count`` [iteration]
  *(Integer)* The number of event durations recorded in each iteration.

``/durations`` [iteration][event duration]
  *(Structured -- see below)*  Event durations for transition events ending
  during a given iteration. These are stored as follows:

    istate
      *(Integer)* Initial state of transition event.
    fstate
      *(Integer)* Final state of transition event.
    duration
      *(Floating-point)* Duration of transition, in units of tau.
    weight
      *(Floating-point)* Weight of trajectory at end of transition, **not**
      normalized by initial state population.
```

Because state-to-state fluxes stored in this file are not normalized by initial macrostate population, they cannot be used as rates without further processing. The `w_direct kinetics` command is used to perform this normalization while

taking statistical fluctuation and correlation into account. See `w_direct kinetics --help` for more information. Target fluxes (total flux into a given state) require no such normalization.

6.1.21.2.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: kintrace.h5).
```

6.1.21.2.3 westpa.cli.tools.w_kinetics module

class westpa.cli.tools.w_kinetics.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_kinetics.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_kinetics.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_kinetics.**DKinetics**(*parent*)

Bases: *WESTKineticsBase*, *WKinetics*

subcommand = 'init'

default_kinetics_file = 'direct.h5'

default_output_file = 'direct.h5'

help_text = 'calculate state-to-state kinetics by tracing trajectories'

description = 'Calculate state-to-state rates and transition event durations by tracing\ntrajectories.\n\nA bin assignment file (usually "assign.h5") including trajectory labeling\nis required (see "w_assign --help" for information on generating this file).\n\nThis subcommand for w_direct is used as input for all other w_direct\nsubcommands, which will convert the flux data in the output file into\naverage rates/fluxes/populations with confidence intervals.\n\n----------\nOutput format\n----------\n\nThe output file (-o/--output, by default "direct.h5") contains the\nfollowing datasets:\n\n ``/conditional_fluxes`` [iteration][state][state]\n *(Floating-point)* Macrostate-to-macrostate fluxes. These are **not**\n normalized by the population of the initial macrostate.\n\n ``/conditional_arrivals`` [iteration][stateA][stateB]\n *(Integer)* Number of trajectories arriving at state *stateB* in a given\n iteration, given that they departed from *stateA*.\n\n ``/total_fluxes``\n [iteration][state]\n *(Floating-point)* Total flux into a given macrostate.\n\n ``/arrivals`` [iteration][state]\n *(Integer)* Number of trajectories arriving at a given state in a given\n iteration, regardless of where they originated.\n\n ``/duration_count`` [iteration]\n *(Integer)* The number of event durations recorded in each iteration.\n\n ``/durations`` [iteration][event duration]\n *(Structured -- see below)* Event durations for transition events ending\n during a given iteration. These are stored as follows:\n\n istate\n *(Integer)* Initial state of transition event.\n fstate\n *(Integer)* Final state of transition event.\n duration\n *(Floating-point)* Duration of transition, in units of tau.\n weight\n *(Floating-point)* Weight of trajectory at end of transition, **not**\n normalized by initial state population.\n\nBecause state-to-state fluxes stored in this file are not normalized by\ninitial macrostate population, they cannot be used as rates without further\nprocessing. The ``w_direct kinetics`` command is used to perform this normalization\nwhile taking statistical fluctuation and correlation into account. See\n``w_direct kinetics --help`` for more information. Target fluxes (total flux\ninto a given state) require no such normalization.\n\n----------\nCommand-line options\n----------\n'

open_files()

go()

class westpa.cli.tools.w_kinetics.WKinetics(*parent*)

Bases: *DKinetics*

subcommand = 'trace'

help_text = 'averages and CIs for path-tracing kinetics analysis'

default_output_file = 'kintrace.h5'

class westpa.cli.tools.w_kinetics.WDirect

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_kinetics'

```
    subcommands = [<class 'westpa.cli.tools.w_kinetics.WKinetics'>]

    subparsers_title = 'calculate state-to-state kinetics by tracing trajectories'

    description = 'Calculate state-to-state rates and transition event durations by
    tracing\ntrajectories.\n\nA bin assignment file (usually "assign.h5") including
    trajectory labeling\nis required (see "w_assign --help" for information on
    generating this file).\n\nThe output generated by this program is used as input for
    the ``w_kinavg``\ntool, which converts the flux data in the output file into average
    rates\nwith confidence intervals. See ``w_kinavg trace --help`` for
    more\ninformation.\n\
    n-------------------------------------------------------------------------------\
    nOutput
    format\
    n-------------------------------------------------------------------------------\n\
    nThe output file (-o/--output, by default "kintrace.h5") contains the\nfollowing
    datasets:\n\n ``/conditional_fluxes`` [iteration][state][state]\n *(Floating-point)*
    Macrostate-to-macrostate fluxes. These are **not**\n normalized by the population of
    the initial macrostate.\n\n ``/conditional_arrivals`` [iteration][stateA][stateB]\n
    *(Integer)* Number of trajectories arriving at state *stateB* in a given\n
    iteration, given that they departed from *stateA*.\n\n ``/total_fluxes``
    [iteration][state]\n *(Floating-point)* Total flux into a given macrostate.\n\n
    ``/arrivals`` [iteration][state]\n *(Integer)* Number of trajectories arriving at a
    given state in a given\n iteration, regardless of where they originated.\n\n
    ``/duration_count`` [iteration]\n *(Integer)* The number of event durations recorded
    in each iteration.\n\n ``/durations`` [iteration][event duration]\n *(Structured --
    see below)* Event durations for transition events ending\n during a given iteration.
    These are stored as follows:\n\n istate\n *(Integer)* Initial state of transition
    event.\n fstate\n *(Integer)* Final state of transition event.\n duration\n
    *(Floating-point)* Duration of transition, in units of tau.\n weight\n
    *(Floating-point)* Weight of trajectory at end of transition, **not**\n normalized
    by initial state population.\n\nBecause state-to-state fluxes stored in this file
    are not normalized by\ninitial macrostate population, they cannot be used as rates
    without further\nprocessing. The ``w_kinavg`` command is used to perform this
    normalization\nwhile taking statistical fluctuation and correlation into account.
    See\n``w_kinavg trace --help`` for more information. Target fluxes (total flux\ninto
    a given state) require no such normalization.\n\
    n-------------------------------------------------------------------------------\
    nCommand-line
    options\
    n-------------------------------------------------------------------------------\n'
```

westpa.cli.tools.w_kinetics.**entry_point**()

6.1.21.3 w_stateprobs

WARNING: w_stateprobs is being deprecated. Please use w_direct instead.

usage:

```
w_stateprobs trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER]
                       [--step-iter STEP] [-a ASSIGNMENTS] [-o OUTPUT] [-k KINETICS]
                       [--disable-bootstrap] [--disable-correl] [--alpha ALPHA]
                       [--autocorrel-alpha ACALPHA] [--nsets NSETS] [-e {cumulative,
↪blocked,none}]
                       [--window-frac WINDOW_FRAC] [--disable-averages]
```

Calculate average populations and associated errors in state populations from weighted ensemble data. Bin assignments, including macrostate definitions, are required. (See "w_assign –help" for more information).

6.1.21.3.1 Output format

The output file (-o/–output, usually "direct.h5") contains the following dataset:

```
/avg_state_probs [state]
  (Structured -- see below) Population of each state across entire
  range specified.

/avg_color_probs [state]
  (Structured -- see below) Population of each ensemble across entire
  range specified.
```

If –evolution-mode is specified, then the following additional datasets are available:

```
/state_pop_evolution [window][state]
  (Structured -- see below). State populations based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.

/color_prob_evolution [window][state]
  (Structured -- see below). Ensemble populations based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.
```

The structure of these datasets is as follows:

```
iter_start
  (Integer) Iteration at which the averaging window begins (inclusive).
```

(continues on next page)

(continued from previous page)

```
iter_stop
  (Integer) Iteration at which the averaging window ends (exclusive).

expected
  (Floating-point) Expected (mean) value of the observable as evaluated within
  this window, in units of inverse tau.

ci_lbound
  (Floating-point) Lower bound of the confidence interval of the observable
  within this window, in units of inverse tau.

ci_ubound
  (Floating-point) Upper bound of the confidence interval of the observable
  within this window, in units of inverse tau.

stderr
  (Floating-point) The standard error of the mean of the observable
  within this window, in units of inverse tau.

corr_len
  (Integer) Correlation length of the observable within this window, in units
  of tau.
```

Each of these datasets is also stamped with a number of attributes:

```
mcbs_alpha
  (Floating-point) Alpha value of confidence intervals. (For example,
  *alpha=0.05* corresponds to a 95% confidence interval.)

mcbs_nsets
  (Integer) Number of bootstrap data sets used in generating confidence
  intervals.

mcbs_acalpha
  (Floating-point) Alpha value for determining correlation lengths.
```

6.1.21.3.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: stateprobs.h5).
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates are stored in KINETICS (default:␣
↪assign.h5).
```

confidence interval calculation options:

```
--disable-bootstrap, -db
                      Enable the use of Monte Carlo Block Bootstrapping.
--disable-correl, -dc
                      Disable the correlation analysis.
--alpha ALPHA         Calculate a (1-ALPHA) confidence interval' (default: 0.05)
--autocorrel-alpha ACALPHA
                      Evaluate autocorrelation to (1-ACALPHA) significance. Note that␣
↪too small an
                      ACALPHA will result in failure to detect autocorrelation in a␣
↪noisy flux signal.
                      (Default: same as ALPHA.)
--nsets NSETS         Use NSETS samples for bootstrapping (default: chosen based on␣
↪ALPHA)
```

calculation options:

```
-e {cumulative,blocked,none}, --evolution-mode {cumulative,blocked,none}
                      How to calculate time evolution of rate estimates. ``cumulative``␣
↪evaluates rates
                      over windows starting with --start-iter and getting progressively␣
↪wider to --stop-
                      iter by steps of --step-iter. ``blocked`` evaluates rates over␣
↪windows of width
                      --step-iter, the first of which begins at --start-iter. ``none``␣
↪(the default)
                      disables calculation of the time evolution of rate estimates.
--window-frac WINDOW_FRAC
                      Fraction of iterations to use in each window when running in␣
↪``cumulative`` mode.
                      The (1 - frac) fraction of iterations will be discarded from the␣
↪start of each
                      window.
```

misc options:

```
--disable-averages, -da
                      Whether or not the averages should be printed to the console (set␣
↪to FALSE if flag
                      is used).
```

6.1.21.3.3 westpa.cli.tools.w_stateprobs module

class westpa.cli.tools.w_stateprobs.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_stateprobs.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_stateprobs.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_stateprobs.**DStateProbs**(*parent*)

Bases: *AverageCommands*

subcommand = 'probs'

help_text = 'Calculates color and state probabilities via tracing.'

default_kinetics_file = 'direct.h5'

description = 'Calculate average populations and associated errors in state
populations from\nweighted ensemble data. Bin assignments, including macrostate
definitions,\nare required. (See "w_assign --help" for more information).\n\
n---\
nOutput
format\
n---\n\
nThe output file (-o/--output, usually "direct.h5") contains the
following\ndataset:\n\n /avg_state_probs [state]\n (Structured -- see below)
Population of each state across entire\n range specified.\n\n /avg_color_probs
[state]\n (Structured -- see below) Population of each ensemble across entire\n
range specified.\n\nIf --evolution-mode is specified, then the following additional
datasets are\navailable:\n\n /state_pop_evolution [window][state]\n (Structured --
see below). State populations based on windows of\n iterations of varying width. If
--evolution-mode=cumulative, then\n these windows all begin at the iteration
specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\n /color_prob_evolution [window][state]\n
(Structured -- see below). Ensemble populations based on windows of\n iterations of
varying width. If --evolution-mode=cumulative, then\n these windows all begin at the
iteration specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\nThe structure of these datasets is as
follows:\n\n iter_start\n (Integer) Iteration at which the averaging window begins
(inclusive).\n\n iter_stop\n (Integer) Iteration at which the averaging window ends
(exclusive).\n\n expected\n (Floating-point) Expected (mean) value of the observable
as evaluated within\n this window, in units of inverse tau.\n\n ci_lbound\n
(Floating-point) Lower bound of the confidence interval of the observable\n within
this window, in units of inverse tau.\n\n ci_ubound\n (Floating-point) Upper bound
of the confidence interval of the observable\n within this window, in units of
inverse tau.\n\n stderr\n (Floating-point) The standard error of the mean of the
observable\n within this window, in units of inverse tau.\n\n corr_len\n (Integer)
Correlation length of the observable within this window, in units\n of tau.\n\nEach
of these datasets is also stamped with a number of attributes:\n\n mcbs_alpha\n
(Floating-point) Alpha value of confidence intervals. (For example,\n *alpha=0.05*
corresponds to a 95% confidence interval.)\n\n mcbs_nsets\n (Integer) Number of
bootstrap data sets used in generating confidence\n intervals.\n\n mcbs_acalpha\n
(Floating-point) Alpha value for determining correlation lengths.\n\n\
n---\
nCommand-line
options\
n---\n'

calculate_state_populations(*pops*)

w_stateprobs()

go()

class westpa.cli.tools.w_stateprobs.WStateProbs(*parent*)

Bases: *DStateProbs*

subcommand = 'trace'

help_text = 'averages and CIs for path-tracing kinetics analysis'

default_output_file = 'stateprobs.h5'

default_kinetics_file = 'assign.h5'

class westpa.cli.tools.w_stateprobs.**WDirect**

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_stateprobs'

subcommands = [<class 'westpa.cli.tools.w_stateprobs.WStateProbs'>]

subparsers_title = 'calculate state-to-state kinetics by tracing trajectories'

description = 'Calculate average populations and associated errors in state
populations from\nweighted ensemble data. Bin assignments, including macrostate
definitions,\nare required. (See "w_assign --help" for more information).\n\
n---\
nOutput
format\
n---\n\
nThe output file (-o/--output, usually "stateprobs.h5") contains the
following\ndataset:\n\n /avg_state_pops [state]\n (Structured -- see below)
Population of each state across entire\n range specified.\n\nIf --evolution-mode is
specified, then the following additional dataset is\navailable:\n\n
/state_pop_evolution [window][state]\n (Structured -- see below). State populations
based on windows of\n iterations of varying width. If --evolution-mode=cumulative,
then\n these windows all begin at the iteration specified with\n --start-iter and
grow in length by --step-iter for each successive\n element. If
--evolution-mode=blocked, then these windows are all of\n width --step-iter
(excluding the last, which may be shorter), the first\n of which begins at iteration
--start-iter.\n\nThe structure of these datasets is as follows:\n\n iter_start\n
(Integer) Iteration at which the averaging window begins (inclusive).\n\n
iter_stop\n (Integer) Iteration at which the averaging window ends (exclusive).\n\n
expected\n (Floating-point) Expected (mean) value of the rate as evaluated within\n
this window, in units of inverse tau.\n\n ci_lbound\n (Floating-point) Lower bound
of the confidence interval on the rate\n within this window, in units of inverse
tau.\n\n ci_ubound\n (Floating-point) Upper bound of the confidence interval on the
rate\n within this window, in units of inverse tau.\n\n corr_len\n (Integer)
Correlation length of the rate within this window, in units\n of tau.\n\nEach of
these datasets is also stamped with a number of attributes:\n\n mcbs_alpha\n
(Floating-point) Alpha value of confidence intervals. (For example,\n *alpha=0.05*
corresponds to a 95% confidence interval.)\n\n mcbs_nsets\n (Integer) Number of
bootstrap data sets used in generating confidence\n intervals.\n\n mcbs_acalpha\n
(Floating-point) Alpha value for determining correlation lengths.\n\n\
n---\
nCommand-line
options\
n---\n'

westpa.cli.tools.w_stateprobs.**entry_point**()

6.1.21.4 w_dumpsegs

6.1.21.4.1 westpa.cli.tools.w_dumpsegs module

westpa.cli.tools.w_dumpsegs.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_dumpsegs.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.cli.tools.w_dumpsegs.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_dumpsegs.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.cli.tools.w_dumpsegs.**WDumpSegs**

Bases: *WESTTool*

prog = 'w_dumpsegs'

description = 'Dump segment data as text. This is very inefficient, so this tool should be used\nas a last resort (use hdfview/h5ls to look at data, and access HDF5 directly for\nsignificant analysis tasks).\n'

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_dumpsegs.**entry_point**()

6.1.21.5 w_postanalysis_matrix

6.1.21.5.1 westpa.cli.tools.w_postanalysis_matrix module

class westpa.cli.tools.w_postanalysis_matrix.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_postanalysis_matrix.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_postanalysis_matrix.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_postanalysis_matrix.**RWMatrix**(*parent*)

Bases: *WESTKineticsBase*, *FluxMatrix*

subcommand = 'init'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

help_text = 'create a color-labeled transition matrix from a WESTPA simulation'

description = 'Generate a colored transition matrix from a WE assignment file. The subsequent\nanalysis requires that the assignments are calculated using only the initial and\nfinal time points of each trajectory segment. This may require downsampling the\nh5file generated by a WE simulation. In the future w_assign may be enhanced to optionally\ngenerate the necessary assignment file from a h5file with intermediate time points.\nAdditionally, this analysis is currently only valid on simulations performed under\neither equilibrium or steady-state conditions without recycling target states.\n\
n--\
nOutput
format\
n--\n\
nThe output file (-o/--output, by default "reweight.h5") contains the\nfollowing datasets:\n\n ``/bin_populations`` [window, bin]\n The reweighted populations of each bin based on windows. Bins contain\n one color each, so to recover the original un-colored spatial bins,\n one must sum over all states.\n\n ``/iterations`` [iteration]\n *(Structured -- see below)* Sparse matrix data from each\n iteration. They are reconstructed and averaged within the\n w_reweight {kinetics/probs} routines so that observables may\n be calculated. Each group contains 4 vectors of data:\n\n flux\n *(Floating-point)* The weight of a series of flux events\n cols\n *(Integer)* The bin from which a flux event began.\n cols\n *(Integer)* The bin into which the walker fluxed.\n obs\n *(Integer)* How many flux events were observed during this\n iteration.\n\
n--\
nCommand-line
options\
n--\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

go()

class westpa.cli.tools.w_postanalysis_matrix.**PAMatrix**(*parent*)

Bases: *RWMatrix*

subcommand = 'init'

help_text = 'averages and CIs for path-tracing kinetics analysis'

default_output_file = 'flux_matrices.h5'

class westpa.cli.tools.w_postanalysis_matrix.**WReweight**

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_postanalysis_matrix'

subcommands = [<class 'westpa.cli.tools.w_postanalysis_matrix.PAMatrix'>]

subparsers_title = 'calculate state-to-state kinetics by tracing trajectories'

```
    description = 'Generate a colored transition matrix from a WE assignment file. The
    subsequent\nanalysis requires that the assignments are calculated using only the
    initial and\nfinal time points of each trajectory segment. This may require
    downsampling the\nh5file generated by a WE simulation. In the future w_assign may be
    enhanced to optionally\ngenerate the necessary assignment file from a h5file with
    intermediate time points.\nAdditionally, this analysis is currently only valid on
    simulations performed under\neither equilibrium or steady-state conditions without
    recycling target states.\n\
    n-------------------------------------------------------------------------\
    nOutput
    format\
    n-------------------------------------------------------------------------\n\
    nThe output file (-o/--output, by default "reweight.h5") contains the\nfollowing
    datasets:\n\n ``/bin_populations`` [window, bin]\n The reweighted populations of
    each bin based on windows. Bins contain\n one color each, so to recover the original
    un-colored spatial bins,\n one must sum over all states.\n\n ``/iterations``
    [iteration]\n *(Structured -- see below)* Sparse matrix data from each\n iteration.
    They are reconstructed and averaged within the\n w_reweight {kinetics/probs}
    routines so that observables may\n be calculated. Each group contains 4 vectors of
    data:\n\n flux\n *(Floating-point)* The weight of a series of flux events\n cols\n
    *(Integer)* The bin from which a flux event began.\n cols\n *(Integer)* The bin into
    which the walker fluxed.\n obs\n *(Integer)* How many flux events were observed
    during this\n iteration.\n\
    n-------------------------------------------------------------------------\
    nCommand-line
    options\
    n-------------------------------------------------------------------------\n'
```

westpa.cli.tools.w_postanalysis_matrix.**entry_point**()

6.1.21.6 w_postanalysis_reweight

6.1.21.6.1 westpa.cli.tools.w_postanalysis_reweight module

class westpa.cli.tools.w_postanalysis_reweight.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_postanalysis_reweight.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

westpa.cli.tools.w_postanalysis_reweight.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

class westpa.cli.tools.w_postanalysis_reweight.**RWAverage**(*parent*)

Bases: *RWStateProbs*, *RWRate*

subcommand = 'average'

help_text = 'Averages and returns fluxes, rates, and color/state populations.'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

```
description = 'A convenience function to run kinetics/probs. Bin
assignments,\nincluding macrostate definitions, are required. (See\n"w_assign
--help" for more information).\n\nFor more information on the individual subcommands
this subs in for, run\nw_reweight {kinetics/probs} --help.\n\
n-------------------------------------------------------------------------------\
nCommand-line
options\
n-------------------------------------------------------------------------------\n'
```

go()

class westpa.cli.tools.w_postanalysis_reweight.**PAAverage**(*parent*)

Bases: *RWAverage*

subcommand = 'average'

help_text = ''

default_output_file = 'kinrw.h5'

default_kinetics_file = 'flux_matrices.h5'

class westpa.cli.tools.w_postanalysis_reweight.**WReweight**

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_postanalysis_reweight'

subcommands = [<class 'westpa.cli.tools.w_postanalysis_reweight.PAAverage'>]

subparsers_title = 'calculate state-to-state kinetics by tracing trajectories'

description = 'A convenience function to run kinetics/probs. Bin assignments,\nincluding macrostate definitions, are required. (See\n"w_assign --help" for more information).\n\nFor more information on the individual subcommands this subs in for, run\nw_reweight {kinetics/probs} --help.\n\n----------\nCommand-line options\n----------\n'

westpa.cli.tools.w_postanalysis_reweight.**entry_point**()

6.1.21.7 w_reweight

6.1.21.7.1 westpa.cli.tools.w_reweight module

class westpa.cli.tools.w_reweight.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.cli.tools.w_reweight.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.cli.tools.w_reweight.**WESTKineticsBase**(*parent*)

Bases: *WESTSubcommand*

Common argument processing for w_direct/w_reweight subcommands. Mostly limited to handling input and output from w_assign.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.cli.tools.w_reweight.**AverageCommands**(*parent*)

Bases: *WESTKineticsBase*

default_output_file = 'direct.h5'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

stamp_mcbs_info(*dataset*)

open_files()

open_assignments()

print_averages(*dataset*, *header*, *dim=1*)

run_calculation(*pi*, *nstates*, *start_iter*, *stop_iter*, *step_iter*, *dataset*, *eval_block*, *name*, *dim*, *do_averages=False*, ***extra*)

westpa.cli.tools.w_reweight.**generate_future**(*work_manager*, *name*, *eval_block*, *kwargs*)

westpa.cli.tools.w_reweight.**mcbs_ci_correl**(*estimator_datasets*, *estimator*, *alpha*, *n_sets=None*, *args=None*, *autocorrel_alpha=None*, *autocorrel_n_sets=None*, *subsample=None*, *do_correl=True*, *mcbs_enable=None*, *estimator_kwargs={}*)

Perform a Monte Carlo bootstrap estimate for the (1-`alpha`) confidence interval on the given `dataset` with the given `estimator`. This routine is appropriate for time-correlated data, using the method described in Huber & Kim, "Weighted-ensemble Brownian dynamics simulations for protein association reactions" (1996), doi:10.1016/S0006-3495(96)79552-8 to determine a statistically-significant correlation time and then reducing the dataset by a factor of that correlation time before running a "classic" Monte Carlo bootstrap.

Returns `(estimate, ci_lb, ci_ub, correl_time)` where `estimate` is the application of the given `estimator` to the input `dataset`, `ci_lb` and `ci_ub` are the lower and upper limits, respectively, of the (1-`alpha`) confidence interval on `estimate`, and `correl_time` is the correlation time of the dataset, significant to (1-`autocorrel_alpha`).

`estimator` is called as `estimator(dataset, *args, **kwargs)`. Common estimators include:

- np.mean – calculate the confidence interval on the mean of `dataset`
- np.median – calculate a confidence interval on the median of `dataset`
- np.std – calculate a confidence interval on the standard deviation of `datset`.

`n_sets` is the number of synthetic data sets to generate using the given `estimator`, which will be chosen using **`get_bssize()`_** if `n_sets` is not given.

`autocorrel_alpha` (which defaults to `alpha`) can be used to adjust the significance level of the autocorrelation calculation. Note that too high a significance level (too low an alpha) for evaluating the significance of autocorrelation values can result in a failure to detect correlation if the autocorrelation function is noisy.

The given `subsample` function is used, if provided, to subsample the dataset prior to running the full Monte Carlo bootstrap. If none is provided, then a random entry from each correlated block is used as the value for that block. Other reasonable choices include `np.mean`, `np.median`, `(lambda x: x[0])` or `(lambda x: x[-1])`. In particular, using `subsample=np.mean` will converge to the block averaged mean and standard error, while accounting for any non-normality in the distribution of the mean.

westpa.cli.tools.w_reweight.**reweight_for_c**(*rows*, *cols*, *obs*, *flux*, *insert*, *indices*, *nstates*, *nbins*, *state_labels*, *state_map*, *nfbins*, *istate*, *jstate*, *stride*, *bin_last_state_map*, *bin_state_map*, *return_obs*, *obs_threshold=1*)

class westpa.cli.tools.w_reweight.**FluxMatrix**

Bases: `object`

w_postanalysis_matrix()

class westpa.cli.tools.w_reweight.**RWMatrix**(*parent*)

Bases: *WESTKineticsBase*, *FluxMatrix*

subcommand = 'init'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

help_text = 'create a color-labeled transition matrix from a WESTPA simulation'

description = 'Generate a colored transition matrix from a WE assignment file. The subsequent\nanalysis requires that the assignments are calculated using only the initial and\nfinal time points of each trajectory segment. This may require downsampling the\nh5file generated by a WE simulation. In the future w_assign may be enhanced to optionally\ngenerate the necessary assignment file from a h5file with intermediate time points.\nAdditionally, this analysis is currently only valid on simulations performed under\neither equilibrium or steady-state conditions without recycling target states.\n\
n--\
nOutput
format\
n--\n\
nThe output file (-o/--output, by default "reweight.h5") contains the\nfollowing datasets:\n\n ``/bin_populations`` [window, bin]\n The reweighted populations of each bin based on windows. Bins contain\n one color each, so to recover the original un-colored spatial bins,\n one must sum over all states.\n\n ``/iterations`` [iteration]\n *(Structured -- see below)* Sparse matrix data from each\n iteration. They are reconstructed and averaged within the\n w_reweight {kinetics/probs} routines so that observables may\n be calculated. Each group contains 4 vectors of data:\n\n flux\n *(Floating-point)* The weight of a series of flux events\n cols\n *(Integer)* The bin from which a flux event began.\n cols\n *(Integer)* The bin into which the walker fluxed.\n obs\n *(Integer)* How many flux events were observed during this\n iteration.\n\
n--\
nCommand-line
options\
n--\n'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

go()

class westpa.cli.tools.w_reweight.RWReweight(*parent*)

Bases: *AverageCommands*

help_text = 'Parent class for all reweighting routines, as they all use the same estimator code.'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

accumulate_statistics(*start_iter*, *stop_iter*)

This function pulls previously generated flux matrix data into memory. The data is assumed to exist within an HDF5 file that is available as a property. The data is kept as a single dimensional numpy array to use with the cython estimator.

generate_reweight_data()

This function ensures all the appropriate files are loaded, sets appropriate attributes necessary for all calling functions/children, and then calls the function to load in the flux matrix data.

class westpa.cli.tools.w_reweight.**RWRate**(*parent*)

Bases: *RWReweight*

subcommand = 'kinetics'

help_text = 'Generates rate and flux values from a WESTPA simulation via reweighting.'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

```
description = 'Calculate average rates from weighted ensemble data using the
postanalysis\nreweighting scheme. Bin assignments (usually "assign.h5") and
pre-calculated\niteration flux matrices (usually "reweight.h5") data files must have
been\npreviously generated using w_reweight matrix (see "w_assign --help"
and\n"w_reweight init --help" for information on generating these files).\n\
n-------------------------------------------------------------------------------\
nOutput
format\
n-------------------------------------------------------------------------------\nThe
output file (-o/--output, usually "kinrw.h5") contains the following\ndataset:\n\n
/avg_rates [state,state]\n (Structured -- see below) State-to-state rates based on
entire window of\n iterations selected.\n\n /avg_total_fluxes [state]\n (Structured
-- see below) Total fluxes into each state based on entire\n window of iterations
selected.\n\n /avg_conditional_fluxes [state,state]\n (Structured -- see below)
State-to-state fluxes based on entire window of\n iterations selected.\n\nIf
--evolution-mode is specified, then the following additional datasets
are\navailable:\n\n /rate_evolution [window][state][state]\n (Structured -- see
below). State-to-state rates based on windows of\n iterations of varying width. If
--evolution-mode=cumulative, then\n these windows all begin at the iteration
specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\n /target_flux_evolution [window,state]\n
(Structured -- see below). Total flux into a given macro state based on\n windows of
iterations of varying width, as in /rate_evolution.\n\n /conditional_flux_evolution
[window,state,state]\n (Structured -- see below). State-to-state fluxes based on
windows of\n varying width, as in /rate_evolution.\n\nThe structure of these
datasets is as follows:\n\n iter_start\n (Integer) Iteration at which the averaging
window begins (inclusive).\n\n iter_stop\n (Integer) Iteration at which the
averaging window ends (exclusive).\n\n expected\n (Floating-point) Expected (mean)
value of the observable as evaluated within\n this window, in units of inverse
tau.\n\n ci_lbound\n (Floating-point) Lower bound of the confidence interval of the
observable\n within this window, in units of inverse tau.\n\n ci_ubound\n
(Floating-point) Upper bound of the confidence interval of the observable\n within
this window, in units of inverse tau.\n\n stderr\n (Floating-point) The standard
error of the mean of the observable\n within this window, in units of inverse
tau.\n\n corr_len\n (Integer) Correlation length of the observable within this
window, in units\n of tau.\n\nEach of these datasets is also stamped with a number
of attributes:\n\n mcbs_alpha\n (Floating-point) Alpha value of confidence
intervals. (For example,\n *alpha=0.05* corresponds to a 95% confidence
interval.)\n\n mcbs_nsets\n (Integer) Number of bootstrap data sets used in
generating confidence\n intervals.\n\n mcbs_acalpha\n (Floating-point) Alpha value
for determining correlation lengths.\n\n\
n-------------------------------------------------------------------------------\
nCommand-line
options\
n-------------------------------------------------------------------------------\n '
```

w_postanalysis_reweight()

This function ensures the data is ready to send in to the estimator and the bootstrapping routine, then does so. Much of this is simply setting up appropriate args and kwargs, then passing them in to the 'run_calculation' function, which sets up future objects to send to the work manager. The results are returned, and then written to the appropriate HDF5 dataset. This function is specific for the rates and fluxes from the reweighting method.

go()

class westpa.cli.tools.w_reweight.RWStateProbs(*parent*)

Bases: *RWReweight*

subcommand = 'probs'

help_text = 'Calculates color and state probabilities via reweighting.'

default_kinetics_file = 'reweight.h5'

```
description = 'Calculate average populations from weighted ensemble data using the
postanalysis\nreweighting scheme. Bin assignments (usually "assign.h5") and
pre-calculated\niteration flux matrices (usually "reweight.h5") data files must have
been\npreviously generated using w_reweight matrix (see "w_assign --help"
and\n"w_reweight init --help" for information on generating these files).\n\
n-------------------------------------------------------------------------------\
nOutput
format\
n-------------------------------------------------------------------------------\n\
nThe output file (-o/--output, usually "direct.h5") contains the
following\ndataset:\n\n /avg_state_probs [state]\n (Structured -- see below)
Population of each state across entire\n range specified.\n\n /avg_color_probs
[state]\n (Structured -- see below) Population of each ensemble across entire\n
range specified.\n\nIf --evolution-mode is specified, then the following additional
datasets are\navailable:\n\n /state_pop_evolution [window][state]\n (Structured --
see below). State populations based on windows of\n iterations of varying width. If
--evolution-mode=cumulative, then\n these windows all begin at the iteration
specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\n /color_prob_evolution [window][state]\n
(Structured -- see below). Ensemble populations based on windows of\n iterations of
varying width. If --evolution-mode=cumulative, then\n these windows all begin at the
iteration specified with\n --start-iter and grow in length by --step-iter for each
successive\n element. If --evolution-mode=blocked, then these windows are all of\n
width --step-iter (excluding the last, which may be shorter), the first\n of which
begins at iteration --start-iter.\n\nThe structure of these datasets is as
follows:\n\n iter_start\n (Integer) Iteration at which the averaging window begins
(inclusive).\n\n iter_stop\n (Integer) Iteration at which the averaging window ends
(exclusive).\n\n expected\n (Floating-point) Expected (mean) value of the observable
as evaluated within\n this window, in units of inverse tau.\n\n ci_lbound\n
(Floating-point) Lower bound of the confidence interval of the observable\n within
this window, in units of inverse tau.\n\n ci_ubound\n (Floating-point) Upper bound
of the confidence interval of the observable\n within this window, in units of
inverse tau.\n\n stderr\n (Floating-point) The standard error of the mean of the
observable\n within this window, in units of inverse tau.\n\n corr_len\n (Integer)
Correlation length of the observable within this window, in units\n of
tau.\n\n\nEach of these datasets is also stamped with a number of attributes:\n\n
mcbs_alpha\n (Floating-point) Alpha value of confidence intervals. (For example,\n
*alpha=0.05* corresponds to a 95% confidence interval.)\n\n mcbs_nsets\n (Integer)
Number of bootstrap data sets used in generating confidence\n intervals.\n\n
mcbs_acalpha\n (Floating-point) Alpha value for determining correlation lengths.\n\
n\n-------------------------------------------------------------------------------\
nCommand-line
options\
n-------------------------------------------------------------------------------\n'
```

w_postanalysis_stateprobs()

This function ensures the data is ready to send in to the estimator and the bootstrapping routine, then does so. Much of this is simply setting up appropriate args and kwargs, then passing them in to the 'run_calculation' function, which sets up future objects to send to the work manager. The results are returned, and then written to the appropriate HDF5 dataset. This function is specific for the color (steady-state) and macrostate probabilities from the reweighting method.

go()

class westpa.cli.tools.w_reweight.**RWAll**(*parent*)

Bases: *RWMatrix*, *RWStateProbs*, *RWRate*

subcommand = 'all'

help_text = 'Runs the full suite, including the generation of the flux matrices.'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

description = 'A convenience function to run init/kinetics/probs. Bin assignments,\nincluding macrostate definitions, are required. (See\n"w_assign --help" for more information).\n\nFor more information on the individual subcommands this subs in for, run\nw_reweight {init/kinetics/probs} --help.\n\n----------\nCommand-line options\n----------\n'

go()

class westpa.cli.tools.w_reweight.**RWAverage**(*parent*)

Bases: *RWStateProbs*, *RWRate*

subcommand = 'average'

help_text = 'Averages and returns fluxes, rates, and color/state populations.'

default_kinetics_file = 'reweight.h5'

default_output_file = 'reweight.h5'

description = 'A convenience function to run kinetics/probs. Bin assignments,\nincluding macrostate definitions, are required. (See\n"w_assign --help" for more information).\n\nFor more information on the individual subcommands this subs in for, run\nw_reweight {kinetics/probs} --help.\n\n----------\nCommand-line options\n----------\n'

go()

class westpa.cli.tools.w_reweight.**WReweight**

Bases: *WESTMasterCommand*, *WESTParallelTool*

prog = 'w_reweight'

subcommands = [<class 'westpa.cli.tools.w_reweight.RWMatrix'>, <class 'westpa.cli.tools.w_reweight.RWAverage'>, <class 'westpa.cli.tools.w_reweight.RWRate'>, <class 'westpa.cli.tools.w_reweight.RWStateProbs'>, <class 'westpa.cli.tools.w_reweight.RWAll'>]

subparsers_title = 'reweighting kinetics analysis scheme'

westpa.cli.tools.w_reweight.**entry_point**()

6.1.21.8 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.

6.1.21.8.1 Overview

usage:

```
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`.

6.1.21.8.2 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)

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

6.1.21.8.3 Examples

Calculate the time evolution flux every 5 iterations:

```
w_fluxanl --evol --evol-step 5
```

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

```
w_fluxanl --alpha 0.01 --autocorrel-alpha 0.05
```

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

```
w_fluxanl --n-sets 500
```

6.1.21.8.4 westpa.cli.tools.w_fluxanl module

`westpa.cli.tools.w_fluxanl.`**`fftconvolve`**(*in1*, *in2*, *mode='full'*, *axes=None*)

Convolve two N-dimensional arrays using FFT.

Convolve *in1* and *in2* using the fast Fourier transform method, with the output size determined by the *mode* argument.

This is generally much faster than *convolve* for large arrays (n > ~500), but can be slower when only a few output values are needed, and can only output float arrays (int or object array inputs will be cast to float).

As of v0.19, *convolve* automatically chooses this method or the direct method based on an estimation of which is faster.

Parameters

- **in1** (*array_like*) – First input.
- **in2** (*array_like*) – Second input. Should have the same number of dimensions as *in1*.
- **mode** (*str {'full', 'valid', 'same'}, optional*) – A string indicating the size of the output:

  full
  The output is the full discrete linear convolution of the inputs. (Default)

  valid
  The output consists only of those elements that do not rely on the zero-padding. In 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

  same
  The output is the same size as *in1*, centered with respect to the 'full' output.

- **axes** (*int or array_like of ints or None, optional*) – Axes over which to compute the convolution. The default is over all axes.

Returns

out – An N-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

Return type

array

See also:

convolve
Uses the direct convolution or FFT convolution algorithm depending on which is faster.

oaconvolve
Uses the overlap-add method to do convolution, which is generally faster when the input arrays are large and significantly different in size.

Examples

Autocorrelation of white noise is an impulse.

```
>>> import numpy as np
>>> from scipy import signal
>>> rng = np.random.default_rng()
>>> sig = rng.standard_normal(1000)
>>> autocorr = signal.fftconvolve(sig, sig[::-1], mode='full')
```

```
>>> import matplotlib.pyplot as plt
>>> fig, (ax_orig, ax_mag) = plt.subplots(2, 1)
>>> ax_orig.plot(sig)
>>> ax_orig.set_title('White noise')
>>> ax_mag.plot(np.arange(-len(sig)+1,len(sig)), autocorr)
>>> ax_mag.set_title('Autocorrelation')
>>> fig.tight_layout()
>>> fig.show()
```

Gaussian blur implemented using FFT convolution. Notice the dark borders around the image, due to the zero-padding beyond its boundaries. The *convolve2d* function allows for other types of image boundaries, but is far

slower.

```
>>> from scipy import datasets
>>> face = datasets.face(gray=True)
>>> kernel = np.outer(signal.windows.gaussian(70, 8),
...                   signal.windows.gaussian(70, 8))
>>> blurred = signal.fftconvolve(face, kernel, mode='same')
```

```
>>> fig, (ax_orig, ax_kernel, ax_blurred) = plt.subplots(3, 1,
...                                                      figsize=(6, 15))
>>> ax_orig.imshow(face, cmap='gray')
>>> ax_orig.set_title('Original')
>>> ax_orig.set_axis_off()
>>> ax_kernel.imshow(kernel, cmap='gray')
>>> ax_kernel.set_title('Gaussian kernel')
>>> ax_kernel.set_axis_off()
>>> ax_blurred.imshow(blurred, cmap='gray')
>>> ax_blurred.set_title('Blurred')
>>> ax_blurred.set_axis_off()
>>> fig.show()
```

westpa.cli.tools.w_fluxanl.**warn**()

Issue a warning, or maybe ignore it or raise an exception.

message
Text of the warning message.

category
The Warning category subclass. Defaults to UserWarning.

stacklevel
How far up the call stack to make this warning appear. A value of 2 for example attributes the warning to the caller of the code calling warn().

source
If supplied, the destroyed object which emitted a ResourceWarning

skip_file_prefixes
An optional tuple of module filename prefixes indicating frames to skip during stacklevel computations for stack frame attribution.

westpa.cli.tools.w_fluxanl.**weight_dtype**

alias of `float64`

westpa.cli.tools.w_fluxanl.**n_iter_dtype**

alias of `uint32`

class westpa.cli.tools.w_fluxanl.**NewWeightEntry**(*source_type*, *weight*, *prev_seg_id=None*, *prev_init_pcoord=None*, *prev_final_pcoord=None*, *new_init_pcoord=None*, *target_state_id=None*, *initial_state_id=None*)

Bases: `object`

NW_SOURCE_RECYCLED = 0

class westpa.cli.tools.w_fluxanl.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.cli.tools.w_fluxanl.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.cli.tools.w_fluxanl.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
: The first iteration included in the calculation.

last_iter
: One past the last iteration included in the calculation.

iter_step
Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

westpa.cli.tools.w_fluxanl.**extract_fluxes**(*iter_start=None*, *iter_stop=None*, *data_manager=None*)

Extract flux values from the WEST HDF5 file for iterations >= iter_start and < iter_stop, optionally using another data manager instance instead of the global one returned by `westpa.rc.get_data_manager()`.

Returns a dictionary mapping target names (if available, target index otherwise) to a 1-D array of type `fluxentry_dtype`, which contains columns for iteration number, flux, and count.

class westpa.cli.tools.w_fluxanl.**WFluxanlTool**

Bases: *WESTTool*

prog = 'w_fluxanl'

description = 'Extract fluxes into pre-defined target states from WEST
data,\naverage, and construct confidence intervals. Monte Carlo bootstrapping\nis
used to account for the correlated and possibly non-Gaussian statistical\nerror in
flux measurements.\n\nAll non-graphical output (including that to the terminal and
HDF5) assumes that\nthe propagation/resampling period ``tau`` is equal to unity; to
obtain results\nin familiar units, divide all fluxes and multiply all correlation
lengths by\nthe true value of ``tau``.\n'

output_format_version = 2

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

calc_store_flux_data()

calc_evol_flux()

go()

Perform the analysis associated with this tool.

westpa.cli.tools.w_fluxanl.**entry_point**()

6.2 westpa.core package

6.2.1 westpa.core.binning package

6.2.1.1 westpa.core.binning module

class westpa.core.binning.**NopMapper**

Bases: *BinMapper*

Put everything into one bin.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**FuncBinMapper**(*func*, *nbins*, *args=None*, *kwargs=None*)

Bases: *BinMapper*

Binning using a custom function which must iterate over input coordinate sets itself.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**PiecewiseBinMapper**(*functions*)

Bases: *BinMapper*

Binning using a set of functions returing boolean values; if the Nth function returns True for a coordinate tuple, then that coordinate is in the Nth bin.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**RectilinearBinMapper**(*boundaries*)

Bases: *BinMapper*

Bin into a rectangular grid based on tuples of float values

property boundaries

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**RecursiveBinMapper**(*base_mapper*, *start_index=0*)

Bases: *BinMapper*

Nest mappers one within another.

property labels

property start_index

add_mapper(*mapper*, *replaces_bin_at*)

Replace the bin containing the coordinate tuple `replaces_bin_at` with the specified `mapper`.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**VectorizingFuncBinMapper**(*func*, *nbins*, *args=None*, *kwargs=None*)

Bases: *BinMapper*

Binning using a custom function which is evaluated once for each (unmasked) coordinate tuple provided.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.**VoronoiBinMapper**(*dfunc*, *centers*, *dfargs=None*, *dfkwargs=None*)

Bases: *BinMapper*

A one-dimensional mapper which assigns a multidimensional pcoord to the closest center based on a distance metric. Both the list of centers and the distance function must be supplied.

assign(*coords*, *mask=None*, *output=None*)

westpa.core.binning.**map_mab**(*coords*, *mask*, *output*, **args*, ***kwargs*)

Binning which adaptively places bins based on the positions of extrema segments and bottleneck segments, which are where the difference in probability is the greatest along the progress coordinate. Operates per dimension and places a fixed number of evenly spaced bins between the segments with the min and max pcoord values. Extrema and bottleneck segments are assigned their own bins.

Parameters

- **coords** (*ndarray*) – An array with pcoord and weight info.
- **mask** (*ndarray*) – Array of 1 (True) and 0 (False), to filter out unwanted segment info.
- **output** (*list*) – The main list that, for each segment, holds the bin assignment.
- ***args** (*list*) – Variable length arguments.
- ****kwargs** (*dict*) – Arbitary keyword arguments. Contains most of the MAB-needed parameters.

Returns

output – The main list that, for each segment, holds the bin assignment.

Return type

list

westpa.core.binning.**map_binless**(*coords*, *mask*, *output*, **args*, ***kwargs*)

Adaptively groups walkers according to a user-defined grouping function that is defined externally. Very general implementation but limited to only a two dimensional progress coordinate (for now).

class westpa.core.binning.**MABBinMapper**(*nbins*, *direction=None*, *skip=None*, *bottleneck=True*, *pca=False*, *mab_log=False*, *bin_log=False*, *bin_log_path='$WEST_SIM_ROOT/binbounds.log'*)

Bases: *FuncBinMapper*

Adaptively place bins in between minimum and maximum segments along the progress coordinte. Extrema and bottleneck segments are assigned to their own bins.

Parameters

- **nbins** (*list of int*) – List of int for nbins in each dimension.
- **direction** (*Union(list of int, None), default: None*) – List of int for 'direction' in each dimension. Direction options are as follows:

  0 : default split at leading and lagging boundaries 1 : split at leading boundary only -1 : split at lagging boundary only 86 : no splitting at either leading or lagging boundary
- **skip** (*Union(list of int, None), default: None*) – List of int for each dimension. Default None for skip=0. Set to 1 to 'skip' running mab in a dimension.
- **bottleneck** (*bool, default: True*) – Whether to turn on or off bottleneck walker splitting.
- **pca** (*bool, default: False*) – Can be True or False (default) to run PCA on pcoords before bin assignment.
- **mab_log** (*bool, default: False*) – Whether to output mab info to west.log.
- **bin_log** (*bool, default: False*) – Whether to output mab bin boundaries to bin_log_path file.
- **bin_log_path** (*str, default: "$WEST_SIM_ROOT/binbounds.log"*) – Path to output bin boundaries.

determine_total_bins(*nbins_per_dim*, *direction*, *skip*, *bottleneck*, ***kwargs*)

The following is neccessary because functional bin mappers need to "reserve" bins and tell the sim manager how many bins they will need to use, this is determined by taking all direction/skipping info into account.

Parameters

- **nbins_per_dim** (*int*) – Number of total bins in each direction.
- **direction** (*list of int*) – Direction in each dimension. See __init__ for more information.
- **skip** (*list of int*) – List of 0s and 1s indicating whether to skip each dimension.
- **bottleneck** (*bool*) – Whether to include separate bin for bottleneck walker(s).
- ****kwargs** (*dict*) – Arbitary keyword arguments. Contains unneeded MAB parameters.

Returns

n_total_bins – Number of total bins.

Return type

int

class westpa.core.binning.**BinlessMapper**(*ngroups*, *ndims*, *group_function*, ***group_function_kwargs*)

Bases: *FuncBinMapper*

Adaptively group walkers according to a user-defined grouping function that is defined externally.

class westpa.core.binning.**MABDriver**(*rc=None*, *system=None*)

Bases: *WEDriver*

assign(*segments*, *initializing=False*)

Assign segments to initial and final bins, and update the (internal) lists of used and available initial states. This function is adapted to the MAB scheme, so that the inital and final segments are sent to the bin mapper at the same time, otherwise the inital and final bin boundaries can be inconsistent.

class westpa.core.binning.**MABSimManager**(*rc=None*)

Bases: *WESimManager*

Subclass of WESimManager, modifying it so bin assignments will be done after all segments are done propagating.

initialize_simulation(*basis_states*, *target_states*, *start_states*, *segs_per_state=1*, *suppress_we=False*)

Making sure that that the MABBinMapper is not the outer bin.

propagate()

prepare_iteration()

class westpa.core.binning.**BinlessDriver**(*rc=None*, *system=None*)

Bases: *WEDriver*

assign(*segments*, *initializing=False*)

Assign segments to initial and final bins, and update the (internal) lists of used and available initial states. This function is adapted to the MAB scheme, so that the inital and final segments are sent to the bin mapper at the same time, otherwise the inital and final bin boundaries can be inconsistent.

class westpa.core.binning.**BinlessSimManager**(*rc=None*)

Bases: *WESimManager*

initialize_simulation(*basis_states*, *target_states*, *start_states*, *segs_per_state=1*, *suppress_we=False*)

Initialize a new weighted ensemble simulation, taking `segs_per_state` initial states from each of the given `basis_states`.

`w_init` is the forward-facing version of this function

propagate()

prepare_iteration()

westpa.core.binning.**accumulate_labeled_populations**(*weights*, *bin_assignments*, *label_assignments*, *labeled_bin_pops*)

For a set of segments in one iteration, calculate the average population in each bin, with separation by last-visited macrostate.

westpa.core.binning.**assign_and_label**(*nsegs_lb*, *nsegs_ub*, *parent_ids*, *assign*, *nstates*, *state_map*, *last_labels*, *pcoords*, *subsample*)

Assign trajectories to bins and last-visted macrostates for each timepoint.

westpa.core.binning.**accumulate_state_populations_from_labeled**(*labeled_bin_pops*, *state_map*, *state_pops*, *check_state_map=True*)

westpa.core.binning.**assignments_list_to_table**(*nsegs*, *nbins*, *assignments*)

Convert a list of bin assignments (integers) to a boolean table indicating indicating if a given segment is in a given bin

westpa.core.binning.**coord_dtype**

alias of `float32`

westpa.core.binning.**index_dtype**

alias of `uint16`

class westpa.core.binning.**Bin**(*iterable=None*, *label=None*)

Bases: `set`

property weight

Total weight of all walkers in this bin

reweight(*new_weight*)

Reweight all walkers in this bin so that the total weight is new_weight

6.2.1.2 westpa.core.binning.assign module

Bin assignment for WEST simulations. This module defines "bin mappers" which take vectors of coordinates (or rather, coordinate tuples), and assign each a definite integer value identifying a bin. Critical portions are implemented in a Cython extension module.

A number of pre-defined bin mappers are available here:

- *RectilinearBinMapper*, for bins divided by N-dimensional grids
- *FuncBinMapper*, for functions which directly calculate bin assignments for a number of coordinate values. This is best used with C/Cython/Numba functions, or intellegently-tuned numpy-based Python functions.
- *VectorizingFuncBinMapper*, for functions which calculate a bin assignment for a single coordinate value. This is best used for arbitrary Python functions.
- *PiecewiseBinMapper*, for using a set of boolean-valued functions, one per bin, to determine assignments. This is likely to be much slower than a *FuncBinMapper* or *VectorizingFuncBinMapper* equipped with an appropriate function, and its use is discouraged.

One "super-mapper" is available, for assembling more complex bin spaces from simpler components:

- *RecursiveBinMapper*, for nesting one set of bins within another.

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 (np.uint16) indicating a 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.

class westpa.core.binning.assign.**Bin**(*iterable=None*, *label=None*)

Bases: `set`

property weight

Total weight of all walkers in this bin

reweight(*new_weight*)

Reweight all walkers in this bin so that the total weight is new_weight

westpa.core.binning.assign.**output_map**(*output*, *omap*, *mask*)

For each output for which mask is true, execute output[i] = omap[output[i]]

westpa.core.binning.assign.**apply_down**(*func*, *args*, *kwargs*, *coords*, *mask*, *output*)

Apply func(coord, *args, **kwargs) to each input coordinate tuple, skipping any for which mask is false and writing results to output.

westpa.core.binning.assign.**apply_down_argmin_across**(*func*, *args*, *kwargs*, *func_output_len*, *coords*, *mask*, *output*)

Apply func(coord, *args, **kwargs) to each input coordinate tuple, skipping any for which mask is false and writing results to output.

westpa.core.binning.assign.**rectilinear_assign**(*coords*, *mask*, *output*, *boundaries*, *boundlens*)

For bins delimited by sets boundaries on a rectilinear grid (`boundaries`), assign coordinates to bins, assuming C ordering of indices within the grid. `boundlens` is the number of boundaries in each dimension.

westpa.core.binning.assign.**index_dtype**

alias of `uint16`

westpa.core.binning.assign.**coord_dtype**

alias of `float32`

class westpa.core.binning.assign.**BinMapper**

Bases: `object`

hashfunc(*, *usedforsecurity=True*)

Returns a sha256 hash object; optionally initialized with a string

construct_bins(*type_=<class 'westpa.core.binning.bins.Bin'>*)

Construct and return an array of bins of type `type`

pickle_and_hash()

Pickle this mapper and calculate a hash of the result (thus identifying the contents of the pickled data), returning a tuple `(pickled_data, hash)`. This will raise PickleError if this mapper cannot be pickled, in which case code that would otherwise rely on detecting a topology change must assume a topology change happened, even if one did not.

class westpa.core.binning.assign.**NopMapper**

Bases: *BinMapper*

Put everything into one bin.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**RectilinearBinMapper**(*boundaries*)

Bases: *BinMapper*

Bin into a rectangular grid based on tuples of float values

property boundaries

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**PiecewiseBinMapper**(*functions*)

Bases: *BinMapper*

Binning using a set of functions returing boolean values; if the Nth function returns True for a coordinate tuple, then that coordinate is in the Nth bin.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**FuncBinMapper**(*func*, *nbins*, *args=None*, *kwargs=None*)

Bases: *BinMapper*

Binning using a custom function which must iterate over input coordinate sets itself.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**VectorizingFuncBinMapper**(*func*, *nbins*, *args=None*, *kwargs=None*)

Bases: *BinMapper*

Binning using a custom function which is evaluated once for each (unmasked) coordinate tuple provided.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**VoronoiBinMapper**(*dfunc*, *centers*, *dfargs=None*, *dfkwargs=None*)

Bases: *BinMapper*

A one-dimensional mapper which assigns a multidimensional pcoord to the closest center based on a distance metric. Both the list of centers and the distance function must be supplied.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.binning.assign.**RecursiveBinMapper**(*base_mapper*, *start_index=0*)

Bases: *BinMapper*

Nest mappers one within another.

property labels

property start_index

add_mapper(*mapper*, *replaces_bin_at*)

Replace the bin containing the coordinate tuple `replaces_bin_at` with the specified `mapper`.

assign(*coords*, *mask=None*, *output=None*)

6.2.1.3 westpa.core.binning.bins module

class westpa.core.binning.bins.**Bin**(*iterable=None*, *label=None*)

Bases: `set`

property weight

Total weight of all walkers in this bin

reweight(*new_weight*)

Reweight all walkers in this bin so that the total weight is new_weight

6.2.2 Minimal Adaptive Binning (MAB) Scheme

6.2.2.1 westpa.core.binning.mab module

class westpa.core.binning.mab.**FuncBinMapper**(*func*, *nbins*, *args=None*, *kwargs=None*)

Bases: *BinMapper*

Binning using a custom function which must iterate over input coordinate sets itself.

assign(*coords*, *mask=None*, *output=None*)

westpa.core.binning.mab.**expandvars**(*path*)

Expand shell variables of form $var and ${var}. Unknown variables are left unchanged.

class westpa.core.binning.mab.**MABBinMapper**(*nbins*, *direction=None*, *skip=None*, *bottleneck=True*, *pca=False*, *mab_log=False*, *bin_log=False*, *bin_log_path='$WEST_SIM_ROOT/binbounds.log'*)

Bases: *FuncBinMapper*

Adaptively place bins in between minimum and maximum segments along the progress coordinte. Extrema and bottleneck segments are assigned to their own bins.

Parameters

- **nbins** (*list of int*) – List of int for nbins in each dimension.
- **direction** (*Union(list of int, None), default: None*) – List of int for 'direction' in each dimension. Direction options are as follows:

  0 : default split at leading and lagging boundaries 1 : split at leading boundary only -1 : split at lagging boundary only 86 : no splitting at either leading or lagging boundary
- **skip** (*Union(list of int, None), default: None*) – List of int for each dimension. Default None for skip=0. Set to 1 to 'skip' running mab in a dimension.
- **bottleneck** (*bool, default: True*) – Whether to turn on or off bottleneck walker splitting.
- **pca** (*bool, default: False*) – Can be True or False (default) to run PCA on pcoords before bin assignment.
- **mab_log** (*bool, default: False*) – Whether to output mab info to west.log.
- **bin_log** (*bool, default: False*) – Whether to output mab bin boundaries to bin_log_path file.
- **bin_log_path** (*str, default: "$WEST_SIM_ROOT/binbounds.log"*) – Path to output bin boundaries.

determine_total_bins(*nbins_per_dim*, *direction*, *skip*, *bottleneck*, ***kwargs*)

The following is neccessary because functional bin mappers need to "reserve" bins and tell the sim manager how many bins they will need to use, this is determined by taking all direction/skipping info into account.

Parameters

- **nbins_per_dim** (*int*) – Number of total bins in each direction.
- **direction** (*list of int*) – Direction in each dimension. See __init__ for more information.
- **skip** (*list of int*) – List of 0s and 1s indicating whether to skip each dimension.

- **bottleneck** (*bool*) – Whether to include separate bin for bottleneck walker(s).
- ****kwargs** (*dict*) – Arbitary keyword arguments. Contains unneeded MAB parameters.

Returns
n_total_bins – Number of total bins.

Return type
int

westpa.core.binning.mab.**map_mab**(*coords*, *mask*, *output*, **args*, ***kwargs*)

Binning which adaptively places bins based on the positions of extrema segments and bottleneck segments, which are where the difference in probability is the greatest along the progress coordinate. Operates per dimension and places a fixed number of evenly spaced bins between the segments with the min and max pcoord values. Extrema and bottleneck segments are assigned their own bins.

Parameters

- **coords** (*ndarray*) – An array with pcoord and weight info.
- **mask** (*ndarray*) – Array of 1 (True) and 0 (False), to filter out unwanted segment info.
- **output** (*list*) – The main list that, for each segment, holds the bin assignment.
- ***args** (*list*) – Variable length arguments.
- ****kwargs** (*dict*) – Arbitary keyword arguments. Contains most of the MAB-needed parameters.

Returns
output – The main list that, for each segment, holds the bin assignment.

Return type
list

6.2.2.2 westpa.core.binning.mab_driver

class westpa.core.binning.mab_driver.**WEDriver**(*rc=None*, *system=None*)

Bases: object

A class implemented Huber & Kim's weighted ensemble algorithm over Segment objects. This class handles all binning, recycling, and preparation of new Segment objects for the next iteration. Binning is accomplished using system.bin_mapper, and per-bin target counts are from system.bin_target_counts.

The workflow is as follows:

1) Call *new_iteration()* every new iteration, providing any recycling targets that are in force and any available initial states for recycling.
2) Call *assign()* to assign segments to bins based on their initial and end points. This returns the number of walkers that were recycled.
3) Call *run_we()*, optionally providing a set of initial states that will be used to recycle walkers.

Note the presence of flux_matrix, transition_matrix, current_iter_segments, next_iter_segments, recycling_segments, initial_binning, final_binning, next_iter_binning, and new_weights (to be documented soon).

weight_split_threshold = 2.0

weight_merge_cutoff = 1.0

largest_allowed_weight = 1.0

smallest_allowed_weight = 1e-310

process_config()

property next_iter_segments

Newly-created segments for the next iteration

property current_iter_segments

Segments for the current iteration

property next_iter_assignments

Bin assignments (indices) for initial points of next iteration.

property current_iter_assignments

Bin assignments (indices) for endpoints of current iteration.

property recycling_segments

Segments designated for recycling

property n_recycled_segs

Number of segments recycled this iteration

property n_istates_needed

Number of initial states needed to support recycling for this iteration

check_threshold_configs()

Check to see if weight thresholds parameters are valid

clear()

Explicitly delete all Segment-related state.

new_iteration(*initial_states=None*, *target_states=None*, *new_weights=None*, *bin_mapper=None*, *bin_target_counts=None*)

Prepare for a new iteration. `initial_states` is a sequence of all InitialState objects valid for use in to generating new segments for the *next* iteration (after the one being begun with the call to new_iteration); that is, these are states available to recycle to. Target states which generate recycling events are specified in `target_states`, a sequence of TargetState objects. Both `initial_states` and `target_states` may be empty as required.

The optional `new_weights` is a sequence of NewWeightEntry objects which will be used to construct the initial flux matrix.

The given `bin_mapper` will be used for assignment, and `bin_target_counts` used for splitting/merging target counts; each will be obtained from the system object if omitted or None.

add_initial_states(*initial_states*)

Add newly-prepared initial states to the pool available for recycling.

property all_initial_states

Return an iterator over all initial states (available or used)

assign(*segments*, *initializing=False*)

Assign segments to initial and final bins, and update the (internal) lists of used and available initial states. If `initializing` is True, then the "final" bin assignments will be identical to the initial bin assignments, a condition required for seeding a new iteration from pre-existing segments.

populate_initial(*initial_states*, *weights*, *system=None*)

Create walkers for a new weighted ensemble simulation.

One segment is created for each provided initial state, then binned and split/merged as necessary. After this function is called, next_iter_segments will yield the new segments to create, used_initial_states will

contain data about which of the provided initial states were used, and avail_initial_states will contain data about which initial states were unused (because their corresponding walkers were merged out of existence).

rebin_current(*parent_segments*)

Reconstruct walkers for the current iteration based on (presumably) new binning. The previous iteration's segments must be provided (as `parent_segments`) in order to update endpoint types appropriately.

construct_next()

Construct walkers for the next iteration, by running weighted ensemble recycling and bin/split/merge on the segments previously assigned to bins using `assign`. Enough unused initial states must be present in `self.avail_initial_states` for every recycled walker to be assigned an initial state.

After this function completes, `self.flux_matrix` contains a valid flux matrix for this iteration (including any contributions from recycling from the previous iteration), and `self.next_iter_segments` contains a list of segments ready for the next iteration, with appropriate values set for weight, endpoint type, parent walkers, and so on.

class westpa.core.binning.mab_driver.**MABDriver**(*rc=None*, *system=None*)

Bases: *WEDriver*

assign(*segments*, *initializing=False*)

Assign segments to initial and final bins, and update the (internal) lists of used and available initial states. This function is adapted to the MAB scheme, so that the inital and final segments are sent to the bin mapper at the same time, otherwise the inital and final bin boundaries can be inconsistent.

6.2.2.3 westpa.core.binning.mab_manager

class westpa.core.binning.mab_manager.**MABBinMapper**(*nbins*, *direction=None*, *skip=None*, *bottleneck=True*, *pca=False*, *mab_log=False*, *bin_log=False*, *bin_log_path='$WEST_SIM_ROOT/binbounds.log'*)

Bases: *FuncBinMapper*

Adaptively place bins in between minimum and maximum segments along the progress coordinte. Extrema and bottleneck segments are assigned to their own bins.

Parameters

- **nbins** (*list of int*) – List of int for nbins in each dimension.
- **direction** (*Union(list of int, None), default: None*) – List of int for 'direction' in each dimension. Direction options are as follows:

  0 : default split at leading and lagging boundaries 1 : split at leading boundary only -1 : split at lagging boundary only 86 : no splitting at either leading or lagging boundary
- **skip** (*Union(list of int, None), default: None*) – List of int for each dimension. Default None for skip=0. Set to 1 to 'skip' running mab in a dimension.
- **bottleneck** (*bool, default: True*) – Whether to turn on or off bottleneck walker splitting.
- **pca** (*bool, default: False*) – Can be True or False (default) to run PCA on pcoords before bin assignment.
- **mab_log** (*bool, default: False*) – Whether to output mab info to west.log.
- **bin_log** (*bool, default: False*) – Whether to output mab bin boundaries to bin_log_path file.

- **bin_log_path** (*str, default: "$WEST_SIM_ROOT/binbounds.log"*) – Path to output bin boundaries.

determine_total_bins(*nbins_per_dim*, *direction*, *skip*, *bottleneck*, ***kwargs*)

The following is neccessary because functional bin mappers need to "reserve" bins and tell the sim manager how many bins they will need to use, this is determined by taking all direction/skipping info into account.

Parameters

- **nbins_per_dim** (*int*) – Number of total bins in each direction.
- **direction** (*list of int*) – Direction in each dimension. See __init__ for more information.
- **skip** (*list of int*) – List of 0s and 1s indicating whether to skip each dimension.
- **bottleneck** (*bool*) – Whether to include separate bin for bottleneck walker(s).
- ****kwargs** (*dict*) – Arbitary keyword arguments. Contains unneeded MAB parameters.

Returns

n_total_bins – Number of total bins.

Return type

int

class westpa.core.binning.mab_manager.**WESimManager**(*rc=None*)

Bases: object

process_config()

register_callback(*hook*, *function*, *priority=0*)

Registers a callback to execute during the given hook into the simulation loop. The optional priority is used to order when the function is called relative to other registered callbacks.

invoke_callbacks(*hook*, **args*, ***kwargs*)

load_plugins(*plugins=None*)

report_bin_statistics(*bins*, *target_states*, *save_summary=False*)

get_bstate_pcoords(*basis_states*, *label='basis'*)

For each of the given basis_states, calculate progress coordinate values as necessary. The HDF5 file is not updated.

report_basis_states(*basis_states*, *label='basis'*)

report_target_states(*target_states*)

initialize_simulation(*basis_states*, *target_states*, *start_states*, *segs_per_state=1*, *suppress_we=False*)

Initialize a new weighted ensemble simulation, taking segs_per_state initial states from each of the given basis_states.

w_init is the forward-facing version of this function

prepare_iteration()

finalize_iteration()

Clean up after an iteration and prepare for the next.

get_istate_futures()

Add n_states initial states to the internal list of initial states assigned to recycled particles. Spare states are used if available, otherwise new states are created. If created new initial states requires generation, then a set of futures is returned representing work manager tasks corresponding to the necessary generation work.

propagate()

save_bin_data()

Calculate and write flux and transition count matrices to HDF5. Population and rate matrices are likely useless at the single-tau level and are no longer written.

check_propagation()

Check for failures in propagation or initial state generation, and raise an exception if any are found.

run_we()

Run the weighted ensemble algorithm based on the binning in self.final_bins and the recycled particles in self.to_recycle, creating and committing the next iteration's segments to storage as well.

prepare_new_iteration()

Commit data for the coming iteration to the HDF5 file.

run()

prepare_run()

Prepare a new run.

finalize_run()

Perform cleanup at the normal end of a run

pre_propagation()

post_propagation()

pre_we()

post_we()

westpa.core.binning.mab_manager.**grouper**(*n*, *iterable*, *fillvalue=None*)

Collect data into fixed-length chunks or blocks

class westpa.core.binning.mab_manager.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).

- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).
- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

westpa.core.binning.mab_manager.**pare_basis_initial_states**(*basis_states*, *initial_states*, *segments=None*)

Given iterables of basis and initial states (and optionally segments that use them), return minimal sets (as in __builtins__.set) of states needed to describe the history of the given segments an initial states.

class westpa.core.binning.mab_manager.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.core.binning.mab_manager.MABSimManager(*rc=None*)

Bases: *WESimManager*

Subclass of WESimManager, modifying it so bin assignments will be done after all segments are done propagating.

initialize_simulation(*basis_states*, *target_states*, *start_states*, *segs_per_state=1*, *suppress_we=False*)

Making sure that that the MABBinMapper is not the outer bin.

propagate()

prepare_iteration()

6.2.3 westpa.core.kinetics package

6.2.3.1 westpa.core.kinetics module

Kinetics analysis library

class westpa.core.kinetics.**RateAverager**(*bin_mapper*, *system=None*, *data_manager=None*, *work_manager=None*)

Bases: object

Calculate bin-to-bin kinetic properties (fluxes, rates, populations) at 1-tau resolution

extract_data(*iter_indices*)

Extract data from the data_manger and place in dict mirroring the same underlying layout.

task_generator(*iter_start*, *iter_stop*, *block_size*)

calculate(*iter_start=None*, *iter_stop=None*, *n_blocks=1*, *queue_size=1*)

Read the HDF5 file and collect flux matrices and population vectors for each bin for each iteration in the range [iter_start, iter_stop). Break the calculation into n_blocks blocks. If the calculation is broken up into more than one block, queue_size specifies the maxmimum number of tasks in the work queue.

westpa.core.kinetics.**calculate_labeled_fluxes**(*nstates*, *weights*, *parent_ids*, *micro_assignments*, *traj_assignments*, *fluxes*)

westpa.core.kinetics.**labeled_flux_to_rate**(*labeled_fluxes*, *labeled_pops*, *output=None*)

Convert a labeled flux matrix and corresponding labeled bin populations to a labeled rate matrix.

westpa.core.kinetics.**calculate_labeled_fluxes_alllags**(*nstates*, *weights*, *parent_ids*, *micro_assignments*, *traj_assignments*, *fluxes*)

westpa.core.kinetics.**nested_to_flat_matrix**(*input*)

Convert nested flux/rate matrix into a flat supermatrix.

westpa.core.kinetics.**nested_to_flat_vector**(*input*)

Convert nested labeled population vector into a flat vector.

westpa.core.kinetics.**flat_to_nested_matrix**(*nstates*, *nbins*, *input*)

Convert flat supermatrix into nested matrix.

westpa.core.kinetics.**flat_to_nested_vector**(*nstates*, *nbins*, *input*)

Convert flat "supervector" into nested vector.

westpa.core.kinetics.**find_macrostate_transitions**(*nstates*, *weights*, *label_assignments*, *state_assignments*, *dt*, *state*, *macro_fluxes*, *macro_counts*, *target_fluxes*, *target_counts*, *durations*)

westpa.core.kinetics.**sequence_macro_flux_to_rate**(*dataset*, *pops*, *istate*, *jstate*, *pairwise=True*, *stride=None*)

Convert a sequence of macrostate fluxes and corresponding list of trajectory ensemble populations to a sequence of rate matrices.

If the optional `pairwise` is true (the default), then rates are normalized according to the relative probability of the initial state among the pair of states (initial, final); this is probably what you want, as these rates will then depend only on the definitions of the states involved (and never the remaining states). Otherwise (``pairwise" is false), the rates are normalized according the probability of the initial state among *all* other states.

class westpa.core.kinetics.**WKinetics**

Bases: `object`

w_kinetics()

6.2.3.2 westpa.core.kinetics.events module

westpa.core.kinetics.events.**weight_dtype**

alias of `float64`

westpa.core.kinetics.events.**index_dtype**

alias of `uint16`

westpa.core.kinetics.events.**find_macrostate_transitions**(*nstates*, *weights*, *label_assignments*, *state_assignments*, *dt*, *state*, *macro_fluxes*, *macro_counts*, *target_fluxes*, *target_counts*, *durations*)

class westpa.core.kinetics.events.**WKinetics**

Bases: `object`

w_kinetics()

6.2.3.3 westpa.core.kinetics.matrates module

Routines for implementing Letteri et al.'s macrostate-to-macrostate rate calculations using extrapolation to steady-state populations from average rate matrices

Internally, "labeled" objects (bin populations labeled by history, rate matrix elements labeled by history) are stored as nested arrays – e.g. rates[initial_label, final_label, initial_bin, final_bin]. These are converted to the flat forms required for, say, eigenvalue calculations internally, and the results converted back. This is because these conversions are not expensive, and saves users of this code from having to know how the flattened indexing works (something I screwed up all too easily during development) – mcz

westpa.core.kinetics.matrates.**weight_dtype**

alias of `float64`

westpa.core.kinetics.matrates.**calculate_labeled_fluxes**(*nstates*, *weights*, *parent_ids*, *micro_assignments*, *traj_assignments*, *fluxes*)

westpa.core.kinetics.matrates.**calculate_labeled_fluxes_alllags**(*nstates*, *weights*, *parent_ids*, *micro_assignments*, *traj_assignments*, *fluxes*)

westpa.core.kinetics.matrates.**labeled_flux_to_rate**(*labeled_fluxes*, *labeled_pops*, *output=None*)

Convert a labeled flux matrix and corresponding labeled bin populations to a labeled rate matrix.

westpa.core.kinetics.matrates.**nested_to_flat_matrix**(*input*)

Convert nested flux/rate matrix into a flat supermatrix.

westpa.core.kinetics.matrates.**nested_to_flat_vector**(*input*)

Convert nested labeled population vector into a flat vector.

westpa.core.kinetics.matrates.**flat_to_nested_vector**(*nstates*, *nbins*, *input*)

Convert flat "supervector" into nested vector.

exception westpa.core.kinetics.matrates.**ConsistencyWarning**

Bases: `UserWarning`

westpa.core.kinetics.matrates.**get_steady_state**(*rates*)

Get steady state solution for a rate matrix. As an optimization, returns the flattened labeled population vector (of length nstates*nbins); to convert to the nested vector used for storage, use nested_to_flat_vector().

westpa.core.kinetics.matrates.**get_macrostate_rates**(*labeled_rates*, *labeled_pops*, *extrapolate=True*)

Using a labeled rate matrix and labeled bin populations, calculate the steady state probability distribution and consequent state-to-state rates.

Returns `(ss, macro_rates)`, where `ss` is the steady-state probability distribution and `macro_rates` is the state-to-state rate matrix.

westpa.core.kinetics.matrates.**estimate_rates**(*nbins*, *state_labels*, *weights*, *parent_ids*, *bin_assignments*, *label_assignments*, *state_map*, *labeled_pops*, *all_lags=False*, *labeled_fluxes=None*, *labeled_rates=None*, *unlabeled_rates=None*)

Estimate fluxes and rates over multiple iterations. The number of iterations is determined by how many vectors of weights, parent IDs, bin assignments, and label assignments are passed.

If `all_lags` is true, then the average is over all possible lags within the length-N window given, otherwise simply the length N lag.

Returns labeled flux matrix, labeled rate matrix, and unlabeled rate matrix.

6.2.3.4 westpa.core.kinetics.rate_averaging module

westpa.core.kinetics.rate_averaging.**namedtuple**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', ['x', 'y'])
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                       # fields also accessible by name
33
```

(continues on next page)

(continued from previous page)

```
>>> d = p._asdict()                # convert to a dictionary
>>> d['x']
11
>>> Point(**d)                     # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)              # _replace() is like str.replace() but targets␣
↪named fields
Point(x=100, y=22)
```

class westpa.core.kinetics.rate_averaging.**zip_longest**

Bases: object

zip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> zip_longest object

Return a zip_longest object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration. When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

westpa.core.kinetics.rate_averaging.**flux_assign**(*weights*, *init_assignments*, *final_assignments*, *flux_matrix*)

westpa.core.kinetics.rate_averaging.**pop_assign**(*weights*, *assignments*, *populations*)

westpa.core.kinetics.rate_averaging.**calc_rates**(*fluxes*, *populations*, *rates*, *mask*)

Calculate a rate matrices from flux and population matrices. A matrix of the same shape as fluxes, is also produced, to be used for generating a mask for the rate matrices where initial state populations are zero.

class westpa.core.kinetics.rate_averaging.**StreamingStats1D**

Bases: object

Calculate mean and variance of a series of one-dimensional arrays of shape (nbins,) using an online algorithm. The statistics are accumulated along what would be axis=0 if the input arrays were stacked vertically.

This code has been adapted from: http://www.johndcook.com/skewness_kurtosis.html

M1

M2

mean

n

update(*x*, *mask*)

Update the running set of statistics given

Parameters

- **x** (*1d ndarray*) – values from a single observation
- **mask** (*1d ndarray*) – A uint8 array to exclude entries from the accumulated statistics.

var

class westpa.core.kinetics.rate_averaging.**StreamingStats2D**

Bases: object

Calculate mean and variance of a series of two-dimensional arrays of shape (nbins, nbins) using an online algorithm. The statistics are accumulated along what would be axis=0 if the input arrays were stacked vertically.

This code has been adapted from: http://www.johndcook.com/skewness_kurtosis.html

M1

M2

mean

n

update(*x*, *mask*)

Update the running set of statistics given

Parameters

- **x** (*2d ndarray*) – values from a single observation
- **mask** (*2d ndarray*) – A uint8 array to exclude entries from the accumulated statistics.

var

class westpa.core.kinetics.rate_averaging.**StreamingStatsTuple**(*M1*, *M2*, *n*)

Bases: tuple

Create new instance of StreamingStatsTuple(M1, M2, n)

M1

Alias for field number 0

M2

Alias for field number 1

n

Alias for field number 2

westpa.core.kinetics.rate_averaging.**grouper**(*n*, *iterable*, *fillvalue=None*)

Collect data into fixed-length chunks or blocks

westpa.core.kinetics.rate_averaging.**tuple2stats**(*stat_tuple*)

westpa.core.kinetics.rate_averaging.**process_iter_chunk**(*bin_mapper*, *iter_indices*, *iter_data=None*)

Calculate the flux matrices and populations of a set of iterations specified by iter_indices. Optionally provide the necessary arrays to perform the calculation in iter_data. Otherwise get data from the data_manager directly.

class westpa.core.kinetics.rate_averaging.**RateAverager**(*bin_mapper*, *system=None*, *data_manager=None*, *work_manager=None*)

Bases: object

Calculate bin-to-bin kinetic properties (fluxes, rates, populations) at 1-tau resolution

extract_data(*iter_indices*)

Extract data from the data_manger and place in dict mirroring the same underlying layout.

task_generator(*iter_start*, *iter_stop*, *block_size*)

calculate(*iter_start=None*, *iter_stop=None*, *n_blocks=1*, *queue_size=1*)

Read the HDF5 file and collect flux matrices and population vectors for each bin for each iteration in the range [iter_start, iter_stop). Break the calculation into n_blocks blocks. If the calculation is broken up into more than one block, queue_size specifies the maxmimum number of tasks in the work queue.

6.2.4 westpa.core.propagators package

6.2.4.1 westpa.core.propagators module

westpa.core.propagators.**blocked_iter**(*blocksize*, *iterable*, *fillvalue=None*)

class westpa.core.propagators.**WESTPropagator**(*rc=None*)

Bases: object

prepare_iteration(*n_iter*, *segments*)

Perform any necessary per-iteration preparation. This is run by the work manager.

finalize_iteration(*n_iter*, *segments*)

Perform any necessary post-iteration cleanup. This is run by the work manager.

get_pcoord(*state*)

Get the progress coordinate of the given basis or initial state.

gen_istate(*basis_state*, *initial_state*)

Generate a new initial state from the given basis state.

propagate(*segments*)

Propagate one or more segments, including any necessary per-iteration setup and teardown for this propagator.

clear_basis_initial_states()

update_basis_initial_states(*basis_states*, *initial_states*)

6.2.4.2 westpa.core.propagators.executable module

class westpa.core.propagators.executable.**BytesIO**(*initial_bytes=b''*)

Bases: _BufferedIOBase

Buffered I/O implementation using an in-memory bytes buffer.

close()

Disable all I/O operations.

closed

True if the file is closed.

flush()

Does nothing.

getbuffer()

Get a read-write view over the contents of the BytesIO object.

getvalue()

Retrieve the entire contents of the BytesIO object.

isatty()

Always returns False.

BytesIO objects are not connected to a TTY-like device.

read(*size=-1*, /)

Read at most size bytes, returned as a bytes object.

If the size argument is negative, read until EOF is reached. Return an empty bytes object at EOF.

read1(*size=-1*, /)

Read at most size bytes, returned as a bytes object.

If the size argument is negative or omitted, read until EOF is reached. Return an empty bytes object at EOF.

readable()

Returns True if the IO object can be read.

readinto(*buffer*, /)

Read bytes into buffer.

Returns number of bytes read (0 for EOF), or None if the object is set not to block and has no data to read.

readline(*size=-1*, /)

Next line from the file, as a bytes object.

Retain newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty bytes object at EOF.

readlines(*size=None*, /)

List of bytes objects, each a line from the file.

Call readline() repeatedly and return a list of the lines so read. The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.

seek(*pos*, *whence=0*, /)

Change stream position.
Seek to byte offset pos relative to position indicated by whence:
0 Start of stream (the default). pos should be >= 0; 1 Current position - pos may be negative; 2 End of stream - pos usually negative.
Returns the new absolute position.

seekable()

Returns True if the IO object can be seeked.

tell()

Current file position, an integer.

truncate(*size=None*, /)

Truncate the file to at most size bytes.

Size defaults to the current file position, as returned by tell(). The current file position is unchanged. Returns the new size.

writable()

Returns True if the IO object can be written.

write(*b*, /)

Write bytes to file.

Return the number of bytes written.

writelines(*lines*, /)

Write lines to the file.

Note that newlines are not added. lines can be any iterable object producing bytes-like objects. This is equivalent to calling write() for each element.

westpa.core.propagators.executable.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

class westpa.core.propagators.executable.**WESTPropagator**(*rc=None*)

Bases: object

prepare_iteration(*n_iter*, *segments*)

Perform any necessary per-iteration preparation. This is run by the work manager.

finalize_iteration(*n_iter*, *segments*)

Perform any necessary post-iteration cleanup. This is run by the work manager.

get_pcoord(*state*)

Get the progress coordinate of the given basis or initial state.

gen_istate(*basis_state*, *initial_state*)

Generate a new initial state from the given basis state.

propagate(*segments*)

Propagate one or more segments, including any necessary per-iteration setup and teardown for this propagator.

clear_basis_initial_states()

update_basis_initial_states(*basis_states*, *initial_states*)

class westpa.core.propagators.executable.**BasisState**(*label*, *probability*, *pcoord=None*, *auxref=None*, *state_id=None*)

Bases: object

Describes an basis (micro)state. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation (i.e. at w_init) or due to recycling.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **probability** – Probability of this state to be selected when creating a new trajectory.
- **pcoord** – The representative progress coordinate of this state.
- **auxref** – A user-provided (string) reference for locating data associated with this state (usually a filesystem path).

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*)

Read a file defining basis states. Each line 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
```

as_numpy_record()

Return the data for this state as a numpy record array.

class westpa.core.propagators.executable.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).
- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

westpa.core.propagators.executable.**return_state_type**(*state_obj*)

Convinience function for returning the state ID and type of the state_obj pointer

class westpa.core.propagators.executable.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

westpa.core.propagators.executable.**check_bool**(*value*, *action='warn'*)

Check that the given `value` is boolean in type. If not, either raise a warning (if `action=='warn'`) or an exception (`action=='raise'`).

westpa.core.propagators.executable.**load_trajectory**(*folder*)

Load trajectory from `folder` using `mdtraj` and return a `mdtraj.Trajectory` object. The folder should contain a trajectory and a topology file (with a recognizable extension) that is supported by `mdtraj`. The topology file is optional if the trajectory file contains topology data (e.g., HDF5 format).

westpa.core.propagators.executable.**safe_extract**(*tar*, *path='.'*, *members=None*, ***, *numeric_owner=False*)

westpa.core.propagators.executable.**pcoord_loader**(*fieldname*, *pcoord_return_filename*, *destobj*, *single_point*)

Read progress coordinate data into the `pcoord` field on `destobj`. An exception will be raised if the data is malformed. If `single_point` is true, then only one (N-dimensional) point will be read, otherwise system.pcoord_len points will be read.

westpa.core.propagators.executable.**aux_data_loader**(*fieldname*, *data_filename*, *segment*, *single_point*)

westpa.core.propagators.executable.**npy_data_loader**(*fieldname*, *coord_file*, *segment*, *single_point*)

westpa.core.propagators.executable.**pickle_data_loader**(*fieldname*, *coord_file*, *segment*, *single_point*)

westpa.core.propagators.executable.**trajectory_loader**(*fieldname*, *coord_folder*, *segment*, *single_point*)

Load data from the trajectory return. `coord_folder` should be the path to a folder containing trajectory files. `segment` is the `Segment` object that the data is associated with. Please see `load_trajectory` for more details. `single_point` is not used by this loader.

westpa.core.propagators.executable.**restart_loader**(*fieldname*, *restart_folder*, *segment*, *single_point*)

Load data from the restart return. The loader will tar all files in `restart_folder` and store it in the per-iteration HDF5 file. `segment` is the `Segment` object that the data is associated with. `single_point` is not used by this loader.

westpa.core.propagators.executable.**restart_writer**(*path*, *segment*)

Prepare the necessary files from the per-iteration HDF5 file to run `segment`.

westpa.core.propagators.executable.**seglog_loader**(*fieldname*, *log_file*, *segment*, *single_point*)

Load data from the log return. The loader will tar all files in `log_file` and store it in the per-iteration HDF5 file. `segment` is the `Segment` object that the data is associated with. `single_point` is not used by this loader.

class westpa.core.propagators.executable.**ExecutablePropagator**(*rc=None*)

Bases: *WESTPropagator*

ENV_CURRENT_ITER = 'WEST_CURRENT_ITER'

ENV_CURRENT_SEG_ID = 'WEST_CURRENT_SEG_ID'

ENV_CURRENT_SEG_DATA_REF = 'WEST_CURRENT_SEG_DATA_REF'

ENV_CURRENT_SEG_INITPOINT = 'WEST_CURRENT_SEG_INITPOINT_TYPE'

ENV_PARENT_SEG_ID = 'WEST_PARENT_ID'

ENV_PARENT_DATA_REF = 'WEST_PARENT_DATA_REF'

ENV_BSTATE_ID = 'WEST_BSTATE_ID'

ENV_BSTATE_DATA_REF = 'WEST_BSTATE_DATA_REF'

ENV_ISTATE_ID = 'WEST_ISTATE_ID'

ENV_ISTATE_DATA_REF = 'WEST_ISTATE_DATA_REF'

ENV_STRUCT_DATA_REF = 'WEST_STRUCT_DATA_REF'

ENV_RAND16 = 'WEST_RAND16'

ENV_RAND32 = 'WEST_RAND32'

ENV_RAND64 = 'WEST_RAND64'

ENV_RAND128 = 'WEST_RAND128'

ENV_RANDFLOAT = 'WEST_RANDFLOAT'

static makepath(*template*, *template_args=None*, *expanduser=True*, *expandvars=True*, *abspath=False*, *realpath=False*)

random_val_env_vars()

Return a set of environment variables containing random seeds. These are returned as a dictionary, suitable for use in `os.environ.update()` or as the `env` argument to `subprocess.Popen()`. Every child process executed by `exec_child()` gets these.

exec_child(*executable*, *environ=None*, *stdin=None*, *stdout=None*, *stderr=None*, *cwd=None*)

Execute a child process with the environment set from the current environment, the values of self.addtl_child_environ, the random numbers returned by self.random_val_env_vars, and the given `environ` (applied in that order). stdin/stdout/stderr are optionally redirected.

This function waits on the child process to finish, then returns (rc, rusage), where rc is the child's return code and rusage is the resource usage tuple from os.wait4()

exec_child_from_child_info(*child_info*, *template_args*, *environ*)

update_args_env_basis_state(*template_args*, *environ*, *basis_state*)

update_args_env_initial_state(*template_args*, *environ*, *initial_state*)

update_args_env_iter(*template_args*, *environ*, *n_iter*)

update_args_env_segment(*template_args*, *environ*, *segment*)

template_args_for_segment(*segment*)

exec_for_segment(*child_info*, *segment*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given segment.

exec_for_iteration(*child_info*, *n_iter*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given iteration number.

exec_for_basis_state(*child_info*, *basis_state*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given basis state

exec_for_initial_state(*child_info*, *initial_state*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given initial state.

prepare_file_system(*segment*, *environ*)

setup_dataset_return(*segment=None*, *subset_keys=None*)

Set up temporary files and environment variables that point to them for segment runners to return data. `segment` is the `Segment` object that the return data is associated with. `subset_keys` specifies the names of a subset of data to be returned.

retrieve_dataset_return(*state*, *return_files*, *del_return_files*, *single_point*)

Retrieve returned data from the temporary locations directed by the environment variables. `state` is a `Segment`, `BasisState` , or `InitialState``object that the return data is associated with. ``return_files` is a `dict` where the keys are the dataset names and the values are the paths to the temporarily files that contain the returned data. `del_return_files` is a `dict` where the keys are the names of datasets to be deleted (if the corresponding value is set to `True`) once the data is retrieved.

get_pcoord(*state*)

Get the progress coordinate of the given basis or initial state.

gen_istate(*basis_state*, *initial_state*)

Generate a new initial state from the given basis state.

prepare_iteration(*n_iter*, *segments*)

Perform any necessary per-iteration preparation. This is run by the work manager.

finalize_iteration(*n_iter*, *segments*)

Perform any necessary post-iteration cleanup. This is run by the work manager.

propagate(*segments*)

Propagate one or more segments, including any necessary per-iteration setup and teardown for this propagator.

6.2.5 westpa.core.reweight package

6.2.5.1 westpa.core.reweight module

Function(s) for the postanalysis toolkit

westpa.core.reweight.**stats_process**(*bin_assignments*, *weights*, *fluxes*, *populations*, *trans*, *mask*, *interval='timepoint'*)

westpa.core.reweight.**reweight_for_c**(*rows*, *cols*, *obs*, *flux*, *insert*, *indices*, *nstates*, *nbins*, *state_labels*, *state_map*, *nfbins*, *istate*, *jstate*, *stride*, *bin_last_state_map*, *bin_state_map*, *return_obs*, *obs_threshold=1*)

class westpa.core.reweight.**FluxMatrix**

Bases: object

w_postanalysis_matrix()

6.2.5.2 westpa.core.reweight.matrix module

westpa.core.reweight.matrix.**weight_dtype**

alias of float64

westpa.core.reweight.matrix.**index_dtype**

alias of uint16

westpa.core.reweight.matrix.**stats_process**(*bin_assignments*, *weights*, *fluxes*, *populations*, *trans*, *mask*, *interval='timepoint'*)

westpa.core.reweight.matrix.**calc_stats**(*bin_assignments*, *weights*, *fluxes*, *populations*, *trans*, *mask*, *sampling_frequency*)

class westpa.core.reweight.matrix.**FluxMatrix**

Bases: object

w_postanalysis_matrix()

6.2.6 westpa.core modules

6.2.6.1 westpa.core module

6.2.6.2 westpa.core.data_manager module

HDF5 data manager for WEST.

Original HDF5 implementation: Joseph W. Kaus Current implementation: Matthew C. Zwier

WEST exclusively uses the cross-platform, self-describing file format HDF5 for data storage. This ensures that data is stored efficiently and portably in a manner that is relatively straightforward for other analysis tools (perhaps written in C/C++/Fortran) to access.

The data is laid out in HDF5 as follows:

- summary – overall summary data for the simulation
- **/iterations/ – data for individual iterations, one group per iteration under /iterations**
  - **iter_00000001/ – data for iteration 1**
    - seg_index – overall information about segments in the iteration, including weight
    - pcoord – progress coordinate data organized as [seg_id][time][dimension]
    - wtg_parents – data used to reconstruct the split/merge history of trajectories
    - recycling – flux and event count for recycled particles, on a per-target-state basis
    - auxdata/ – auxiliary datasets (data stored on the 'data' field of Segment objects)

The file root object has an integer attribute 'west_file_format_version' which can be used to determine how to access data even as the file format (i.e. organization of data within HDF5 file) evolves.

Version history:

Version 9

- Basis states are now saved as iter_segid instead of just segid as a pointer label.
- Initial states are also saved in the iteration 0 file, with a negative sign.

Version 8

- Added external links to trajectory files in iterations/iter_* groups, if the HDF5 framework was used.
- Added an iter group for the iteration 0 to store conformations of basis states.

Version 7

- Removed bin_assignments, bin_populations, and bin_rates from iteration group.
- Added new_segments subgroup to iteration group

Version 6

- ???

Version 5

- moved iter_* groups into a top-level iterations/ group,
- added in-HDF5 storage for basis states, target states, and generated states

`class` `westpa.core.data_manager.`**`attrgetter`**(*attr*, */*, **attrs*)

Bases: `object`

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

`westpa.core.data_manager.`**`relpath`**(*path*, *start=None*)

Return a relative version of a path

`westpa.core.data_manager.`**`dirname`**(*p*)

Returns the directory component of a pathname

class westpa.core.data_manager.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.core.data_manager.**BasisState**(*label*, *probability*, *pcoord=None*, *auxref=None*, *state_id=None*)

Bases: object

Describes an basis (micro)state. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation (i.e. at w_init) or due to recycling.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **probability** – Probability of this state to be selected when creating a new trajectory.
- **pcoord** – The representative progress coordinate of this state.
- **auxref** – A user-provided (string) reference for locating data associated with this state (usually a filesystem path).

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*)

Read a file defining basis states. Each line 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
```

as_numpy_record()

Return the data for this state as a numpy record array.

class westpa.core.data_manager.**TargetState**(*label*, *pcoord*, *state_id=None*)

Bases: object

Describes a target state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **pcoord** – The representative progress coordinate of this state.

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*, *dtype*)

Read a file defining target states. 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.

class westpa.core.data_manager.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: `object`

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).
- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

class westpa.core.data_manager.**NewWeightEntry**(*source_type*, *weight*, *prev_seg_id=None*, *prev_init_pcoord=None*, *prev_final_pcoord=None*, *new_init_pcoord=None*, *target_state_id=None*, *initial_state_id=None*)

Bases: object

NW_SOURCE_RECYCLED = 0

class westpa.core.data_manager.**ExecutablePropagator**(*rc=None*)

Bases: *WESTPropagator*

ENV_CURRENT_ITER = 'WEST_CURRENT_ITER'

ENV_CURRENT_SEG_ID = 'WEST_CURRENT_SEG_ID'

ENV_CURRENT_SEG_DATA_REF = 'WEST_CURRENT_SEG_DATA_REF'

ENV_CURRENT_SEG_INITPOINT = 'WEST_CURRENT_SEG_INITPOINT_TYPE'

ENV_PARENT_SEG_ID = 'WEST_PARENT_ID'

ENV_PARENT_DATA_REF = 'WEST_PARENT_DATA_REF'

ENV_BSTATE_ID = 'WEST_BSTATE_ID'

ENV_BSTATE_DATA_REF = 'WEST_BSTATE_DATA_REF'

ENV_ISTATE_ID = 'WEST_ISTATE_ID'

ENV_ISTATE_DATA_REF = 'WEST_ISTATE_DATA_REF'

ENV_STRUCT_DATA_REF = 'WEST_STRUCT_DATA_REF'

ENV_RAND16 = 'WEST_RAND16'

ENV_RAND32 = 'WEST_RAND32'

ENV_RAND64 = 'WEST_RAND64'

ENV_RAND128 = 'WEST_RAND128'

ENV_RANDFLOAT = 'WEST_RANDFLOAT'

static makepath(*template*, *template_args=None*, *expanduser=True*, *expandvars=True*, *abspath=False*, *realpath=False*)

random_val_env_vars()

Return a set of environment variables containing random seeds. These are returned as a dictionary, suitable for use in `os.environ.update()` or as the `env` argument to `subprocess.Popen()`. Every child process executed by `exec_child()` gets these.

exec_child(*executable*, *environ=None*, *stdin=None*, *stdout=None*, *stderr=None*, *cwd=None*)

Execute a child process with the environment set from the current environment, the values of self.addtl_child_environ, the random numbers returned by self.random_val_env_vars, and the given `environ` (applied in that order). stdin/stdout/stderr are optionally redirected.

This function waits on the child process to finish, then returns (rc, rusage), where rc is the child's return code and rusage is the resource usage tuple from os.wait4()

exec_child_from_child_info(*child_info*, *template_args*, *environ*)

update_args_env_basis_state(*template_args*, *environ*, *basis_state*)

update_args_env_initial_state(*template_args*, *environ*, *initial_state*)

update_args_env_iter(*template_args*, *environ*, *n_iter*)

update_args_env_segment(*template_args*, *environ*, *segment*)

template_args_for_segment(*segment*)

exec_for_segment(*child_info*, *segment*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given segment.

exec_for_iteration(*child_info*, *n_iter*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given iteration number.

exec_for_basis_state(*child_info*, *basis_state*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given basis state

exec_for_initial_state(*child_info*, *initial_state*, *addtl_env=None*)

Execute a child process with environment and template expansion from the given initial state.

prepare_file_system(*segment*, *environ*)

setup_dataset_return(*segment=None*, *subset_keys=None*)

Set up temporary files and environment variables that point to them for segment runners to return data. `segment` is the `Segment` object that the return data is associated with. `subset_keys` specifies the names of a subset of data to be returned.

retrieve_dataset_return(*state*, *return_files*, *del_return_files*, *single_point*)

Retrieve returned data from the temporary locations directed by the environment variables. `state` is a `Segment`, `BasisState` , or `InitialState``object that the return data is associated with. ``return_files` is a `dict` where the keys are the dataset names and the values are the paths to the temporarily files that contain the returned data. `del_return_files` is a `dict` where the keys are the names of datasets to be deleted (if the corresponding value is set to `True`) once the data is retrieved.

get_pcoord(*state*)

Get the progress coordinate of the given basis or initial state.

gen_istate(*basis_state*, *initial_state*)

Generate a new initial state from the given basis state.

prepare_iteration(*n_iter*, *segments*)

Perform any necessary per-iteration preparation. This is run by the work manager.

finalize_iteration(*n_iter*, *segments*)

Perform any necessary post-iteration cleanup. This is run by the work manager.

propagate(*segments*)

Propagate one or more segments, including any necessary per-iteration setup and teardown for this propagator.

westpa.core.data_manager.**makepath**(*template*, *template_args=None*, *expanduser=True*, *expandvars=True*, *abspath=False*, *realpath=False*)

class westpa.core.data_manager.**flushing_lock**(*lock*, *fileobj*)

Bases: object

class westpa.core.data_manager.**expiring_flushing_lock**(*lock*, *flush_method*, *nextsync*)

Bases: object

westpa.core.data_manager.**seg_id_dtype**

alias of int64

westpa.core.data_manager.**n_iter_dtype**

alias of uint32

westpa.core.data_manager.**weight_dtype**

alias of float64

westpa.core.data_manager.**utime_dtype**

alias of float64

westpa.core.data_manager.**seg_status_dtype**

alias of uint8

westpa.core.data_manager.**seg_initpoint_dtype**

alias of uint8

westpa.core.data_manager.**seg_endpoint_dtype**

alias of uint8

westpa.core.data_manager.**istate_type_dtype**

alias of uint8

westpa.core.data_manager.**istate_status_dtype**

alias of uint8

westpa.core.data_manager.**nw_source_dtype**

alias of uint8

class westpa.core.data_manager.**WESTDataManager**(*rc=None*)

Bases: object

Data manager for assisiting the reading and writing of WEST data from/to HDF5 files.

default_iter_prec = 8

default_we_h5filename = 'west.h5'

default_we_h5file_driver = None

default_flush_period = 60

default_aux_compression_threshold = 1048576

binning_hchunksize = 4096

table_scan_chunksize = 1024

flushing_lock()

expiring_flushing_lock()

process_config()

property system

property closed

iter_group_name(*n_iter*, *absolute=True*)

require_iter_group(*n_iter*)

Get the group associated with n_iter, creating it if necessary.

del_iter_group(*n_iter*)

get_iter_group(*n_iter*)

get_seg_index(*n_iter*)

property current_iteration

open_backing(*mode=None*)

Open the (already-created) HDF5 file named in self.west_h5filename.

prepare_backing()

Create new HDF5 file

close_backing()

flush_backing()

save_target_states(*tstates*, *n_iter=None*)

Save the given target states in the HDF5 file; they will be used for the next iteration to be propagated. A complete set is required, even if nominally appending to an existing set, which simplifies the mapping of IDs to the table.

find_tstate_group(*n_iter*)

find_ibstate_group(*n_iter*)

get_target_states(*n_iter*)

Return a list of Target objects representing the target (sink) states that are in use for iteration n_iter. Future iterations are assumed to continue from the most recent set of states.

create_ibstate_group(*basis_states*, *n_iter=None*)

Create the group used to store basis states and initial states (whose definitions are always coupled). This group is hard-linked into all iteration groups that use these basis and initial states.

create_ibstate_iter_h5file(*basis_states*)

Create the per-iteration HDF5 file for the basis states (i.e., iteration 0). This special treatment is needed so that the analysis tools can access basis states more easily.

update_iter_h5file(*n_iter*, *segments*)

Write out the per-iteration HDF5 file with given segments and add an external link to it in the main HDF5 file (west.h5) if the link is not present.

get_basis_states(*n_iter=None*)

Return a list of BasisState objects representing the basis states that are in use for iteration n_iter.

create_initial_states(*n_states*, *n_iter=None*)

Create storage for `n_states` initial states associated with iteration `n_iter`, and return bare InitialState objects with only state_id set.

update_initial_states(*initial_states*, *n_iter=None*)

Save the given initial states in the HDF5 file

get_initial_states(*n_iter=None*)

get_segment_initial_states(*segments*, *n_iter=None*)

Retrieve all initial states referenced by the given segments.

get_unused_initial_states(*n_states=None*, *n_iter=None*)

Retrieve any prepared but unused initial states applicable to the given iteration. Up to `n_states` states are returned; if `n_states` is None, then all unused states are returned.

prepare_iteration(*n_iter*, *segments*)

Prepare for a new iteration by creating space to store the new iteration's data. The number of segments, their IDs, and their lineage must be determined and included in the set of segments passed in.

update_iter_group_links(*n_iter*)

Update the per-iteration hard links pointing to the tables of target and initial/basis states for the given iteration. These links are not used by this class, but are remarkably convenient for third-party analysis tools and hdfview.

get_iter_summary(*n_iter=None*)

update_iter_summary(*summary*, *n_iter=None*)

del_iter_summary(*min_iter*)

update_segments(*n_iter*, *segments*)

Update segment information in the HDF5 file; all prior information for each `segment` is overwritten, except for parent and weight transfer information.

get_segments(*n_iter=None*, *seg_ids=None*, *load_pcoords=True*)

Return the given (or all) segments from a given iteration.

If the optional parameter `load_auxdata` is true, then all auxiliary datasets available are loaded and mapped onto the `data` dictionary of each segment. If `load_auxdata` is None, then use the default `self.auto_load_auxdata`, which can be set by the option `load_auxdata` in the `[data]` section of `west.cfg`. This essentially requires as much RAM as there is per-iteration auxiliary data, so this behavior is not on by default.

prepare_segment_restarts(*segments*, *basis_states=None*, *initial_states=None*)

Prepare the necessary folder and files given the data stored in parent per-iteration HDF5 file for propagating the simulation. `basis_states` and `initial_states` should be provided if the segments are newly created

get_all_parent_ids(*n_iter*)

get_parent_ids(*n_iter*, *seg_ids=None*)

Return a sequence of the parent IDs of the given seg_ids.

get_weights(*n_iter*, *seg_ids*)

Return the weights associated with the given seg_ids

get_child_ids(*n_iter*, *seg_id*)

Return the seg_ids of segments who have the given segment as a parent.

get_children(*segment*)

Return all segments which have the given segment as a parent

prepare_run()

finalize_run()

save_new_weight_data(*n_iter*, *new_weights*)

Save a set of NewWeightEntry objects to HDF5. Note that this should be called for the iteration in which the weights appear in their new locations (e.g. for recycled walkers, the iteration following recycling).

get_new_weight_data(*n_iter*)

find_bin_mapper(*hashval*)

Check to see if the given has value is in the binning table. Returns the index in the bin data tables if found, or raises KeyError if not.

get_bin_mapper(*hashval*)

Look up the given hash value in the binning table, unpickling and returning the corresponding bin mapper if available, or raising KeyError if not.

save_bin_mapper(*hashval*, *pickle_data*)

Store the given mapper in the table of saved mappers. If the mapper cannot be stored, PickleError will be raised. Returns the index in the bin data tables where the mapper is stored.

save_iter_binning(*n_iter*, *hashval*, *pickled_mapper*, *target_counts*)

Save information about the binning used to generate segments for iteration n_iter.

westpa.core.data_manager.**normalize_dataset_options**(*dsopts*, *path_prefix=''*, *n_iter=0*)

westpa.core.data_manager.**create_dataset_from_dsopts**(*group*, *dsopts*, *shape=None*, *dtype=None*, *data=None*, *autocompress_threshold=None*, *n_iter=None*)

westpa.core.data_manager.**require_dataset_from_dsopts**(*group*, *dsopts*, *shape=None*, *dtype=None*, *data=None*, *autocompress_threshold=None*, *n_iter=None*)

westpa.core.data_manager.**calc_chunksize**(*shape*, *dtype*, *max_chunksize=262144*)

Calculate a chunk size for HDF5 data, anticipating that access will slice along lower dimensions sooner than higher dimensions.

6.2.6.3 westpa.core.extloader module

`westpa.core.extloader.`**`load_module`**(*module_name*, *path=None*)

Load and return the given module, recursively loading containing packages as necessary.

`westpa.core.extloader.`**`get_object`**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

6.2.6.4 westpa.core.h5io module

Miscellaneous routines to help with HDF5 input and output of WEST-related data.

`class` `westpa.core.h5io.`**`Trajectory`**(*xyz*, *topology*, *time=None*, *unitcell_lengths=None*, *unitcell_angles=None*)

Bases: `object`

Container object for a molecular dynamics trajectory

A Trajectory represents a collection of one or more molecular structures, generally (but not necessarily) from a molecular dynamics trajectory. The Trajectory stores a number of fields describing the system through time, including the cartesian coordinates of each atoms (`xyz`), the topology of the molecular system (`topology`), and information about the unitcell if appropriate (`unitcell_vectors`, `unitcell_length`, `unitcell_angles`).

A Trajectory should generally be constructed by loading a file from disk. Trajectories can be loaded from (and saved to) the PDB, XTC, TRR, DCD, binpos, NetCDF or MDTraj HDF5 formats.

Trajectory supports fancy indexing, so you can extract one or more frames from a Trajectory as a separate trajectory. For example, to form a trajectory with every other frame, you can slice with `traj[::2]`.

Trajectory uses the nanometer, degree & picosecond unit system.

Examples

```
>>> # loading a trajectory
>>> import mdtraj as md
>>> md.load('trajectory.xtc', top='native.pdb')
<mdtraj.Trajectory with 1000 frames, 22 atoms at 0x1058a73d0>
```

```
>>> # slicing a trajectory
>>> t = md.load('trajectory.h5')
>>> print(t)
<mdtraj.Trajectory with 100 frames, 22 atoms>
>>> print(t[::2])
<mdtraj.Trajectory with 50 frames, 22 atoms>
```

```
>>> # calculating the average distance between two atoms
>>> import mdtraj as md
>>> import numpy as np
>>> t = md.load('trajectory.h5')
>>> np.mean(np.sqrt(np.sum((t.xyz[:, 0, :] - t.xyz[:, 21, :])**2, axis=1)))
```

See also:

`mdtraj.load`
High-level function that loads files and returns an `md.Trajectory`

n_frames

Type
int

n_atoms

Type
int

n_residues

Type
int

time

Type
np.ndarray, shape=(n_frames,)

timestep

Type
float

topology

Type
md.Topology

top

Type
md.Topology

xyz

Type
np.ndarray, shape=(n_frames, n_atoms, 3)

unitcell_vectors

Type
{np.ndarray, shape=(n_frames, 3, 3), None}

unitcell_lengths

Type
{np.ndarray, shape=(n_frames, 3), None}

unitcell_angles

Type
{np.ndarray, shape=(n_frames, 3), None}

property n_frames

Number of frames in the trajectory

Returns
n_frames – The number of frames in the trajectory

Return type
int

property n_atoms

Number of atoms in the trajectory

Returns
n_atoms – The number of atoms in the trajectory

Return type
int

property n_residues

Number of residues (amino acids) in the trajectory

Returns
n_residues – The number of residues in the trajectory's topology

Return type
int

property n_chains

Number of chains in the trajectory

Returns
n_chains – The number of chains in the trajectory's topology

Return type
int

property top

Alias for self.topology, describing the organization of atoms into residues, bonds, etc

Returns
topology – The topology object, describing the organization of atoms into residues, bonds, etc

Return type
md.Topology

property timestep

Timestep between frames, in picoseconds

Returns
timestep – The timestep between frames, in picoseconds.

Return type
float

property unitcell_vectors

The vectors that define the shape of the unit cell in each frame

Returns
vectors – Vectors defining the shape of the unit cell in each frame. The semantics of this array are that the shape of the unit cell in frame `i` are given by the three vectors, `value[i, 0, :]`, `value[i, 1, :]`, and `value[i, 2, :]`.

Return type
np.ndarray, shape(n_frames, 3, 3)

property unitcell_volumes

Volumes of unit cell for each frame.

Returns
volumes – Volumes of the unit cell in each frame, in nanometers^3, or None if the Trajectory contains no unitcell information.

Return type
{np.ndarray, shape=(n_frames), None}

superpose(*reference*, *frame=0*, *atom_indices=None*, *ref_atom_indices=None*, *parallel=True*)

Superpose each conformation in this trajectory upon a reference

Parameters

- **reference** (*md.Trajectory*) – Align self to a particular frame in *reference*
- **frame** (*int*) – The index of the conformation in *reference* to align to.
- **atom_indices** (*array_like, or None*) – The indices of the atoms to superpose. If not supplied, all atoms will be used.
- **ref_atom_indices** (*array_like, or None*) – Use these atoms on the reference structure. If not supplied, the same atom indices will be used for this trajectory and the reference one.
- **parallel** (*bool*) – Use OpenMP to run the superposition in parallel over multiple cores

Return type
self

join(*other*, *check_topology=True*, *discard_overlapping_frames=False*)

Join two trajectories together along the time/frame axis.

This method joins trajectories along the time axis, giving a new trajectory of length equal to the sum of the lengths of *self* and *other*. It can also be called by using *self + other*

Parameters

- **other** (*Trajectory or list of Trajectory*) – One or more trajectories to join with this one. These trajectories are *appended* to the end of this trajectory.
- **check_topology** (*bool*) – Ensure that the topology of *self* and *other* are identical before joining them. If false, the resulting trajectory will have the topology of *self*.
- **discard_overlapping_frames** (*bool, optional*) – If True, compare coordinates at trajectory edges to discard overlapping frames. Default: False.

See also:

stack
join two trajectories along the atom axis

stack(*other*, *keep_resSeq=True*)

Stack two trajectories along the atom axis

This method joins trajectories along the atom axis, giving a new trajectory with a number of atoms equal to the sum of the number of atoms in *self* and *other*.

Notes

The resulting trajectory will have the unitcell and time information the left operand.

Examples

```
>>> t1 = md.load('traj1.h5')
>>> t2 = md.load('traj2.h5')
>>> # even when t2 contains no unitcell information
>>> t2.unitcell_vectors = None
>>> stacked = t1.stack(t2)
>>> # the stacked trajectory inherits the unitcell information
>>> # from the first trajectory
```

(continues on next page)

(continued from previous page)

```
>>> np.all(stacked.unitcell_vectors == t1.unitcell_vectors)
True
```

Parameters

- **other** (Trajectory) – The other trajectory to join
- **keep_resSeq** (*bool, optional, default=True*) – see `mdtraj.core.topology.Topology.join` method documentation

See also:

join
join two trajectories along the time/frame axis.

slice(*key*, *copy=True*)

Slice trajectory, by extracting one or more frames into a separate object

This method can also be called using index bracket notation, i.e *traj[1] == traj.slice(1)*

Parameters

- **key** (*{int, np.ndarray, slice}*) – The slice to take. Can be either an int, a list of ints, or a slice object.
- **copy** (*bool, default=True*) – Copy the arrays after slicing. If you set this to false, then if you modify a slice, you'll modify the original array since they point to the same data.

property topology

Topology of the system, describing the organization of atoms into residues, bonds, etc

Returns

topology – The topology object, describing the organization of atoms into residues, bonds, etc

Return type

md.Topology

property xyz

Cartesian coordinates of each atom in each simulation frame

Returns

xyz – A three dimensional numpy array, with the cartesian coordinates of each atoms in each frame.

Return type

np.ndarray, shape=(n_frames, n_atoms, 3)

property unitcell_lengths

Lengths that define the shape of the unit cell in each frame.

Returns

lengths – Lengths of the unit cell in each frame, in nanometers, or None if the Trajectory contains no unitcell information.

Return type

{np.ndarray, shape=(n_frames, 3), None}

property unitcell_angles

Angles that define the shape of the unit cell in each frame.

Returns

lengths – The angles between the three unitcell vectors in each frame, `alpha`, `beta`, and `gamma`. `alpha' gives the angle between vectors ``b` and `c`, `beta` gives the angle between vectors `c` and `a`, and `gamma` gives the angle between vectors `a` and `b`. The angles are in degrees.

Return type

np.ndarray, shape=(n_frames, 3)

property time

The simulation time corresponding to each frame, in picoseconds

Returns

time – The simulation time corresponding to each frame, in picoseconds

Return type

np.ndarray, shape=(n_frames,)

openmm_positions(*frame*)

OpenMM-compatable positions of a single frame.

Examples

```
>>> t = md.load('trajectory.h5')
>>> context.setPositions(t.openmm_positions(0))
```

Parameters

frame (*int*) – The index of frame of the trajectory that you wish to extract

Returns

positions – The cartesian coordinates of specific trajectory frame, formatted for input to OpenMM

Return type

list

openmm_boxes(*frame*)

OpenMM-compatable box vectors of a single frame.

Examples

```
>>> t = md.load('trajectory.h5')
>>> context.setPeriodicBoxVectors(t.openmm_positions(0))
```

Parameters

frame (*int*) – Return box for this single frame.

Returns

box – The periodic box vectors for this frame, formatted for input to OpenMM.

Return type

tuple

static load(*filenames*, ***kwargs*)

Load a trajectory from disk

Parameters

- **filenames** (*{path-like, [path-like]}*) – Either a path or list of paths

- **extension** (*As requested by the various load functions -- it depends on the*)

save(*filename*, ***kwargs*)

Save trajectory to disk, in a format determined by the filename extension

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory. The extension will be parsed and will control the format.
- **lossy** (*bool*) – For .h5 or .lh5, whether or not to use compression.
- **no_models** (*bool*) – For .pdb. TODO: Document this?
- **force_overwrite** (*bool*) – If *filename* already exists, overwrite it.

save_hdf5(*filename*, *force_overwrite=True*)

Save trajectory to MDTraj HDF5 format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_lammpstrj(*filename*, *force_overwrite=True*)

Save trajectory to LAMMPS custom dump format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_xyz(*filename*, *force_overwrite=True*)

Save trajectory to .xyz format.

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_pdb(*filename*, *force_overwrite=True*, *bfactors=None*)

Save trajectory to RCSB PDB format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there
- **bfactors** (*array_like, default=None, shape=(n_frames, n_atoms) or (n_atoms,)*) – Save bfactors with pdb file. If the array is two dimensional it should contain a bfactor for each atom in each frame of the trajectory. Otherwise, the same bfactor will be saved in each frame.

save_xtc(*filename*, *force_overwrite=True*)

Save trajectory to Gromacs XTC format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory

- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_trr(*filename*, *force_overwrite=True*)

Save trajectory to Gromacs TRR format

Notes

Only the xyz coordinates and the time are saved, the velocities and forces in the trr will be zeros

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_dcd(*filename*, *force_overwrite=True*)

Save trajectory to CHARMM/NAMD DCD format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filenames, if its already there

save_dtr(*filename*, *force_overwrite=True*)

Save trajectory to DESMOND DTR format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filenames, if its already there

save_binpos(*filename*, *force_overwrite=True*)

Save trajectory to AMBER BINPOS format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_mdcrd(*filename*, *force_overwrite=True*)

Save trajectory to AMBER mdcrd format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_netcdf(*filename*, *force_overwrite=True*)

Save trajectory in AMBER NetCDF format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if it's already there

save_netcdfrst(*filename*, *force_overwrite=True*)

Save trajectory in AMBER NetCDF restart format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the restart
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if it's already there

Notes

NetCDF restart files can only store a single frame. If only one frame exists, "filename" will be written. Otherwise, "filename.#" will be written, where # is a zero-padded number from 1 to the total number of frames in the trajectory

save_amberrst7(*filename*, *force_overwrite=True*)

Save trajectory in AMBER ASCII restart format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the restart
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if it's already there

Notes

Amber restart files can only store a single frame. If only one frame exists, "filename" will be written. Otherwise, "filename.#" will be written, where # is a zero-padded number from 1 to the total number of frames in the trajectory

save_lh5(*filename*, *force_overwrite=True*)

Save trajectory in deprecated MSMBuilder2 LH5 (lossy HDF5) format.

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if it's already there

save_gro(*filename*, *force_overwrite=True*, *precision=3*)

Save trajectory in Gromacs .gro format

Parameters

- **filename** (*path-like*) – Path to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at that filename if it exists
- **precision** (*int, default=3*) – The number of decimal places to use for coordinates in GRO file

save_tng(*filename*, *force_overwrite=True*)

Save trajectory to Gromacs TNG format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filename, if its already there

save_gsd(*filename*, *force_overwrite=True*)

Save trajectory to HOOMD GSD format

Parameters

- **filename** (*path-like*) – filesystem path in which to save the trajectory
- **force_overwrite** (*bool, default=True*) – Overwrite anything that exists at filenames, if its already there

center_coordinates(*mass_weighted=False*)

Center each trajectory frame at the origin (0,0,0).

This method acts inplace on the trajectory. The centering can be either uniformly weighted (mass_weighted=False) or weighted by the mass of each atom (mass_weighted=True).

Parameters

mass_weighted (*bool, optional (default = False)*) – If True, weight atoms by mass when removing COM.

Return type

self

restrict_atoms(***kwargs*)

DEPRECATED: restrict_atoms was replaced by atom_slice and will be removed in 2.0

Retain only a subset of the atoms in a trajectory

Deletes atoms not in *atom_indices*, and re-indexes those that remain

atom_indices

[array-like, dtype=int, shape=(n_atoms)] List of atom indices to keep.

inplace

[bool, default=True] If `True`, the operation is done inplace, modifying `self`. Otherwise, a copy is returned with the restricted atoms, and `self` is not modified.

traj

[md.Trajectory] The return value is either `self`, or the new trajectory, depending on the value of `inplace`.

atom_slice(*atom_indices*, *inplace=False*)

Create a new trajectory from a subset of atoms

Parameters

- **atom_indices** (*array-like, dtype=int, shape=(n_atoms)*) – List of indices of atoms to retain in the new trajectory.
- **inplace** (*bool, default=False*) – If `True`, the operation is done inplace, modifying `self`. Otherwise, a copy is returned with the sliced atoms, and `self` is not modified.

Returns

traj – The return value is either `self`, or the new trajectory, depending on the value of `inplace`.

Return type

md.Trajectory

See also:

stack

stack multiple trajectories along the atom axis

remove_solvent(*exclude=None*, *inplace=False*)

Create a new trajectory without solvent atoms

Parameters

- **exclude** (*array-like, dtype=str, shape=(n_solvent_types)*) – List of solvent residue names to retain in the new trajectory.
- **inplace** (*bool, default=False*) – The return value is either `self`, or the new trajectory, depending on the value of `inplace`.

Returns

traj – The return value is either `self`, or the new trajectory, depending on the value of `inplace`.

Return type

md.Trajectory

smooth(*width*, *order=3*, *atom_indices=None*, *inplace=False*)

Smoothen a trajectory using a zero-delay Buttersworth filter. Please note that for optimal results the trajectory should be properly aligned prior to smoothing (see *md.Trajectory.superpose*).

Parameters

- **width** (*int*) – This acts very similar to the window size in a moving average smoother. In this implementation, the frequency of the low-pass filter is taken to be two over this width, so it's like "half the period" of the sinusiod where the filter starts to kick in. Must be an integer greater than one.
- **order** (*int, optional, default=3*) – The order of the filter. A small odd number is recommended. Higher order filters cutoff more quickly, but have worse numerical properties.
- **atom_indices** (*array-like, dtype=int, shape=(n_atoms), default=None*) – List of indices of atoms to retain in the new trajectory. Default is set to *None*, which applies smoothing to all atoms.
- **inplace** (*bool, default=False*) – The return value is either `self`, or the new trajectory, depending on the value of `inplace`.

Returns

traj – The return value is either `self`, or the new smoothed trajectory, depending on the value of `inplace`.

Return type

md.Trajectory

References

make_molecules_whole(*inplace=False*, *sorted_bonds=None*)

Only make molecules whole

Parameters

- **inplace** (*bool*) – If False, a new Trajectory is created and returned. If True, this Trajectory is modified directly.
- **sorted_bonds** (*array of shape (n_bonds, 2)*) – Pairs of atom indices that define bonds, in sorted order. If not specified, these will be determined from the trajectory's topology.

See also:

image_molecules

image_molecules(*inplace=False*, *anchor_molecules=None*, *other_molecules=None*, *sorted_bonds=None*, *make_whole=True*)

Recenter and apply periodic boundary conditions to the molecules in each frame of the trajectory.

This method is useful for visualizing a trajectory in which molecules were not wrapped to the periodic unit cell, or in which the macromolecules are not centered with respect to the solvent. It tries to be intelligent in deciding what molecules to center, so you can simply call it and trust that it will "do the right thing".

Parameters

- **inplace** (*bool, default=False*) – If False, a new Trajectory is created and returned. If True, this Trajectory is modified directly.
- **anchor_molecules** (*list of atom sets, optional, default=None*) – Molecule that should be treated as an "anchor". These molecules will be centered in the box and put near each other. If not specified, anchor molecules are guessed using a heuristic.
- **other_molecules** (*list of atom sets, optional, default=None*) – Molecules that are not anchors. If not specified, these will be molecules other than the anchor molecules
- **sorted_bonds** (*array of shape (n_bonds, 2)*) – Pairs of atom indices that define bonds, in sorted order. If not specified, these will be determined from the trajectory's topology. Only relevant if `make_whole` is True.
- **make_whole** (*bool*) – Whether to make molecules whole.

Returns

traj – The return value is either `self` or the new trajectory, depending on the value of `inplace`.

Return type

md.Trajectory

See also:

`Topology.guess_anchor_molecules`

`westpa.core.h5io.`**join_traj**(*trajs*, *check_topology=True*, *discard_overlapping_frames=False*)

Concatenate multiple trajectories into one long trajectory

Parameters

- **trajs** (*iterable of trajectories*) – Combine these into one trajectory
- **check_topology** (*bool*) – Make sure topologies match before joining
- **discard_overlapping_frames** (*bool*) – Check for overlapping frames and discard

`westpa.core.h5io.`**in_units_of**(*quantity*, *units_in*, *units_out*, *inplace=False*)

Convert a numerical quantity between unit systems.

Parameters

- **quantity** (*{number, np.ndarray, openmm.unit.Quantity}*) – quantity can either be a unitted quantity – i.e. instance of openmm.unit.Quantity, or just a bare number or numpy array
- **units_in** (*str*) – If you supply a quantity that's not a openmm.unit.Quantity, you should tell me what units it is in. If you don't, i'm just going to echo you back your quantity without doing any unit checking.
- **units_out** (*str*) – A string description of the units you want out. This should look like "nanometers/picosecond" or "nanometers**3" or whatever

- **inplace** (*bool*) – Attempt to do the transformation inplace, by mutating the *quantity* argument and avoiding a copy. This is only possible if *quantity* is a writable numpy array.

Returns

rquantity – The resulting quantity, in the new unit system. If the function was called with *inplace=True* and *quantity* was a writable numpy array, *rquantity* will alias the same memory as the input *quantity*, which will have been changed inplace. Otherwise, if a copy was required, *rquantity* will point to new memory.

Return type

{number, np.ndarray}

Examples

```
>>> in_units_of(1, 'meter**2/second', 'nanometers**2/picosecond')
1000000.0
```

westpa.core.h5io.**import_**(*module*)

Import a module, and issue a nice message to stderr if the module isn't installed.

Currently, this function will print nice error messages for networkx, tables, netCDF4, and openmm.unit, which are optional MDTraj dependencies.

Parameters

module (*str*) – The module you'd like to import, as a string

Returns

module – The module object

Return type

{module, object}

Examples

```
>>> # the following two lines are equivalent. the difference is that the
>>> # second will check for an ImportError and print you a very nice
>>> # user-facing message about what's wrong (where you can install the
>>> # module from, etc) if the import fails
>>> import tables
>>> tables = import_('tables')
```

westpa.core.h5io.**ensure_type**(*val*, *dtype*, *ndim*, *name*, *length=None*, *can_be_none=False*, *shape=None*, *warn_on_cast=True*, *add_newaxis_on_deficient_ndim=False*)

Typecheck the size, shape and dtype of a numpy array, with optional casting.

Parameters

- **val** (*{np.ndaraay, None}*) – The array to check
- **dtype** (*{nd.dtype, str}*) – The dtype you'd like the array to have
- **ndim** (*int*) – The number of dimensions you'd like the array to have
- **name** (*str*) – name of the array. This is used when throwing exceptions, so that we can describe to the user which array is messed up.
- **length** (*int, optional*) – How long should the array be?
- **can_be_none** (*bool*) – Is `val == None` acceptable?

- **shape** (*tuple, optional*) – What should be shape of the array be? If the provided tuple has Nones in it, those will be semantically interpreted as matching any length in that dimension. So, for example, using the shape spec `(None, None, 3)` will ensure that the last dimension is of length three without constraining the first two dimensions
- **warn_on_cast** (*bool, default=True*) – Raise a warning when the dtypes don't match and a cast is done.
- **add_newaxis_on_deficient_ndim** (*bool, default=True*) – Add a new axis to the beginining of the array if the number of dimensions is deficient by one compared to your specification. For instance, if you're trying to get out an array of `ndim == 3`, but the user provides an array of `shape == (10, 10)`, a new axis will be created with length 1 in front, so that the return value is of shape `(1, 10, 10)`.

Notes

The returned value will always be C-contiguous.

Returns

typechecked_val – If *val=None* and *can_be_none=True*, then this will return None. Otherwise, it will return val (or a copy of val). If the dtype wasn't right, it'll be casted to the right shape. If the array was not C-contiguous, it'll be copied as well.

Return type

np.ndarray, None

class westpa.core.h5io.**HDF5TrajectoryFile**(*filename*, *mode='r'*, *force_overwrite=True*, *compression='zlib'*)

Bases: `object`

Interface for reading and writing to a MDTraj HDF5 molecular dynamics trajectory file, whose format is described here.

This is a file-like object, that both reading or writing depending on the *mode* flag. It implements the context manager protocol, so you can also use it with the python 'with' statement.

The format is extremely flexible and high performance. It can hold a wide variety of information about a trajectory, including fields like the temperature and energies. Because it's built on the fantastic HDF5 library, it's easily extensible too.

Parameters

- **filename** (*path-like*) – Path to the file to open
- **mode** (*{'r, 'w'}*) – Mode in which to open the file. 'r' is for reading and 'w' is for writing
- **force_overwrite** (*bool*) – In mode='w', how do you want to behave if a file by the name of *filename* already exists? if *force_overwrite=True*, it will be overwritten.
- **compression** (*{'zlib', None}*) – Apply compression to the file? This will save space, and does not cost too many cpu cycles, so it's recommended.

root

title

application

topology

randomState

forcefield

reference

constraints

See also:

mdtraj.load_hdf5
: High-level wrapper that returns a md.Trajectory

distance_unit = 'nanometers'

property root
: Direct access to the root group of the underlying Tables HDF5 file handle.

  This can be used for random or specific access to the underlying arrays on disk

property title
: User-defined title for the data represented in the file

property application
: Suite of programs that created the file

property topology
: Get the topology out from the file

  Returns
  topology – A topology object

  Return type
  mdtraj.Topology

property randomState
: State of the creators internal random number generator at the start of the simulation

property forcefield
: Description of the hamiltonian used. A short, human readable string, like AMBER99sbildn.

property reference
: A published reference that documents the program or parameters used to generate the data

property constraints
: Constraints applied to the bond lengths

  Returns
  constraints – A one dimensional array with the a int, int, float type giving the index of the two atoms involved in the constraints and the distance of the constraint. If no constraint information is in the file, the return value is None.

  Return type
  {None, np.array, dtype=[('atom1', '<i4'), ('atom2', '<i4'), ('distance', '<f4')])}

read_as_traj(*n_frames=None*, *stride=None*, *atom_indices=None*)
: Read a trajectory from the HDF5 file

  Parameters

  - **n_frames** (*{int, None}*) – The number of frames to read. If not supplied, all of the remaining frames will be read.
  - **stride** (*{int, None}*) – By default all of the frames will be read, but you can pass this flag to read a subset of of the data by grabbing only every *stride*-th frame from disk.

- **atom_indices** (*{int, None}*) – By default all of the atom will be read, but you can pass this flag to read only a subsets of the atoms for the *coordinates* and *velocities* fields. Note that you will have to carefully manage the indices and the offsets, since the *i*-th atom in the topology will not necessarily correspond to the *i*-th atom in your subset.

Returns
trajectory – A trajectory object containing the loaded portion of the file.

Return type
Trajectory

read(*n_frames=None*, *stride=None*, *atom_indices=None*)

Read one or more frames of data from the file

Parameters

- **n_frames** (*{int, None}*) – The number of frames to read. If not supplied, all of the remaining frames will be read.
- **stride** (*{int, None}*) – By default all of the frames will be read, but you can pass this flag to read a subset of of the data by grabbing only every *stride*-th frame from disk.
- **atom_indices** (*{int, None}*) – By default all of the atom will be read, but you can pass this flag to read only a subsets of the atoms for the *coordinates* and *velocities* fields. Note that you will have to carefully manage the indices and the offsets, since the *i*-th atom in the topology will not necessarily correspond to the *i*-th atom in your subset.

Notes

If you'd like more flexible access to the data, that is available by using the pytables group directly, which is accessible via the *root* property on this class.

Returns
frames – The returned namedtuple will have the fields "coordinates", "time", "cell_lengths", "cell_angles", "velocities", "kineticEnergy", "potentialEnergy", "temperature" and "alchemicalLambda". Each of the fields in the returned namedtuple will either be a numpy array or None, dependening on if that data was saved in the trajectory. All of the data shall be n units of "nanometers", "picoseconds", "kelvin", "degrees" and "kilojoules_per_mole".

Return type
namedtuple

write(*coordinates*, *time=None*, *cell_lengths=None*, *cell_angles=None*, *velocities=None*, *kineticEnergy=None*, *potentialEnergy=None*, *temperature=None*, *alchemicalLambda=None*)

Write one or more frames of data to the file

This method saves data that is associated with one or more simulation frames. Note that all of the arguments can either be raw numpy arrays or unitted arrays (with openmm.unit.Quantity). If the arrays are unittted, a unit conversion will be automatically done from the supplied units into the proper units for saving on disk. You won't have to worry about it.

Furthermore, if you wish to save a single frame of simulation data, you can do so naturally, for instance by supplying a 2d array for the coordinates and a single float for the time. This "shape deficiency" will be recognized, and handled appropriately.

Parameters

- **coordinates** (*np.ndarray, shape=(n_frames, n_atoms, 3)*) – The cartesian coordinates of the atoms to write. By convention, the lengths should be in units of nanometers.
- **time** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the simulation time, in picoseconds corresponding to each frame.
- **cell_lengths** (*np.ndarray, shape=(n_frames, 3), dtype=float32, optional*) – You may optionally specify the unitcell lengths. The length of the periodic box in each frame, in each direction, *a*, *b*, *c*. By convention the lengths should be in units of angstroms.
- **cell_angles** (*np.ndarray, shape=(n_frames, 3), dtype=float32, optional*) – You may optionally specify the unitcell angles in each frame. Organized analogously to cell_lengths. Gives the alpha, beta and gamma angles respectively. By convention, the angles should be in units of degrees.
- **velocities** (*np.ndarray, shape=(n_frames, n_atoms, 3), optional*) – You may optionally specify the cartesian components of the velocity for each atom in each frame. By convention, the velocities should be in units of nanometers / picosecond.
- **kineticEnergy** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the kinetic energy in each frame. By convention the kinetic energies should b in units of kilojoules per mole.
- **potentialEnergy** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the potential energy in each frame. By convention the kinetic energies should b in units of kilojoules per mole.
- **temperature** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the temperature in each frame. By convention the temperatures should b in units of Kelvin.
- **alchemicalLambda** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the alchemical lambda in each frame. These have no units, but are generally between zero and one.

seek(*offset*, *whence=0*)

Move to a new file position

Parameters

- **offset** (*int*) – A number of frames.
- **whence** (*{0, 1, 2}*) – 0: offset from start of file, offset should be >=0. 1: move relative to the current position, positive or negative 2: move relative to the end of file, offset should be <= 0. Seeking beyond the end of a file is not supported

tell()

Current file position

Returns

offset – The current frame in the file.

Return type

int

close()

Close the HDF5 file handle

flush()

Write all buffered data in the to the disk file.

class westpa.core.h5io.**Frames**(*coordinates*, *time*, *cell_lengths*, *cell_angles*, *velocities*, *kineticEnergy*, *potentialEnergy*, *temperature*, *alchemicalLambda*)

Bases: tuple

Create new instance of Frames(coordinates, time, cell_lengths, cell_angles, velocities, kineticEnergy, potentialEnergy, temperature, alchemicalLambda)

alchemicalLambda

Alias for field number 8

cell_angles

Alias for field number 3

cell_lengths

Alias for field number 2

coordinates

Alias for field number 0

kineticEnergy

Alias for field number 5

potentialEnergy

Alias for field number 6

temperature

Alias for field number 7

time

Alias for field number 1

velocities

Alias for field number 4

class westpa.core.h5io.**WESTTrajectory**(*coordinates*, *topology=None*, *time=None*, *iter_labels=None*, *seg_labels=None*, *pcoords=None*, *parent_ids=None*, *unitcell_lengths=None*, *unitcell_angles=None*)

Bases: *Trajectory*

A subclass of mdtraj.Trajectory that contains the trajectory of atom coordinates with pointers denoting the iteration number and segment index of each frame.

iter_label_values()

seg_label_values(*iteration=None*)

property label_values

property iter_labels

Iteration index corresponding to each frame

Returns

time – The iteration index corresponding to each frame

Return type

np.ndarray, shape=(n_frames,)

property seg_labels

Segment index corresponding to each frame

Returns

time – The segment index corresponding to each frame

Return type

np.ndarray, shape=(n_frames,)

property pcoords

property parent_ids

join(*other*, *check_topology=True*, *discard_overlapping_frames=False*)

Join two Trajectory``s. This overrides ``mdtraj.Trajectory.join so that it also handles WESTPA pointers. mdtraj.Trajectory.join's documentation for more details.

slice(*key*, *copy=True*)

Slice the Trajectory. This overrides mdtraj.Trajectory.slice so that it also handles WESTPA pointers. Please see mdtraj.Trajectory.slice's documentation for more details.

westpa.core.h5io.**resolve_filepath**(*path*, *constructor=<class 'h5py._hl.files.File'>*, *cargs=None*, *ckwargs=None*, ***addtlkwargs*)

Use a combined filesystem and HDF5 path to open an HDF5 file and return the appropriate object. Returns (h5file, h5object). The file is opened using constructor(filename, *cargs, **ckwargs).

westpa.core.h5io.**calc_chunksize**(*shape*, *dtype*, *max_chunksize=262144*)

Calculate a chunk size for HDF5 data, anticipating that access will slice along lower dimensions sooner than higher dimensions.

westpa.core.h5io.**tostr**(*b*)

Convert a nonstandard string object b to str with the handling of the case where b is bytes.

westpa.core.h5io.**is_within_directory**(*directory*, *target*)

westpa.core.h5io.**safe_extract**(*tar*, *path='.'*, *members=None*, ***, *numeric_owner=False*)

westpa.core.h5io.**create_hdf5_group**(*parent_group*, *groupname*, *replace=False*, *creating_program=None*)

Create (or delete and recreate) and HDF5 group named groupname within the enclosing Group (object) parent_group. If replace is True, then the group is replaced if present; if False, then an error is raised if the group is present. After the group is created, HDF5 attributes are set using *stamp_creator_data*.

westpa.core.h5io.**stamp_creator_data**(*h5group*, *creating_program=None*)

Mark the following on the HDF5 group h5group:

creation_program

The name of the program that created the group

creation_user

The username of the user who created the group

creation_hostname

The hostname of the machine on which the group was created

creation_time

The date and time at which the group was created, in the current locale.

creation_unix_time

The Unix time (seconds from the epoch, UTC) at which the group was created.

This is meant to facilitate tracking the flow of data, but should not be considered a secure paper trail (after all, anyone with write access to the HDF5 file can modify these attributes).

westpa.core.h5io.**get_creator_data**(*h5group*)

Read back creator data as written by stamp_creator_data, returning a dictionary with keys as described for stamp_creator_data. Missing fields are denoted with None. The creation_time field is returned as a string.

westpa.core.h5io.**load_west**(*filename*)

Load WESTPA trajectory files from disk.

Parameters

filename (*str*) – String filename of HDF Trajectory file.

westpa.core.h5io.**stamp_iter_range**(*h5object*, *start_iter*, *stop_iter*)

Mark that the HDF5 object h5object (dataset or group) contains data from iterations start_iter <= n_iter < stop_iter.

westpa.core.h5io.**get_iter_range**(*h5object*)

Read back iteration range data written by stamp_iter_range

westpa.core.h5io.**stamp_iter_step**(*h5group*, *iter_step*)

Mark that the HDF5 object h5object (dataset or group) contains data with an iteration step (stride) of iter_step).

westpa.core.h5io.**get_iter_step**(*h5group*)

Read back iteration step (stride) written by stamp_iter_step

westpa.core.h5io.**check_iter_range_least**(*h5object*, *iter_start*, *iter_stop*)

Return True if the iteration range [iter_start, iter_stop) is the same as or entirely contained within the iteration range stored on h5object.

westpa.core.h5io.**check_iter_range_equal**(*h5object*, *iter_start*, *iter_stop*)

Return True if the iteration range [iter_start, iter_stop) is the same as the iteration range stored on h5object.

westpa.core.h5io.**get_iteration_entry**(*h5object*, *n_iter*)

Create a slice for data corresponding to iteration n_iter in h5object.

westpa.core.h5io.**get_iteration_slice**(*h5object*, *iter_start*, *iter_stop=None*, *iter_stride=None*)

Create a slice for data corresponding to iterations [iter_start,iter_stop), with stride iter_step, in the given h5object.

westpa.core.h5io.**label_axes**(*h5object*, *labels*, *units=None*)

Stamp the given HDF5 object with axis labels. This stores the axis labels in an array of strings in an attribute called axis_labels on the given object. units if provided is a corresponding list of units.

class westpa.core.h5io.**WESTPAH5File**(**args*, ***kwargs*)

Bases: File

Generalized input/output for WESTPA simulation (or analysis) data.

Create a new file object.

See the h5py user guide for a detailed explanation of the options.

name

Name of the file on disk, or file-like object. Note: for files created with the 'core' driver, HDF5 still requires this be non-empty.

mode

r Readonly, file must exist (default) r+ Read/write, file must exist w Create file, truncate if exists w- or x Create file, fail if exists a Read/write if exists, create otherwise

driver

Name of the driver to use. Legal values are None (default, recommended), 'core', 'sec2', 'direct', 'stdio', 'mpio', 'ros3'.

libver
Library version bounds. Supported values: 'earliest', 'v108', 'v110', 'v112' and 'latest'. The 'v108', 'v110' and 'v112' options can only be specified with the HDF5 1.10.2 library or later.

userblock_size
Desired size of user block. Only allowed when creating a new file (mode w, w- or x).

swmr
Open the file in SWMR read mode. Only used when mode = 'r'.

rdcc_nbytes
Total size of the dataset chunk cache in bytes. The default size is 1024**2 (1 MiB) per dataset. Applies to all datasets unless individually changed.

rdcc_w0
The chunk preemption policy for all datasets. This must be between 0 and 1 inclusive and indicates the weighting according to which chunks which have been fully read or written are penalized when determining which chunks to flush from cache. A value of 0 means fully read or written chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of 1 means fully read or written chunks are always preempted before other chunks. If your application only reads or writes data once, this can be safely set to 1. Otherwise, this should be set lower depending on how often you re-read or re-write the same data. The default value is 0.75. Applies to all datasets unless individually changed.

rdcc_nslots
The number of chunk slots in the raw data chunk cache for this file. Increasing this value reduces the number of cache collisions, but slightly increases the memory used. Due to the hashing strategy, this value should ideally be a prime number. As a rule of thumb, this value should be at least 10 times the number of chunks that can fit in rdcc_nbytes bytes. For maximum performance, this value should be set approximately 100 times that number of chunks. The default value is 521. Applies to all datasets unless individually changed.

track_order
Track dataset/group/attribute creation order under root group if True. If None use global default h5.get_config().track_order.

fs_strategy
The file space handling strategy to be used. Only allowed when creating a new file (mode w, w- or x). Defined as: "fsm" FSM, Aggregators, VFD "page" Paged FSM, VFD "aggregate" Aggregators, VFD "none" VFD If None use HDF5 defaults.

fs_page_size
File space page size in bytes. Only used when fs_strategy="page". If None use the HDF5 default (4096 bytes).

fs_persist
A boolean value to indicate whether free space should be persistent or not. Only allowed when creating a new file. The default value is False.

fs_threshold
The smallest free-space section size that the free space manager will track. Only allowed when creating a new file. The default value is 1.

page_buf_size
Page buffer size in bytes. Only allowed for HDF5 files created with fs_strategy="page". Must be a power of two value and greater or equal than the file space page size when creating the file. It is not used by default.

min_meta_keep
Minimum percentage of metadata to keep in the page buffer before allowing pages containing metadata to be evicted. Applicable only if page_buf_size is set. Default value is zero.

min_raw_keep
Minimum percentage of raw data to keep in the page buffer before allowing pages containing raw data to be evicted. Applicable only if page_buf_size is set. Default value is zero.

locking
The file locking behavior. Defined as:
- False (or "false") – Disable file locking

- True (or "true") – Enable file locking
- "best-effort" – Enable file locking but ignore some errors
- None – Use HDF5 defaults

> **Warning:** The HDF5_USE_FILE_LOCKING environment variable can override this parameter.

Only available with HDF5 >= 1.12.1 or 1.10.x >= 1.10.7.

alignment_threshold
Together with `alignment_interval`, this property ensures that any file object greater than or equal in size to the alignment threshold (in bytes) will be aligned on an address which is a multiple of alignment interval.

alignment_interval
This property should be used in conjunction with `alignment_threshold`. See the description above. For more details, see https://portal.hdfgroup.org/display/HDF5/H5P_SET_ALIGNMENT

meta_block_size
Set the current minimum size, in bytes, of new metadata block allocations. See https://portal.hdfgroup.org/display/HDF5/H5P_SET_META_BLOCK_SIZE

Additional keywords
Passed on to the selected file driver.

default_iter_prec = 8

replace_dataset(**args*, ***kwargs*)

iter_object_name(*n_iter*, *prefix=''*, *suffix=''*)
Return a properly-formatted per-iteration name for iteration `n_iter`. (This is used in create/require/get_iter_group, but may also be useful for naming datasets on a per-iteration basis.)

create_iter_group(*n_iter*, *group=None*)
Create a per-iteration data storage group for iteration number `n_iter` in the group `group` (which is '/iterations' by default).

require_iter_group(*n_iter*, *group=None*)
Ensure that a per-iteration data storage group for iteration number `n_iter` is available in the group `group` (which is '/iterations' by default).

get_iter_group(*n_iter*, *group=None*)
Get the per-iteration data group for iteration number `n_iter` from within the group `group` ('/iterations' by default).

class westpa.core.h5io.**WESTIterationFile**(*file*, *mode='r'*, *force_overwrite=True*, *compression='zlib'*, *link=None*)

Bases: *HDF5TrajectoryFile*

read(*frame_indices=None*, *atom_indices=None*)
Read one or more frames of data from the file

Parameters

- **n_frames** (*{int, None}*) – The number of frames to read. If not supplied, all of the remaining frames will be read.
- **stride** (*{int, None}*) – By default all of the frames will be read, but you can pass this flag to read a subset of of the data by grabbing only every *stride*-th frame from disk.
- **atom_indices** (*{int, None}*) – By default all of the atom will be read, but you can pass this flag to read only a subsets of the atoms for the *coordinates* and

velocities fields. Note that you will have to carefully manage the indices and the offsets, since the *i*-th atom in the topology will not necessarily correspond to the *i*-th atom in your subset.

Notes

If you'd like more flexible access to the data, that is available by using the pytables group directly, which is accessible via the *root* property on this class.

Returns

frames – The returned namedtuple will have the fields "coordinates", "time", "cell_lengths", "cell_angles", "velocities", "kineticEnergy", "potentialEnergy", "temperature" and "alchemicalLambda". Each of the fields in the returned namedtuple will either be a numpy array or None, dependening on if that data was saved in the trajectory. All of the data shall be n units of "nanometers", "picoseconds", "kelvin", "degrees" and "kilojoules_per_mole".

Return type

namedtuple

`has_topology`()

`has_pointer`()

`has_restart`(*segment*)

`write_data`(*where*, *name*, *data*)

`read_data`(*where*, *name*)

`read_as_traj`(*iteration=None*, *segment=None*, *atom_indices=None*)

Read a trajectory from the HDF5 file

Parameters

- **n_frames** (*{int, None}*) – The number of frames to read. If not supplied, all of the remaining frames will be read.
- **stride** (*{int, None}*) – By default all of the frames will be read, but you can pass this flag to read a subset of of the data by grabbing only every *stride*-th frame from disk.
- **atom_indices** (*{int, None}*) – By default all of the atom will be read, but you can pass this flag to read only a subsets of the atoms for the *coordinates* and *velocities* fields. Note that you will have to carefully manage the indices and the offsets, since the *i*-th atom in the topology will not necessarily correspond to the *i*-th atom in your subset.

Returns

trajectory – A trajectory object containing the loaded portion of the file.

Return type

Trajectory

`read_restart`(*segment*)

`write_segment`(*segment*, *pop=False*)

class westpa.core.h5io.**DSSpec**

Bases: object

Generalized WE dataset access

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

get_segment_data(*n_iter*, *seg_id*)

class westpa.core.h5io.**FileLinkedDSSpec**(*h5file_or_name*)

Bases: *DSSpec*

Provide facilities for accessing WESTPA HDF5 files, including auto-opening and the ability to pickle references to such files for transmission (through, e.g., the work manager), provided that the HDF5 file can be accessed by the same path on both the sender and receiver.

property h5file

Lazily open HDF5 file. This is required because allowing an open HDF5 file to cross a fork() boundary generally corrupts the internal state of the HDF5 library.

class westpa.core.h5io.**SingleDSSpec**(*h5file_or_name*, *dsname*, *alias=None*, *slice=None*)

Bases: *FileLinkedDSSpec*

classmethod from_string(*dsspec_string*, *default_h5file*)

class westpa.core.h5io.**SingleIterDSSpec**(*h5file_or_name*, *dsname*, *alias=None*, *slice=None*)

Bases: *SingleDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.core.h5io.**SingleSegmentDSSpec**(*h5file_or_name*, *dsname*, *alias=None*, *slice=None*)

Bases: *SingleDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

get_segment_data(*n_iter*, *seg_id*)

class westpa.core.h5io.**FnDSSpec**(*h5file_or_name*, *fn*)

Bases: *FileLinkedDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.core.h5io.**MultiDSSpec**(*dsspecs*)

Bases: *DSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.core.h5io.**IterBlockedDataset**(*dataset_or_array*, *attrs=None*)

Bases: object

classmethod empty_like(*blocked_dataset*)

cache_data(*max_size=None*)

Cache this dataset in RAM. If `max_size` is given, then only cache if the entire dataset fits in `max_size` bytes. If `max_size` is the string 'available', then only cache if the entire dataset fits in available RAM, as defined by the `psutil` module.

drop_cache()

iter_entry(*n_iter*)

iter_slice(*start=None*, *stop=None*)

6.2.6.5 westpa.core.progress module

westpa.core.progress.**linregress**(*x*, *y=None*, *alternative='two-sided'*)

Calculate a linear least-squares regression for two sets of measurements.

Parameters

- **x** (*array_like*) – Two sets of measurements. Both arrays should have the same length. If only *x* is given (and y=None), then it must be a two-dimensional array where one dimension has length 2. The two sets of measurements are then found by splitting the array along the length-2 dimension. In the case where y=None and *x* is a 2x2 array, linregress(x) is equivalent to linregress(x[0], x[1]).
- **y** (*array_like*) – Two sets of measurements. Both arrays should have the same length. If only *x* is given (and y=None), then it must be a two-dimensional array where one dimension has length 2. The two sets of measurements are then found by splitting the array along the length-2 dimension. In the case where y=None and *x* is a 2x2 array, linregress(x) is equivalent to linregress(x[0], x[1]).
- **alternative** (*{'two-sided', 'less', 'greater'}, optional*) – Defines the alternative hypothesis. Default is 'two-sided'. The following options are available:
  - 'two-sided': the slope of the regression line is nonzero
  - 'less': the slope of the regression line is less than zero
  - 'greater': the slope of the regression line is greater than zero

  Added in version 1.7.0.

Returns

result – The return value is an object with the following attributes:

slope
[float] Slope of the regression line.

intercept
[float] Intercept of the regression line.

rvalue
[float] The Pearson correlation coefficient. The square of rvalue is equal to the coefficient of determination.

pvalue
[float] The p-value for a hypothesis test whose null hypothesis is that the slope is zero, using Wald Test with t-distribution of the test statistic. See *alternative* above for alternative hypotheses.

stderr
[float] Standard error of the estimated slope (gradient), under the assumption of residual normality.

intercept_stderr
[float] Standard error of the estimated intercept, under the assumption of residual normality.

Return type
LinregressResult instance

See also:

scipy.optimize.curve_fit
: Use non-linear least squares to fit a function to data.

scipy.optimize.leastsq
: Minimize the sum of squares of a set of equations.

Notes

Missing values are considered pair-wise: if a value is missing in *x*, the corresponding value in *y* is masked.

For compatibility with older versions of SciPy, the return value acts like a `namedtuple` of length 5, with fields `slope`, `intercept`, `rvalue`, `pvalue` and `stderr`, so one can continue to write:

```
slope, intercept, r, p, se = linregress(x, y)
```

With that style, however, the standard error of the intercept is not available. To have access to all the computed values, including the standard error of the intercept, use the return value as an object with attributes, e.g.:

```
result = linregress(x, y)
print(result.intercept, result.intercept_stderr)
```

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> rng = np.random.default_rng()
```

Generate some data:

```
>>> x = rng.random(10)
>>> y = 1.6*x + rng.random(10)
```

Perform the linear regression:

```
>>> res = stats.linregress(x, y)
```

Coefficient of determination (R-squared):

```
>>> print(f"R-squared: {res.rvalue**2:.6f}")
R-squared: 0.717533
```

Plot the data along with the fitted line:

```
>>> plt.plot(x, y, 'o', label='original data')
>>> plt.plot(x, res.intercept + res.slope*x, 'r', label='fitted line')
>>> plt.legend()
>>> plt.show()
```

Calculate 95% confidence interval on slope and intercept:

```
>>> # Two-sided inverse Students t-distribution
>>> # p - probability, df - degrees of freedom
>>> from scipy.stats import t
>>> tinv = lambda p, df: abs(t.ppf(p/2, df))
```

```
>>> ts = tinv(0.05, len(x)-2)
>>> print(f"slope (95%): {res.slope:.6f} +/- {ts*res.stderr:.6f}")
slope (95%): 1.453392 +/- 0.743465
>>> print(f"intercept (95%): {res.intercept:.6f}"
...       f" +/- {ts*res.intercept_stderr:.6f}")
intercept (95%): 0.616950 +/- 0.544475
```

westpa.core.progress.**nop**()

class westpa.core.progress.**ProgressIndicator**(*stream=None*, *interval=1*)

Bases: object

draw_fancy()

draw_simple()

draw()

clear()

property operation

property extent

property progress

new_operation(*operation*, *extent=None*, *progress=0*)

start()

stop()

6.2.6.6 westpa.core.segment module

class westpa.core.segment.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

6.2.6.7 westpa.core.sim_manager module

class westpa.core.sim_manager.timedelta

Bases: object

Difference between two datetime values.

timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

days

Number of days.

max = datetime.timedelta(days=999999999, seconds=86399, microseconds=999999)

microseconds

Number of microseconds (>= 0 and less than 1 second).

min = datetime.timedelta(days=-999999999)

resolution = datetime.timedelta(microseconds=1)

seconds

Number of seconds (>= 0 and less than 1 day).

total_seconds()

Total seconds in the duration.

class westpa.core.sim_manager.**zip_longest**

Bases: object

zip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> zip_longest object

Return a zip_longest object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration. When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

exception westpa.core.sim_manager.**PickleError**

Bases: Exception

westpa.core.sim_manager.**weight_dtype**

alias of float64

class westpa.core.sim_manager.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.core.sim_manager.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).

- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

westpa.core.sim_manager.**grouper**(*n*, *iterable*, *fillvalue=None*)

Collect data into fixed-length chunks or blocks

exception westpa.core.sim_manager.**PropagationError**

Bases: RuntimeError

class westpa.core.sim_manager.**WESimManager**(*rc=None*)

Bases: object

process_config()

register_callback(*hook*, *function*, *priority=0*)

Registers a callback to execute during the given hook into the simulation loop. The optional priority is used to order when the function is called relative to other registered callbacks.

invoke_callbacks(*hook*, **args*, ***kwargs*)

load_plugins(*plugins=None*)

report_bin_statistics(*bins*, *target_states*, *save_summary=False*)

get_bstate_pcoords(*basis_states*, *label='basis'*)

For each of the given `basis_states`, calculate progress coordinate values as necessary. The HDF5 file is not updated.

report_basis_states(*basis_states*, *label='basis'*)

report_target_states(*target_states*)

initialize_simulation(*basis_states*, *target_states*, *start_states*, *segs_per_state=1*, *suppress_we=False*)

Initialize a new weighted ensemble simulation, taking `segs_per_state` initial states from each of the given `basis_states`.

`w_init` is the forward-facing version of this function

prepare_iteration()

finalize_iteration()

Clean up after an iteration and prepare for the next.

get_istate_futures()

Add `n_states` initial states to the internal list of initial states assigned to recycled particles. Spare states are used if available, otherwise new states are created. If created new initial states requires generation, then a set of futures is returned representing work manager tasks corresponding to the necessary generation work.

propagate()

save_bin_data()

Calculate and write flux and transition count matrices to HDF5. Population and rate matrices are likely useless at the single-tau level and are no longer written.

check_propagation()

Check for failures in propagation or initial state generation, and raise an exception if any are found.

run_we()

Run the weighted ensemble algorithm based on the binning in self.final_bins and the recycled particles in self.to_recycle, creating and committing the next iteration's segments to storage as well.

prepare_new_iteration()

Commit data for the coming iteration to the HDF5 file.

run()

prepare_run()

Prepare a new run.

finalize_run()

Perform cleanup at the normal end of a run

pre_propagation()

post_propagation()

pre_we()

post_we()

6.2.6.8 westpa.core.states module

class westpa.core.states.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.core.states.**BasisState**(*label*, *probability*, *pcoord=None*, *auxref=None*, *state_id=None*)

Bases: object

Describes an basis (micro)state. These basis states are used to generate initial states for new trajectories, either at the beginning of the simulation (i.e. at w_init) or due to recycling.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **probability** – Probability of this state to be selected when creating a new trajectory.
- **pcoord** – The representative progress coordinate of this state.
- **auxref** – A user-provided (string) reference for locating data associated with this state (usually a filesystem path).

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*)

Read a file defining basis states. Each line 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
```

as_numpy_record()

Return the data for this state as a numpy record array.

class westpa.core.states.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.
- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).

- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

class westpa.core.states.**TargetState**(*label*, *pcoord*, *state_id=None*)

Bases: object

Describes a target state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **label** – A descriptive label for this microstate (may be empty)
- **pcoord** – The representative progress coordinate of this state.

classmethod states_to_file(*states*, *fileobj*)

Write a file defining basis states, which may then be read by *states_from_file()*.

classmethod states_from_file(*statefile*, *dtype*)

Read a file defining target states. 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.

westpa.core.states.**pare_basis_initial_states**(*basis_states*, *initial_states*, *segments=None*)

Given iterables of basis and initial states (and optionally segments that use them), return minimal sets (as in __builtins__.set) of states needed to describe the history of the given segments an initial states.

westpa.core.states.**return_state_type**(*state_obj*)

Convinience function for returning the state ID and type of the state_obj pointer

6.2.6.9 westpa.core.systems module

class westpa.core.systems.**NopMapper**

Bases: *BinMapper*

Put everything into one bin.

assign(*coords*, *mask=None*, *output=None*)

class westpa.core.systems.**WESTSystem**(*rc=None*)

Bases: object

A description of the system being simulated, including the dimensionality and data type of the progress coordinate, the number of progress coordinate entries expected from each segment, and binning. To construct a simulation, the user must subclass WESTSystem and set several instance variables.

At a minimum, the user must subclass WESTSystem and override **:method:`initialize`** to set the data type and dimensionality of progress coordinate data and define a bin mapper.

Variables

- **pcoord_ndim** – The number of dimensions in the progress coordinate. Defaults to 1 (i.e. a one-dimensional progress coordinate).
- **pcoord_dtype** – The data type of the progress coordinate, which must be callable (e.g. np.float32 and long will work, but '<f4' and '<i8' will not). Defaults to np.float64.
- **pcoord_len** – The length of the progress coordinate time series generated by each segment, including *both* the initial and final values. Defaults to 2 (i.e. only the initial and final progress coordinate values for a segment are returned from propagation).
- **bin_mapper** – A bin mapper describing the progress coordinate space.
- **bin_target_counts** – A vector of target counts, one per bin.

property bin_target_counts

initialize()

Prepare this system object for use in simulation or analysis, creating a bin space, setting replicas per bin, and so on. This function is called whenever a WEST tool creates an instance of the system driver.

prepare_run()

Prepare this system for use in a simulation run. Called by w_run in all worker processes.

finalize_run()

A hook for system-specific processing for the end of a simulation run (as defined by such things as maximum wallclock time, rather than perhaps more scientifically-significant definitions of "the end of a simulation run")

new_pcoord_array(*pcoord_len=None*)

Return an appropriately-sized and -typed pcoord array for a timepoint, segment, or number of segments. If `pcoord_len` is not specified (or None), then a length appropriate for a segment is returned.

new_region_set()

6.2.6.10 westpa.core.textio module

Miscellaneous routines to help with input and output of WEST-related data in text format

class westpa.core.textio.**NumericTextOutputFormatter**(*output_file*, *mode='wt'*, *emit_header=None*)

Bases: `object`

comment_string = '# '

emit_header = True

close()

write(*str*)

writelines(*sequence*)

write_comment(*line*)

Writes a line beginning with the comment string

write_header(*line*)

Appends a line to those written when the file header is written. The appropriate comment string will be prepended, so `line` should not include a comment character.

6.2.6.11 westpa.core.we_driver module

class westpa.core.we_driver.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: `object`

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.core.we_driver.**InitialState**(*state_id*, *basis_state_id*, *iter_created*, *iter_used=None*, *istate_type=None*, *istate_status=None*, *pcoord=None*, *basis_state=None*, *basis_auxref=None*)

Bases: object

Describes an initial state for a new trajectory. These are generally constructed by appropriate modification of a basis state.

Variables

- **state_id** – Integer identifier of this state, usually set by the data manager.
- **basis_state_id** – Identifier of the basis state from which this state was generated, or None.
- **basis_state** – The *BasisState* from which this state was generated, or None.

- **iter_created** – Iteration in which this state was generated (0 for simulation initialization).
- **iter_used** – Iteration in which this state was used to initiate a trajectory (None for unused).
- **istate_type** – Integer describing the type of this initial state (ISTATE_TYPE_BASIS for direct use of a basis state, ISTATE_TYPE_GENERATED for a state generated from a basis state, ISTATE_TYPE_RESTART for a state corresponding to the endpoint of a segment in another simulation, or ISTATE_TYPE_START for a state generated from a start state).
- **istate_status** – Integer describing whether this initial state has been properly prepared.
- **pcoord** – The representative progress coordinate of this state.

ISTATE_TYPE_UNSET = 0

ISTATE_TYPE_BASIS = 1

ISTATE_TYPE_GENERATED = 2

ISTATE_TYPE_RESTART = 3

ISTATE_TYPE_START = 4

ISTATE_UNUSED = 0

ISTATE_STATUS_PENDING = 0

ISTATE_STATUS_PREPARED = 1

ISTATE_STATUS_FAILED = 2

istate_types = {'ISTATE_TYPE_BASIS': 1, 'ISTATE_TYPE_GENERATED': 2, 'ISTATE_TYPE_RESTART': 3, 'ISTATE_TYPE_START': 4, 'ISTATE_TYPE_UNSET': 0}

istate_type_names = {0: 'ISTATE_TYPE_UNSET', 1: 'ISTATE_TYPE_BASIS', 2: 'ISTATE_TYPE_GENERATED', 3: 'ISTATE_TYPE_RESTART', 4: 'ISTATE_TYPE_START'}

istate_statuses = {'ISTATE_STATUS_FAILED': 2, 'ISTATE_STATUS_PENDING': 0, 'ISTATE_STATUS_PREPARED': 1}

istate_status_names = {0: 'ISTATE_STATUS_PENDING', 1: 'ISTATE_STATUS_PREPARED', 2: 'ISTATE_STATUS_FAILED'}

as_numpy_record()

exception westpa.core.we_driver.**ConsistencyError**

Bases: RuntimeError

exception westpa.core.we_driver.**AccuracyError**

Bases: RuntimeError

class westpa.core.we_driver.**NewWeightEntry**(*source_type*, *weight*, *prev_seg_id=None*, *prev_init_pcoord=None*, *prev_final_pcoord=None*, *new_init_pcoord=None*, *target_state_id=None*, *initial_state_id=None*)

Bases: object

NW_SOURCE_RECYCLED = 0

class westpa.core.we_driver.**WEDriver**(*rc=None*, *system=None*)

Bases: object

A class implemented Huber & Kim's weighted ensemble algorithm over Segment objects. This class handles all binning, recycling, and preparation of new Segment objects for the next iteration. Binning is accomplished using system.bin_mapper, and per-bin target counts are from system.bin_target_counts.

The workflow is as follows:

1) Call *new_iteration()* every new iteration, providing any recycling targets that are in force and any available initial states for recycling.
2) Call *assign()* to assign segments to bins based on their initial and end points. This returns the number of walkers that were recycled.
3) Call *run_we()*, optionally providing a set of initial states that will be used to recycle walkers.

Note the presence of flux_matrix, transition_matrix, current_iter_segments, next_iter_segments, recycling_segments, initial_binning, final_binning, next_iter_binning, and new_weights (to be documented soon).

weight_split_threshold = 2.0

weight_merge_cutoff = 1.0

largest_allowed_weight = 1.0

smallest_allowed_weight = 1e-310

process_config()

property next_iter_segments

Newly-created segments for the next iteration

property current_iter_segments

Segments for the current iteration

property next_iter_assignments

Bin assignments (indices) for initial points of next iteration.

property current_iter_assignments

Bin assignments (indices) for endpoints of current iteration.

property recycling_segments

Segments designated for recycling

property n_recycled_segs

Number of segments recycled this iteration

property n_istates_needed

Number of initial states needed to support recycling for this iteration

check_threshold_configs()

Check to see if weight thresholds parameters are valid

clear()

Explicitly delete all Segment-related state.

new_iteration(*initial_states=None*, *target_states=None*, *new_weights=None*, *bin_mapper=None*, *bin_target_counts=None*)

Prepare for a new iteration. `initial_states` is a sequence of all InitialState objects valid for use in to generating new segments for the *next* iteration (after the one being begun with the call to new_iteration); that is, these are states available to recycle to. Target states which generate recycling events are specified in `target_states`, a sequence of TargetState objects. Both `initial_states` and `target_states` may be empty as required.

The optional `new_weights` is a sequence of NewWeightEntry objects which will be used to construct the initial flux matrix.

The given `bin_mapper` will be used for assignment, and `bin_target_counts` used for splitting/merging target counts; each will be obtained from the system object if omitted or None.

add_initial_states(*initial_states*)

Add newly-prepared initial states to the pool available for recycling.

property all_initial_states

Return an iterator over all initial states (available or used)

assign(*segments*, *initializing=False*)

Assign segments to initial and final bins, and update the (internal) lists of used and available initial states. If `initializing` is True, then the "final" bin assignments will be identical to the initial bin assignments, a condition required for seeding a new iteration from pre-existing segments.

populate_initial(*initial_states*, *weights*, *system=None*)

Create walkers for a new weighted ensemble simulation.

One segment is created for each provided initial state, then binned and split/merged as necessary. After this function is called, next_iter_segments will yield the new segments to create, used_initial_states will contain data about which of the provided initial states were used, and avail_initial_states will contain data about which initial states were unused (because their corresponding walkers were merged out of existence).

rebin_current(*parent_segments*)

Reconstruct walkers for the current iteration based on (presumably) new binning. The previous iteration's segments must be provided (as `parent_segments`) in order to update endpoint types appropriately.

construct_next()

Construct walkers for the next iteration, by running weighted ensemble recycling and bin/split/merge on the segments previously assigned to bins using `assign`. Enough unused initial states must be present in `self.avail_initial_states` for every recycled walker to be assigned an initial state.

After this function completes, `self.flux_matrix` contains a valid flux matrix for this iteration (including any contributions from recycling from the previous iteration), and `self.next_iter_segments` contains a list of segments ready for the next iteration, with appropriate values set for weight, endpoint type, parent walkers, and so on.

6.2.6.12 westpa.core.wm_ops module

westpa.core.wm_ops.**get_pcoord**(*state*)

westpa.core.wm_ops.**gen_istate**(*basis_state*, *initial_state*)

westpa.core.wm_ops.**prep_iter**(*n_iter*, *segments*)

westpa.core.wm_ops.**post_iter**(*n_iter*, *segments*)

westpa.core.wm_ops.**propagate**(*basis_states*, *initial_states*, *segments*)

6.2.6.13 westpa.core.yamlcfg module

YAML-based configuration files for WESTPA

westpa.core.yamlcfg.**YLoader**

alias of CLoader

class westpa.core.yamlcfg.**NopMapper**

Bases: *BinMapper*

Put everything into one bin.

assign(*coords*, *mask=None*, *output=None*)

exception westpa.core.yamlcfg.**ConfigValueWarning**

Bases: UserWarning

westpa.core.yamlcfg.**warn_dubious_config_entry**(*entry*, *value*, *expected_type=None*, *category=<class 'westpa.core.yamlcfg.ConfigValueWarning'>*, *stacklevel=1*)

westpa.core.yamlcfg.**check_bool**(*value*, *action='warn'*)

Check that the given value is boolean in type. If not, either raise a warning (if action=='warn') or an exception (action=='raise').

exception westpa.core.yamlcfg.**ConfigItemMissing**(*key*, *message=None*)

Bases: KeyError

exception westpa.core.yamlcfg.**ConfigItemTypeError**(*key*, *expected_type*, *message=None*)

Bases: TypeError

exception westpa.core.yamlcfg.**ConfigValueError**(*key*, *value*, *message=None*)

Bases: ValueError

class westpa.core.yamlcfg.**YAMLConfig**

Bases: object

preload_config_files = ['/etc/westpa/westrc', '/home/docs/.westrc']

update_from_file(*file*, *required=True*)

require(*key*, *type_=None*)

Ensure that a configuration item with the given key is present. If the optional type_ is given, additionally require that the item has that type.

require_type_if_present(*key*, *type_*)

Ensure that the configuration item with the given key has the given type.

coerce_type_if_present(*key*, *type_*)

get(*key*, *default=None*)

get_typed(*key*, *type_*, *default=<object object>*)

get_path(*key*, *default=<object object>*, *expandvars=True*, *expanduser=True*, *realpath=True*, *abspath=True*)

get_pathlist(*key*, *default=<object object>*, *sep=':'*, *expandvars=True*, *expanduser=True*, *realpath=True*, *abspath=True*)

get_python_object(*key*, *default=<object object>*, *path=None*)

get_choice(*key*, *choices*, *default=<object object>*, *value_transform=None*)

class westpa.core.yamlcfg.**YAMLSystem**(*rc=None*)

Bases: object

A description of the system being simulated, including the dimensionality and data type of the progress coordinate, the number of progress coordinate entries expected from each segment, and binning. To construct a simulation, the user must subclass WESTSystem and set several instance variables.

At a minimum, the user must subclass WESTSystem and override **:method:`initialize`** to set the data type and dimensionality of progress coordinate data and define a bin mapper.

Variables

- **pcoord_ndim** – The number of dimensions in the progress coordinate. Defaults to 1 (i.e. a one-dimensional progress coordinate).
- **pcoord_dtype** – The data type of the progress coordinate, which must be callable (e.g. np.float32 and long will work, but '<f4' and '<i8' will not). Defaults to np.float64.
- **pcoord_len** – The length of the progress coordinate time series generated by each segment, including *both* the initial and final values. Defaults to 2 (i.e. only the initial and final progress coordinate values for a segment are returned from propagation).
- **bin_mapper** – A bin mapper describing the progress coordinate space.
- **bin_target_counts** – A vector of target counts, one per bin.

property bin_target_counts

initialize()

Prepare this system object for use in simulation or analysis, creating a bin space, setting replicas per bin, and so on. This function is called whenever a WEST tool creates an instance of the system driver.

prepare_run()

Prepare this system for use in a simulation run. Called by w_run in all worker processes.

finalize_run()

A hook for system-specific processing for the end of a simulation run (as defined by such things as maximum wallclock time, rather than perhaps more scientifically-significant definitions of "the end of a simulation run")

new_pcoord_array(*pcoord_len=None*)

Return an appropriately-sized and -typed pcoord array for a timepoint, segment, or number of segments. If pcoord_len is not specified (or None), then a length appropriate for a segment is returned.

new_region_set()

6.3 westpa.work_managers package

6.3.1 westpa.work_managers package

6.3.1.1 westpa.work_managers module

A system for parallel, remote execution of multiple arbitrary tasks. Much of this, both in concept and execution, was inspired by (and in some cases based heavily on) the `concurrent.futures` package from Python 3.2, with some simplifications and adaptations (thanks to Brian Quinlan and his futures implementation).

class `westpa.work_managers.`**SerialWorkManager**

Bases: *WorkManager*

classmethod from_environ(*wmenv=None*)

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

class `westpa.work_managers.`**ThreadsWorkManager**(*n_workers=None*)

Bases: *WorkManager*

A work manager using threads.

classmethod from_environ(*wmenv=None*)

runtask(*task_queue*)

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

class `westpa.work_managers.`**ProcessWorkManager**(*n_workers=None*, *shutdown_timeout=1*)

Bases: *WorkManager*

A work manager using the `multiprocessing` module.

Notes

On MacOS, as of Python 3.8 the default start method for multiprocessing launching new processes was changed from fork to spawn. In general, spawn is more robust and efficient, however it requires serializability of everything being passed to the child process. In contrast, fork is much less memory efficient, as it makes a full copy of everything in the parent process. However, it does not require picklability.

So, on MacOS, the method for launching new processes is explicitly changed to fork from the (MacOS-specific) default of spawn. Unix should default to fork.

See https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods and https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods for more details.

classmethod from_environ(*wmenv=None*)

task_loop()

results_loop()

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

`westpa.work_managers.`**make_work_manager**()

Using cues from the environment, instantiate a pre-configured work manager.

6.3.1.2 westpa.work_managers.core module

class `westpa.work_managers.core.`**islice**

Bases: `object`

islice(iterable, stop) –> islice object islice(iterable, start, stop[, step]) –> islice object

Return an iterator whose next() method returns selected values from an iterable. If start is specified, will skip all preceding elements; otherwise, start defaults to zero. Step defaults to one. If specified as another value, step determines how many values are skipped between successive calls. Works like a slice() on a list but returns an iterator.

`westpa.work_managers.core.`**contextmanager**(*func*)

@contextmanager decorator.

Typical usage:

@contextmanager def some_generator(<arguments>):

<setup> try:

yield <value>

finally:

<cleanup>

This makes this:

with some_generator(<arguments>) as <variable>:
<body>

equivalent to this:

<setup> try:
<variable> = <value> <body>

finally:
<cleanup>

class westpa.work_managers.core.**WorkManager**

Bases: object

Base class for all work managers. At a minimum, work managers must provide a submit() function and a n_workers attribute (which may be a property), though most will also override startup() and shutdown().

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. fn(*args,**kwargs) will be executed by a worker, and the return value assigned as the result of the returned future. The function fn and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in tasks should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function fn and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given futures as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of futures as they become available. Futures are generated by the task_generator, which must return a triple of the form expected by submit. The method also accepts an int queue_size that dictates the maximum number of Futures that should be pending at any given time. The default value of None submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given futures and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given `futures` in order. This function returns the same `futures` as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.core.**FutureWatcher**(*futures*, *threshold=1*)

Bases: `object`

A device to wait on multiple results and/or exceptions with only one lock.

signal(*future*)

Signal this watcher that the given future has results available. If this brings the number of available futures above signal_threshold, this watcher's event object will be signalled as well.

wait()

Wait on one or more futures.

reset()

Reset this watcher's list of completed futures, returning the list of completed futures prior to resetting it.

add(*futures*)

Add watchers to all futures in the iterable of futures.

class westpa.work_managers.core.**WMFuture**(*task_id=None*)

Bases: `object`

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given `futures`. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If `discard` is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

6.3.1.3 westpa.work_managers.environment module

Routines for configuring the work manager environment

class westpa.work_managers.environment.**WMEnvironment**(*use_arg_prefixes=False*, *valid_work_managers=None*)

Bases: object

A class to encapsulate the environment in which work managers are instantiated; this controls how environment variables and command-line arguments are used to set up work managers. This could be used to cleanly instantiate two work managers within one application, but is really more about providing facilities to make it easier for individual work managers to configure themselves according to precendence of configuration information:

1. command-line arguments
2. environment variables
3. defaults

group_title = 'parallelization options'

group_description = None

env_prefix = 'WM'

arg_prefix = 'wm'

default_work_manager = 'serial'

default_parallel_work_manager = 'processes'

valid_work_managers = ['serial', 'threads', 'processes', 'zmq', 'mpi']

env_name(*name*)

arg_name(*name*)

arg_flag(*name*)

get_val(*name*, *default=None*, *type_=None*)

add_wm_args(*parser*)

process_wm_args(*args*)

make_work_manager()

Using cues from the environment, instantiate a pre-configured work manager.

westpa.work_managers.environment.**make_work_manager**()

Using cues from the environment, instantiate a pre-configured work manager.

westpa.work_managers.environment.**add_wm_args**(*parser*)

westpa.work_managers.environment.**process_wm_args**(*args*)

6.3.1.4 westpa.work_managers.mpi module

A work manager which uses MPI to distribute tasks and collect results.

class westpa.work_managers.mpi.**deque**

Bases: object

deque([iterable[, maxlen]]) –> deque object

A list-like sequence optimized for data accesses near its endpoints.

append()

Add an element to the right side of the deque.

appendleft()

Add an element to the left side of the deque.

clear()

Remove all elements from the deque.

copy()

Return a shallow copy of a deque.

count()

D.count(value) – return number of occurrences of value

extend()

Extend the right side of the deque with elements from the iterable

extendleft()

Extend the left side of the deque with elements from the iterable

index()

D.index(value, [start, [stop]]) – return first index of value. Raises ValueError if the value is not present.

insert()

D.insert(index, object) – insert object before index

maxlen

maximum size of a deque or None if unbounded

pop()

Remove and return the rightmost element.

popleft()

Remove and return the leftmost element.

remove()

D.remove(value) – remove first occurrence of value.

reverse()

D.reverse() – reverse *IN PLACE*

rotate()

Rotate the deque n steps to the right (default n=1). If n is negative, rotates left.

class westpa.work_managers.mpi.**WorkManager**

Bases: object

Base class for all work managers. At a minimum, work managers must provide a submit() function and a n_workers attribute (which may be a property), though most will also override startup() and shutdown().

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. fn(*args,**kwargs) will be executed by a worker, and the return value assigned as the result of the returned future. The function fn and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in tasks should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function fn and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given futures as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of futures as they become available. Futures are generated by the task_generator, which must return a triple of the form expected by submit. The method also accepts an int queue_size that dictates the maximum number of Futures that should be pending at any given time. The default value of None submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given futures and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given futures in order. This function returns the same futures as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.mpi.**WMFuture**(*task_id=None*)

Bases: object

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given futures. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If discard is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

class westpa.work_managers.mpi.**Task**(*task_id*, *fn*, *args*, *kwargs*)

Bases: object

Tasks are tuples of (task_id, function, args, keyword args)

class westpa.work_managers.mpi.**MPIWorkManager**

Bases: *WorkManager*

MPIWorkManager factory.

Initialize info shared by Manager and Worker classes.

classmethod from_environ(*wmenv=None*)

submit(*fn*, *args=None*, *kwargs=None*)

Adhere to WorkManager interface. This method should never be called.

class westpa.work_managers.mpi.**Serial**

Bases: *MPIWorkManager*

Replication of the serial work manager. This is a fallback for MPI runs that request only 1 (size=1) processor.

Initialize info shared by Manager and Worker classes.

submit(*fn*, *args=None*, *kwargs=None*)

Adhere to WorkManager interface. This method should never be called.

class westpa.work_managers.mpi.**Manager**

Bases: *MPIWorkManager*

Manager of the MPIWorkManage. Distributes tasks to Worker as they are received from the sim_manager. In addition to the main thread, this class spawns two threads, a receiver and a dispatcher.

Initialize different state variables used by Manager.

startup()

Spawns the dispatcher and receiver threads.

submit(*fn*, *args=None*, *kwargs=None*)

Receive task from simulation manager and add it to pending_futures.

shutdown()

Send shutdown tag to all worker processes, and set the shutdown sentinel to stop the receiver and dispatcher loops.

class westpa.work_managers.mpi.**Worker**

Bases: *MPIWorkManager*

Client class for executing tasks as distributed by the Manager in the MPI Work Manager

Initialize info shared by Manager and Worker classes.

startup()

Clock the worker in for work.

clockIn()

Do each task as it comes in. The completion of a task is notice to the manager that more work is welcome.

property is_master

Worker processes need to be marked as not manager. This ensures that the proper branching is followed in w_run.py.

6.3.1.5 westpa.work_managers.processes module

exception westpa.work_managers.processes.**Empty**

Bases: `Exception`

Exception raised by Queue.get(block=0)/get_nowait().

class westpa.work_managers.processes.**WorkManager**

Bases: `object`

Base class for all work managers. At a minimum, work managers must provide a `submit()` function and a `n_workers` attribute (which may be a property), though most will also override `startup()` and `shutdown()`.

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given `futures` as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of `futures` as they become available. Futures are generated by the `task_generator`, which must return a triple of the form expected by `submit`. The method also accepts an int `queue_size` that dictates the maximum number of Futures that should be pending at any given time. The default value of `None` submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given `futures` and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given `futures` in order. This function returns the same `futures` as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.processes.**WMFuture**(*task_id=None*)

Bases: `object`

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given `futures`. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If `discard` is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

class westpa.work_managers.processes.**ProcessWorkManager**(*n_workers=None*, *shutdown_timeout=1*)

Bases: *WorkManager*

A work manager using the `multiprocessing` module.

Notes

On MacOS, as of Python 3.8 the default start method for multiprocessing launching new processes was changed from fork to spawn. In general, spawn is more robust and efficient, however it requires serializability of everything being passed to the child process. In contrast, fork is much less memory efficient, as it makes a full copy of everything in the parent process. However, it does not require picklability.

So, on MacOS, the method for launching new processes is explicitly changed to fork from the (MacOS-specific) default of spawn. Unix should default to fork.

See https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods and https://docs.python.org/3/library/multiprocessing.html#the-spawn-and-forkserver-start-methods for more details.

classmethod from_environ(*wmenv=None*)

task_loop()

results_loop()

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

6.3.1.6 westpa.work_managers.serial module

class westpa.work_managers.serial.**WorkManager**

Bases: object

Base class for all work managers. At a minimum, work managers must provide a `submit()` function and a `n_workers` attribute (which may be a property), though most will also override `startup()` and `shutdown()`.

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given `futures` as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of `futures` as they become available. Futures are generated by the `task_generator`, which must return a triple of the form expected by `submit`. The method also accepts an int `queue_size` that dictates the maximum number of Futures that should be pending at any given time. The default value of `None` submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given `futures` and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given `futures` in order. This function returns the same `futures` as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.serial.**WMFuture**(*task_id=None*)

Bases: object

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given futures. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If discard is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

class westpa.work_managers.serial.**SerialWorkManager**

Bases: *WorkManager*

classmethod from_environ(*wmenv=None*)

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. fn(*args,**kwargs) will be executed by a worker, and the return value assigned as the result of the returned future. The function fn and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

6.3.1.7 westpa.work_managers.threads module

class westpa.work_managers.threads.**WorkManager**

Bases: object

Base class for all work managers. At a minimum, work managers must provide a `submit()` function and a `n_workers` attribute (which may be a property), though most will also override `startup()` and `shutdown()`.

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given `futures` as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of `futures` as they become available. Futures are generated by the `task_generator`, which must return a triple of the form expected by `submit`. The method also accepts an int `queue_size` that dictates the maximum number of Futures that should be pending at any given time. The default value of `None` submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given `futures` and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given `futures` in order. This function returns the same `futures` as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.threads.**WMFuture**(*task_id=None*)

Bases: object

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given futures. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If discard is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

class westpa.work_managers.threads.**Task**(*fn*, *args*, *kwargs*, *future*)

Bases: object

run()

class westpa.work_managers.threads.**ThreadsWorkManager**(*n_workers=None*)

Bases: *WorkManager*

A work manager using threads.

classmethod from_environ(*wmenv=None*)

runtask(*task_queue*)

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. fn(*args,**kwargs) will be executed by a worker, and the return value assigned as the result of the returned future. The function fn and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

6.3.2 westpa.work_managers.zeromq package

6.3.2.1 westpa.work_managers.zeromq module

exception westpa.work_managers.zeromq.**ZMQWMError**

Bases: RuntimeError

Base class for errors related to the ZeroMQ work manager itself

exception westpa.work_managers.zeromq.**ZMQWMTimeout**

Bases: *ZMQWMEnvironmentError*

A timeout of a sort that indicatess that a master or worker has failed or never started.

exception westpa.work_managers.zeromq.**ZMQWMEnvironmentError**

Bases: *ZMQWMError*

Class representing an error in the environment in which the ZeroMQ work manager is running. This includes such things as master/worker ID mismatches.

exception westpa.work_managers.zeromq.**ZMQWorkerMissing**

Bases: *ZMQWMError*

Exception representing that a worker processing a task died or disappeared

class westpa.work_managers.zeromq.**ZMQCore**

Bases: object

PROTOCOL_MAJOR = 3

PROTOCOL_MINOR = 0

PROTOCOL_UPDATE = 0

PROTOCOL_VERSION = (3, 0, 0)

internal_transport = 'ipc'

default_comm_mode = 'ipc'

default_master_heartbeat = 20.0

default_worker_heartbeat = 20.0

default_timeout_factor = 5.0

default_startup_timeout = 120.0

default_shutdown_timeout = 5.0

classmethod make_ipc_endpoint()

classmethod remove_ipc_endpoints()

classmethod make_tcp_endpoint(*address='127.0.0.1'*)

classmethod make_internal_endpoint()

get_identification()

validate_message(*message*)

Validate incoming message. Raises an exception if the message is improperly formatted (TypeError) or does not correspond to the appropriate master (ZMQWMEnvironmentError).

message_validation(*msg*)

A context manager for message validation. The instance variable `validation_fail_action` controls the behavior of this context manager:

- 'raise': re-raise the exception that indicated failed validation. Useful for development.
- 'exit' (default): report the error and exit the program.
- 'warn': report the error and continue.

recv_message(*socket*, *flags=0*, *validate=True*, *timeout=None*)

Receive a message object from the given socket, using the given flags. Message validation is performed if `validate` is true. If `timeout` is given, then it is the number of milliseconds to wait prior to raising a ZMQWMTimeout exception. `timeout` is ignored if `flags` includes `zmq.NOBLOCK`.

recv_all(*socket*, *flags=0*, *validate=True*)

Receive all messages currently available from the given socket.

recv_ack(*socket*, *flags=0*, *validate=True*, *timeout=None*)

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

send_reply(*socket*, *original_message*, *reply='ok'*, *payload=None*, *flags=0*)

Send a reply to `original_message` on `socket`. The reply message is a Message object or a message identifier. The reply master_id and worker_id are set from `original_message`, unless master_id is not set, in which case it is set from self.master_id.

send_ack(*socket*, *original_message*)

Send an acknowledgement message, which is mostly just to respect REQ/REP recv/send patterns.

send_nak(*socket*, *original_message*)

Send a negative acknowledgement message.

send_inproc_message(*message*, *payload=None*, *flags=0*)

signal_shutdown()

shutdown_handler(*signal=None*, *frame=None*)

install_signal_handlers(*signals=None*)

install_sigint_handler()

startup()

shutdown()

join()

class westpa.work_managers.zeromq.**ZMQNode**(*upstream_rr_endpoint*, *upstream_ann_endpoint*, *n_local_workers=None*)

Bases: *ZMQCore*, *IsNode*

run()

property is_master

comm_loop()

startup()

class westpa.work_managers.zeromq.**ZMQWorker**(*rr_endpoint*, *ann_endpoint*)

Bases: *ZMQCore*

This is the outward facing worker component of the ZMQ work manager. This forms the interface to the master. This process cannot hang or crash due to an error in tasks it executes, so tasks are isolated in ZMQExecutor, which communicates with ZMQWorker via (what else?) ZeroMQ.

property is_master

update_master_info(*msg*)

identify(*rr_socket*)

request_task(*rr_socket*, *task_socket*)

handle_reconfigure_timeout(*msg*, *timers*)

handle_result(*result_socket*, *rr_socket*)

comm_loop()

Master communication loop for the worker process.

shutdown_executor()

install_signal_handlers(*signals=None*)

startup(*process_index=None*)

class westpa.work_managers.zeromq.**ZMQWorkManager**(*n_local_workers=1*)

Bases: *ZMQCore*, *WorkManager*, *IsNode*

classmethod add_wm_args(*parser*, *wmenv=None*)

classmethod from_environ(*wmenv=None*)

classmethod read_host_info(*filename*)

classmethod canonicalize_endpoint(*endpoint*, *allow_wildcard_host=True*)

property n_workers

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

handle_result(*socket*, *msg*)

handle_task_request(*socket*, *msg*)

update_worker_information(*msg*)

check_workers()

remove_worker(*worker_id*)

shutdown_clear_tasks()

Abort pending tasks with error on shutdown.

comm_loop()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

6.3.2.2 westpa.work_managers.zeromq.core module

Created on May 29, 2015

@author: mzwier

`westpa.work_managers.zeromq.core.`**randport**(*address='127.0.0.1'*)

Select a random unused TCP port number on the given address.

exception `westpa.work_managers.zeromq.core.`**ZMQWMError**

Bases: `RuntimeError`

Base class for errors related to the ZeroMQ work manager itself

exception `westpa.work_managers.zeromq.core.`**ZMQWorkerMissing**

Bases: *ZMQWMError*

Exception representing that a worker processing a task died or disappeared

exception `westpa.work_managers.zeromq.core.`**ZMQWMEnvironmentError**

Bases: *ZMQWMError*

Class representing an error in the environment in which the ZeroMQ work manager is running. This includes such things as master/worker ID mismatches.

exception westpa.work_managers.zeromq.core.**ZMQWMTimeout**

Bases: *ZMQWMEnvironmentError*

A timeout of a sort that indicatess that a master or worker has failed or never started.

class westpa.work_managers.zeromq.core.**Message**(*message=None*, *payload=None*, *master_id=None*, *src_id=None*)

Bases: object

SHUTDOWN = 'shutdown'

ACK = 'ok'

NAK = 'no'

IDENTIFY = 'identify'

TASKS_AVAILABLE = 'tasks_available'

TASK_REQUEST = 'task_request'

MASTER_BEACON = 'master_alive'

RECONFIGURE_TIMEOUT = 'reconfigure_timeout'

TASK = 'task'

RESULT = 'result'

idempotent_announcement_messages = {'master_alive', 'shutdown', 'tasks_available'}

classmethod coalesce_announcements(*messages*)

class westpa.work_managers.zeromq.core.**Task**(*fn*, *args*, *kwargs*, *task_id=None*)

Bases: object

execute()

Run this task, returning a Result object.

class westpa.work_managers.zeromq.core.**Result**(*task_id*, *result=None*, *exception=None*, *traceback=None*)

Bases: object

class westpa.work_managers.zeromq.core.**PassiveTimer**(*duration*, *started=None*)

Bases: object

started

duration

property expired

property expires_in

reset(*at=None*)

start(*at=None*)

class westpa.work_managers.zeromq.core.**PassiveMultiTimer**

Bases: object

add_timer(*identifier*, *duration*)

remove_timer(*identifier*)

change_duration(*identifier*, *duration*)

reset(*identifier=None*, *at=None*)

expired(*identifier*, *at=None*)

next_expiration()

next_expiration_in()

which_expired(*at=None*)

class westpa.work_managers.zeromq.core.ZMQCore

Bases: object

PROTOCOL_MAJOR = 3

PROTOCOL_MINOR = 0

PROTOCOL_UPDATE = 0

PROTOCOL_VERSION = (3, 0, 0)

internal_transport = 'ipc'

default_comm_mode = 'ipc'

default_master_heartbeat = 20.0

default_worker_heartbeat = 20.0

default_timeout_factor = 5.0

default_startup_timeout = 120.0

default_shutdown_timeout = 5.0

classmethod make_ipc_endpoint()

classmethod remove_ipc_endpoints()

classmethod make_tcp_endpoint(*address='127.0.0.1'*)

classmethod make_internal_endpoint()

get_identification()

validate_message(*message*)

Validate incoming message. Raises an exception if the message is improperly formatted (TypeError) or does not correspond to the appropriate master (ZMQWMEnvironmentError).

message_validation(*msg*)

A context manager for message validation. The instance variable `validation_fail_action` controls the behavior of this context manager:

- 'raise': re-raise the exception that indicated failed validation. Useful for development.
- 'exit' (default): report the error and exit the program.
- 'warn': report the error and continue.

recv_message(*socket*, *flags=0*, *validate=True*, *timeout=None*)

Receive a message object from the given socket, using the given flags. Message validation is performed if `validate` is true. If `timeout` is given, then it is the number of milliseconds to wait prior to raising a ZMQWMTimeout exception. `timeout` is ignored if `flags` includes `zmq.NOBLOCK`.

recv_all(*socket*, *flags=0*, *validate=True*)

Receive all messages currently available from the given socket.

recv_ack(*socket*, *flags=0*, *validate=True*, *timeout=None*)

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

send_reply(*socket*, *original_message*, *reply='ok'*, *payload=None*, *flags=0*)

Send a reply to `original_message` on `socket`. The reply message is a Message object or a message identifier. The reply master_id and worker_id are set from `original_message`, unless master_id is not set, in which case it is set from self.master_id.

send_ack(*socket*, *original_message*)

Send an acknowledgement message, which is mostly just to respect REQ/REP recv/send patterns.

send_nak(*socket*, *original_message*)

Send a negative acknowledgement message.

send_inproc_message(*message*, *payload=None*, *flags=0*)

signal_shutdown()

shutdown_handler(*signal=None*, *frame=None*)

install_signal_handlers(*signals=None*)

install_sigint_handler()

startup()

shutdown()

join()

westpa.work_managers.zeromq.core.**shutdown_process**(*process*, *timeout=1.0*)

class westpa.work_managers.zeromq.core.**IsNode**(*n_local_workers=None*)

Bases: `object`

write_host_info(*filename=None*)

startup()

shutdown()

6.3.2.3 westpa.work_managers.zeromq.node module

Created on Jun 11, 2015

@author: mzwier

class westpa.work_managers.zeromq.node.**ZMQCore**

Bases: object

PROTOCOL_MAJOR = 3

PROTOCOL_MINOR = 0

PROTOCOL_UPDATE = 0

PROTOCOL_VERSION = (3, 0, 0)

internal_transport = 'ipc'

default_comm_mode = 'ipc'

default_master_heartbeat = 20.0

default_worker_heartbeat = 20.0

default_timeout_factor = 5.0

default_startup_timeout = 120.0

default_shutdown_timeout = 5.0

classmethod make_ipc_endpoint()

classmethod remove_ipc_endpoints()

classmethod make_tcp_endpoint(*address='127.0.0.1'*)

classmethod make_internal_endpoint()

get_identification()

validate_message(*message*)

Validate incoming message. Raises an exception if the message is improperly formatted (TypeError) or does not correspond to the appropriate master (ZMQWMEnvironmentError).

message_validation(*msg*)

A context manager for message validation. The instance variable `validation_fail_action` controls the behavior of this context manager:

- 'raise': re-raise the exception that indicated failed validation. Useful for development.
- 'exit' (default): report the error and exit the program.
- 'warn': report the error and continue.

recv_message(*socket*, *flags=0*, *validate=True*, *timeout=None*)

Receive a message object from the given socket, using the given flags. Message validation is performed if `validate` is true. If `timeout` is given, then it is the number of milliseconds to wait prior to raising a ZMQWMTimeout exception. `timeout` is ignored if `flags` includes `zmq.NOBLOCK`.

recv_all(*socket*, *flags=0*, *validate=True*)

Receive all messages currently available from the given socket.

recv_ack(*socket*, *flags=0*, *validate=True*, *timeout=None*)

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

send_reply(*socket*, *original_message*, *reply='ok'*, *payload=None*, *flags=0*)

Send a reply to `original_message` on `socket`. The reply message is a Message object or a message identifier. The reply master_id and worker_id are set from `original_message`, unless master_id is not set, in which case it is set from self.master_id.

send_ack(*socket*, *original_message*)

Send an acknowledgement message, which is mostly just to respect REQ/REP recv/send patterns.

send_nak(*socket*, *original_message*)

Send a negative acknowledgement message.

send_inproc_message(*message*, *payload=None*, *flags=0*)

signal_shutdown()

shutdown_handler(*signal=None*, *frame=None*)

install_signal_handlers(*signals=None*)

install_sigint_handler()

startup()

shutdown()

join()

class westpa.work_managers.zeromq.node.**Message**(*message=None*, *payload=None*, *master_id=None*, *src_id=None*)

Bases: `object`

SHUTDOWN = 'shutdown'

ACK = 'ok'

NAK = 'no'

IDENTIFY = 'identify'

TASKS_AVAILABLE = 'tasks_available'

TASK_REQUEST = 'task_request'

MASTER_BEACON = 'master_alive'

RECONFIGURE_TIMEOUT = 'reconfigure_timeout'

TASK = 'task'

RESULT = 'result'

idempotent_announcement_messages = {'master_alive', 'shutdown', 'tasks_available'}

classmethod coalesce_announcements(*messages*)

class westpa.work_managers.zeromq.node.**PassiveMultiTimer**

Bases: object

add_timer(*identifier*, *duration*)

remove_timer(*identifier*)

change_duration(*identifier*, *duration*)

reset(*identifier=None*, *at=None*)

expired(*identifier*, *at=None*)

next_expiration()

next_expiration_in()

which_expired(*at=None*)

class westpa.work_managers.zeromq.node.**IsNode**(*n_local_workers=None*)

Bases: object

write_host_info(*filename=None*)

startup()

shutdown()

class westpa.work_managers.zeromq.node.**ThreadProxy**(*in_type*, *out_type*, *mon_type=SocketType.PUB*)

Bases: ProxyBase, ThreadDevice

Proxy in a Thread. See Proxy for more.

class westpa.work_managers.zeromq.node.**ZMQNode**(*upstream_rr_endpoint*, *upstream_ann_endpoint*, *n_local_workers=None*)

Bases: *ZMQCore*, *IsNode*

run()

property is_master

comm_loop()

startup()

6.3.2.4 westpa.work_managers.zeromq.work_manager module

class westpa.work_managers.zeromq.work_manager.**ZMQCore**

Bases: object

PROTOCOL_MAJOR = 3

PROTOCOL_MINOR = 0

PROTOCOL_UPDATE = 0

PROTOCOL_VERSION = (3, 0, 0)

internal_transport = 'ipc'

default_comm_mode = 'ipc'

default_master_heartbeat = 20.0

default_worker_heartbeat = 20.0

default_timeout_factor = 5.0

default_startup_timeout = 120.0

default_shutdown_timeout = 5.0

classmethod make_ipc_endpoint()

classmethod remove_ipc_endpoints()

classmethod make_tcp_endpoint(*address='127.0.0.1'*)

classmethod make_internal_endpoint()

get_identification()

validate_message(*message*)

Validate incoming message. Raises an exception if the message is improperly formatted (TypeError) or does not correspond to the appropriate master (ZMQWMEnvironmentError).

message_validation(*msg*)

A context manager for message validation. The instance variable `validation_fail_action` controls the behavior of this context manager:

- 'raise': re-raise the exception that indicated failed validation. Useful for development.
- 'exit' (default): report the error and exit the program.
- 'warn': report the error and continue.

recv_message(*socket*, *flags=0*, *validate=True*, *timeout=None*)

Receive a message object from the given socket, using the given flags. Message validation is performed if `validate` is true. If `timeout` is given, then it is the number of milliseconds to wait prior to raising a ZMQWMTimeout exception. `timeout` is ignored if `flags` includes `zmq.NOBLOCK`.

recv_all(*socket*, *flags=0*, *validate=True*)

Receive all messages currently available from the given socket.

recv_ack(*socket*, *flags=0*, *validate=True*, *timeout=None*)

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

send_reply(*socket*, *original_message*, *reply='ok'*, *payload=None*, *flags=0*)

Send a reply to `original_message` on `socket`. The reply message is a Message object or a message identifier. The reply master_id and worker_id are set from `original_message`, unless master_id is not set, in which case it is set from self.master_id.

send_ack(*socket*, *original_message*)

Send an acknowledgement message, which is mostly just to respect REQ/REP recv/send patterns.

send_nak(*socket*, *original_message*)

Send a negative acknowledgement message.

send_inproc_message(*message*, *payload=None*, *flags=0*)

signal_shutdown()

shutdown_handler(*signal=None*, *frame=None*)

install_signal_handlers(*signals=None*)

install_sigint_handler()

startup()

shutdown()

join()

class westpa.work_managers.zeromq.work_manager.Message(*message=None*, *payload=None*, *master_id=None*, *src_id=None*)

Bases: object

SHUTDOWN = 'shutdown'

ACK = 'ok'

NAK = 'no'

IDENTIFY = 'identify'

TASKS_AVAILABLE = 'tasks_available'

TASK_REQUEST = 'task_request'

MASTER_BEACON = 'master_alive'

RECONFIGURE_TIMEOUT = 'reconfigure_timeout'

TASK = 'task'

RESULT = 'result'

idempotent_announcement_messages = {'master_alive', 'shutdown', 'tasks_available'}

classmethod coalesce_announcements(*messages*)

class westpa.work_managers.zeromq.work_manager.Task(*fn*, *args*, *kwargs*, *task_id=None*)

Bases: object

execute()

Run this task, returning a Result object.

class westpa.work_managers.zeromq.work_manager.Result(*task_id*, *result=None*, *exception=None*, *traceback=None*)

Bases: object

exception westpa.work_managers.zeromq.work_manager.**ZMQWorkerMissing**

Bases: *ZMQWMError*

Exception representing that a worker processing a task died or disappeared

exception westpa.work_managers.zeromq.work_manager.**ZMQWMEnvironmentError**

Bases: *ZMQWMError*

Class representing an error in the environment in which the ZeroMQ work manager is running. This includes such things as master/worker ID mismatches.

class westpa.work_managers.zeromq.work_manager.**IsNode**(*n_local_workers=None*)

Bases: object

write_host_info(*filename=None*)

startup()

shutdown()

class westpa.work_managers.zeromq.work_manager.**PassiveMultiTimer**

Bases: object

add_timer(*identifier*, *duration*)

remove_timer(*identifier*)

change_duration(*identifier*, *duration*)

reset(*identifier=None*, *at=None*)

expired(*identifier*, *at=None*)

next_expiration()

next_expiration_in()

which_expired(*at=None*)

westpa.work_managers.zeromq.work_manager.**randport**(*address='127.0.0.1'*)

Select a random unused TCP port number on the given address.

class westpa.work_managers.zeromq.work_manager.**ZMQWorker**(*rr_endpoint*, *ann_endpoint*)

Bases: *ZMQCore*

This is the outward facing worker component of the ZMQ work manager. This forms the interface to the master. This process cannot hang or crash due to an error in tasks it executes, so tasks are isolated in ZMQExecutor, which communicates with ZMQWorker via (what else?) ZeroMQ.

property is_master

update_master_info(*msg*)

identify(*rr_socket*)

request_task(*rr_socket*, *task_socket*)

handle_reconfigure_timeout(*msg*, *timers*)

handle_result(*result_socket*, *rr_socket*)

comm_loop()

Master communication loop for the worker process.

shutdown_executor()

install_signal_handlers(*signals=None*)

startup(*process_index=None*)

class westpa.work_managers.zeromq.work_manager.**ZMQNode**(*upstream_rr_endpoint*, *upstream_ann_endpoint*, *n_local_workers=None*)

Bases: *ZMQCore*, *IsNode*

run()

property is_master

comm_loop()

startup()

class westpa.work_managers.zeromq.work_manager.**WorkManager**

Bases: object

Base class for all work managers. At a minimum, work managers must provide a `submit()` function and a `n_workers` attribute (which may be a property), though most will also override `startup()` and `shutdown()`.

classmethod from_environ(*wmenv=None*)

classmethod add_wm_args(*parser*, *wmenv=None*)

sigint_handler(*signum*, *frame*)

install_sigint_handler()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

run()

Run the worker loop (in clients only).

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

as_completed(*futures*)

Return a generator which yields results from the given `futures` as they become available.

submit_as_completed(*task_generator*, *queue_size=None*)

Return a generator which yields results from a set of `futures` as they become available. Futures are generated by the `task_generator`, which must return a triple of the form expected by `submit`. The method also accepts an int `queue_size` that dictates the maximum number of Futures that should be pending at any given time. The default value of `None` submits all of the tasks at once.

wait_any(*futures*)

Wait on any of the given `futures` and return the first one which has a result available. If more than one result is or becomes available simultaneously, any completed future may be returned.

wait_all(*futures*)

A convenience function which waits on all the given `futures` in order. This function returns the same `futures` as submitted to the function as a list, indicating the order in which waits occurred.

property is_master

True if this is the master process for task distribution. This is necessary, e.g., for MPI, where all processes start identically and then must branch depending on rank.

class westpa.work_managers.zeromq.work_manager.**WMFuture**(*task_id=None*)

Bases: `object`

A "future", representing work which has been dispatched for completion asynchronously.

static all_acquired(*futures*)

Context manager to acquire all locks on the given `futures`. Primarily for internal use.

get_result(*discard=True*)

Get the result associated with this future, blocking until it is available. If `discard` is true, then removes the reference to the result contained in this instance, so that a collection of futures need not turn into a cache of all associated results.

property result

wait()

Wait until this future has a result or exception available.

get_exception()

Get the exception associated with this future, blocking until it is available.

property exception

Get the exception associated with this future, blocking until it is available.

get_traceback()

Get the traceback object associated with this future, if any.

property traceback

Get the traceback object associated with this future, if any.

is_done()

Indicates whether this future is done executing (may block if this future is being updated).

property done

Indicates whether this future is done executing (may block if this future is being updated).

class westpa.work_managers.zeromq.work_manager.**deque**

Bases: object

deque([iterable[, maxlen]]) –> deque object

A list-like sequence optimized for data accesses near its endpoints.

append()

Add an element to the right side of the deque.

appendleft()

Add an element to the left side of the deque.

clear()

Remove all elements from the deque.

copy()

Return a shallow copy of a deque.

count()

D.count(value) – return number of occurrences of value

extend()

Extend the right side of the deque with elements from the iterable

extendleft()

Extend the left side of the deque with elements from the iterable

index()

D.index(value, [start, [stop]]) – return first index of value. Raises ValueError if the value is not present.

insert()

D.insert(index, object) – insert object before index

maxlen

maximum size of a deque or None if unbounded

pop()

Remove and return the rightmost element.

popleft()

Remove and return the leftmost element.

remove()

D.remove(value) – remove first occurrence of value.

reverse()

D.reverse() – reverse *IN PLACE*

rotate()

Rotate the deque n steps to the right (default n=1). If n is negative, rotates left.

class westpa.work_managers.zeromq.work_manager.**ZMQWorkManager**(*n_local_workers=1*)

Bases: *ZMQCore*, *WorkManager*, *IsNode*

classmethod add_wm_args(*parser*, *wmenv=None*)

classmethod from_environ(*wmenv=None*)

classmethod read_host_info(*filename*)

classmethod canonicalize_endpoint(*endpoint*, *allow_wildcard_host=True*)

property n_workers

submit(*fn*, *args=None*, *kwargs=None*)

Submit a task to the work manager, returning a *WMFuture* object representing the pending result. `fn(*args,**kwargs)` will be executed by a worker, and the return value assigned as the result of the returned future. The function `fn` and all arguments must be picklable; note particularly that off-path modules (like the system module and any active plugins) are not picklable unless pre-loaded in the worker process (i.e. prior to forking the master).

submit_many(*tasks*)

Submit a set of tasks to the work manager, returning a list of *WMFuture* objects representing pending results. Each entry in `tasks` should be a triple (fn, args, kwargs), which will result in fn(*args, **kwargs) being executed by a worker. The function `fn` and all arguments must be picklable; note particularly that off-path modules are not picklable unless pre-loaded in the worker process.

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

handle_result(*socket*, *msg*)

handle_task_request(*socket*, *msg*)

update_worker_information(*msg*)

check_workers()

remove_worker(*worker_id*)

shutdown_clear_tasks()

Abort pending tasks with error on shutdown.

comm_loop()

startup()

Perform any necessary startup work, such as spawning clients.

shutdown()

Cleanly shut down any active workers.

6.3.2.5 westpa.work_managers.zeromq.worker module

Created on May 29, 2015

@author: mzwier

class westpa.work_managers.zeromq.worker.**ZMQCore**

Bases: `object`

PROTOCOL_MAJOR = 3

PROTOCOL_MINOR = 0

PROTOCOL_UPDATE = 0

PROTOCOL_VERSION = (3, 0, 0)

internal_transport = 'ipc'

default_comm_mode = 'ipc'

default_master_heartbeat = 20.0

default_worker_heartbeat = 20.0

default_timeout_factor = 5.0

default_startup_timeout = 120.0

default_shutdown_timeout = 5.0

classmethod make_ipc_endpoint()

classmethod remove_ipc_endpoints()

classmethod make_tcp_endpoint(*address='127.0.0.1'*)

classmethod make_internal_endpoint()

get_identification()

validate_message(*message*)

Validate incoming message. Raises an exception if the message is improperly formatted (TypeError) or does not correspond to the appropriate master (ZMQWMEnvironmentError).

message_validation(*msg*)

A context manager for message validation. The instance variable `validation_fail_action` controls the behavior of this context manager:

- 'raise': re-raise the exception that indicated failed validation. Useful for development.
- 'exit' (default): report the error and exit the program.
- 'warn': report the error and continue.

recv_message(*socket*, *flags=0*, *validate=True*, *timeout=None*)

Receive a message object from the given socket, using the given flags. Message validation is performed if `validate` is true. If `timeout` is given, then it is the number of milliseconds to wait prior to raising a ZMQWMTimeout exception. `timeout` is ignored if `flags` includes `zmq.NOBLOCK`.

recv_all(*socket*, *flags=0*, *validate=True*)

Receive all messages currently available from the given socket.

recv_ack(*socket*, *flags=0*, *validate=True*, *timeout=None*)

send_message(*socket*, *message*, *payload=None*, *flags=0*)

Send a message object. Subclasses may override this to decorate the message with appropriate IDs, then delegate upward to actually send the message. `message` may either be a pre-constructed `Message` object or a message identifier, in which (latter) case `payload` will become the message payload. `payload` is ignored if `message` is a `Message` object.

send_reply(*socket*, *original_message*, *reply='ok'*, *payload=None*, *flags=0*)

Send a reply to `original_message` on `socket`. The reply message is a Message object or a message identifier. The reply master_id and worker_id are set from `original_message`, unless master_id is not set, in which case it is set from self.master_id.

send_ack(*socket*, *original_message*)

Send an acknowledgement message, which is mostly just to respect REQ/REP recv/send patterns.

send_nak(*socket*, *original_message*)

Send a negative acknowledgement message.

send_inproc_message(*message*, *payload=None*, *flags=0*)

signal_shutdown()

shutdown_handler(*signal=None*, *frame=None*)

install_signal_handlers(*signals=None*)

install_sigint_handler()

startup()

shutdown()

join()

class westpa.work_managers.zeromq.worker.**Message**(*message=None*, *payload=None*, *master_id=None*, *src_id=None*)

Bases: `object`

SHUTDOWN = 'shutdown'

ACK = 'ok'

NAK = 'no'

IDENTIFY = 'identify'

TASKS_AVAILABLE = 'tasks_available'

TASK_REQUEST = 'task_request'

MASTER_BEACON = 'master_alive'

RECONFIGURE_TIMEOUT = 'reconfigure_timeout'

TASK = 'task'

RESULT = 'result'

idempotent_announcement_messages = {'master_alive', 'shutdown', 'tasks_available'}

classmethod coalesce_announcements(*messages*)

exception westpa.work_managers.zeromq.worker.**ZMQWMTimeout**

Bases: *ZMQWMEnvironmentError*

A timeout of a sort that indicatess that a master or worker has failed or never started.

class westpa.work_managers.zeromq.worker.**PassiveMultiTimer**

Bases: object

add_timer(*identifier*, *duration*)

remove_timer(*identifier*)

change_duration(*identifier*, *duration*)

reset(*identifier=None*, *at=None*)

expired(*identifier*, *at=None*)

next_expiration()

next_expiration_in()

which_expired(*at=None*)

class westpa.work_managers.zeromq.worker.**Task**(*fn*, *args*, *kwargs*, *task_id=None*)

Bases: object

execute()

Run this task, returning a Result object.

class westpa.work_managers.zeromq.worker.**Result**(*task_id*, *result=None*, *exception=None*, *traceback=None*)

Bases: object

class westpa.work_managers.zeromq.worker.**ZMQWorker**(*rr_endpoint*, *ann_endpoint*)

Bases: *ZMQCore*

This is the outward facing worker component of the ZMQ work manager. This forms the interface to the master. This process cannot hang or crash due to an error in tasks it executes, so tasks are isolated in ZMQExecutor, which communicates with ZMQWorker via (what else?) ZeroMQ.

property is_master

update_master_info(*msg*)

identify(*rr_socket*)

request_task(*rr_socket*, *task_socket*)

handle_reconfigure_timeout(*msg*, *timers*)

handle_result(*result_socket*, *rr_socket*)

comm_loop()

Master communication loop for the worker process.

shutdown_executor()

install_signal_handlers(*signals=None*)

startup(*process_index=None*)

class westpa.work_managers.zeromq.worker.**ZMQExecutor**(*task_endpoint*, *result_endpoint*)

Bases: *ZMQCore*

The is the component of the ZMQ WM worker that actually executes tasks. This is isolated in a separate process and controlled via ZMQ from the ZMQWorker.

comm_loop()

startup(*process_index=None*)

6.4 westpa.tools package

6.4.1 westpa.tools module

tools – classes for implementing command-line tools for WESTPA

class westpa.tools.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.tools.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.tools.**WESTToolComponent**

Bases: `object`

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

class westpa.tools.**WESTSubcommand**(*parent*)

Bases: *WESTToolComponent*

Base class for command-line tool subcommands. A little sugar for making this more uniform.

subcommand = None

help_text = None

description = None

add_to_subparsers(*subparsers*)

go()

property work_manager

The work manager for this tool. Raises AttributeError if this is not a parallel tool.

class westpa.tools.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

class westpa.tools.**WESTMultiTool**(*wm_env=None*)

Bases: *WESTParallelTool*

Base class for command-line tools which work with multiple simulations. Automatically parses for and gives commands to load multiple files.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

parse_from_yaml(*yamlfilepath*)

Parse options from YAML input file. Command line arguments take precedence over options specified in the YAML hierarchy. TODO: add description on how YAML files should be constructed.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

exception NoSimulationsException

Bases: Exception

generate_file_list(*key_list*)

A convenience function which takes in a list of keys that are filenames, and returns a dictionary which contains all the individual files loaded inside of a dictionary keyed to the filename.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.tools.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.tools.**WESTDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

Tool for synthesizing a dataset for analysis from other datasets. This may be done using a custom function, or a list of "data set specifications". It is anticipated that if several source datasets are required, then a tool will have multiple instances of this class.

group_name = 'input dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.tools.**WESTWDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

group_name = 'weight dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.tools.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
: The first iteration included in the calculation.

last_iter
: One past the last iteration included in the calculation.

iter_step
: Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.tools.**SegSelector**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

parse_segsel_file(*filename*)

class westpa.tools.**BinMappingComponent**

Bases: *WESTToolComponent*

Component for obtaining a bin mapper from one of several places based on command-line arguments. Such locations include an HDF5 file that contains pickled mappers (including the primary WEST HDF5 file), the system object, an external function, or (in the common case of rectilinear bins) a list of lists of bin boundaries.

Some configuration is necessary prior to calling process_args() if loading a mapper from HDF5. Specifically, either set_we_h5file_info() or set_other_h5file_info() must be called to describe where to find the appropriate mapper. In the case of set_we_h5file_info(), the mapper used for WE at the end of a given iteration will be loaded. In the case of set_other_h5file_info(), an arbitrary group and hash value are specified; the mapper corresponding to that hash in the given group will be returned.

In the absence of arguments, the mapper contained in an existing HDF5 file is preferred; if that is not available, the mapper from the system driver is used.

This component adds the following arguments to argument parsers:

--bins-from-system Obtain bins from the system driver

—bins-from-expr=EXPR Construct rectilinear bins by parsing EXPR and calling RectilinearBinMapper() with the result. EXPR must therefore be a list of lists.

–bins-from-function=[PATH:]MODULE.FUNC
Call an external function FUNC in module MODULE (optionally adding PATH to the search path when loading MODULE) which, when called, returns a fully-constructed bin mapper.

—bins-from-file Load bin definitions from a YAML configuration file.

--bins-from-h5file Load bins from the file being considered; this is intended to mean the master WEST HDF5 file or results of other binning calculations, as appropriate.

add_args(*parser*, *description='binning options'*, *suppress=[]*)

Add arguments specific to this component to the given argparse parser.

add_target_count_args(*parser*, *description='bin target count options'*)

Add options to the given parser corresponding to target counts.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

set_we_h5file_info(*n_iter=None*, *data_manager=None*, *required=False*)

Set up to load a bin mapper from the master WEST HDF5 file. The mapper is actually loaded from the file when self.load_bin_mapper() is called, if and only if command line arguments direct this. If `required` is true, then a mapper must be available at iteration `n_iter`, or else an exception will be raised.

set_other_h5file_info(*topology_group*, *hashval*)

Set up to load a bin mapper from (any) open HDF5 file, where bin topologies are stored in `topology_group` (an h5py Group object) and the desired mapper has hash value `hashval`. The mapper itself is loaded when self.load_bin_mapper() is called.

westpa.tools.**mapper_from_dict**(*ybins*)

class westpa.tools.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.tools.**Plotter**(*h5file*, *h5key*, *iteration=-1*, *interface='matplotlib'*)

Bases: object

This is a semi-generic plotting interface that has a built in curses based terminal plotter. It's fairly specific to what we're using it for here, but we could (and maybe should) build it out into a little library that we can use via the command line to plot things. Might be useful for looking at data later. That would also cut the size of this tool down by a good bit.

plot(*i=0*, *j=1*, *tau=1*, *iteration=None*, *dim=0*, *interface=None*)

class westpa.tools.**WIPIDataset**(*raw*, *key*)

Bases: object

keys()

class westpa.tools.**KineticsIteration**(*kin_h5file*, *index*, *assign*, *iteration=-1*)

Bases: object

keys()

class westpa.tools.**WIPIScheme**(*scheme*, *name*, *parent*, *settings*)

Bases: object

property scheme

property list_schemes

Lists what schemes are configured in west.cfg file. Schemes should be structured as follows, in west.cfg:
west:

system:

analysis:
directory: analysis analysis_schemes:

scheme.1:
enabled: True states:

- label: unbound coords: [[7.0]]
- label: bound coords: [[2.7]]

bins:

- type: RectilinearBinMapper boundaries: [[0.0, 2.80, 7, 10000]]

property iteration

property assign

property direct

The output from w_direct.py from the current scheme.

property state_labels

property bin_labels

property west

property reweight

property current

The current iteration. See help for __get_data_for_iteration__

property past

The previous iteration. See help for __get_data_for_iteration__

6.4.2 westpa.tools.binning module

class westpa.tools.binning.**count**(*start=0*, *step=1*)

Bases: object

Return a count object whose .__next__() method returns consecutive values.
Equivalent to:

def count(firstval=0, step=1):

x = firstval while 1:

yield x x += step

exception westpa.tools.binning.**PickleError**

Bases: Exception

class westpa.tools.binning.**RectilinearBinMapper**(*boundaries*)

Bases: *BinMapper*

Bin into a rectangular grid based on tuples of float values

property boundaries

assign(*coords*, *mask=None*, *output=None*)

westpa.tools.binning.**weight_dtype**

alias of float64

westpa.tools.binning.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

class westpa.tools.binning.**WESTToolComponent**

Bases: object

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

westpa.tools.binning.**mapper_from_expr**(*expr*)

westpa.tools.binning.**mapper_from_system**()

westpa.tools.binning.**mapper_from_function**(*funcspec*)

Return a mapper constructed by calling a function in a named module. `funcspec` should be formatted as `[PATH]:MODULE.FUNC`. This function loads MODULE, optionally adding PATH to the search path, then returns MODULE.FUNC()

westpa.tools.binning.**mapper_from_hdf5**(*topol_group*, *hashval*)

Retrieve the mapper identified by `hashval` from the given bin topology group `topol_group`. Returns `(mapper, pickle, hashval)`

westpa.tools.binning.**mapper_from_yaml**(*yamlfilename*)

westpa.tools.binning.**mapper_from_dict**(*ybins*)

westpa.tools.binning.**write_bin_info**(*mapper*, *assignments*, *weights*, *n_target_states*, *outfile=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *detailed=False*)

Write information about binning to `outfile`, given a mapper (`mapper`) and the weights (`weights`) and bin assignments (`assignments`) of a set of segments, along with a target state count (`n_target_states`). If `detailed` is true, then per-bin information is written as well as summary information about all bins.

westpa.tools.binning.**write_bin_labels**(*mapper*, *dest*, *header='# bin labels:\n'*, *fmt='# bin {index:{max_iwidth}d} -- {label!s}\n'*)

Print labels for all bins in `mapper` to the file-like object``dest``.

If provided, `header` is printed prior to any labels. A number of expansions are available in `header`:

- `mapper` – the mapper itself (from which most of the following can be obtained)
- `classname` – the class name of the mapper
- `nbins` – number of bins in the mapper

The `fmt` string specifies how bin labels are to be printed. A number of expansions are available in `fmt`:

- `index` – the zero-based index of the bin
- `label` – the label of the bin
- `max_iwidth` – the maximum width (in characters) of the bin index, for pretty alignment

class westpa.tools.binning.**BinMappingComponent**

Bases: *WESTToolComponent*

Component for obtaining a bin mapper from one of several places based on command-line arguments. Such locations include an HDF5 file that contains pickled mappers (including the primary WEST HDF5 file), the system object, an external function, or (in the common case of rectilinear bins) a list of lists of bin boundaries.

Some configuration is necessary prior to calling process_args() if loading a mapper from HDF5. Specifically, either set_we_h5file_info() or set_other_h5file_info() must be called to describe where to find the appropriate mapper. In the case of set_we_h5file_info(), the mapper used for WE at the end of a given iteration will be loaded. In the case of set_other_h5file_info(), an arbitrary group and hash value are specified; the mapper corresponding to that hash in the given group will be returned.

In the absence of arguments, the mapper contained in an existing HDF5 file is preferred; if that is not available, the mapper from the system driver is used.

This component adds the following arguments to argument parsers:

--bins-from-system Obtain bins from the system driver

—bins-from-expr=EXPR Construct rectilinear bins by parsing EXPR and calling RectilinearBinMapper() with the result. EXPR must therefore be a list of lists.

–bins-from-function=[PATH:]MODULE.FUNC
Call an external function FUNC in module MODULE (optionally adding PATH to the search path when loading MODULE) which, when called, returns a fully-constructed bin mapper.

—bins-from-file Load bin definitions from a YAML configuration file.

--bins-from-h5file Load bins from the file being considered; this is intended to mean the master WEST HDF5 file or results of other binning calculations, as appropriate.

`add_args`(*parser*, *description='binning options'*, *suppress=[]*)

Add arguments specific to this component to the given argparse parser.

`add_target_count_args`(*parser*, *description='bin target count options'*)

Add options to the given parser corresponding to target counts.

`process_args`(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

`set_we_h5file_info`(*n_iter=None*, *data_manager=None*, *required=False*)

Set up to load a bin mapper from the master WEST HDF5 file. The mapper is actually loaded from the file when self.load_bin_mapper() is called, if and only if command line arguments direct this. If `required` is true, then a mapper must be available at iteration `n_iter`, or else an exception will be raised.

`set_other_h5file_info`(*topology_group*, *hashval*)

Set up to load a bin mapper from (any) open HDF5 file, where bin topologies are stored in `topology_group` (an h5py Group object) and the desired mapper has hash value `hashval`. The mapper itself is loaded when self.load_bin_mapper() is called.

6.4.3 westpa.tools.core module

Core classes for creating WESTPA command-line tools

`class westpa.tools.core.`**`WESTToolComponent`**

Bases: `object`

Base class for WEST command line tools and components used in constructing tools

`include_arg`(*argname*)

`exclude_arg`(*argname*)

`set_arg_default`(*argname*, *value*)

`add_args`(*parser*)

Add arguments specific to this component to the given argparse parser.

`process_args`(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

class westpa.tools.core.**WESTTool**

Bases: *WESTToolComponent*

Base class for WEST command line tools

prog = None

usage = None

description = None

epilog = None

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

make_parser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then call self.go()

class westpa.tools.core.**WESTParallelTool**(*wm_env=None*)

Bases: *WESTTool*

Base class for command-line tools parallelized with wwmgr. This automatically adds and processes wwmgr command-line arguments and creates a work manager at self.work_manager.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.tools.core.**WESTMultiTool**(*wm_env=None*)

Bases: *WESTParallelTool*

Base class for command-line tools which work with multiple simulations. Automatically parses for and gives commands to load multiple files.

make_parser_and_process(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *args=None*)

A convenience function to create a parser, call add_all_args(), and then call process_all_args(). The argument namespace is returned.

parse_from_yaml(*yamlfilepath*)

Parse options from YAML input file. Command line arguments take precedence over options specified in the YAML hierarchy. TODO: add description on how YAML files should be constructed.

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

exception NoSimulationsException

Bases: Exception

generate_file_list(*key_list*)

A convenience function which takes in a list of keys that are filenames, and returns a dictionary which contains all the individual files loaded inside of a dictionary keyed to the filename.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

main()

A convenience function to make a parser, parse and process arguments, then run self.go() in the master process.

class westpa.tools.core.**WESTSubcommand**(*parent*)

Bases: *WESTToolComponent*

Base class for command-line tool subcommands. A little sugar for making this more uniform.

subcommand = None

help_text = None

description = None

add_to_subparsers(*subparsers*)

go()

property work_manager

The work manager for this tool. Raises AttributeError if this is not a parallel tool.

class westpa.tools.core.**WESTMasterCommand**

Bases: *WESTTool*

Base class for command-line tools that employ subcommands

subparsers_title = None

subcommands = None

include_help_command = True

add_args(*parser*)

Add arguments specific to this tool to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this tool and deal with them appropriately (setting instance variables, etc)

go()

Perform the analysis associated with this tool.

6.4.4 westpa.tools.data_reader module

class westpa.tools.data_reader.**WESTToolComponent**

Bases: object

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

westpa.tools.data_reader.**get_object**(*object_name*, *path=None*)

Attempt to load the given object, using additional path information if given.

class westpa.tools.data_reader.**FnDSSpec**(*h5file_or_name*, *fn*)

Bases: *FileLinkedDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.tools.data_reader.**MultiDSSpec**(*dsspecs*)

Bases: *DSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.tools.data_reader.**SingleSegmentDSSpec**(*h5file_or_name*, *dsname*, *alias=None*, *slice=None*)

Bases: *SingleDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

get_segment_data(*n_iter*, *seg_id*)

class westpa.tools.data_reader.**SingleIterDSSpec**(*h5file_or_name*, *dsname*, *alias=None*, *slice=None*)

Bases: *SingleDSSpec*

get_iter_data(*n_iter*, *seg_slice=(slice(None, None, None),)*)

class westpa.tools.data_reader.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.tools.data_reader.**WESTDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

Tool for synthesizing a dataset for analysis from other datasets. This may be done using a custom function, or a list of "data set specifications". It is anticipated that if several source datasets are required, then a tool will have multiple instances of this class.

group_name = 'input dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.tools.data_reader.**WESTWDSSynthesizer**(*default_dsname=None*, *h5filename=None*)

Bases: *WESTToolComponent*

group_name = 'weight dataset options'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

6.4.5 westpa.tools.dtypes module

Numpy/HDF5 data types shared among several WESTPA tools

westpa.tools.dtypes.**n_iter_dtype**

alias of `uint32`

westpa.tools.dtypes.**seg_id_dtype**

alias of `int64`

westpa.tools.dtypes.**weight_dtype**

alias of `float64`

6.4.6 westpa.tools.iter_range module

class westpa.tools.iter_range.**WESTToolComponent**

Bases: `object`

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

class westpa.tools.iter_range.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter

The first iteration included in the calculation.

last_iter
One past the last iteration included in the calculation.

iter_step
Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

6.4.7 westpa.tools.kinetics_tool module

class westpa.tools.kinetics_tool.**WESTDataReader**

Bases: *WESTToolComponent*

Tool for reading data from WEST-related HDF5 files. Coordinates finding the main HDF5 file from west.cfg or command line arguments, caching of certain kinds of data (eventually), and retrieving auxiliary data sets from various places.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

open(*mode='r'*)

close()

property weight_dsspec

property parent_id_dsspec

class westpa.tools.kinetics_tool.**IterRangeSelection**(*data_manager=None*)

Bases: *WESTToolComponent*

Select and record limits on iterations used in analysis and/or reporting. This class provides both the user-facing command-line options and parsing, and the application-side API for recording limits in HDF5.

HDF5 datasets calculated based on a restricted set of iterations should be tagged with the following attributes:

first_iter
: The first iteration included in the calculation.

last_iter
: One past the last iteration included in the calculation.

iter_step
: Blocking or sampling period for iterations included in the calculation.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*, *override_iter_start=None*, *override_iter_stop=None*, *default_iter_step=1*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

iter_block_iter()

Return an iterable of (block_start,block_end) over the blocks of iterations selected by –first-iter/–last-iter/–step-iter.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first-iter/–last-iter/–step-iter.

record_data_iter_range(*h5object*, *iter_start=None*, *iter_stop=None*)

Store attributes `iter_start` and `iter_stop` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data at least for the iteration range specified.

check_data_iter_range_equal(*h5object*, *iter_start=None*, *iter_stop=None*)

Check that the given HDF5 object contains (as denoted by its `iter_start`/`iter_stop` attributes) data exactly for the iteration range specified.

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride (in other words, the given `iter_step` is a multiple of the stride with which data was recorded).

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *iter_start=None*, *iter_stop=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*iter_start=None*, *iter_stop=None*, *iter_step=None*, *dtype=None*)

Return a sequence for the given iteration numbers and stride, filling in missing values from those stored on `self`. The smallest data type capable of holding `iter_stop` is returned unless otherwise specified using the `dtype` argument.

class westpa.tools.kinetics_tool.**WESTSubcommand**(*parent*)

Bases: *WESTToolComponent*

Base class for command-line tool subcommands. A little sugar for making this more uniform.

subcommand = None

help_text = None

description = None

add_to_subparsers(*subparsers*)

go()

property work_manager

The work manager for this tool. Raises AttributeError if this is not a parallel tool.

class westpa.tools.kinetics_tool.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

westpa.tools.kinetics_tool.**generate_future**(*work_manager*, *name*, *eval_block*, *kwargs*)

class westpa.tools.kinetics_tool.**WESTKineticsBase**(*parent*)

Bases: *WESTSubcommand*

Common argument processing for w_direct/w_reweight subcommands. Mostly limited to handling input and output from w_assign.

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

class westpa.tools.kinetics_tool.**AverageCommands**(*parent*)

Bases: *WESTKineticsBase*

default_output_file = 'direct.h5'

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

stamp_mcbs_info(*dataset*)

open_files()

open_assignments()

print_averages(*dataset*, *header*, *dim=1*)

run_calculation(*pi*, *nstates*, *start_iter*, *stop_iter*, *step_iter*, *dataset*, *eval_block*, *name*, *dim*, *do_averages=False*, ***extra*)

6.4.8 westpa.tools.plot module

class westpa.tools.plot.**Plotter**(*h5file*, *h5key*, *iteration=-1*, *interface='matplotlib'*)

Bases: object

This is a semi-generic plotting interface that has a built in curses based terminal plotter. It's fairly specific to what we're using it for here, but we could (and maybe should) build it out into a little library that we can use via the command line to plot things. Might be useful for looking at data later. That would also cut the size of this tool down by a good bit.

plot(*i=0*, *j=1*, *tau=1*, *iteration=None*, *dim=0*, *interface=None*)

6.4.9 westpa.tools.progress module

class westpa.tools.progress.**ProgressIndicator**(*stream=None*, *interval=1*)

Bases: object

draw_fancy()

draw_simple()

draw()

clear()

property operation

property extent

property progress

new_operation(*operation*, *extent=None*, *progress=0*)

start()

stop()

class westpa.tools.progress.**WESTToolComponent**

Bases: object

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

class westpa.tools.progress.**ProgressIndicatorComponent**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

6.4.10 westpa.tools.selected_segs module

class westpa.tools.selected_segs.**WESTToolComponent**

Bases: object

Base class for WEST command line tools and components used in constructing tools

include_arg(*argname*)

exclude_arg(*argname*)

set_arg_default(*argname*, *value*)

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

add_all_args(*parser*)

Add arguments for all components from which this class derives to the given parser, starting with the class highest up the inheritance chain (most distant ancestor).

process_all_args(*args*)

westpa.tools.selected_segs.**seg_id_dtype**

alias of `int64`

class westpa.tools.selected_segs.**SegmentSelection**(*iterable=None*)

Bases: `object`

Initialize this segment selection from an iterable of (n_iter,seg_id) pairs.

add(*pair*)

from_iter(*n_iter*)

property start_iter

property stop_iter

classmethod from_text(*filename*)

class westpa.tools.selected_segs.**AllSegmentSelection**(*start_iter=None*, *stop_iter=None*, *data_manager=None*)

Bases: *SegmentSelection*

Initialize this segment selection from an iterable of (n_iter,seg_id) pairs.

add(*pair*)

from_iter(*n_iter*)

class westpa.tools.selected_segs.**SegSelector**

Bases: *WESTToolComponent*

add_args(*parser*)

Add arguments specific to this component to the given argparse parser.

process_args(*args*)

Take argparse-processed arguments associated with this component and deal with them appropriately (setting instance variables, etc)

parse_segsel_file(*filename*)

6.4.11 westpa.tools.wipi module

class westpa.tools.wipi.**Plotter**(*h5file*, *h5key*, *iteration=-1*, *interface='matplotlib'*)

Bases: object

This is a semi-generic plotting interface that has a built in curses based terminal plotter. It's fairly specific to what we're using it for here, but we could (and maybe should) build it out into a little library that we can use via the command line to plot things. Might be useful for looking at data later. That would also cut the size of this tool down by a good bit.

plot(*i=0*, *j=1*, *tau=1*, *iteration=None*, *dim=0*, *interface=None*)

class westpa.tools.wipi.**WIPIDataset**(*raw*, *key*)

Bases: object

keys()

class westpa.tools.wipi.**KineticsIteration**(*kin_h5file*, *index*, *assign*, *iteration=-1*)

Bases: object

keys()

class westpa.tools.wipi.**WIPIScheme**(*scheme*, *name*, *parent*, *settings*)

Bases: object

property scheme

property list_schemes

Lists what schemes are configured in west.cfg file. Schemes should be structured as follows, in west.cfg:
west:

system:

analysis:
directory: analysis analysis_schemes:

scheme.1:
enabled: True states:

- label: unbound coords: [[7.0]]
- label: bound coords: [[2.7]]

bins:

- type: RectilinearBinMapper boundaries: [[0.0, 2.80, 7, 10000]]

property iteration

property assign

property direct

The output from w_direct.py from the current scheme.

property state_labels

property bin_labels

property west

property reweight

property current

The current iteration. See help for __get_data_for_iteration__

property past

The previous iteration. See help for __get_data_for_iteration__

6.5 Other Packages

6.5.1 westpa.fasthist package

6.5.1.1 Module contents

westpa.fasthist.**histnd**(*values*, *binbounds*, *weights=1.0*, *out=None*, *binbound_check=True*, *ignore_out_of_range=False*)

Generate an N-dimensional PDF (or contribution to a PDF) from the given values. `binbounds` is a list of arrays of boundary values, with one entry for each dimension (`values` must have as many columns as there are entries in `binbounds`) `weight`, if provided, specifies the weight each value contributes to the histogram; this may be a scalar (for equal weights for all values) or a vector of the same length as `values` (for unequal weights). If `binbound_check` is True, then the boundaries are checked for strict positive monotonicity; set to False to shave a few microseconds if you know your bin boundaries to be monotonically increasing.

westpa.fasthist.**normhistnd**(*hist*, *binbounds*)

Normalize the N-dimensional histogram `hist` with corresponding bin boundaries `binbounds`. Modifies `hist` in place and returns the normalization factor used.

6.5.2 westpa.mclib package

6.5.2.1 Module contents

A package for performing Monte Carlo bootstrap estimates of statistics.

westpa.mclib.**mcbs_correltime**(*dataset*, *alpha*, *n_sets=None*)

Calculate the correlation time of the given `dataset`, significant to the (1-`alpha`) level, using the method described in Huber & Kim, "Weighted-ensemble Brownian dynamics simulations for protein association reactions" (1996), doi:10.1016/S0006-3495(96)79552-8. An appropriate balance between space and speed is chosen based on the size of the input data.

Returns 0 for data statistically uncorrelated with (1-alpha) confidence, otherwise the correlation length. (Thus, the appropriate stride for blocking is the result of this function plus one.)

westpa.mclib.**get_bssize**(*alpha*)

Return a bootstrap data set size appropriate for the given confidence level.

westpa.mclib.**mcbs_ci**(*dataset*, *estimator*, *alpha*, *dlen*, *n_sets=None*, *args=None*, *kwargs=None*, *sort=<function msort>*)

Perform a Monte Carlo bootstrap estimate for the (1-`alpha`) confidence interval on the given `dataset` with the given `estimator`. This routine is not appropriate for time-correlated data.

Returns `(estimate, ci_lb, ci_ub)` where `estimate` is the application of the given `estimator` to the input `dataset`, and `ci_lb` and `ci_ub` are the lower and upper limits, respectively, of the (1-`alpha`) confidence interval on `estimate`.

estimator is called as estimator(dataset, *args, **kwargs). Common estimators include:

- numpy.mean – calculate the confidence interval on the mean of dataset
- numpy.median – calculate a confidence interval on the median of dataset
- numpy.std – calculate a confidence interval on the standard deviation of datset.

n_sets is the number of synthetic data sets to generate using the given estimator, which will be chosen using `get_bssize()` if n_sets is not given.

sort can be used to override the sorting routine used to calculate the confidence interval, which should only be necessary for estimators returning vectors rather than scalars.

westpa.mclib.**mcbs_ci_correl**(*estimator_datasets*, *estimator*, *alpha*, *n_sets=None*, *args=None*, *autocorrel_alpha=None*, *autocorrel_n_sets=None*, *subsample=None*, *do_correl=True*, *mcbs_enable=None*, *estimator_kwargs={}*)

Perform a Monte Carlo bootstrap estimate for the (1-alpha) confidence interval on the given dataset with the given estimator. This routine is appropriate for time-correlated data, using the method described in Huber & Kim, "Weighted-ensemble Brownian dynamics simulations for protein association reactions" (1996), doi:10.1016/S0006-3495(96)79552-8 to determine a statistically-significant correlation time and then reducing the dataset by a factor of that correlation time before running a "classic" Monte Carlo bootstrap.

Returns (estimate, ci_lb, ci_ub, correl_time) where estimate is the application of the given estimator to the input dataset, ci_lb and ci_ub are the lower and upper limits, respectively, of the (1-alpha) confidence interval on estimate, and correl_time is the correlation time of the dataset, significant to (1-autocorrel_alpha).

estimator is called as estimator(dataset, *args, **kwargs). Common estimators include:

- np.mean – calculate the confidence interval on the mean of dataset
- np.median – calculate a confidence interval on the median of dataset
- np.std – calculate a confidence interval on the standard deviation of datset.

n_sets is the number of synthetic data sets to generate using the given estimator, which will be chosen using `get_bssize()` if n_sets is not given.

autocorrel_alpha (which defaults to alpha) can be used to adjust the significance level of the autocorrelation calculation. Note that too high a significance level (too low an alpha) for evaluating the significance of autocorrelation values can result in a failure to detect correlation if the autocorrelation function is noisy.

The given subsample function is used, if provided, to subsample the dataset prior to running the full Monte Carlo bootstrap. If none is provided, then a random entry from each correlated block is used as the value for that block. Other reasonable choices include np.mean, np.median, (lambda x: x[0]) or (lambda x: x[-1]). In particular, using subsample=np.mean will converge to the block averaged mean and standard error, while accounting for any non-normality in the distribution of the mean.

6.5.3 westpa.trajtree package

6.5.3.1 westpa.trajtree module

class westpa.trajtree.**TrajTreeSet**(*segsel=None*, *data_manager=None*)

Bases: _trajtree_base

get_roots()

get_root_indices()

trace_trajectories(*visit*, *get_visitor_state=None*, *set_visitor_state=None*, *vargs=None*, *vkwargs=None*)

6.5.3.2 westpa.trajtree.trajtree module

class westpa.trajtree.trajtree.**AllSegmentSelection**(*start_iter=None*, *stop_iter=None*, *data_manager=None*)

Bases: *SegmentSelection*

Initialize this segment selection from an iterable of (n_iter,seg_id) pairs.

add(*pair*)

from_iter(*n_iter*)

class westpa.trajtree.trajtree.**trajnode**(*n_iter*, *seg_id*)

Bases: tuple

Create new instance of trajnode(n_iter, seg_id)

n_iter

Alias for field number 0

seg_id

Alias for field number 1

class westpa.trajtree.trajtree.**TrajTreeSet**(*segsel=None*, *data_manager=None*)

Bases: _trajtree_base

get_roots()

get_root_indices()

trace_trajectories(*visit*, *get_visitor_state=None*, *set_visitor_state=None*, *vargs=None*, *vkwargs=None*)

class westpa.trajtree.trajtree.**FakeTrajTreeSet**

Bases: *TrajTreeSet*

6.5.4 WESTPA Old Tools

6.5.4.1 westpa.oldtools package

6.5.4.1.1 westpa.oldtools module

6.5.4.1.2 westpa.oldtools.files module

westpa.oldtools.files.**load_npy_or_text**(*filename*)

Load an array from an existing .npy file, or read a text file and convert to a NumPy array. In either case, return a NumPy array. If a pickled NumPy dataset is found, memory-map it read-only. If the specified file does not contain a pickled NumPy array, attempt to read the file using numpy.loadtxt(filename, **kwargs).

6.5.4.1.3 westpa.oldtools.miscfn module

Miscellaneous support functions for WEST and WEST tools

`westpa.oldtools.miscfn.`**parse_int_list**(*list_string*)

Parse a simple list consisting of integers or ranges of integers separated by commas. Ranges are specified as min:max, and include the maximum value (unlike Python's `range`). Duplicate values are ignored. Returns the result as a sorted list. Raises ValueError if the list cannot be parsed.

6.5.4.2 westpa.oldtools.aframe package

6.5.4.2.1 westpa.oldtools.aframe

WEST Analyis framework – an unholy mess of classes exploiting each other

class `westpa.oldtools.aframe.`**AnalysisMixin**

Bases: `object`

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

exception `westpa.oldtools.aframe.`**ArgumentError**(**args*, ***kwargs*)

Bases: `RuntimeError`

class `westpa.oldtools.aframe.`**WESTAnalysisTool**

Bases: `object`

add_args(*parser*, *upcall=True*)

Add arguments to a parser common to all analyses of this type.

process_args(*args*, *upcall=True*)

open_analysis_backing()

close_analysis_backing()

require_analysis_group(*groupname*, *replace=False*)

class `westpa.oldtools.aframe.`**IterRangeMixin**

Bases: *AnalysisMixin*

A mixin for limiting the range of data considered for a given analysis. This should go after DataManagerMixin

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

check_iter_range()

iter_block_iter()

Return an iterable of (block_first,block_last+1) over the blocks of iterations selected by –first/–last/–step. NOTE WELL that the second of the pair follows Python iterator conventions and returns one past the last element of the block.

n_iter_blocks()

Return the number of blocks of iterations (as returned by `iter_block_iter`) selected by –first/–last/–step.

record_data_iter_range(*h5object*, *first_iter=None*, *last_iter=None*)

Store attributes `first_iter` and `last_iter` on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute `iter_step` on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *first_iter=None*, *last_iter=None*)

Check that the given HDF5 object contains (as denoted by its `first_iter`/`last_iter` attributes) at least the data range specified.

check_data_iter_range_equal(*h5object*, *first_iter=None*, *last_iter=None*)

Check that the given HDF5 object contains per-iteration data for exactly the specified iterations (as denoted by the object's `first_iter` and `last_iter` attributes

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride. (In other words, is the given `iter_step` a multiple of the stride with which data was recorded.)

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *first_iter=None*, *last_iter=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*first_iter=None*, *last_iter=None*, *iter_step=None*)

class westpa.oldtools.aframe.**WESTDataReaderMixin**

Bases: *AnalysisMixin*

A mixin for analysis requiring access to the HDF5 files generated during a WEST run.

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

clear_run_cache()

property cache_pcoords

Whether or not to cache progress coordinate data. While caching this data can significantly speed up some analysis operations, this requires copious RAM.

Setting this to False when it was formerly True will release any cached data.

get_summary_table()

get_iter_group(*n_iter*)

Return the HDF5 group corresponding to `n_iter`

get_segments(*n_iter*, *include_pcoords=True*)

Return all segments present in iteration n_iter

get_segments_by_id(*n_iter*, *seg_ids*, *include_pcoords=True*)

Get segments from the data manager, employing caching where possible

get_children(*segment*, *include_pcoords=True*)

get_seg_index(*n_iter*)

get_wtg_parent_array(*n_iter*)

get_parent_array(*n_iter*)

get_pcoord_array(*n_iter*)

get_pcoord_dataset(*n_iter*)

get_pcoords(*n_iter*, *seg_ids*)

get_seg_ids(*n_iter*, *bool_array=None*)

get_created_seg_ids(*n_iter*)

Return a list of seg_ids corresponding to segments which were created for the given iteration (are not continuations).

max_iter_segs_in_range(*first_iter*, *last_iter*)

Return the maximum number of segments present in any iteration in the range selected

total_segs_in_range(*first_iter*, *last_iter*)

Return the total number of segments present in all iterations in the range selected

get_pcoord_len(*n_iter*)

Get the length of the progress coordinate array for the given iteration.

get_total_time(*first_iter=None*, *last_iter=None*, *dt=None*)

Return the total amount of simulation time spanned between first_iter and last_iter (inclusive).

class westpa.oldtools.aframe.**ExtDataReaderMixin**

Bases: *AnalysisMixin*

An external data reader, primarily designed for reading brute force data, but also suitable for any auxiliary datasets required for analysis.

default_chunksize = 8192

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

is_npy(*filename*)

load_npy_or_text(*filename*)

Load an array from an existing .npy file, or read a text file and convert to a NumPy array. In either case, return a NumPy array. If a pickled NumPy dataset is found, memory-map it read-only. If the specified file does not contain a pickled NumPy array, attempt to read the file using numpy.loadtxt(filename).

text_to_h5dataset(*fileobj*, *group*, *dsname*, *dtype=<class 'numpy.float64'>*, *skiprows=0*, *usecols=None*, *chunksize=None*)

Read text-format data from the given filename or file-like object `fileobj` and write to a newly-created dataset called `dsname` in the HDF5 group `group`. The data is stored as type `dtype`. By default, the shape is taken as (number of lines, number of columns); columns can be omitted by specifying a list for `usecols`, and lines can be skipped by using `skiprows`. Data is read in chunks of `chunksize` rows.

npy_to_h5dataset(*array*, *group*, *dsname*, *usecols=None*, *chunksize=None*)

Store the given array into a newly-created dataset named `dsname` in the HDF5 group `group`, optionally only storing a subset of columns. Data is written `chunksize` rows at a time, allowing very large memory-mapped arrays to be copied.

class westpa.oldtools.aframe.**BFDataManager**

Bases: *AnalysisMixin*

A class to manage brute force trajectory data. The primary purpose is to read in and manage brute force progress coordinate data for one or more trajectories. The trajectories need not be the same length, but they do need to have the same time spacing for progress coordinate values.

traj_index_dtype = dtype([('pcoord_len', '<u8'), ('source_data', 'O')])

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

update_traj_index(*traj_id*, *pcoord_len*, *source_data*)

get_traj_group(*traj_id*)

create_traj_group()

get_n_trajs()

get_traj_len(*traj_id*)

get_max_traj_len()

get_pcoord_array(*traj_id*)

get_pcoord_dataset(*traj_id*)

require_bf_h5file()

close_bf_h5file()

class westpa.oldtools.aframe.**BinningMixin**

Bases: *AnalysisMixin*

A mixin for performing binning on WEST data.

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

mapper_from_expr(*expr*)

write_bin_labels(*dest*, *header='# bin labels:\n'*, *format='# bin {bin_index:{max_iwidth}d} -- {label!s}\n'*)

Print labels for all bins in `self.mapper` to `dest`. If provided, `header` is printed before any labels. The `format` string specifies how bin labels are to be printed. Valid entries are:

- `bin_index` – the zero-based index of the bin
- `label` – the label, as obtained by `bin.label`
- `max_iwidth` – the maximum width (in characters) of the bin index, for pretty alignment

require_binning_group()

delete_binning_group()

record_data_binhash(*h5object*)

Record the identity hash for self.mapper as an attribute on the given HDF5 object (group or dataset)

check_data_binhash(*h5object*)

Check whether the recorded bin identity hash on the given HDF5 object matches the identity hash for self.mapper

assign_to_bins()

Assign WEST segment data to bins. Requires the DataReader mixin to be in the inheritance tree

require_bin_assignments()

get_bin_assignments(*first_iter=None*, *last_iter=None*)

get_bin_populations(*first_iter=None*, *last_iter=None*)

class westpa.oldtools.aframe.**MCBSMixin**

Bases: *AnalysisMixin*

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

calc_mcbs_nsets(*alpha=None*)

calc_ci_bound_indices(*n_sets=None*, *alpha=None*)

class westpa.oldtools.aframe.**TrajWalker**(*data_reader*, *history_chunksize=100*)

Bases: object

A class to perform analysis by walking the trajectory tree. A stack is used rather than recursion, or else the highest number of iterations capable of being considered would be the same as the Python recursion limit.

trace_to_root(*n_iter*, *seg_id*)

Trace the given segment back to its starting point, returning a list of Segment objects describing the entire trajectory.

get_trajectory_roots(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments which start new trajectories. If min_iter or max_iter is specified, restrict the set of iterations within which the search is conducted.

get_initial_nodes(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments with which to begin a tree walk – those alive or created within [first_iter,last_iter].

trace_trajectories(*first_iter*, *last_iter*, *callable*, *include_pcoords=True*, *cargs=None*, *ckwargs=None*, *get_state=None*, *set_state=None*)

Walk the trajectory tree depth-first, calling
`callable(segment, children, history, *cargs, **ckwargs)` for each segment visited. `segment` is the segment being visited, `children` is that segment's children, `history` is the chain of segments leading to `segment` (not including `segment`). get_state and set_state are used to record and reset, respectively, any state specific to `callable` when a new branch is traversed.

class westpa.oldtools.aframe.**TransitionAnalysisMixin**

Bases: *AnalysisMixin*

require_transitions_group()

delete_transitions_group()

get_transitions_ds()

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

require_transitions()

find_transitions()

class westpa.oldtools.aframe.**TransitionEventAccumulator**(*n_bins*, *output_group*, *calc_fpts=True*)

Bases: object

index_dtype

alias of uint64

count_dtype

alias of uint64

weight_dtype

alias of float64

output_tdat_chunksize = 4096

tdat_buffersize = 524288

max_acc = 32768

clear()

clear_state()

get_state()

set_state(*state_dict*)

record_transition_data(*tdat*)

Update running statistics and write transition data to HDF5 (with buffering)

flush_transition_data()

Flush any unwritten output that may be present

start_accumulation(*assignments*, *weights*, *bin_pops*, *traj=0*, *n_iter=0*)

continue_accumulation(*assignments*, *weights*, *bin_pops*, *traj=0*, *n_iter=0*)

class westpa.oldtools.aframe.**BFTransitionAnalysisMixin**

Bases: *TransitionAnalysisMixin*

require_transitions()

find_transitions(*chunksize=65536*)

class westpa.oldtools.aframe.**KineticsAnalysisMixin**

Bases: *AnalysisMixin*

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

parse_bin_range(*range_string*)

check_bin_selection(*n_bins=None*)

Check to see that the bin ranges selected by the user conform to the available bins (i.e., bin indices are within the permissible range). Also assigns the complete bin range if the user has not explicitly limited the bins to be considered.

property selected_bin_pair_iter

class westpa.oldtools.aframe.**CommonOutputMixin**

Bases: *AnalysisMixin*

add_common_output_args(*parser_or_group*)

process_common_output_args(*args*)

class westpa.oldtools.aframe.**PlottingMixin**

Bases: *AnalysisMixin*

require_matplotlib()

6.5.4.2.2 westpa.oldtools.aframe.atool module

class westpa.oldtools.aframe.atool.**WESTAnalysisTool**

Bases: object

add_args(*parser*, *upcall=True*)

Add arguments to a parser common to all analyses of this type.

process_args(*args*, *upcall=True*)

open_analysis_backing()

close_analysis_backing()

require_analysis_group(*groupname*, *replace=False*)

6.5.4.2.3 westpa.oldtools.aframe.base_mixin module

exception westpa.oldtools.aframe.base_mixin.**ArgumentError**(**args*, ***kwargs*)

Bases: RuntimeError

class westpa.oldtools.aframe.base_mixin.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

6.5.4.2.4 westpa.oldtools.aframe.binning module

class westpa.oldtools.aframe.binning.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.binning.**BinningMixin**

Bases: *AnalysisMixin*

A mixin for performing binning on WEST data.

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

mapper_from_expr(*expr*)

write_bin_labels(*dest*, *header='# bin labels:\n'*, *format='# bin {bin_index:{max_iwidth}d} -- {label!s}\n'*)

Print labels for all bins in `self.mapper` to `dest`. If provided, `header` is printed before any labels. The `format` string specifies how bin labels are to be printed. Valid entries are:

- `bin_index` – the zero-based index of the bin
- `label` – the label, as obtained by `bin.label`
- `max_iwidth` – the maximum width (in characters) of the bin index, for pretty alignment

require_binning_group()

delete_binning_group()

record_data_binhash(*h5object*)

Record the identity hash for self.mapper as an attribute on the given HDF5 object (group or dataset)

check_data_binhash(*h5object*)

Check whether the recorded bin identity hash on the given HDF5 object matches the identity hash for self.mapper

assign_to_bins()

Assign WEST segment data to bins. Requires the DataReader mixin to be in the inheritance tree

require_bin_assignments()

get_bin_assignments(*first_iter=None*, *last_iter=None*)

get_bin_populations(*first_iter=None*, *last_iter=None*)

6.5.4.2.5 westpa.oldtools.aframe.data_reader module

class westpa.oldtools.aframe.data_reader.**Segment**(*n_iter=None*, *seg_id=None*, *weight=None*, *endpoint_type=None*, *parent_id=None*, *wtg_parent_ids=None*, *pcoord=None*, *status=None*, *walltime=None*, *cputime=None*, *data=None*)

Bases: object

A class wrapping segment data that must be passed through the work manager or data manager. Most fields are self-explanatory. One item worth noting is that a negative parent ID means that the segment starts from the initial state with ID -(segment.parent_id+1)

SEG_STATUS_UNSET = 0

SEG_STATUS_PREPARED = 1

SEG_STATUS_COMPLETE = 2

SEG_STATUS_FAILED = 3

SEG_INITPOINT_UNSET = 0

SEG_INITPOINT_CONTINUES = 1

SEG_INITPOINT_NEWTRAJ = 2

SEG_ENDPOINT_UNSET = 0

SEG_ENDPOINT_CONTINUES = 1

SEG_ENDPOINT_MERGED = 2

SEG_ENDPOINT_RECYCLED = 3

statuses = {'SEG_STATUS_COMPLETE': 2, 'SEG_STATUS_FAILED': 3, 'SEG_STATUS_PREPARED': 1, 'SEG_STATUS_UNSET': 0}

initpoint_types = {'SEG_INITPOINT_CONTINUES': 1, 'SEG_INITPOINT_NEWTRAJ': 2, 'SEG_INITPOINT_UNSET': 0}

endpoint_types = {'SEG_ENDPOINT_CONTINUES': 1, 'SEG_ENDPOINT_MERGED': 2, 'SEG_ENDPOINT_RECYCLED': 3, 'SEG_ENDPOINT_UNSET': 0}

status_names = {0: 'SEG_STATUS_UNSET', 1: 'SEG_STATUS_PREPARED', 2: 'SEG_STATUS_COMPLETE', 3: 'SEG_STATUS_FAILED'}

initpoint_type_names = {0: 'SEG_INITPOINT_UNSET', 1: 'SEG_INITPOINT_CONTINUES', 2: 'SEG_INITPOINT_NEWTRAJ'}

endpoint_type_names = {0: 'SEG_ENDPOINT_UNSET', 1: 'SEG_ENDPOINT_CONTINUES', 2: 'SEG_ENDPOINT_MERGED', 3: 'SEG_ENDPOINT_RECYCLED'}

static initial_pcoord(*segment*)

Return the initial progress coordinate point of this segment.

static final_pcoord(*segment*)

Return the final progress coordinate point of this segment.

property initpoint_type

property initial_state_id

property status_text

property endpoint_type_text

class westpa.oldtools.aframe.data_reader.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

westpa.oldtools.aframe.data_reader.**parse_int_list**(*list_string*)

Parse a simple list consisting of integers or ranges of integers separated by commas. Ranges are specified as min:max, and include the maximum value (unlike Python's `range`). Duplicate values are ignored. Returns the result as a sorted list. Raises ValueError if the list cannot be parsed.

class westpa.oldtools.aframe.data_reader.**WESTDataReaderMixin**

Bases: *AnalysisMixin*

A mixin for analysis requiring access to the HDF5 files generated during a WEST run.

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

clear_run_cache()

property cache_pcoords

Whether or not to cache progress coordinate data. While caching this data can significantly speed up some analysis operations, this requires copious RAM.

Setting this to False when it was formerly True will release any cached data.

get_summary_table()

get_iter_group(*n_iter*)

Return the HDF5 group corresponding to `n_iter`

get_segments(*n_iter*, *include_pcoords=True*)

Return all segments present in iteration n_iter

get_segments_by_id(*n_iter*, *seg_ids*, *include_pcoords=True*)

Get segments from the data manager, employing caching where possible

get_children(*segment*, *include_pcoords=True*)

get_seg_index(*n_iter*)

get_wtg_parent_array(*n_iter*)

get_parent_array(*n_iter*)

get_pcoord_array(*n_iter*)

get_pcoord_dataset(*n_iter*)

get_pcoords(*n_iter*, *seg_ids*)

get_seg_ids(*n_iter*, *bool_array=None*)

get_created_seg_ids(*n_iter*)

Return a list of seg_ids corresponding to segments which were created for the given iteration (are not continuations).

max_iter_segs_in_range(*first_iter*, *last_iter*)

Return the maximum number of segments present in any iteration in the range selected

total_segs_in_range(*first_iter*, *last_iter*)

Return the total number of segments present in all iterations in the range selected

get_pcoord_len(*n_iter*)

Get the length of the progress coordinate array for the given iteration.

get_total_time(*first_iter=None*, *last_iter=None*, *dt=None*)

Return the total amount of simulation time spanned between first_iter and last_iter (inclusive).

class westpa.oldtools.aframe.data_reader.**ExtDataReaderMixin**

Bases: *AnalysisMixin*

An external data reader, primarily designed for reading brute force data, but also suitable for any auxiliary datasets required for analysis.

default_chunksize = 8192

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

is_npy(*filename*)

load_npy_or_text(*filename*)

Load an array from an existing .npy file, or read a text file and convert to a NumPy array. In either case, return a NumPy array. If a pickled NumPy dataset is found, memory-map it read-only. If the specified file does not contain a pickled NumPy array, attempt to read the file using numpy.loadtxt(filename).

text_to_h5dataset(*fileobj*, *group*, *dsname*, *dtype=<class 'numpy.float64'>*, *skiprows=0*, *usecols=None*, *chunksize=None*)

Read text-format data from the given filename or file-like object `fileobj` and write to a newly-created dataset called `dsname` in the HDF5 group `group`. The data is stored as type `dtype`. By default, the shape is taken as (number of lines, number of columns); columns can be omitted by specifying a list for `usecols`, and lines can be skipped by using `skiprows`. Data is read in chunks of `chunksize` rows.

npy_to_h5dataset(*array*, *group*, *dsname*, *usecols=None*, *chunksize=None*)

Store the given array into a newly-created dataset named `dsname` in the HDF5 group `group`, optionally only storing a subset of columns. Data is written `chunksize` rows at a time, allowing very large memory-mapped arrays to be copied.

class westpa.oldtools.aframe.data_reader.**BFDataManager**

Bases: *AnalysisMixin*

A class to manage brute force trajectory data. The primary purpose is to read in and manage brute force progress coordinate data for one or more trajectories. The trajectories need not be the same length, but they do need to have the same time spacing for progress coordinate values.

traj_index_dtype = dtype([('pcoord_len', '<u8'), ('source_data', 'O')])

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

update_traj_index(*traj_id*, *pcoord_len*, *source_data*)

get_traj_group(*traj_id*)

create_traj_group()

get_n_trajs()

get_traj_len(*traj_id*)

get_max_traj_len()

get_pcoord_array(*traj_id*)

get_pcoord_dataset(*traj_id*)

require_bf_h5file()

close_bf_h5file()

6.5.4.2.6 westpa.oldtools.aframe.iter_range module

class westpa.oldtools.aframe.iter_range.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

exception westpa.oldtools.aframe.iter_range.**ArgumentError**(**args*, ***kwargs*)

Bases: RuntimeError

class westpa.oldtools.aframe.iter_range.**IterRangeMixin**

Bases: *AnalysisMixin*

A mixin for limiting the range of data considered for a given analysis. This should go after DataManagerMixin

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

check_iter_range()

iter_block_iter()

Return an iterable of (block_first,block_last+1) over the blocks of iterations selected by –first/–last/–step. NOTE WELL that the second of the pair follows Python iterator conventions and returns one past the last element of the block.

n_iter_blocks()

Return the number of blocks of iterations (as returned by iter_block_iter) selected by –first/–last/–step.

record_data_iter_range(*h5object*, *first_iter=None*, *last_iter=None*)

Store attributes first_iter and last_iter on the given HDF5 object (group/dataset)

record_data_iter_step(*h5object*, *iter_step=None*)

Store attribute iter_step on the given HDF5 object (group/dataset).

check_data_iter_range_least(*h5object*, *first_iter=None*, *last_iter=None*)

Check that the given HDF5 object contains (as denoted by its first_iter/last_iter attributes) at least the data range specified.

check_data_iter_range_equal(*h5object*, *first_iter=None*, *last_iter=None*)

Check that the given HDF5 object contains per-iteration data for exactly the specified iterations (as denoted by the object's first_iter and last_iter attributes

check_data_iter_step_conformant(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride suitable for extracting data with the given stride. (In other words, is the given `iter_step` a multiple of the stride with which data was recorded.)

check_data_iter_step_equal(*h5object*, *iter_step=None*)

Check that the given HDF5 object contains per-iteration data at an iteration stride the same as that specified.

slice_per_iter_data(*dataset*, *first_iter=None*, *last_iter=None*, *iter_step=None*, *axis=0*)

Return the subset of the given dataset corresponding to the given iteration range and stride. Unless otherwise specified, the first dimension of the dataset is the one sliced.

iter_range(*first_iter=None*, *last_iter=None*, *iter_step=None*)

6.5.4.2.7 westpa.oldtools.aframe.kinetics module

class westpa.oldtools.aframe.kinetics.**AnalysisMixin**

Bases: `object`

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.kinetics.**KineticsAnalysisMixin**

Bases: *AnalysisMixin*

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

parse_bin_range(*range_string*)

check_bin_selection(*n_bins=None*)

Check to see that the bin ranges selected by the user conform to the available bins (i.e., bin indices are within the permissible range). Also assigns the complete bin range if the user has not explicitly limited the bins to be considered.

property selected_bin_pair_iter

6.5.4.2.8 westpa.oldtools.aframe.mcbs module

Tools for Monte Carlo bootstrap error analysis

class westpa.oldtools.aframe.mcbs.**AnalysisMixin**

Bases: `object`

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.mcbs.**MCBSMixin**

Bases: *AnalysisMixin*

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

calc_mcbs_nsets(*alpha=None*)

calc_ci_bound_indices(*n_sets=None*, *alpha=None*)

westpa.oldtools.aframe.mcbs.**calc_mcbs_nsets**(*alpha*)

Return a bootstrap data set size appropriate for the given confidence level.

westpa.oldtools.aframe.mcbs.**calc_ci_bound_indices**(*n_sets*, *alpha*)

westpa.oldtools.aframe.mcbs.**bootstrap_ci_ll**(*estimator*, *data*, *alpha*, *n_sets*, *storage*, *sort*, *eargs=()*, *ekwargs={}*, *fhat=None*)

Low-level routine for calculating bootstrap error estimates. Arguments and return values are as those for `bootstrap_ci`, except that no argument is optional except additional arguments for the estimator (`eargs`, `ekwargs`). `data` must be an array (or subclass), and an additional array `storage` must be provided, which must be appropriately shaped and typed to hold `n_sets` results from `estimator`. Further, if the value `fhat` of the estimator must be pre-calculated to allocate `storage`, then its value may be passed; otherwise, `estimator(data, *eargs,**kwargs)` will be called to calculate it.

westpa.oldtools.aframe.mcbs.**bootstrap_ci**(*estimator*, *data*, *alpha*, *n_sets=None*, *sort=<function msort>*, *eargs=()*, *ekwargs={}*)

Perform a Monte Carlo bootstrap of a (1-alpha) confidence interval for the given `estimator`. Returns (fhat, ci_lower, ci_upper), where fhat is the result of `estimator(data, *eargs, **ekwargs)`, and `ci_lower` and `ci_upper` are the lower and upper bounds of the surrounding confidence interval, calculated by calling `estimator(syndata, *eargs, **ekwargs)` on each synthetic data set `syndata`. If `n_sets` is provided, that is the number of synthetic data sets generated, otherwise an appropriate size is selected automatically (see `calc_mcbs_nsets()`).

`sort`, if given, is applied to sort the results of calling `estimator` on each synthetic data set prior to obtaining the confidence interval. This function must sort on the last index.

Individual entries in synthetic data sets are selected by the first index of `data`, allowing this function to be used on arrays of multidimensional data.

Returns (fhat, lb, ub, ub-lb, abs((ub-lb)/fhat), and max(ub-fhat,fhat-lb)) (that is, the estimated value, the lower and upper bounds of the confidence interval, the width of the confidence interval, the relative width of the confidence interval, and the symmetrized error bar of the confidence interval).

6.5.4.2.9 westpa.oldtools.aframe.output module

class westpa.oldtools.aframe.output.**AnalysisMixin**

Bases: `object`

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.output.**CommonOutputMixin**

Bases: *AnalysisMixin*

add_common_output_args(*parser_or_group*)

process_common_output_args(*args*)

6.5.4.2.10 westpa.oldtools.aframe.plotting module

class westpa.oldtools.aframe.plotting.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.plotting.**PlottingMixin**

Bases: *AnalysisMixin*

require_matplotlib()

6.5.4.2.11 westpa.oldtools.aframe.trajwalker module

class westpa.oldtools.aframe.trajwalker.**TrajWalker**(*data_reader*, *history_chunksize=100*)

Bases: object

A class to perform analysis by walking the trajectory tree. A stack is used rather than recursion, or else the highest number of iterations capable of being considered would be the same as the Python recursion limit.

trace_to_root(*n_iter*, *seg_id*)

Trace the given segment back to its starting point, returning a list of Segment objects describing the entire trajectory.

get_trajectory_roots(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments which start new trajectories. If min_iter or max_iter is specified, restrict the set of iterations within which the search is conducted.

get_initial_nodes(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments with which to begin a tree walk – those alive or created within [first_iter,last_iter].

trace_trajectories(*first_iter*, *last_iter*, *callable*, *include_pcoords=True*, *cargs=None*, *ckwargs=None*, *get_state=None*, *set_state=None*)

Walk the trajectory tree depth-first, calling
`callable(segment, children, history, *cargs, **ckwargs)` for each segment visited. `segment` is the segment being visited, `children` is that segment's children, `history` is the chain of segments leading to `segment` (not including `segment`). get_state and set_state are used to record and reset, respectively, any state specific to `callable` when a new branch is traversed.

6.5.4.2.12 westpa.oldtools.aframe.transitions module

class westpa.oldtools.aframe.transitions.**AnalysisMixin**

Bases: object

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

class westpa.oldtools.aframe.transitions.**TrajWalker**(*data_reader*, *history_chunksize=100*)

Bases: object

A class to perform analysis by walking the trajectory tree. A stack is used rather than recursion, or else the highest number of iterations capable of being considered would be the same as the Python recursion limit.

trace_to_root(*n_iter*, *seg_id*)

Trace the given segment back to its starting point, returning a list of Segment objects describing the entire trajectory.

get_trajectory_roots(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments which start new trajectories. If min_iter or max_iter is specified, restrict the set of iterations within which the search is conducted.

get_initial_nodes(*first_iter*, *last_iter*, *include_pcoords=True*)

Get segments with which to begin a tree walk – those alive or created within [first_iter,last_iter].

trace_trajectories(*first_iter*, *last_iter*, *callable*, *include_pcoords=True*, *cargs=None*, *ckwargs=None*, *get_state=None*, *set_state=None*)

Walk the trajectory tree depth-first, calling `callable(segment, children, history, *cargs, **ckwargs)` for each segment visited. `segment` is the segment being visited, `children` is that segment's children, `history` is the chain of segments leading to `segment` (not including `segment`). get_state and set_state are used to record and reset, respectively, any state specific to `callable` when a new branch is traversed.

class westpa.oldtools.aframe.transitions.**TransitionEventAccumulator**(*n_bins*, *output_group*, *calc_fpts=True*)

Bases: `object`

index_dtype

alias of `uint64`

count_dtype

alias of `uint64`

weight_dtype

alias of `float64`

output_tdat_chunksize = 4096

tdat_buffersize = 524288

max_acc = 32768

clear()

clear_state()

get_state()

set_state(*state_dict*)

record_transition_data(*tdat*)

Update running statistics and write transition data to HDF5 (with buffering)

flush_transition_data()

Flush any unwritten output that may be present

start_accumulation(*assignments*, *weights*, *bin_pops*, *traj=0*, *n_iter=0*)

continue_accumulation(*assignments*, *weights*, *bin_pops*, *traj=0*, *n_iter=0*)

class westpa.oldtools.aframe.transitions.**TransitionAnalysisMixin**

Bases: *AnalysisMixin*

require_transitions_group()

delete_transitions_group()

get_transitions_ds()

add_args(*parser*, *upcall=True*)

process_args(*args*, *upcall=True*)

require_transitions()

find_transitions()

class westpa.oldtools.aframe.transitions.**BFTransitionAnalysisMixin**

Bases: *TransitionAnalysisMixin*

require_transitions()

find_transitions(*chunksize=65536*)

6.5.4.3 westpa.oldtools.cmds package

6.5.4.3.1 westpa.oldtools.cmds module

6.5.4.3.2 westpa.oldtools.cmds.w_ttimes module

6.5.4.4 westpa.oldtools.stats package

6.5.4.4.1 westpa.oldtools.stats module

class westpa.oldtools.stats.**RunningStatsAccumulator**(*shape*, *dtype=<class 'numpy.float64'>*, *count_dtype=<class 'numpy.uint64'>*, *weight_dtype=<class 'numpy.float64'>*, *mask_value=nan*)

Bases: `object`

incorporate(*index*, *value*, *weight*)

average()

mean()

std()

6.5.4.4.2 westpa.oldtools.stats.accumulator module

class westpa.oldtools.stats.accumulator.**RunningStatsAccumulator**(*shape*, *dtype=<class 'numpy.float64'>*, *count_dtype=<class 'numpy.uint64'>*, *weight_dtype=<class 'numpy.float64'>*, *mask_value=nan*)

Bases: object

incorporate(*index*, *value*, *weight*)

average()

mean()

std()

6.5.4.4.3 westpa.oldtools.stats.edfs module

class westpa.oldtools.stats.edfs.**EDF**(*values*, *weights=None*)

Bases: object

A class for creating and manipulating empirical distribution functions (cumulative distribution functions derived from sample data).

Construct a new EDF from the given values and (optionally) weights.

static from_array(*array*)

static from_arrays(*x*, *F*)

as_array()

Return this EDF as a (N,2) array, where N is the number of unique values passed to the constructor. Numpy type casting rules are applied (so, for instance, integral abcissae are converted to floating-point values).

quantiles(*p*)

Treating the EDF as a quantile function, return the values of the (statistical) variable whose probabilities are at least p. That is, Q(p) = inf {x: p <= F(x) }.

quantile(*p*)

median()

moment(*n*)

Calculate the nth moment of this probability distribution

<x^n> = int_{-inf}^{inf} x^n dF(x)

cmoment(*n*)

Calculate the nth central moment of this probability distribution

mean()

var()

Return the second central moment of this probability distribution.

std()

Return the standard deviation (root of the variance) of this probability distribution.

6.5.4.4.4 westpa.oldtools.stats.mcbs module

Tools for Monte Carlo bootstrap error analysis

`westpa.oldtools.stats.mcbs.`**`add_mcbs_options`**(*parser*)

Add arguments concerning Monte Carlo bootstrap (`confidence` and `bssize`) to the given parser

`westpa.oldtools.stats.mcbs.`**`get_bssize`**(*alpha*)

Return a bootstrap data set size appropriate for the given confidence level

`westpa.oldtools.stats.mcbs.`**`bootstrap_ci`**(*estimator*, *data*, *alpha*, *n_sets=None*, *args=()*, *kwargs={}*, *sort=<function msort>*, *extended_output=False*)

Perform a Monte Carlo bootstrap of a (1-alpha) confidence interval for the given `estimator`. Returns (fhat, ci_lower, ci_upper), where fhat is the result of `estimator(data, *args, **kwargs)`, and `ci_lower` and `ci_upper` are the lower and upper bounds of the surrounding confidence interval, calculated by calling `estimator(syndata, *args, **kwargs)` on each synthetic data set `syndata`. If `n_sets` is provided, that is the number of synthetic data sets generated, otherwise an appropriate size is selected automatically (see `get_bssize()`).

`sort`, if given, is applied to sort the results of calling `estimator` on each synthetic data set prior to obtaining the confidence interval.

Individual entries in synthetic data sets are selected by the first index of `data`, allowing this function to be used on arrays of multidimensional data.

If `extended_output` is True (by default not), instead of returning (fhat, lb, ub), this function returns (fhat, lb, ub, ub-lb, abs((ub-lb)/fhat), and max(ub-fhat,fhat-lb)) (that is, the estimated value, the lower and upper bounds of the confidence interval, the width of the confidence interval, the relative width of the confidence interval, and the symmetrized error bar of the confidence interval).

6.6 westpa.westext package

6.6.1 Currently Supported

6.6.1.1 westpa.westext.adaptvoronoi package

6.6.1.1.1 Submodules

6.6.1.1.2 westpa.westext.adaptvoronoi.adaptVor_driver module

`westpa.westext.adaptvoronoi.adaptVor_driver.`**`check_bool`**(*value*, *action='warn'*)

Check that the given `value` is boolean in type. If not, either raise a warning (if `action=='warn'`) or an exception (`action=='raise'`).

exception `westpa.westext.adaptvoronoi.adaptVor_driver.`**`ConfigItemMissing`**(*key*, *message=None*)

Bases: `KeyError`

class `westpa.westext.adaptvoronoi.adaptVor_driver.`**`VoronoiBinMapper`**(*dfunc*, *centers*, *dfargs=None*, *dfkwargs=None*)

Bases: *BinMapper*

A one-dimensional mapper which assigns a multidimensional pcoord to the closest center based on a distance metric. Both the list of centers and the distance function must be supplied.

assign(*coords*, *mask=None*, *output=None*)

class westpa.westext.adaptvoronoi.adaptVor_driver.**AdaptiveVoronoiDriver**(*sim_manager*, *plugin_config*)

Bases: object

This plugin implements an adaptive scheme using voronoi bins from Zhang 2010, J Chem Phys, 132. The options exposed to the configuration file are:

- av_enabled (bool, default False): Enables adaptive binning
- max_centers (int, default 10): The maximum number of voronoi centers to be placed
- walk_count (integer, default 5): Number of walkers per voronoi center
- center_freq (ingeter, default 1): Frequency of center placement
- priority (integer, default 1): Priority in the plugin order
- **dfunc_method (function, non-optional, no default): Non-optional user defined** function that will be used to calculate distances between voronoi centers and data points
- **mapper_func (function, optional): Optional user defined function for building bin** mappers for more complicated binning schemes e.g. embedding the voronoi binning in a portion of the state space. If not defined the plugin will build a VoronoiBinMapper with the information it has.

dfunc()

Distance function to be used by the plugin. This function will be used to calculate the distance between each point.

get_dfunc_method(*plugin_config*)

get_mapper_func(*plugin_config*)

get_initial_centers()

This function pulls from the centers from either the previous bin mapper or uses the definition from the system to calculate the number of centers

update_bin_mapper()

Update the bin_mapper using the current set of voronoi centers

update_centers(*iter_group*)

Update the set of Voronoi centers according to Zhang 2010, J Chem Phys, 132. A short description of the algorithm can be found in the text:

1) First reference structure is chosen randomly from the first set of given structure 2) Given a set of n reference structures, for each configuration in the iteration the distances to each reference structure is calculated and the minimum distance is found 3) The configuration with the minimum distance is selected as the next reference

prepare_new_iteration()

6.6.1.1.3 Module contents

class westpa.westext.adaptvoronoi.**AdaptiveVoronoiDriver**(*sim_manager*, *plugin_config*)

Bases: object

This plugin implements an adaptive scheme using voronoi bins from Zhang 2010, J Chem Phys, 132. The options exposed to the configuration file are:

- av_enabled (bool, default False): Enables adaptive binning
- max_centers (int, default 10): The maximum number of voronoi centers to be placed
- walk_count (integer, default 5): Number of walkers per voronoi center
- center_freq (ingeter, default 1): Frequency of center placement
- priority (integer, default 1): Priority in the plugin order

- **dfunc_method (function, non-optional, no default): Non-optional user defined**
  function that will be used to calculate distances between voronoi centers and data points
- **mapper_func (function, optional): Optional user defined function for building bin**
  mappers for more complicated binning schemes e.g. embedding the voronoi binning in a portion of the state space. If not defined the plugin will build a VoronoiBinMapper with the information it has.

dfunc()

Distance function to be used by the plugin. This function will be used to calculate the distance between each point.

get_dfunc_method(*plugin_config*)

get_mapper_func(*plugin_config*)

get_initial_centers()

This function pulls from the centers from either the previous bin mapper or uses the definition from the system to calculate the number of centers

update_bin_mapper()

Update the bin_mapper using the current set of voronoi centers

update_centers(*iter_group*)

Update the set of Voronoi centers according to Zhang 2010, J Chem Phys, 132. A short description of the algorithm can be found in the text:

1) First reference structure is chosen randomly from the first set of given structure 2) Given a set of n reference structures, for each configuration in the iteration the distances to each reference structure is calculated and the minimum distance is found 3) The configuration with the minimum distance is selected as the next reference

prepare_new_iteration()

6.6.1.2 westpa.westext.stringmethod package

6.6.1.2.1 Submodules

6.6.1.2.2 westpa.westext.stringmethod.fourier_fitting module

6.6.1.2.3 westpa.westext.stringmethod.string_driver module

6.6.1.2.4 westpa.westext.stringmethod.string_method module

6.6.1.2.5 Module contents

6.6.1.3 westpa.westext.hamsm_restarting package

6.6.1.3.1 Description

This plugin leverages haMSM analysis *[1]* to provide simulation post-analysis. This post-analysis can be used on its own, or can be used to initialize and run new WESTPA simulations using structures in the haMSM's best estimate of steady-state as described in *[2]*, which may accelerate convergence to steady-state.

haMSM analysis is performed using the msm_we library.

Sample files necessary to run the restarting plugin (as described below) can be found in the WESTPA GitHub Repo.

6.6.1.3.2 Usage

Configuration

west.cfg

This plugin requires the following section in `west.cfg` (or whatever your WE configuration file is named):

```
west:
  plugins:
  - plugin: westpa.westext.hamsm_restarting.restart_driver.RestartDriver
    n_restarts: 0             # Number of restarts to perform
    n_runs: 5                 # Number of runs within each restart
    n_restarts_to_use: 0.5    # Amount of prior restarts' data to use. -1, a decimal in␣
↪(0,1), or an integer. Details below.
    extension_iters: 5        # Number of iterations to continue runs for, if target is␣
↪not reached by first restart period
    coord_len: 2                                        # Length of pcoords returned
    initialization_file: restart_initialization.json  # JSON describing w_run parameters␣
↪for new runs
    ref_pdb_file: common_files/bstate.pdb             # File containing reference␣
↪structure/topology
    model_name: NaClFlux                                # Name for msm_we model
    n_clusters: 2                                       # Number of clusters in haMSM␣
↪building
    we_folder: .                                        # Should point to the same␣
↪directory as WEST_SIM_ROOT
    target_pcoord_bounds: [[-inf, 2.60]]              # Progress coordinate boundaries␣
↪for the target state
    basis_pcoord_bounds: [[12.0, inf]]               # Progress coordinate boundaries␣
↪for the basis state
    tau: 5e-13                                          # Resampling time, i.e. length of␣
↪a WE iteration in physical units
    pcoord_ndim0: 1                                     # Dimensionality of progress␣
↪coordinate
    dim_reduce_method: pca                              # Dimensionality reduction scheme,␣
↪either "pca", "vamp", or "none"
    parent_traj_filename: parent.xml                    # Name of parent file in each␣
↪segment
    child_traj_filename: seg.xml                        # Name of child file in each␣
↪segment
    user_functions: westpa_scripts/restart_overrides.py       # Python file defining␣
↪coordinate processing
    struct_filetype: mdtraj.formats.PDBTrajectoryFile         # Filetype for output␣
↪start-structures
    debug: False              # Optional, defaults to False. If true, enables debug-mode␣
↪logging.
    streaming: True           # Does clustering in a streaming fashion, versus trying to␣
↪load all coords in memory
    n_cpus: 1                 # Number of CPUs to use for parallel calculations
```

Some sample parameters are provided in the above, but of course should be modified to your specific system.

Note about `restarts_to_use` **:** `restarts_to_use` can be specified in a few different ways. A value of `-1`

means to use all available data. A decimal `0 < restarts_to_use < 1` will use the last `restarts_to_use * current_restart` iterations of data – so, for example, set to 0.5 to use the last half of the data, or 0.75 to use the last 3/4. Finally, and integer value will just use the last `restarts_to_use` iterations.

Note that ref_pdb_file can be any filetype supported by `msm_we.initialize()`'s structure loading. At the time of writing, this is limited to PDB, however that is planned to be extended. Also at the time of writing, that's only used to set model.nAtoms, so if you're using some weird topology that's unsupported, you should be able to scrap that and manually set nAtoms on the object.

Also in this file, `west.data.data_refs.basis_state` MUST point to `$WEST_SIM_ROOT/{basis_state.auxref}` and not a subdirectory if restarts are being used. This is because when the plugin initiates a restart, start_state references in `$WEST_SIM_ROOT/restartXX/start_states.txt` are set relative to `$WEST_SIM_ROOT`. All basis/start state references are defined relative to `west.data.data_refs.basis_state`, so if that points to a subdirectory of `$WEST_SIM_ROOT`, those paths will not be accurate.

Running

Once configured, just run your WESTPA simulation normally with `w_run`, and the plugin will automatically handle performing restarts, and extensions if necessary.

6.6.1.3.3 Extensions

To be clear: these are extensions in the sense of extending a simulation to be longer – not in the sense of "an extension to the WESTPA software package"!

Running with `extension_iters` greater than 0 will enable extensions before the first restart if the target state is not reached. This is useful to avoid restarting when you don't yet have structures spanning all the way from your basis to target. At the time of writing, it's not yet clear whether restarting from "incomplete" WE runs like this will help or hinder the total number of iterations it takes to reach the target.

Extensions are simple and work as follows: before doing the first restart, after all runs are complete, the output WESTPA h5 files are scanned to see if any recycling has occurred. If it hasn't, then each run is extended by `extension_iters` iterations.

restart_initialization.json

```
{
    "bstates":["start,1,bstates/bstate.pdb"],
    "tstates":["bound,2.6"],
    "bstate-file":"bstates/bstates.txt",
    "tstate-file" :"tstate.file",
    "segs-per-state": 1
}
```

It is not necessary to specify both in-line states and a state-file for each, but that is shown in the sample for completeness.

It is important that `bstates` and `tstates` are lists of strings, and not just strings, even if only one bstate/tstate is being used!

With `n_runs > 1`, before doing any restart, multiple independent runs are performed. However, before the first restart (this applies if no restarts are performed as well), the plugin has no way of accessing the parameters that were initially passed to `w_init` and `w_run`.

Therefore, it is necessary to store those parameters in a file, so the plugin can read them and initiate subsequent runs.

After the first restart is performed, the plugin writes this file itself, so it is only necessary to manually configure for that first set of runs.

Featurization overrides

```
import numpy as np
import mdtraj as md

def processCoordinates(self, coords):
        log.debug("Processing coordinates")

        if self.dimReduceMethod == "none":
            nC = np.shape(coords)
            nC = nC[0]
            ndim = 3 * self.nAtoms
            data = coords.reshape(nC, 3 * self.nAtoms)
            return data

        if self.dimReduceMethod == "pca" or self.dimReduceMethod == "vamp":

            ### NaCl RMSD dimensionality reduction
            log.warning("Hardcoded selection: Doing dim reduction for Na, Cl. This is␣
↪only for testing!")
            indNA = self.reference_structure.topology.select("element Na")
            indCL = self.reference_structure.topology.select("element Cl")

            diff = np.subtract(coords[:, indNA], coords[:, indCL])

            dist = np.array(np.sqrt(
                np.mean(
                    np.power(
                        diff,
                        2)
                , axis=-1)
            ))

            return dist
```

This is the file whose path is provided in the configuration file in `plugin.user_functions`, and must be a Python file defining a function named `processCoordinates(self, coords)` which takes a numpy array of coordinates, featurizes it, and returns the numpy array of feature-coordinates.

This is left to be user-provided because whatever featurization you do will be system-specific. The provided function is monkey-patched into the `msm_we.modelWE` class.

An example is provided above, which does a simple RMSD coordinate reduction for the NaCl association tutorial system.

Doing only post-analysis

If you want to ONLY use this for haMSM post-analysis, and not restarting, just set `n_restarts: 0` in the configuration.

Work manager for restarting

If you're using some parallelism (which you should), and you're using the plugin to do restarts or multiple runs, then your choice of work manager can be important. This plugin handles starting new WESTPA runs using the Python API. The process work manager, by default, uses `fork` to start new workers which seems to eventually causes memory issues, since `fork` passes the entire contents of the parent to each child. Switching the spawn method to `forkserver` or `spawn` may introduce other issues.

Using the ZMQ work manager works well. The MPI work manager should also work well, though is untested. Both of these handle starting new workers in a more efficient way, without copying the full state of the parent.

Continuing a failed run

The restarting plugin has a few different things it expects to find when it runs. Crashes during the WE run should not affect this. However, if the plugin itself crashes while running, these may be left in a weird state.

If the plugin crashes while running, make sure:

- `restart.dat` contains the correct entries. `restarts_completed` is the number of restarts *successfully* completed, and same for `runs_completed` within that restart.
- `restart_initialization.json` is pointing to the correct restart

It may help to `w_truncate` the very last iteration and allow WESTPA to re-do it.

Potential Pitfalls/Troubleshooting

- Basis state calculation may take a LONG time with a large number of start-states. A simple RMSD calculation using cpptraj and 500,000 start-states took over 6 hours. Reducing the number of runs used through `n_restarts_to_use` will ameliorate this.
- If `restart_driver.prepare_coordinates()` has written a coordinate for an iteration, subsequent runs will NOT overwrite it, and will skip it.
- In general: verify that msm_we is installed
- Verify that `restart_initialization.json` has been correctly set
- This plugin does not yet attempt to resolve environment variables in the config, so things like say, $WEST_SIM_ROOT, will be interpreted literally in paths

References

[1] Suárez, E., Adelman, J. L. & Zuckerman, D. M. Accurate Estimation of Protein Folding and Unfolding Times: Beyond Markov State Models. J Chem Theory Comput 12, 3473–3481 (2016).

[2] Copperman, J. & Zuckerman, D. M. Accelerated Estimation of Long-Timescale Kinetics from Weighted Ensemble Simulation via Non-Markovian "Microbin" Analysis. J Chem Theory Comput 16, 6763–6775 (2020).

6.6.2 Depreciated

6.6.2.1 westpa.westext.weed package

6.6.2.1.1 Submodules

6.6.2.1.2 westpa.westext.weed.BinCluster module

class westpa.westext.weed.BinCluster.**ClusterList**(*ratios*, *nbins*)

Bases: object

join(*pairs*)

Join clusters given a tuple (i,j) of bin pairs

join_simple(*pairs*)

Join clusters using direct ratios given a tuple (i,j) of bin pairs

6.6.2.1.3 westpa.westext.weed.ProbAdjustEquil module

westpa.westext.weed.ProbAdjustEquil.**probAdjustEquil**(*binProb*, *rates*, *uncert*, *threshold=0.0*, *fullCalcClust=False*, *fullCalcBins=False*)

This function adjusts bin pops in binProb using rates and uncert matrices fullCalcBins –> True for weighted avg, False for simple calc fullCalcClust –> True for weighted avg, False for simple calc threshold –> minimum weight (relative to max) for another value to be averaged

only matters if fullCalcBins == True (or later perhaps if fullCalcClust == True)

6.6.2.1.4 westpa.westext.weed.UncertMath module

class westpa.westext.weed.UncertMath.**UncertContainer**(*vals*, *vals_dmin*, *vals_dmax*, *mask=False*)

Bases: object

Container to hold uncertainty measurements. Data is convert to np masked arrays to avoid possible numerical problems

transpose()

recip()

update_mask()

concatenate(*value*, *axis=0*)

Concatentate UncertContainer value to self. Assumes that if dimensions of self and value do not match, to add a np.newaxis along axis of value

weighted_average(*axis=0*, *expaxis=None*)

Calculate weighted average of data along axis after optionally inserting a new dimension into the shape array at position expaxis

6.6.2.1.5 westpa.westext.weed.weed_driver module

westpa.westext.weed.weed_driver.**check_bool**(*value*, *action='warn'*)

Check that the given `value` is boolean in type. If not, either raise a warning (if `action=='warn'`) or an exception (`action=='raise'`).

class westpa.westext.weed.weed_driver.**RateAverager**(*bin_mapper*, *system=None*, *data_manager=None*, *work_manager=None*)

Bases: `object`

Calculate bin-to-bin kinetic properties (fluxes, rates, populations) at 1-tau resolution

extract_data(*iter_indices*)

Extract data from the data_manger and place in dict mirroring the same underlying layout.

task_generator(*iter_start*, *iter_stop*, *block_size*)

calculate(*iter_start=None*, *iter_stop=None*, *n_blocks=1*, *queue_size=1*)

Read the HDF5 file and collect flux matrices and population vectors for each bin for each iteration in the range [iter_start, iter_stop). Break the calculation into n_blocks blocks. If the calculation is broken up into more than one block, queue_size specifies the maxmimum number of tasks in the work queue.

westpa.westext.weed.weed_driver.**probAdjustEquil**(*binProb*, *rates*, *uncert*, *threshold=0.0*, *fullCalcClust=False*, *fullCalcBins=False*)

This function adjusts bin pops in binProb using rates and uncert matrices fullCalcBins –> True for weighted avg, False for simple calc fullCalcClust –> True for weighted avg, False for simple calc threshold –> minimum weight (relative to max) for another value to be averaged
only matters if fullCalcBins == True (or later perhaps if fullCalcClust == True)

westpa.westext.weed.weed_driver.**bins_from_yaml_dict**(*bin_dict*)

class westpa.westext.weed.weed_driver.**WEEDDriver**(*sim_manager*, *plugin_config*)

Bases: `object`

get_rates(*n_iter*, *mapper*)

Get rates and associated uncertainties as of n_iter, according to the window size the user has selected (self.windowsize)

prepare_new_iteration()

6.6.2.1.6 Module contents

westext.weed – Support for weighted ensemble equilibrium dynamics

Initial code by Dan Zuckerman (May 2011), integration by Matt Zwier, and testing by Carsen Stringer. Re-factoring and optimization of probability adjustment routines by Joshua L. Adelman (January 2012).

westpa.westext.weed.**probAdjustEquil**(*binProb*, *rates*, *uncert*, *threshold=0.0*, *fullCalcClust=False*, *fullCalcBins=False*)

This function adjusts bin pops in binProb using rates and uncert matrices fullCalcBins –> True for weighted avg, False for simple calc fullCalcClust –> True for weighted avg, False for simple calc threshold –> minimum weight (relative to max) for another value to be averaged

only matters if fullCalcBins == True (or later perhaps if fullCalcClust == True)

class westpa.westext.weed.**WEEDDriver**(*sim_manager*, *plugin_config*)

Bases: object

get_rates(*n_iter*, *mapper*)

Get rates and associated uncertainties as of n_iter, according to the window size the user has selected (self.windowsize)

prepare_new_iteration()

6.6.2.2 westpa.westext.wess package

6.6.2.2.1 Submodules

6.6.2.2.2 westpa.westext.wess.ProbAdjust module

westpa.westext.wess.ProbAdjust.**solve_steady_state**(*T*, *U*, *target_bins_index*)

westpa.westext.wess.ProbAdjust.**prob_adjust**(*binprob*, *rates*, *uncert*, *oldindex*, *targets=[]*)

6.6.2.2.3 westpa.westext.wess.wess_driver module

westpa.westext.wess.wess_driver.**check_bool**(*value*, *action='warn'*)

Check that the given value is boolean in type. If not, either raise a warning (if action=='warn') or an exception (action=='raise').

class westpa.westext.wess.wess_driver.**RateAverager**(*bin_mapper*, *system=None*, *data_manager=None*, *work_manager=None*)

Bases: object

Calculate bin-to-bin kinetic properties (fluxes, rates, populations) at 1-tau resolution

extract_data(*iter_indices*)

Extract data from the data_manger and place in dict mirroring the same underlying layout.

task_generator(*iter_start*, *iter_stop*, *block_size*)

calculate(*iter_start=None*, *iter_stop=None*, *n_blocks=1*, *queue_size=1*)

Read the HDF5 file and collect flux matrices and population vectors for each bin for each iteration in the range [iter_start, iter_stop). Break the calculation into n_blocks blocks. If the calculation is broken up into more than one block, queue_size specifies the maxmimum number of tasks in the work queue.

westpa.westext.wess.wess_driver.**prob_adjust**(*binprob*, *rates*, *uncert*, *oldindex*, *targets=[]*)

westpa.westext.wess.wess_driver.**bins_from_yaml_dict**(*bin_dict*)

westpa.westext.wess.wess_driver.**reduce_array**(*Aij*)

Remove empty rows and columns from an array Aij and return the reduced array Bij and the list of non-empty states

class westpa.westext.wess.wess_driver.**WESSDriver**(*sim_manager*, *plugin_config*)

Bases: object

get_rates(*n_iter*, *mapper*)

Get rates and associated uncertainties as of n_iter, according to the window size the user has selected (self.windowsize)

prepare_new_iteration()

6.6.2.2.4 Module contents

westpa.westext.wess.**prob_adjust**(*binprob*, *rates*, *uncert*, *oldindex*, *targets=[]*)

class westpa.westext.wess.**WESSDriver**(*sim_manager*, *plugin_config*)

Bases: object

get_rates(*n_iter*, *mapper*)

Get rates and associated uncertainties as of n_iter, according to the window size the user has selected (self.windowsize)

prepare_new_iteration()

6.6.3 Module contents

6.7 westpa.analysis package

This subpackage provides an API to facilitate the analysis of WESTPA simulation data. Its core abstraction is the Run class. A Run instance provides a read-only view of a WEST HDF5 ("west.h5") file.

API reference: https://westpa.readthedocs.io/en/latest/documentation/analysis/

6.7.1 How To

Open a run:

```
>>> from westpa.analysis import Run
>>> run = Run.open('west.h5')
>>> run
<WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>
```

Iterate over iterations and walkers:

```
>>> for iteration in run:
...     for walker in iteration:
...         pass
...
```

Access a particular iteration:

```
>>> iteration = run.iteration(10)
>>> iteration
Iteration(10, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
```

Access a particular walker:

```
>>> walker = iteration.walker(4)
>>> walker
Walker(4, Iteration(10, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
```

Get the weight and progress coordinate values of a walker:

```
>>> walker.weight
9.876543209876543e-06
>>> walker.pcoords
array([[3.1283207],
       [3.073721 ],
       [2.959221 ],
       [2.6756208],
       [2.7888207]], dtype=float32)
```

Get the parent and children of a walker:

```
>>> walker.parent
Walker(2, Iteration(9, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
>>> for child in walker.children:
...     print(child)
...
Walker(0, Iteration(11, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(1, Iteration(11, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(2, Iteration(11, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(3, Iteration(11, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(4, Iteration(11, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
```

Trace the ancestry of a walker:

```
>>> trace = walker.trace()
>>> trace
Trace(Walker(4, Iteration(10, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>)))
>>> for walker in trace:
...     print(walker)
...
Walker(1, Iteration(1, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(4, Iteration(2, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(5, Iteration(3, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(6, Iteration(4, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(9, Iteration(5, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(8, Iteration(6, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(8, Iteration(7, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(13, Iteration(8, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(2, Iteration(9, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
Walker(4, Iteration(10, <WESTPA Run with 500 iterations at 0x7fcaf8f0d5b0>))
```

Close a run (and its underlying HDF5 file):

```
>>> run.close()
>>> run
<Closed WESTPA Run at 0x7fcaf8f0d5b0>
```

(continues on next page)

(continued from previous page)

```
>>> run.h5file
<Closed HDF5 file>
```

6.7.2 Retrieving Trajectories

6.7.2.1 Built-in Reader

MD trajectory data stored in an identical manner as in the Basic NaCl tutorial may be retrieved using the built-in `BasicMDTrajectory` reader with its default settings:

```
>>> from westpa.analysis import BasicMDTrajectory
>>> trajectory = BasicMDTrajectory()
```

Here `trajectory` is a callable object that takes either a `Walker` or a `Trace` instance as input and returns an MDTraj Trajectory:

```
>>> traj = trajectory(walker)
>>> traj
<mdtraj.Trajectory with 5 frames, 33001 atoms, 6625 residues, and unitcells at␣
↪0x7fcae484ad00>
>>> traj = trajectory(trace)
>>> traj
<mdtraj.Trajectory with 41 frames, 33001 atoms, 6625 residues, and unitcells at␣
↪0x7fcae487c790>
```

Minor variations of the "basic" trajectory storage protocol (e.g., use of different file formats) can be handled by changing the parameters of the `BasicMDTrajectory` reader. For example, suppose that instead of storing the coordinate and topology data for trajectory segments in separate files ("seg.dcd" and "bstate.pdb"), we store them together in a MDTraj HDF5 trajectory file ("seg.h5"). This change can be accommodated by explicitly setting the `traj_ext` and `top` parameters of the trajectory reader:

```
>>> trajectory = BasicMDTrajectory(traj_ext='.h5', top=None)
```

Trajectories that are saved with the HDF5 Framework can use `HDF5MDTrajectory` reader instead.

6.7.2.2 Custom Readers

For users requiring greater flexibility, custom trajectory readers can be implemented using the `westpa.analysis.Trajectory` class. Implementing a custom reader requires two ingredients:

1. A function for retrieving individual trajectory segments. The function must take a `Walker` instance as its first argument and return a sequence (e.g., a list, NumPy array, or MDTraj Trajectory) representing the trajectory of the walker. Moreover, it must accept a Boolean keyword argument `include_initpoint`, which specifies whether the returned trajectory includes its initial point.
2. A function for concatenating trajectory segments. A default implementation is provided by the `concatenate()` function in the `westpa.analysis.trajectories` module.

6.7.3 westpa.analysis.core module

class westpa.analysis.core.**Run**(*h5filename='west.h5'*)

A read-only view of a WESTPA simulation run.

Parameters

h5filename (*str or file-like object, default 'west.h5'*) – Pathname or stream of a main WESTPA HDF5 data file.

classmethod open(*h5filename='west.h5'*)

Alternate constructor.

Parameters

h5filename (*str or file-like object, default 'west.h5'*) – Pathname or stream of a main WESTPA HDF5 data file.

close()

Close the Run instance by closing the underlying WESTPA HDF5 file.

property closed

Whether the Run instance is closed.

Type

bool

property summary

Summary data by iteration.

Type

pd.DataFrame

property num_iterations

Number of completed iterations.

Type

int

property iterations

Sequence of iterations.

Type

Sequence[*Iteration*]

property num_walkers

Total number of walkers.

Type

int

property num_segments

Total number of trajectory segments (alias self.num_walkers).

Type

int

property walkers

All walkers in the run.

Type

Iterable[*Walker*]

property recycled_walkers

Walkers that stopped in the sink.

Type

Iterable[*Walker*]

property initial_walkers

Walkers whose parents are initial states.

Type

Iterable[*Walker*]

iteration(*number*)

Return a specific iteration.

Parameters

number (*int*) – Iteration number (1-based).

Returns

The iteration indexed by *number*.

Return type

Iteration

class westpa.analysis.core.Iteration(*number*, *run*)

An iteration of a WESTPA simulation.

Parameters

- **number** (*int*) – Iteration number (1-based).
- **run** (Run) – Simulation run to which the iteration belongs.

property h5group

HDF5 group containing the iteration data.

Type

h5py.Group

property prev

Previous iteration.

Type

Iteration

property next

Next iteration.

Type

Iteration

property summary

Iteration summary.

Type

pd.DataFrame

property segment_summaries

Segment summary data for the iteration.

Type

pd.DataFrame

property pcoords

Progress coordinate snaphots of each walker.

Type

3D ndarray

property weights

Statistical weight of each walker.

Type

1D ndarray

property bin_target_counts

Target count for each bin.

Type

1D ndarray, dtype=uint64

property bin_mapper

Bin mapper used in the iteration.

Type

BinMapper

property num_bins

Number of bins.

Type

int

property bins

Bins.

Type

Iterable[*Bin*]

property num_walkers

Number of walkers in the iteration.

Type

int

property num_segments

Number of trajectory segments (alias self.num_walkers).

Type

int

property walkers

Walkers in the iteration.

Type

Iterable[*Walker*]

property recycled_walkers

Walkers that stopped in the sink.

Type

Iterable[*Walker*]

property initial_walkers

Walkers whose parents are initial states.

Type

Iterable[*Walker*]

property auxiliary_data

Auxiliary data stored for the iteration.

Type

h5py.Group or None

property basis_state_summaries

Basis state summary data.

Type

pd.DataFrame

property basis_state_pcoords

Progress coordinates of each basis state.

Type

2D ndarray

property basis_states

Basis states in use for the iteration.

Type

list[*BasisState*]

property has_target_states

Whether target (sink) states are defined for this iteration.

Type

bool

property target_state_summaries

Target state summary data.

Type

pd.DataFrame or None

property target_state_pcoords

Progress coordinates of each target state.

Type

2D ndarray or None

property target_states

Target states in use for the iteration.

Type

list[*TargetState*]

property sink

Union of bins serving as the recycling sink.

Type

BinUnion or None

bin(*index*)

Return the bin with the given index.

Parameters

index (*int*) – Bin index (0-based).

Returns

The bin indexed by *index*.

Return type

Bin

walker(*index*)

Return the walker with the given index.

Parameters

index (*int*) – Walker index (0-based).

Returns

The walker indexed by *index*.

Return type

Walker

basis_state(*index*)

Return the basis state with the given index.

Parameters

index (*int*) – Basis state index (0-based).

Returns

The basis state indexed by *index*.

Return type

BasisState

target_state(*index*)

Return the target state with the given index.

Parameters

index (*int*) – Target state index (0-based).

Returns

The target state indexed by *index*.

Return type

TargetState

class westpa.analysis.core.**Walker**(*index*, *iteration*)

A walker in an iteration of a WESTPA simulation.

Parameters

- **index** (*int*) – Walker index (0-based).
- **iteration** (Iteration) – Iteration to which the walker belongs.

property run

Run to which the walker belongs.

Type

Run

property weight

Statistical weight of the walker.

Type

float64

property pcoords

Progress coordinate snapshots.

Type

2D ndarray

property num_snapshots

Number of snapshots.

Type

int

property segment_summary

Segment summary data.

Type

pd.Series

property parent

The parent of the walker.

Type

Walker or *InitialState*

property children

The children of the walker.

Type

Iterable[*Walker*]

property recycled

True if the walker stopped in the sink, False otherwise.

Type

bool

property initial

True if the parent of the walker is an initial state, False otherwise.

Type

bool

property auxiliary_data

Auxiliary data for the walker.

Type

dict

trace(***kwargs*)

Return the trace (ancestral line) of the walker.

For full documentation see *Trace*.

Returns

The trace of the walker.

Return type

Trace

class westpa.analysis.core.**BinUnion**(*indices*, *mapper*)

A (disjoint) union of bins defined by a common bin mapper.

Parameters

- **indices** (*iterable of int*) – The indices of the bins comprising the union.
- **mapper** (BinMapper) – The bin mapper defining the bins.

union(**others*)

Return the union of the bin union and all others.

Parameters

***others** (BinUnion) – Other *BinUnion* instances, consisting of bins defined by the same underlying bin mapper.

Returns

The union of *self* and *others*.

Return type

BinUnion

intersection(**others*)

Return the intersection of the bin union and all others.

Parameters

***others** (BinUnion) – Other *BinUnion* instances, consisting of bins defined by the same underlying bin mapper.

Returns

The itersection of *self* and *others*.

Return type
BinUnion

class westpa.analysis.core.**Bin**(*index*, *mapper*)

A bin defined by a bin mapper.

Parameters

- **index** (*int*) – The index of the bin.
- **mapper** (BinMapper) – The bin mapper defining the bin.

class westpa.analysis.core.**Trace**(*walker*, *source=None*, *max_length=None*)

A trace of a walker's ancestry.

Parameters

- **walker** (Walker) – The terminal walker.
- **source** (Bin, BinUnion, *or collections.abc.Container*, *optional*) – A source (macro)state, specified as a container object whose `__contains__()` method is the indicator function for the corresponding subset of progress coordinate space. The trace is stopped upon encountering a walker that stopped in *source*.
- **max_length** (*int*, *optional*) – The maximum number of walkers in the trace.

6.7.4 westpa.analysis.trajectories module

class westpa.analysis.trajectories.**Trajectory**(*fget=None*, ***, *fconcat=None*)

A callable that returns the trajectory of a walker or trace.

Parameters

- **fget** (*callable*) – Function for retrieving a single trajectory segment. Must take a `Walker` instance as its first argument and accept a boolean keyword argument *include_initpoint*. The function should return a sequence (e.g., a list or ndarray) representing the trajectory of the walker. If *include_initpoint* is True, the trajectory segment should include its initial point. Otherwise, the trajectory segment should exclude its initial point.
- **fconcat** (*callable*, *optional*) – Function for concatenating trajectory segments. Must take a sequence of trajectory segments as input and return their concatenation. The default concatenation function is *concatenate()*.

property **segment_collector**

Segment retrieval manager.

Type
SegmentCollector

property **fget**

Function for getting trajectory segments.

Type
callable

property **fconcat**

Function for concatenating trajectory segments.

Type
callable

class westpa.analysis.trajectories.**SegmentCollector**(*trajectory*, *use_threads=False*, *max_workers=None*, *show_progress=False*)

An object that manages the retrieval of trajectory segments.

Parameters

- **trajectory** (Trajectory) – The trajectory to which the segment collector is attached.
- **use_threads** (*bool, default False*) – Whether to use a pool of threads to retrieve trajectory segments asynchronously. Setting this parameter to True may be may be useful when segment retrieval is an I/O bound task.
- **max_workers** (*int, optional*) – Maximum number of threads to use. The default value is specified in the ThreadPoolExecutor documentation.
- **show_progress** (*bool, default False*) – Whether to show a progress bar when retrieving multiple segments.

get_segments(*walkers*, *initpoint_mask=None*, ***kwargs*)

Retrieve the trajectories of multiple walkers.

Parameters

- **walkers** (*sequence of* Walker) – The walkers for which to retrieve trajectories.
- **initpoint_mask** (*sequence of bool, optional*) – A Boolean mask indicating whether each trajectory segment should include (True) or exclude (False) its initial point. Default is all True.

Returns

The trajectory of each walker.

Return type

list of sequences

class westpa.analysis.trajectories.**BasicMDTrajectory**(*top='bstate.pdb'*, *traj_ext='.dcd'*, *state_ext='.xml'*, *sim_root='.'*)

Trajectory reader for MD trajectories stored as in the Basic Tutorial.

Parameters

- **top** (*str or mdtraj.Topology, default 'bstate.pdb'*)
- **traj_ext** (*str, default '.dcd'*)
- **state_ext** (*str, default '.xml'*)
- **sim_root** (*str, default '.'*)

class westpa.analysis.trajectories.**HDF5MDTrajectory**

Trajectory reader for MD trajectories stored by the HDF5 framework.

westpa.analysis.trajectories.**concatenate**(*segments*)

Return the concatenation of a sequence of trajectory segments.

Parameters

segments (*sequence of sequences*) – A sequence of trajectory segments.

Returns

The concatenation of *segments*.

Return type

sequence

6.7.5 westpa.analysis.statistics module

westpa.analysis.statistics.**time_average**(*observable*, *iterations*)

Compute the time average of an observable.

Parameters

- **observable** (*Callable[[Walker], ArrayLike]*) – Function that takes a walker as input and returns a number or a fixed-size array of numbers.
- **iterations** (*Sequence[Iteration]*) – Sequence of iterations over which to compute the average.

Returns

The time average of *observable* over *iterations*.

Return type

ArrayLike

6.8 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.

6.8.1 Overall structure

```
/
    #ibstates/
        index
        naming
            bstate_index
            bstate_pcoord
            istate_index
            istate_pcoord
    #tstates/
        index
    bin_topologies/
        index
        pickles
    iterations/
        iter_XXXXXXXX/\|iter_XXXXXXXX/
            auxdata/
            bin_target_counts
            ibstates/
                bstate_index
                bstate_pcoord
                istate_index
                istate_pcoord
            pcoord
```

(continues on next page)

(continued from previous page)

```
            seg_index
            wtgraph
        ...
    summary
```

6.8.2 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

6.8.2.1 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

6.8.3 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_co	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)	

6.8.3.1 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	

6.8.4 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

6.9 Overview

6.10 Style Guide

6.10.1 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.

6.10.2 Style and Usage

6.10.2.1 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

6.10.2.2 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`.

6.10.2.3 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.

6.10.2.4 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.

6.10.2.5 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.

6.10.2.6 Non-English words

- Avoid Latin words and abbreviations.

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

6.10.2.7 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".

6.10.2.8 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`.

6.10.2.9 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`.

6.10.2.10 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.

6.10.2.11 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.

6.10.2.12 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.

6.10.3 Computer Language Elements

6.10.3.1 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)
  ```

6.10.3.2 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.

6.10.3.3 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`.

6.10.3.4 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.

6.10.3.5 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.

6.11 Source Code Management

6.12 Documentation Practices

6.12.1 Introduction to Editing the Sphinx Documentation

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

Make sure `sphinx` and `sphinx_rtd_theme` are installed on the system. The settings for the documentation are specified in `/westpa/doc/conf.py`. In order to successfully build the documentation, your system has to statisfy the minimum environment to install WESTPA.

The documentation may be built locally in the `_build` folder 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.

6.12.2 Uploading to ReadTheDocs

The online copy of WESTPA Sphinx documentation is hosted on ReadtheDocs. The Sphinx documentations on the main branch are updated whenever the main branch is updated, via a webhook setup on ReadtheDocs and `/westpa/.readthedocs.yml`. The environment used to build the documentation on the RTD servers are described in `/westpa/doc/doc_env.yaml`.

6.12.3 In Cases of Major Revisions in Code Base

Currently, each `.rst` file contains pre-written descriptions and autogenerated sections generated from docstrings via `automodule`. In cases where the WESTPA code base has significantly changed, the structure of the code base can be regenerated into the `test` folder by running the following command in the `doc` folder:

```
sphinx-apidoc -f -o test ../src/westpa
```

6.13 WESTPA Modules API

6.13.1 Binning

Bin assignment for WEST simulations. This module defines "bin mappers" which take vectors of coordinates (or rather, coordinate tuples), and assign each a definite integer value identifying a bin. Critical portions are implemented in a Cython extension module.

A number of pre-defined bin mappers are available here:

- *RectilinearBinMapper*, for bins divided by N-dimensional grids
- *FuncBinMapper*, for functions which directly calculate bin assignments for a number of coordinate values. This is best used with C/Cython/Numba functions, or intellegently-tuned numpy-based Python functions.
- *VectorizingFuncBinMapper*, for functions which calculate a bin assignment for a single coordinate value. This is best used for arbitrary Python functions.
- *PiecewiseBinMapper*, for using a set of boolean-valued functions, one per bin, to determine assignments. This is likely to be much slower than a *FuncBinMapper* or *VectorizingFuncBinMapper* equipped with an appropriate function, and its use is discouraged.

One "super-mapper" is available, for assembling more complex bin spaces from simpler components:

- *RecursiveBinMapper*, for nesting one set of bins within another.

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 (np.uint16) indicating a 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.

6.13.2 YAMLCFG

YAML-based configuration files for WESTPA

6.13.3 RC

class westpa.core._rc.**WESTRC**

A class, an instance of which is accessible as `westpa.rc`, to handle global issues for WEST-PA code, such as loading modules and plugins, writing output based on verbosity level, adding default command line options, and so on.

6.14 WESTPA Tools

6.15 WEST

6.15.1 Setup

6.15.1.1 Defining and Calculating Progress Coordinates

6.15.1.2 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.

6.15.1.2.1 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.

6.15.1.2.2 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`.

6.15.1.2.3 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

....
```

(continues on next page)

(continued from previous page)

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

6.15.1.2.4 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,))
```

6.15.1.2.5 PiecewiseBinMapper

6.15.1.2.6 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.

6.15.1.3 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`.

6.15.1.4 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.

6.15.1.5 Propagators

6.15.1.5.1 The Executable Propagator

6.15.1.5.2 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

6.15.1.6 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

6.15.1.7 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.

6.15.1.7.1 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

6.15.1.7.2 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_I	Integer >=1	Current iteration number
WEST_CURRENT_S	Integer >=0	Current segment ID
WEST_CURRENT_S	String	General-purpose reference, based on current segment information, configured in west.cfg. Usually used for storage paths
WEST_CURRENT_S	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_DA	String	General purpose reference, based on parent segment information, configured in west.cfg. Usually used for storage paths
WEST_PCOORD_RE	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.

6.15.1.7.3 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_D	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_DA	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_DA	get_pcoord for initial state, gen_istate	String	Initial state data references, usually a pathname
WEST_PCOORD_R	get_pcoord for basis or initial state	Pathname	Where progress coordinate data is expected to be found after execution

6.15.1.8 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

6.15.1.9 Weighted Ensemble Algorithm (Resampling)

6.15.2 Running

6.15.2.1 Overview

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

6.15.2.2 Setting simulation limits

6.15.2.3 Running a simulation

6.15.2.3.1 Running on a single node

6.15.2.3.2 Running on multiple nodes with MPI

6.15.2.3.3 Running on multiple nodes with ZeroMQ

6.15.2.4 Managing data

6.15.2.5 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.

6.15.3 Analysis

6.15.3.1 Gauging simulation progress and convergence

6.15.3.1.1 Progress coordinate distribution (w_pcpdist)

w_pcpdist and plothist

6.15.3.1.2 Kinetics for source/sink simulations

w_fluxanl

6.15.3.1.3 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.

6.16 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.

6.16.1 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.

6.16.1.1 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_truncat	Truncates the weighted ensemble simulation from a given iteration.

6.16.1.2 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_fluxani	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*).

6.16.2 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
```

6.16.2.1 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`

6.16.3 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.

6.16.3.1 Overview

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

- *w_init*
- *w_run*

6.16.3.2 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.

6.16.3.3 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
  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
```

(continues on next page)

(continued from previous page)

```
  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)
```

6.16.4 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*

6.17 WEST Work Manager

6.17.1 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
thread	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).

6.17.2 Environment variables

6.17.2.1 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	(none)	processes	Use the given task distribution system: "serial", "threads", "processes", or "zmq"
WM_N_WO	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_	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_	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_	zmq	Random port	Master distributes tasks at this address
WM_ZMQ_	zmq	Random port	Master receives task results at this address |
WM_ZMQ_	zmq	Random port	Master publishes announcements (such as "shut down now") at this address
WM_ZMQ_	zmq	`zmq_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)

6.17.2.2 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_IN	processes, zmq	Contains an integer, 0 based, identifying the process among the set of processes started on a given node.

6.17.3 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
```

6.18 WEST Extensions

6.18.1 Post-Analysis Reweighting

6.18.2 String Method

6.18.3 Weighted Ensemble Equilibrium Dynamics

6.18.4 Weighted Ensemble Steady State

6.19 Command Line Tool Index

6.19.1 w_init

usage:

```
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]
             [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
             [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
             [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
             [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
             [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
```

(continues on next page)

(continued from previous page)

```
↪HEARTBEAT]
             [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
             [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

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.

optional arguments:

```
-h, --help            show this help message and exit
--force               Overwrite any existing simulation data
--bstate-file BSTATE_FILE, --bstates-from BSTATE_FILE
                      Read basis state names, probabilities, and (optionally) data␣
↪references from
                      BSTATE_FILE.
--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.
--tstate-file TSTATE_FILE, --tstates-from TSTATE_FILE
                      Read target state names and representative progress coordinates␣
↪from
                      TSTATE_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.
--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.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
```

(continues on next page)

(continued from previous page)

```
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution. Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do not support
                      this option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

--zmq-mode MODE Operate as a master (server) or a node (workers/client). "server" is a deprecated synonym for "master" and "client" is a deprecated synonym for "node".

--zmq-comm-mode COMM_MODE Use the given communication mode – TCP or IPC (Unix-domain) – sockets for communication within a node. IPC (the default) may be more efficient but is not available on (exceptionally rare) systems without node-local storage (e.g. /tmp); on such systems, TCP may be used instead.

--zmq-write-host-info INFO_FILE Store hostname and port information needed to connect to this instance in INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly. Downstream nodes read this file with –zmq-read-host-info and know where how to connect.

--zmq-read-host-info INFO_FILE Read hostname and port information needed to connect to the master (or other coordinating node) from INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly, writing that information with –zmq-write-host-info for this instance to read.

--zmq-upstream-rr-endpoint ENDPOINT ZeroMQ endpoint to which to send request/response (task and result) traffic toward the master.

--zmq-upstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to receive announcement (heartbeat and shutdown notification) traffic from the master.

--zmq-downstream-rr-endpoint ENDPOINT ZeroMQ endpoint on which to listen for request/response (task and result) traffic from subsidiary workers.

--zmq-downstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to send announcement (heartbeat and shutdown notification) traffic toward workers.

--zmq-master-heartbeat MASTER_HEARTBEAT Every MASTER_HEARTBEAT seconds, the master announces its presence to workers.

--zmq-worker-heartbeat WORKER_HEARTBEAT Every WORKER_HEARTBEAT seconds, workers announce their presence to the master.

--zmq-timeout-factor FACTOR Scaling factor for heartbeat timeouts. If the master doesn't hear from a worker in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If a worker doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds, the master is assumed to have crashed. Both cases result in shutdown.

--zmq-startup-timeout STARTUP_TIMEOUT Amount of time (in seconds) to wait for communication between the master and at least one worker. This may need to be changed on very large, heavily-loaded computer systems that start all processes simultaneously.

--zmq-shutdown-timeout SHUTDOWN_TIMEOUT Amount of time (in seconds) to wait for workers to shut down.

6.19.2 w_bins

`w_bins` deals with binning modification and statistics

6.19.2.1 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.

6.19.2.2 Command-Line Options

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

6.19.2.2.1 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).
```

6.19.2.2.2 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
```

(continues on next page)

(continued from previous page)

```
  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.
  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).
```

6.19.2.3 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).
```

6.19.2.4 Examples

(TODO: Write up an example)

6.19.3 w_run

usage:

```
w_run [-h]
```

Start/continue a WEST simulation

optional arguments:

```
-h, --help            show this help message and exit
--oneseg              only propagate one segment (useful for debugging propagators)
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
```

(continues on next page)

(continued from previous page)

```
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is
↪'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do␣
↪not support
                      this option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a
                      deprecated synonym for "master" and "client" is a deprecated␣
↪synonym for
                      "node".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g.
                      /tmp); on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read
                      this file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting
                      in coordinating the communication of other nodes to choose ports␣
↪randomly,
                      writing that information with --zmq-write-host-info for this␣
```

(continues on next page)

(continued from previous page)

```
↪instance to read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic
                      toward the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result)
                      traffic from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker
                      in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed.␣
↪If a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.4 w_truncate

NOTE: w_truncate only deletes iteration groups from the HDF5 data store. It is recommended that any iteration data saved to the file system (e.g. in the traj_segs directory) is deleted or moved for the corresponding iterations.

usage:

```
w_truncate [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-n N_ITER]
```

Remove all iterations after a certain point in a WESTPA simulation.

optional arguments:

```
-h, --help            show this help message and exit
-n N_ITER, --iter N_ITER
                      Truncate this iteration and those following.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

6.19.5 w_fork

usage:

```
w_fork [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-i INPUT_H5FILE]
             [-I N_ITER] [-o OUTPUT_H5FILE] [--istate-map ISTATE_MAP] [--no-headers]
```

Prepare a new weighted ensemble simulation from an existing one at a particular point. A new HDF5 file is generated. In the case of executable propagation, it is the user's responsibility to prepare the new simulation directory appropriately, particularly making the old simulation's restart data from the appropriate iteration available as the new simulations initial state data; a mapping of old simulation segment to new simulation initial states is created, both in the new HDF5 file and as a flat text file, to aid in this. Target states and basis states for the new simulation are taken from those in the original simulation.

optional arguments:

```
-h, --help            show this help message and exit
-i INPUT_H5FILE, --input INPUT_H5FILE
                      Create simulation from the given INPUT_H5FILE (default: read from␣
↪configuration
                      file.
-I N_ITER, --iteration N_ITER
                      Take initial distribution for new simulation from iteration N_ITER␣
↪(default:
                      last complete iteration).
-o OUTPUT_H5FILE, --output OUTPUT_H5FILE
                      Save new simulation HDF5 file as OUTPUT (default: forked.h5).
--istate-map ISTATE_MAP
                      Write text file describing mapping of existing segments to new␣
↪initial states
                      in ISTATE_MAP (default: istate_map.txt).
--no-headers          Do not write header to ISTATE_MAP
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
```

(continues on next page)

(continued from previous page)

```
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

6.19.6 w_assign

usage:

```
w_assign [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE]
               [--bins-from-system | --bins-from-expr BINS_FROM_EXPR | --bins-from-
↪function BINS_FROM_FUNCTION | --bins-from-file BINFILE | --bins-from-h5file]
               [--construct-dataset CONSTRUCT_DATASET | --dsspecs DSSPEC [DSSPEC ...]]
               [--states STATEDEF [STATEDEF ...] | --states-from-file STATEFILE |
               --states-from-function STATEFUNC] [-o OUTPUT] [--subsample] [--config-
↪from-file]
               [--scheme-name SCHEME] [--serial | --parallel | --work-manager WORK_
↪MANAGER]
               [--n-workers N_WORKERS] [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE]
               [--zmq-write-host-info INFO_FILE] [--zmq-read-host-info INFO_FILE]
               [--zmq-upstream-rr-endpoint ENDPOINT] [--zmq-upstream-ann-endpoint␣
↪ENDPOINT]
               [--zmq-downstream-rr-endpoint ENDPOINT] [--zmq-downstream-ann-endpoint␣
↪ENDPOINT]
               [--zmq-master-heartbeat MASTER_HEARTBEAT] [--zmq-worker-heartbeat WORKER_
↪HEARTBEAT]
               [--zmq-timeout-factor FACTOR] [--zmq-startup-timeout STARTUP_TIMEOUT]
               [--zmq-shutdown-timeout SHUTDOWN_TIMEOUT]
```

Assign walkers to bins, producing a file (by default named “assign.h5”) which can be used in subsequent analysis.

For consistency in subsequent analysis operations, the entire dataset must be assigned, even if only a subset of the data will be used. This ensures that analyses that rely on tracing trajectories always know the originating bin of each trajectory.

6.19.6.1 Source data

Source data is provided either by a user-specified function (–construct-dataset) or a list of “data set specifications” (–dsspecs). If neither is provided, the progress coordinate dataset ‘’pcoord” is used.

To use a custom function to extract or calculate data whose probability distribution will be calculated, specify the function in standard Python MODULE.FUNCTION syntax as the argument to –construct-dataset. This function will be called as function(n_iter,iter_group), where n_iter is the iteration whose data are being considered and iter_group is the corresponding group in the main WEST HDF5 file (west.h5). The function must return data which can be indexed as [segment][timepoint][dimension].

To use a list of data set specifications, specify –dsspecs and then list the desired datasets one-by-one (space-separated in most shells). These data set specifications are formatted as NAME[,file=FILENAME,slice=SLICE], which will use the dataset called NAME in the HDF5 file FILENAME (defaulting to the main WEST HDF5 file west.h5), and slice it with the Python slice expression SLICE (as in [0:2] to select the first two elements of the first axis of the dataset). The `slice` option is most useful for selecting one column (or more) from a multi-column dataset, such as arises when using a progress coordinate of multiple dimensions.

6.19.6.2 Specifying macrostates

Optionally, kinetic macrostates may be defined in terms of sets of bins. Each trajectory will be labeled with the kinetic macrostate it was most recently in at each timepoint, for use in subsequent kinetic analysis. This is required for all kinetics analysis (w_kintrace and w_kinmat).

There are three ways to specify macrostates:

1. States corresponding to single bins may be identified on the command line using the –states option, which takes multiple arguments, one for each state (separated by spaces in most shells). Each state is specified as a coordinate tuple, with an optional label prepended, as in `bound:1.0` or `unbound:(2.5,2.5)`. Unlabeled states are named `stateN`, where N is the (zero-based) position in the list of states supplied to –states.

2. States corresponding to multiple bins may use a YAML input file specified with –states-from-file. This file defines a list of states, each with a name and a list of coordinate tuples; bins containing these coordinates will be mapped to the containing state. For instance, the following file:

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

   produces two macrostates: the first state is called “unbound” and consists of bins containing the (2-dimensional) progress coordinate values (9.0, 1.0) and (9.0, 2.0); the second state is called “bound” and consists of the single bin containing the point (0.1, 0.0).

3. Arbitrary state definitions may be supplied by a user-defined function, specified as –states-from-function=MODULE.FUNCTION. This function is called with the bin mapper as an argument (`function(mapper)`) and must return a list of dictionaries, one per state. Each dictionary must contain a vector of coordinate tuples with key “coords”; the bins into which each of these tuples falls define the state. An optional name for the state (with key “label”) may also be provided.

6.19.6.3 Output format

The output file (-o/–output, by default “assign.h5”) contains the following attributes datasets:

```
``nbins`` attribute
  *(Integer)* Number of valid bins. Bin assignments range from 0 to
  *nbins*-1, inclusive.

``nstates`` attribute
  *(Integer)* Number of valid macrostates (may be zero if no such states are
  specified). Trajectory ensemble assignments range from 0 to *nstates*-1,
  inclusive, when states are defined.

``/assignments`` [iteration][segment][timepoint]
  *(Integer)* Per-segment and -timepoint assignments (bin indices).

``/npts`` [iteration]
  *(Integer)* Number of timepoints in each iteration.
```

(continues on next page)

(continued from previous page)

```
``/nsegs`` [iteration]
  *(Integer)* Number of segments in each iteration.

``/labeled_populations`` [iterations][state][bin]
  *(Floating-point)* Per-iteration and -timepoint bin populations, labeled
  by most recently visited macrostate. The last state entry (*nstates-1*)
  corresponds to trajectories initiated outside of a defined macrostate.

``/bin_labels`` [bin]
  *(String)* Text labels of bins.
```

When macrostate assignments are given, the following additional datasets are present:

```
``/trajlabels`` [iteration][segment][timepoint]
  *(Integer)* Per-segment and -timepoint trajectory labels, indicating the
  macrostate which each trajectory last visited.

``/state_labels`` [state]
  *(String)* Labels of states.

``/state_map`` [bin]
  *(Integer)* Mapping of bin index to the macrostate containing that bin.
  An entry will contain *nbins+1* if that bin does not fall into a
  macrostate.
```

Datasets indexed by state and bin contain one more entry than the number of valid states or bins. For *N* bins, axes indexed by bin are of size *N+1*, and entry *N* (0-based indexing) corresponds to a walker outside of the defined bin space (which will cause most mappers to raise an error). More importantly, for *M* states (including the case *M=0* where no states are specified), axes indexed by state are of size *M+1* and entry *M* refers to trajectories initiated in a region not corresponding to a defined macrostate.

Thus, `labeled_populations[:,:,:].sum(axis=1)[:,:-1]` gives overall per-bin populations, for all defined bins and `labeled_populations[:,:,:].sum(axis=2)[:,:-1]` gives overall per-trajectory-ensemble populations for all defined states.

6.19.6.4 Parallelization

This tool supports parallelized binning, including reading/calculating input data.

6.19.6.5 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
```

(continues on next page)

(continued from previous page)

```
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks
                      that have very large requests/response. Default: no limit.
```

WEST input data options:

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

binning options:

--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 BINFILE, --binfile BINFILE Load bin specification from the YAML file BINFILE. This currently takes the form {'bins': {'type': 'RectilinearBinMapper', 'boundaries': [[boundset1], [boundset2], ...]}}; only rectilinear bin bounds are supported.

--bins-from-h5file Load bin specification from the data file being examined (default where stored bin definitions available).

input dataset options:

```
--construct-dataset CONSTRUCT_DATASET
                      Use the given function (as in module.function) to extract source␣
↪data. This
                      function will be called once per iteration as function(n_iter,␣
↪iter_group) to
                      construct data for one iteration. Data returned must be indexable␣
↪as
                      [seg_id][timepoint][dimension]
```

(continues on next page)

(continued from previous page)

```
--dsspecs DSSPEC [DSSPEC ...]
                      Construct source data from one or more DSSPECs.
```

macrostate definitions:

```
--states STATEDEF [STATEDEF ...]
                      Single-bin kinetic macrostate, specified by a coordinate tuple (e.
↪g. '1.0' or
                      '[1.0,1.0]'), optionally labeled (e.g. 'bound:[1.0,1.0]'). States␣
↪corresponding
                      to multiple bins must be specified with --states-from-file.
--states-from-file STATEFILE
                      Load kinetic macrostates from the YAML file STATEFILE. See␣
↪description above
                      for the appropriate structure.
--states-from-function STATEFUNC
                      Load kinetic macrostates from the function STATEFUNC, specified as
                      module_name.func_name. This function is called with the bin mapper␣
↪as an
                      argument, and must return a list of dictionaries {'label': state_
↪label,
                      'coords': 2d_array_like} one for each macrostate; the 'coords'␣
↪entry must
                      contain enough rows to identify all bins in the macrostate.
```

other options:

```
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: assign.h5).
--subsample           Determines whether or not the data should be subsampled. This is␣
↪rather useful
                      for analysing steady state simulations.
--config-from-file    Load bins/macrostates from a scheme specified in west.cfg.
--scheme-name SCHEME  Name of scheme specified in west.cfg.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is
↪'processes'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do␣
↪not support
                      this option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

--zmq-mode MODE Operate as a master (server) or a node (workers/client). "server" is a dep-

recated synonym for "master" and "client" is a deprecated synonym for "node".

--zmq-comm-mode COMM_MODE Use the given communication mode – TCP or IPC (Unix-domain) – sockets for communication within a node. IPC (the default) may be more efficient but is not available on (exceptionally rare) systems without node-local storage (e.g. /tmp); on such systems, TCP may be used instead.

--zmq-write-host-info INFO_FILE Store hostname and port information needed to connect to this instance in INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly. Downstream nodes read this file with –zmq-read-host-info and know where how to connect.

--zmq-read-host-info INFO_FILE Read hostname and port information needed to connect to the master (or other coordinating node) from INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly, writing that information with –zmq-write-host-info for this instance to read.

--zmq-upstream-rr-endpoint ENDPOINT ZeroMQ endpoint to which to send request/response (task and result) traffic toward the master.

--zmq-upstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to receive announcement (heartbeat and shutdown notification) traffic from the master.

--zmq-downstream-rr-endpoint ENDPOINT ZeroMQ endpoint on which to listen for request/response (task and result) traffic from subsidiary workers.

--zmq-downstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to send announcement (heartbeat and shutdown notification) traffic toward workers.

--zmq-master-heartbeat MASTER_HEARTBEAT Every MASTER_HEARTBEAT seconds, the master announces its presence to workers.

--zmq-worker-heartbeat WORKER_HEARTBEAT Every WORKER_HEARTBEAT seconds, workers announce their presence to the master.

--zmq-timeout-factor FACTOR Scaling factor for heartbeat timeouts. If the master doesn't hear from a worker in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If a worker doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds, the master is assumed to have crashed. Both cases result in shutdown.

--zmq-startup-timeout STARTUP_TIMEOUT Amount of time (in seconds) to wait for communication between the master and at least one worker. This may need to be changed on very large, heavily-loaded computer systems that start all processes simultaneously.

--zmq-shutdown-timeout SHUTDOWN_TIMEOUT Amount of time (in seconds) to wait for workers to shut down.

6.19.7 w_trace

usage:

```
w_trace [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-W WEST_H5FILE]
          [-d DSNAME] [--output-pattern OUTPUT_PATTERN] [-o OUTPUT]
          N_ITER:SEG_ID [N_ITER:SEG_ID ...]
```

Trace individual WEST trajectories and emit (or calculate) quantities along the trajectory.

Trajectories are specified as N_ITER:SEG_ID pairs. Each segment is traced back to its initial point, and then various quantities (notably n_iter and seg_id) are printed in order from initial point up until the given segment in the given iteration.

Output is stored in several files, all named according to the pattern given by the -o/–output-pattern parameter. The default output pattern is "traj_%d_%d", where the printf-style format codes are replaced by the iteration number and segment ID of the terminal segment of the trajectory being traced.

Individual datasets can be selected for writing using the `-d/--dataset` option (which may be specified more than once). The simplest form is `-d dsname`, which causes data from dataset `dsname` along the trace to be stored to HDF5. The dataset is assumed to be stored on a per-iteration basis, with the first dimension corresponding to seg_id and the second dimension corresponding to time within the segment. Further options are specified as comma-separated key=value pairs after the data set name, as in:

```
-d dsname,alias=newname,index=idsname,file=otherfile.h5,slice=[100,...]
```

The following options for datasets are supported:

```
alias=newname
    When writing this data to HDF5 or text files, use ``newname``
    instead of ``dsname`` to identify the dataset. This is mostly of
    use in conjunction with the ``slice`` option in order, e.g., to
    retrieve two different slices of a dataset and store then with
    different names for future use.

index=idsname
    The dataset is not stored on a per-iteration basis for all
    segments, but instead is stored as a single dataset whose
    first dimension indexes n_iter/seg_id pairs. The index to
    these n_iter/seg_id pairs is ``idsname``.

file=otherfile.h5
    Instead of reading data from the main WEST HDF5 file (usually
    ``west.h5``), read data from ``otherfile.h5``.

slice=[100,...]
    Retrieve only the given slice from the dataset. This can be
    used to pick a subset of interest to minimize I/O.
```

6.19.7.1 positional arguments

```
N_ITER:SEG_ID         Trace trajectory ending (or at least alive at) N_ITER:SEG_ID.
```

6.19.7.2 optional arguments

```
-h, --help            show this help message and exit
-d DSNAME, --dataset DSNAME
                      Include the dataset named DSNAME in trace output. An extended form␣
↪like
                      DSNAME[,alias=ALIAS][,index=INDEX][,file=FILE][,slice=SLICE] will␣
↪obtain the
                      dataset from the given FILE instead of the main WEST HDF5 file,␣
↪slice it by
                      SLICE, call it ALIAS in output, and/or access per-segment data by a
                      n_iter,seg_id INDEX instead of a seg_id indexed dataset in the␣
↪group for
                      n_iter.
```

6.19.7.3 general options

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

6.19.7.4 WEST input data options

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

6.19.7.5 output options

```
--output-pattern OUTPUT_PATTERN
                      Write per-trajectory data to output files/HDF5 groups whose names␣
↪begin with
                      OUTPUT_PATTERN, which must contain two printf-style format flags␣
↪which will be
                      replaced with the iteration number and segment ID of the terminal␣
↪segment of
                      the trajectory being traced. (Default: traj_%d_%d.)
-o OUTPUT, --output OUTPUT
```

(continues on next page)

(continued from previous page)

```
                    Store intermediate data and analysis results to OUTPUT (default:␣
↪trajs.h5).
```

6.19.8 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.

6.19.8.1 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`.

6.19.8.2 Command-Line Options

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

6.19.8.2.1 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**)
```

6.19.8.2.2 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)
```

6.19.8.2.3 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)

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

6.19.8.3 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
```

6.19.9 w_ipa

usage:

```
w_ipa [-h] [-r RCFILE] [--quiet] [--verbose] [--version] [--max-queue-length MAX_QUEUE_
↪LENGTH]
            [-W WEST_H5FILE] [--analysis-only] [--reanalyze] [--ignore-hash] [--debug] [-
↪-terminal]
            [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_WORKERS]
            [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
            [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
            [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
            [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
            [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
            [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

-r RCFILE, --rcfile RCFILE use RCFILE as the WEST run-time configuration file (default: west.cfg)

--quiet emit only essential information

--verbose emit extra information

--version show program's version number and exit

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

WEST input data options:

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

runtime options:

```
--analysis-only, -ao  Use this flag to run the analysis and return to the terminal.
--reanalyze, -ra      Use this flag to delete the existing files and reanalyze.
--ignore-hash, -ih    Ignore hash and don't regenerate files.
--debug, -d           Debug output largely intended for development.
--terminal, -t        Plot output in terminal.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is
↪'processes'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
```

(continues on next page)

(continued from previous page)

```
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.10 w_pdist

usage:

```
w_pdist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE] [--first-iter N_
↪ITER]
              [--last-iter N_ITER] [-b BINEXPR] [-o OUTPUT] [-C] [--loose]
              [--construct-dataset CONSTRUCT_DATASET | --dsspecs DSSPEC [DSSPEC ...]]
              [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
              [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
              [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
              [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
              [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
              [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
```

(continues on next page)

(continued from previous page)

```
              [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

Calculate time-resolved, multi-dimensional probability distributions of WE datasets.

6.19.10.1 Source data

Source data is provided either by a user-specified function (–construct-dataset) or a list of "data set specifications" (–dsspecs). If neither is provided, the progress coordinate dataset ''pcoord" is used.

To use a custom function to extract or calculate data whose probability distribution will be calculated, specify the function in standard Python MODULE.FUNCTION syntax as the argument to –construct-dataset. This function will be called as function(n_iter,iter_group), where n_iter is the iteration whose data are being considered and iter_group is the corresponding group in the main WEST HDF5 file (west.h5). The function must return data which can be indexed as [segment][timepoint][dimension].

To use a list of data set specifications, specify –dsspecs and then list the desired datasets one-by-one (space-separated in most shells). These data set specifications are formatted as NAME[,file=FILENAME,slice=SLICE], which will use the dataset called NAME in the HDF5 file FILENAME (defaulting to the main WEST HDF5 file west.h5), and slice it with the Python slice expression SLICE (as in [0:2] to select the first two elements of the first axis of the dataset). The `slice` option is most useful for selecting one column (or more) from a multi-column dataset, such as arises when using a progress coordinate of multiple dimensions.

6.19.10.2 Histogram binning

By default, histograms are constructed with 100 bins in each dimension. This can be overridden by specifying -b/–bins, which accepts a number of different kinds of arguments:

```
a single integer N
  N uniformly spaced bins will be used in each dimension.

a sequence of integers N1,N2,... (comma-separated)
  N1 uniformly spaced bins will be used for the first dimension, N2 for the
  second, and so on.

a list of lists [[B11, B12, B13, ...], [B21, B22, B23, ...], ...]
  The bin boundaries B11, B12, B13, ... will be used for the first dimension,
  B21, B22, B23, ... for the second dimension, and so on. These bin
  boundaries need not be uniformly spaced. These expressions will be
  evaluated with Python's ``eval`` construct, with ``np`` available for
  use [e.g. to specify bins using np.arange()].
```

The first two forms (integer, list of integers) will trigger a scan of all data in each dimension in order to determine the minimum and maximum values, which may be very expensive for large datasets. This can be avoided by explicitly providing bin boundaries using the list-of-lists form.

Note that these bins are *NOT* at all related to the bins used to drive WE sampling.

6.19.10.3 Output format

The output file produced (specified by -o/–output, defaulting to "pdist.h5") may be fed to plothist to generate plots (or appropriately processed text or HDF5 files) from this data. In short, the following datasets are created:

```
``histograms``
  Normalized histograms. The first axis corresponds to iteration, and
  remaining axes correspond to dimensions of the input dataset.

``/binbounds_0``
  Vector of bin boundaries for the first (index 0) dimension. Additional
  datasets similarly named (/binbounds_1, /binbounds_2, ...) are created
  for additional dimensions.

``/midpoints_0``
  Vector of bin midpoints for the first (index 0) dimension. Additional
  datasets similarly named are created for additional dimensions.

``n_iter``
  Vector of iteration numbers corresponding to the stored histograms (i.e.
  the first axis of the ``histograms`` dataset).
```

6.19.10.4 Subsequent processing

The output generated by this program (-o/–output, default "pdist.h5") may be plotted by the `plothist` program. See `plothist --help` for more information.

6.19.10.5 Parallelization

This tool supports parallelized binning, including reading of input data. Parallel processing is the default. For simple cases (reading pre-computed input data, modest numbers of segments), serial processing (–serial) may be more efficient.

6.19.10.6 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
-b BINEXPR, --bins BINEXPR
                      Use BINEXPR for bins. This may be an integer, which will be used␣
↪for each
                      dimension of the progress coordinate; a list of integers␣
↪(formatted as
                      [n1,n2,...]) which will use n1 bins for the first dimension, n2␣
↪for the second
                      dimension, and so on; or a list of lists of boundaries (formatted␣
↪as [[a1, a2,
                      ...], [b1, b2, ...], ... ]), which will use [a1, a2, ...] as bin␣
↪boundaries for
                      the first dimension, [b1, b2, ...] as bin boundaries for the␣
↪second dimension,
                      and so on. (Default: 100 bins in each dimension.)
```

(continues on next page)

(continued from previous page)

```
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: pdist.h5).
-C, --compress        Compress histograms. May make storage of higher-dimensional␣
↪histograms more
                      tractable, at the (possible extreme) expense of increased analysis␣
↪time.
                      (Default: no compression.)
--loose               Ignore values that do not fall within bins. (Risky, as this can␣
↪make buggy bin
                      boundaries appear as reasonable data. Only use if you are sure of␣
↪your bin
                      boundary specification.)
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks
                      that have very large requests/response. Default: no limit.
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

input dataset options:

```
--construct-dataset CONSTRUCT_DATASET
                      Use the given function (as in module.function) to extract source␣
↪data. This
                      function will be called once per iteration as function(n_iter,␣
↪iter_group) to
                      construct data for one iteration. Data returned must be indexable␣
↪as
                      [seg_id][timepoint][dimension]
--dsspecs DSSPEC [DSSPEC ...]
                      Construct probability distribution from one or more DSSPECs.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is
↪'processes'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do␣
↪not support
                      this option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

--zmq-mode MODE Operate as a master (server) or a node (workers/client). "server" is a deprecated synonym for "master" and "client" is a deprecated synonym for "node".

--zmq-comm-mode COMM_MODE Use the given communication mode – TCP or IPC (Unix-domain) – sockets for communication within a node. IPC (the default) may be more efficient but is not available on (exceptionally rare) systems without node-local storage (e.g. /tmp); on such systems, TCP may be used instead.

--zmq-write-host-info INFO_FILE Store hostname and port information needed to connect to this instance in INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly. Downstream nodes read this file with –zmq-read-host-info and know where how to connect.

--zmq-read-host-info INFO_FILE Read hostname and port information needed to connect to the master (or other coordinating node) from INFO_FILE. This allows the master and nodes assisting in coordinating the communication of other nodes to choose ports randomly, writing that information with –zmq-write-host-info for this instance to read.

--zmq-upstream-rr-endpoint ENDPOINT ZeroMQ endpoint to which to send request/response (task and result) traffic toward the master.

--zmq-upstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to receive announcement (heartbeat and shutdown notification) traffic from the master.

--zmq-downstream-rr-endpoint ENDPOINT ZeroMQ endpoint on which to listen for request/response (task and result) traffic from subsidiary workers.

--zmq-downstream-ann-endpoint ENDPOINT ZeroMQ endpoint on which to send announcement (heartbeat and shutdown notification) traffic toward workers.

--zmq-master-heartbeat MASTER_HEARTBEAT Every MASTER_HEARTBEAT seconds, the master announces its presence to workers.

--zmq-worker-heartbeat WORKER_HEARTBEAT Every WORKER_HEARTBEAT seconds, workers announce their presence to the master.

--zmq-timeout-factor FACTOR Scaling factor for heartbeat timeouts. If the master doesn't hear from a worker in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If a worker doesn't hear from the master

in MASTER_HEARTBEAT*FACTOR seconds, the master is assumed to have crashed. Both cases result in shutdown.

--zmq-startup-timeout STARTUP_TIMEOUT Amount of time (in seconds) to wait for communication between the master and at least one worker. This may need to be changed on very large, heavily-loaded computer systems that start all processes simultaneously.

--zmq-shutdown-timeout SHUTDOWN_TIMEOUT Amount of time (in seconds) to wait for workers to shut down.

6.19.11 w_succ

usage:

```
w_succ [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-A H5FILE] [-W WEST_
↪H5FILE]
            [-o OUTPUT_FILE]
```

List segments which successfully reach a target state.

optional arguments:

```
-h, --help            show this help message and exit
-o OUTPUT_FILE, --output OUTPUT_FILE
                      Store output in OUTPUT_FILE (default: write to standard output).
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

general analysis options:

```
-A H5FILE, --analysis-file H5FILE
                      Store intermediate and final results in H5FILE (default: analysis.
↪h5).
```

WEST input data options:

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

6.19.12 w_crawl

usage:

```
w_crawl [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
              [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE] [--first-iter N_
↪ITER]
              [--last-iter N_ITER] [-c CRAWLER_INSTANCE]
              [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
              [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
              [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
              [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
              [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
              [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
              [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
              task_callable
```

Crawl a weighted ensemble dataset, executing a function for each iteration. This can be used for postprocessing of trajectories, cleanup of datasets, or anything else that can be expressed as "do X for iteration N, then do something with the result". Tasks are parallelized by iteration, and no guarantees are made about evaluation order.

6.19.12.1 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks
                      that have very large requests/response. Default: no limit.
```

WEST input data options:

```
-W WEST_H5FILE, --west-data WEST_H5FILE
                      Take WEST data from WEST_H5FILE (default: read from the HDF5 file␣
```

(continues on next page)

(continued from previous page)

```
↪specified in
                      west.cfg).
```

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

task options:

```
-c CRAWLER_INSTANCE, --crawler-instance CRAWLER_INSTANCE
                      Use CRAWLER_INSTANCE (specified as module.instance) as an instance␣
↪of
                      WESTPACrawler to coordinate the calculation. Required only if␣
↪initialization,
                      finalization, or task result processing is required.
task_callable         Run TASK_CALLABLE (specified as module.function) on each iteration.
↪ Required.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work
                      managers are ('serial', 'threads', 'processes', 'zmq'); default is
↪'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option.
                      Use 0 for a dedicated server. (Ignored by work managers which do␣
↪not support
                      this option.)
```

options for ZeroMQ (“zmq”) work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a
                      deprecated synonym for "master" and "client" is a deprecated␣
↪synonym for
                      "node".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g.
                      /tmp); on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
```

(continues on next page)

(continued from previous page)

```
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read
                      this file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting
                      in coordinating the communication of other nodes to choose ports␣
↪randomly,
                      writing that information with --zmq-write-host-info for this␣
↪instance to read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic
                      toward the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result)
                      traffic from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker
                      in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed.␣
↪If a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
```

(continues on next page)

(continued from previous page)

```
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.13 w_direct

usage:

```
w_direct [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--max-queue-length MAX_QUEUE_LENGTH]
                [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
                [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
                [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
                [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
                [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
                [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
                [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
                {help,init,average,kinetics,probs,all} ...
```

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

direct kinetics analysis schemes:

```
{help,init,average,kinetics,probs,all}
  help                print help for this command or individual subcommands
  init                calculate state-to-state kinetics by tracing trajectories
  average             Averages and returns fluxes, rates, and color/state populations.
  kinetics            Generates rate and flux values from a WESTPA simulation via␣
↪tracing.
```

(continues on next page)

(continued from previous page)

```
  probs                 Calculates color and state probabilities via tracing.
  all                   Runs the full suite, including the tracing of events.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
↪support this
                      option.)
```

options for ZeroMQ (“zmq”) work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
```

(continues on next page)

(continued from previous page)

```
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.14 w_select

usage:

```
w_select [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
                [--max-queue-length MAX_QUEUE_LENGTH] [-W WEST_H5FILE] [--first-iter N_
↪ITER]
                [--last-iter N_ITER] [-p MODULE.FUNCTION] [-v] [-a] [-o OUTPUT]
                [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
                [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
                [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
                [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
```

(continues on next page)

(continued from previous page)

```
              [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
              [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
              [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

Select dynamics segments matching various criteria. This requires a user-provided prediate function. By default, only matching segments are stored. If the -a/–include-ancestors option is given, then matching segments and their ancestors will be stored.

6.19.14.1 Predicate function

Segments are selected based on a predicate function, which must be callable as `predicate(n_iter, iter_group)` and return a collection of segment IDs matching the predicate in that iteration.

The predicate may be inverted by specifying the -v/–invert command-line argument.

6.19.14.2 Output format

The output file (-o/–output, by default "select.h5") contains the following datasets:

```
``/n_iter`` [iteration]
  *(Integer)* Iteration numbers for each entry in other datasets.

``/n_segs`` [iteration]
  *(Integer)* Number of segment IDs matching the predicate (or inverted
  predicate, if -v/--invert is specified) in the given iteration.

``/seg_ids`` [iteration][segment]
  *(Integer)* Matching segments in each iteration. For an iteration
  ``n_iter``, only the first ``n_iter`` entries are valid. For example,
  the full list of matching seg_ids in the first stored iteration is
  ``seg_ids[0][:n_segs[0]]``.

``/weights`` [iteration][segment]
  *(Floating-point)* Weights for each matching segment in ``/seg_ids``.
```

6.19.14.3 Command-line arguments

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
```

(continues on next page)

(continued from previous page)

```
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

```
--max-queue-length MAX_QUEUE_LENGTH
                      Maximum number of tasks that can be queued. Useful to limit RAM␣
↪use for tasks that
                      have very large requests/response. Default: no limit.
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

selection options:

```
-p MODULE.FUNCTION, --predicate-function MODULE.FUNCTION
                      Use the given predicate function to match segments. This function␣
↪should take an
                      iteration number and the HDF5 group corresponding to that␣
↪iteration and return a
                      sequence of seg_ids matching the predicate, as in ``match_
↪predicate(n_iter,
                      iter_group)``.
-v, --invert          Invert the match predicate.
-a, --include-ancestors
                      Include ancestors of matched segments in output.
```

output options:

-o OUTPUT, --output OUTPUT Write output to OUTPUT (default: select.h5).

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
```

(continues on next page)

(continued from previous page)

```
↪support this
                        option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
```

(continues on next page)

(continued from previous page)

```
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.15 w_states

usage:

```
w_states [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               [--show | --append | --replace] [--bstate-file BSTATE_FILE] [--bstate␣
↪BSTATES]
               [--tstate-file TSTATE_FILE] [--tstate TSTATES]
               [--serial | --parallel | --work-manager WORK_MANAGER] [--n-workers N_
↪WORKERS]
               [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE] [--zmq-write-host-info INFO_
↪FILE]
               [--zmq-read-host-info INFO_FILE] [--zmq-upstream-rr-endpoint ENDPOINT]
               [--zmq-upstream-ann-endpoint ENDPOINT] [--zmq-downstream-rr-endpoint␣
↪ENDPOINT]
               [--zmq-downstream-ann-endpoint ENDPOINT] [--zmq-master-heartbeat MASTER_
↪HEARTBEAT]
               [--zmq-worker-heartbeat WORKER_HEARTBEAT] [--zmq-timeout-factor FACTOR]
               [--zmq-startup-timeout STARTUP_TIMEOUT] [--zmq-shutdown-timeout SHUTDOWN_
↪TIMEOUT]
```

Display or manipulate basis (initial) or target (recycling) states for a WEST simulation. By default, states are displayed (or dumped to files). If `--replace` is specified, all basis/target states are replaced for the next iteration. If `--append` is specified, the given target state(s) are appended to the list for the next iteration. Appending basis states is not permitted, as this would require renormalizing basis state probabilities in ways that may be error-prone. Instead, use `w_states --show --bstate-file=bstates.txt` and then edit the resulting `bstates.txt` file to include the new desired basis states, then use `w_states --replace --bstate-file=bstates.txt` to update the WEST HDF5 file appropriately.

optional arguments:

```
-h, --help            show this help message and exit
--bstate-file BSTATE_FILE
                      Read (--append/--replace) or write (--show) basis state names,␣
↪probabilities, and
                      data references from/to BSTATE_FILE.
--bstate BSTATES      Add the given basis state (specified as a string 'label,
↪probability[,auxref]') to
                      the list of basis states (after those specified in --bstate-file,␣
↪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.
--tstate-file TSTATE_FILE
                      Read (--append/--replace) or write (--show) target state names and␣
↪representative
                      progress coordinates from/to TSTATE_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.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

modes of operation:

```
--show                Display current basis/target states (or dump to files).
--append              Append the given basis/target states to those currently in use.
--replace             Replace current basis/target states with those specified.
```

parallelization options:

```
--serial              run in serial mode
--parallel            run in parallel mode (using processes)
--work-manager WORK_MANAGER
                      use the given work manager for parallel task distribution.␣
↪Available work managers
                      are ('serial', 'threads', 'processes', 'zmq'); default is 'serial'
--n-workers N_WORKERS
                      Use up to N_WORKERS on this host, for work managers which support␣
↪this option. Use
                      0 for a dedicated server. (Ignored by work managers which do not␣
```

(continues on next page)

(continued from previous page)

```
↪support this
                      option.)
```

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a deprecated
                      synonym for "master" and "client" is a deprecated synonym for "node
↪".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g. /tmp);
                      on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read this
                      file with --zmq-read-host-info and know where how to connect.
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting in
                      coordinating the communication of other nodes to choose ports␣
↪randomly, writing
                      that information with --zmq-write-host-info for this instance to␣
↪read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic toward
                      the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result) traffic
                      from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
```

(continues on next page)

(continued from previous page)

```
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker in
                      WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed. If␣
↪a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.16 w_eddist

usage:

```
w_eddist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               [--max-queue-length MAX_QUEUE_LENGTH] [-b BINEXPR] [-C] [--loose] --
↪istate ISTATE
               --fstate FSTATE [--first-iter ITER_START] [--last-iter ITER_STOP] [-k␣
↪KINETICS]
               [-o OUTPUT] [--serial | --parallel | --work-manager WORK_MANAGER]
               [--n-workers N_WORKERS] [--zmq-mode MODE] [--zmq-comm-mode COMM_MODE]
               [--zmq-write-host-info INFO_FILE] [--zmq-read-host-info INFO_FILE]
               [--zmq-upstream-rr-endpoint ENDPOINT] [--zmq-upstream-ann-endpoint␣
↪ENDPOINT]
               [--zmq-downstream-rr-endpoint ENDPOINT] [--zmq-downstream-ann-endpoint␣
↪ENDPOINT]
               [--zmq-master-heartbeat MASTER_HEARTBEAT] [--zmq-worker-heartbeat WORKER_
↪HEARTBEAT]
               [--zmq-timeout-factor FACTOR] [--zmq-startup-timeout STARTUP_TIMEOUT]
               [--zmq-shutdown-timeout SHUTDOWN_TIMEOUT]
```

Calculate time-resolved transition-event duration distribution from kinetics results

6.19.16.1 Source data

Source data is collected from the results of 'w_kinetics trace' (see w_kinetics trace –help for more information on generating this dataset).

6.19.16.2 Histogram binning

By default, histograms are constructed with 100 bins in each dimension. This can be overridden by specifying -b/–bins, which accepts a number of different kinds of arguments:

```
a single integer N
  N uniformly spaced bins will be used in each dimension.

a sequence of integers N1,N2,... (comma-separated)
  N1 uniformly spaced bins will be used for the first dimension, N2 for the
  second, and so on.

a list of lists [[B11, B12, B13, ...], [B21, B22, B23, ...], ...]
  The bin boundaries B11, B12, B13, ... will be used for the first dimension,
  B21, B22, B23, ... for the second dimension, and so on. These bin
  boundaries need not be uniformly spaced. These expressions will be
  evaluated with Python's ``eval`` construct, with ``np`` available for
  use [e.g. to specify bins using np.arange()].
```

The first two forms (integer, list of integers) will trigger a scan of all data in each dimension in order to determine the minimum and maximum values, which may be very expensive for large datasets. This can be avoided by explicitly providing bin boundaries using the list-of-lists form.

Note that these bins are *NOT* at all related to the bins used to drive WE sampling.

6.19.16.3 Output format

The output file produced (specified by -o/–output, defaulting to "pdist.h5") may be fed to plothist to generate plots (or appropriately processed text or HDF5 files) from this data. In short, the following datasets are created:

```
``histograms``
  Normalized histograms. The first axis corresponds to iteration, and
  remaining axes correspond to dimensions of the input dataset.

``/binbounds_0``
  Vector of bin boundaries for the first (index 0) dimension. Additional
  datasets similarly named (/binbounds_1, /binbounds_2, ...) are created
  for additional dimensions.

``/midpoints_0``
  Vector of bin midpoints for the first (index 0) dimension. Additional
  datasets similarly named are created for additional dimensions.

``n_iter``
  Vector of iteration numbers corresponding to the stored histograms (i.e.
  the first axis of the ``histograms`` dataset).
```

6.19.16.4 Subsequent processing

The output generated by this program (-o/–output, default "pdist.h5") may be plotted by the `plothist` program. See `plothist --help` for more information.

6.19.16.5 Parallelization

This tool supports parallelized binning, including reading of input data. Parallel processing is the default. For simple cases (reading pre-computed input data, modest numbers of segments), serial processing (–serial) may be more efficient.

6.19.16.6 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
-b BINEXPR, --bins BINEXPR
                      Use BINEXPR for bins. This may be an integer, which will be used␣
↪for each
                      dimension of the progress coordinate; a list of integers␣
↪(formatted as
                      [n1,n2,...]) which will use n1 bins for the first dimension, n2␣
↪for the second
                      dimension, and so on; or a list of lists of boundaries (formatted␣
↪as [[a1, a2,
                      ...], [b1, b2, ...], ... ]), which will use [a1, a2, ...] as bin␣
↪boundaries for
                      the first dimension, [b1, b2, ...] as bin boundaries for the␣
↪second dimension,
                      and so on. (Default: 100 bins in each dimension.)
-C, --compress        Compress histograms. May make storage of higher-dimensional␣
↪histograms more
                      tractable, at the (possible extreme) expense of increased analysis␣
↪time.
                      (Default: no compression.)
--loose               Ignore values that do not fall within bins. (Risky, as this can␣
↪make buggy bin
                      boundaries appear as reasonable data. Only use if you are sure of␣
↪your bin
                      boundary specification.)
--istate ISTATE       Initial state defining transition event
--fstate FSTATE       Final state defining transition event
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

parallelization options:

--max-queue-length MAX_QUEUE_LENGTH Maximum number of tasks that can be queued. Useful to limit RAM use for tasks that have very large requests/response. Default: no limit.

iteration range options:

```
--first-iter ITER_START
                      Iteration to begin analysis (default: 1)
--last-iter ITER_STOP
                      Iteration to end analysis
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates (including evolution) are stored␣
↪in KINETICS
                      (default: kintrace.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: eddist.h5).
```

parallelization options:

--serial run in serial mode

--parallel run in parallel mode (using processes)

--work-manager WORK_MANAGER use the given work manager for parallel task distribution. Available work managers are ('serial', 'threads', 'processes', 'zmq'); default is 'processes'

--n-workers N_WORKERS Use up to N_WORKERS on this host, for work managers which support this option. Use 0 for a dedicated server. (Ignored by work managers which do not support this option.)

options for ZeroMQ ("zmq") work manager (master or node):

```
--zmq-mode MODE       Operate as a master (server) or a node (workers/client). "server"␣
↪is a
                      deprecated synonym for "master" and "client" is a deprecated␣
↪synonym for
                      "node".
--zmq-comm-mode COMM_MODE
                      Use the given communication mode -- TCP or IPC (Unix-domain) --␣
↪sockets for
                      communication within a node. IPC (the default) may be more␣
↪efficient but is not
                      available on (exceptionally rare) systems without node-local␣
↪storage (e.g.
                      /tmp); on such systems, TCP may be used instead.
--zmq-write-host-info INFO_FILE
                      Store hostname and port information needed to connect to this␣
↪instance in
                      INFO_FILE. This allows the master and nodes assisting in␣
↪coordinating the
                      communication of other nodes to choose ports randomly. Downstream␣
↪nodes read
                      this file with --zmq-read-host-info and know where how to connect.
```

(continues on next page)

(continued from previous page)

```
--zmq-read-host-info INFO_FILE
                      Read hostname and port information needed to connect to the master␣
↪(or other
                      coordinating node) from INFO_FILE. This allows the master and␣
↪nodes assisting
                      in coordinating the communication of other nodes to choose ports␣
↪randomly,
                      writing that information with --zmq-write-host-info for this␣
↪instance to read.
--zmq-upstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint to which to send request/response (task and␣
↪result) traffic
                      toward the master.
--zmq-upstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to receive announcement (heartbeat and␣
↪shutdown
                      notification) traffic from the master.
--zmq-downstream-rr-endpoint ENDPOINT
                      ZeroMQ endpoint on which to listen for request/response (task and␣
↪result)
                      traffic from subsidiary workers.
--zmq-downstream-ann-endpoint ENDPOINT
                      ZeroMQ endpoint on which to send announcement (heartbeat and␣
↪shutdown
                      notification) traffic toward workers.
--zmq-master-heartbeat MASTER_HEARTBEAT
                      Every MASTER_HEARTBEAT seconds, the master announces its presence␣
↪to workers.
--zmq-worker-heartbeat WORKER_HEARTBEAT
                      Every WORKER_HEARTBEAT seconds, workers announce their presence to␣
↪the master.
--zmq-timeout-factor FACTOR
                      Scaling factor for heartbeat timeouts. If the master doesn't hear␣
↪from a worker
                      in WORKER_HEARTBEAT*FACTOR, the worker is assumed to have crashed.␣
↪If a worker
                      doesn't hear from the master in MASTER_HEARTBEAT*FACTOR seconds,␣
↪the master is
                      assumed to have crashed. Both cases result in shutdown.
--zmq-startup-timeout STARTUP_TIMEOUT
                      Amount of time (in seconds) to wait for communication between the␣
↪master and at
                      least one worker. This may need to be changed on very large,␣
↪heavily-loaded
                      computer systems that start all processes simultaneously.
--zmq-shutdown-timeout SHUTDOWN_TIMEOUT
                      Amount of time (in seconds) to wait for workers to shut down.
```

6.19.17 w_ntop

usage:

```
w_ntop [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version] [-W WEST_H5FILE]
             [--first-iter N_ITER] [--last-iter N_ITER] [-a ASSIGNMENTS] [-n COUNT] [-t␣
↪TIMEPOINT]
             [--highweight | --lowweight | --random] [-o OUTPUT]
```

Select walkers from bins . An assignment file mapping walkers to bins at each timepoint is required (see``w_assign –help`` for further information on generating this file). By default, high-weight walkers are selected (hence the name `w_ntop`: select the N top-weighted walkers from each bin); however, minimum weight walkers and randomly-selected walkers may be selected instead.

6.19.17.1 Output format

The output file (-o/–output, by default "ntop.h5") contains the following datasets:

```
``/n_iter`` [iteration]
  *(Integer)* Iteration numbers for each entry in other datasets.

``/n_segs`` [iteration][bin]
  *(Integer)* Number of segments in each bin/state in the given iteration.
  This will generally be the same as the number requested with
  ``--n/--count`` but may be smaller if the requested number of walkers
  does not exist.

``/seg_ids`` [iteration][bin][segment]
  *(Integer)* Matching segments in each iteration for each bin.
  For an iteration ``n_iter``, only the first ``n_iter`` entries are
  valid. For example, the full list of matching seg_ids in bin 0 in the
  first stored iteration is ``seg_ids[0][0][:n_segs[0]]``.

``/weights`` [iteration][bin][segment]
  *(Floating-point)* Weights for each matching segment in ``/seg_ids``.
```

6.19.17.2 Command-line arguments

optional arguments:

```
-h, --help            show this help message and exit
--highweight          Select COUNT highest-weight walkers from each bin.
--lowweight           Select COUNT lowest-weight walkers from each bin.
--random              Select COUNT walkers randomly from each bin.
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
```

(continues on next page)

(continued from previous page)

```
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
```

input options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Use assignments from the given ASSIGNMENTS file (default: assign.
↪h5).
```

selection options:

```
-n COUNT, --count COUNT
                      Select COUNT walkers from each iteration for each bin (default: 1).
-t TIMEPOINT, --timepoint TIMEPOINT
                      Base selection on the given TIMEPOINT within each iteration.␣
↪Default (-1)
                      corresponds to the last timepoint.
```

output options:

```
-o OUTPUT, --output OUTPUT
                      Write output to OUTPUT (default: ntop.h5).
```

6.19.18 plothist

6.19.18.1 plothist_instant

usage:

```
plothist instant [-h] [-o PLOT_OUTPUT] [--hdf5-output HDF5_OUTPUT] [--plot-contour]
                       [--title TITLE] [--linear | --energy | --zero-energy E | --log10]
                       [--range RANGE] [--postprocess-function POSTPROCESS_FUNCTION]
                       [--text-output TEXT_OUTPUT] [--iter N_ITER]
                       input [DIMENSION] [ADDTLDIM]
```

Plot a probability distribution for a single WE iteration. The probability distribution must have been previously extracted with `w_pdist` (or, at least, must be compatible with the output format of `w_pdist`; see `w_pdist --help` for more information).

optional arguments:

```
-h, --help            show this help message and exit
```

input options:

```
input                 HDF5 file containing histogram data
DIMENSION             Plot for the given DIMENSION, specified as INT[:[LB,UB]:LABEL],␣
↪where INT is a
                      zero-based integer identifying the dimension in the histogram, LB␣
↪and UB are
                      lower and upper bounds for plotting, and LABEL is the label for␣
↪the plot axis.
                      (Default: dimension 0, full range.)
ADDTLDIM              For instantaneous/average plots, plot along the given additional␣
↪dimension,
                      producing a color map.
--iter N_ITER         Plot distribution for iteration N_ITER (default: last completed␣
↪iteration).
```

output options:

```
-o PLOT_OUTPUT, --output PLOT_OUTPUT, --plot-output PLOT_OUTPUT
                      Store plot as PLOT_OUTPUT. This may be set to an empty string (e.g.
↪ --plot-
                      output='') to suppress plotting entirely. The output format is␣
↪determined by
                      filename extension (and thus defaults to PDF). Default: "hist.pdf".
--hdf5-output HDF5_OUTPUT
                      Store plot data in the HDF5 file HDF5_OUTPUT.
--plot-contour        Determines whether or not to superimpose a contour plot over the␣
↪heatmap for 2D
                      objects.
--text-output TEXT_OUTPUT
                      Store plot data in a text format at TEXT_OUTPUT. This option is␣
↪only valid for
                      1-D histograms. (Default: no text output.)
```

plot options:

```
--title TITLE         Include TITLE as the top-of-graph title
--linear              Plot the histogram on a linear scale.
--energy              Plot the histogram on an inverted natural log scale, corresponding␣
↪to (free)
                      energy (default).
--zero-energy E       Set the zero of energy to E, which may be a scalar, "min" or "max"
--log10               Plot the histogram on a base-10 log scale.
--range RANGE         Plot histogram ordinates over the given RANGE, specified as "LB,UB
↪", where LB
                      and UB are the lower and upper bounds, respectively. For 1-D plots,
↪ this is the
                      Y axis. For 2-D plots, this is the colorbar axis. (Default: full␣
↪range.)
--postprocess-function POSTPROCESS_FUNCTION
                      Names a function (as in module.function) that will be called just␣
```

(continues on next page)

(continued from previous page)

```
↪prior to
                      saving the plot. The function will be called as ``postprocess(hist,
↪ midpoints,
                      binbounds)`` where ``hist`` is the histogram that was plotted,␣
↪``midpoints`` is
                      the bin midpoints for each dimension, and ``binbounds`` is the bin␣
↪boundaries
                      for each dimension for 2-D plots, or None otherwise. The plot must␣
↪be modified
                      in place using the pyplot stateful interface.
```

6.19.18.2 plothist_average

usage:

```
plothist average [-h] [-o PLOT_OUTPUT] [--hdf5-output HDF5_OUTPUT] [--plot-contour]
                      [--title TITLE] [--linear | --energy | --zero-energy E | --log10]
                      [--range RANGE] [--postprocess-function POSTPROCESS_FUNCTION]
                      [--text-output TEXT_OUTPUT] [--first-iter N_ITER] [--last-iter N_
↪ITER]
                      input [DIMENSION] [ADDTLDIM]
```

Plot a probability distribution averaged over multiple iterations. The probability distribution must have been previously extracted with `w_pdist` (or, at least, must be compatible with the output format of `w_pdist`; see `w_pdist --help` for more information).

optional arguments:

```
-h, --help            show this help message and exit
```

input options:

```
input                 HDF5 file containing histogram data
DIMENSION             Plot for the given DIMENSION, specified as INT[:[LB,UB]:LABEL],␣
↪where INT is a
                      zero-based integer identifying the dimension in the histogram, LB␣
↪and UB are
                      lower and upper bounds for plotting, and LABEL is the label for␣
↪the plot axis.
                      (Default: dimension 0, full range.)
ADDTLDIM              For instantaneous/average plots, plot along the given additional␣
↪dimension,
                      producing a color map.
--first-iter N_ITER   Begin averaging at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude averaging with N_ITER, inclusive (default: last completed␣
↪iteration).
```

output options:

```
-o PLOT_OUTPUT, --output PLOT_OUTPUT, --plot-output PLOT_OUTPUT
                      Store plot as PLOT_OUTPUT. This may be set to an empty string (e.g.
↪ --plot-
```

(continues on next page)

(continued from previous page)

```
                      output='') to suppress plotting entirely. The output format is␣
↪determined by
                      filename extension (and thus defaults to PDF). Default: "hist.pdf".
--hdf5-output HDF5_OUTPUT
                      Store plot data in the HDF5 file HDF5_OUTPUT.
--plot-contour        Determines whether or not to superimpose a contour plot over the␣
↪heatmap for 2D
                      objects.
--text-output TEXT_OUTPUT
                      Store plot data in a text format at TEXT_OUTPUT. This option is␣
↪only valid for
                      1-D histograms. (Default: no text output.)
```

plot options:

```
--title TITLE         Include TITLE as the top-of-graph title
--linear              Plot the histogram on a linear scale.
--energy              Plot the histogram on an inverted natural log scale, corresponding␣
↪to (free)
                      energy (default).
--zero-energy E       Set the zero of energy to E, which may be a scalar, "min" or "max"
--log10               Plot the histogram on a base-10 log scale.
--range RANGE         Plot histogram ordinates over the given RANGE, specified as "LB,UB
↪", where LB
                      and UB are the lower and upper bounds, respectively. For 1-D plots,
↪ this is the
                      Y axis. For 2-D plots, this is the colorbar axis. (Default: full␣
↪range.)
--postprocess-function POSTPROCESS_FUNCTION
                      Names a function (as in module.function) that will be called just␣
↪prior to
                      saving the plot. The function will be called as ``postprocess(hist,
↪ midpoints,
                      binbounds)`` where ``hist`` is the histogram that was plotted,␣
↪``midpoints`` is
                      the bin midpoints for each dimension, and ``binbounds`` is the bin␣
↪boundaries
                      for each dimension for 2-D plots, or None otherwise. The plot must␣
↪be modified
                      in place using the pyplot stateful interface.
```

6.19.18.3 plothist_evolution

usage:

```
plothist evolution [-h] [-o PLOT_OUTPUT] [--hdf5-output HDF5_OUTPUT] [--plot-contour]
                       [--title TITLE] [--linear | --energy | --zero-energy E | --
↪log10]
                       [--range RANGE] [--postprocess-function POSTPROCESS_FUNCTION]
                       [--first-iter N_ITER] [--last-iter N_ITER] [--step-iter STEP]
                       input [DIMENSION]
```

Plot a probability distribution as it evolves over iterations. The probability distribution must have been previously extracted with `w_pdist` (or, at least, must be compatible with the output format of `w_pdist`; see `w_pdist --help` for more information).

optional arguments:

```
-h, --help            show this help message and exit
```

input options:

```
input                 HDF5 file containing histogram data
DIMENSION             Plot for the given DIMENSION, specified as INT[:[LB,UB]:LABEL],␣
↪where INT is a
                      zero-based integer identifying the dimension in the histogram, LB␣
↪and UB are
                      lower and upper bounds for plotting, and LABEL is the label for␣
↪the plot axis.
                      (Default: dimension 0, full range.)
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Average in blocks of STEP iterations.
```

output options:

```
-o PLOT_OUTPUT, --output PLOT_OUTPUT, --plot-output PLOT_OUTPUT
                      Store plot as PLOT_OUTPUT. This may be set to an empty string (e.g.
↪ --plot-
                      output='') to suppress plotting entirely. The output format is␣
↪determined by
                      filename extension (and thus defaults to PDF). Default: "hist.pdf".
--hdf5-output HDF5_OUTPUT
                      Store plot data in the HDF5 file HDF5_OUTPUT.
--plot-contour        Determines whether or not to superimpose a contour plot over the␣
↪heatmap for 2D
                      objects.
```

plot options:

```
--title TITLE         Include TITLE as the top-of-graph title
--linear              Plot the histogram on a linear scale.
--energy              Plot the histogram on an inverted natural log scale, corresponding␣
↪to (free)
                      energy (default).
--zero-energy E       Set the zero of energy to E, which may be a scalar, "min" or "max"
--log10               Plot the histogram on a base-10 log scale.
--range RANGE         Plot histogram ordinates over the given RANGE, specified as "LB,UB
↪", where LB
                      and UB are the lower and upper bounds, respectively. For 1-D plots,
↪ this is the
                      Y axis. For 2-D plots, this is the colorbar axis. (Default: full␣
↪range.)
--postprocess-function POSTPROCESS_FUNCTION
                      Names a function (as in module.function) that will be called just␣
↪prior to
```

(continues on next page)

(continued from previous page)

```
                      saving the plot. The function will be called as ``postprocess(hist,
↪ midpoints,
                      binbounds)`` where ``hist`` is the histogram that was plotted,␣
↪``midpoints`` is
                      the bin midpoints for each dimension, and ``binbounds`` is the bin␣
↪boundaries
                      for each dimension for 2-D plots, or None otherwise. The plot must␣
↪be modified
                      in place using the pyplot stateful interface.
```

usage:

```
plothist [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               {help,instant,average,evolution} ...
```

Plot probability density functions (histograms) generated by w_pdist or other programs conforming to the same output format. This program operates in one of three modes:

```
instant
  Plot 1-D and 2-D histograms for an individual iteration. See
  ``plothist instant --help`` for more information.

average
  Plot 1-D and 2-D histograms, averaged over several iterations. See
  ``plothist average --help`` for more information.

evolution
  Plot the time evolution 1-D histograms as waterfall (heat map) plots.
  See ``plothist evolution --help`` for more information.
```

This program takes the output of `w_pdist` as input (see `w_pdist --help` for more information), and can generate any kind of graphical output that matplotlib supports.

6.19.18.4 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

plotting modes:

```
{help,instant,average,evolution}
  help                print help for this command or individual subcommands
  instant             plot probability distribution for a single WE iteration
  average             plot average of a probability distribution over a WE simulation
  evolution           plot evolution of a probability distribution over the course of a␣
↪WE simulation
```

6.19.19 ploterr

usage:

```
ploterrs [-h] [-r RCFILE] [--quiet | --verbose | --debug] [--version]
               {help,d.kinetics,d.probs,rw.probs,rw.kinetics,generic} ...
```

Plots error ranges for weighted ensemble datasets.

6.19.19.1 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

general options:

```
-r RCFILE, --rcfile RCFILE
                      use RCFILE as the WEST run-time configuration file (default: west.
↪cfg)
--quiet               emit only essential information
--verbose             emit extra information
--debug               enable extra checks and emit copious information
--version             show program's version number and exit
```

supported input formats:

```
{help,d.kinetics,d.probs,rw.probs,rw.kinetics,generic}
  help                print help for this command or individual subcommands
  d.kinetics          output of w_direct kinetics
  d.probs             output of w_direct probs
  rw.probs            output of w_reweight probs
  rw.kinetics         output of w_reweight kinetics
  generic             arbitrary HDF5 file and dataset
```

6.19.20 w_kinavg

WARNING: w_kinavg is being deprecated. Please use w_direct instead.

usage:

```
w_kinavg trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER] [--step-
↪iter STEP]
                      [-a ASSIGNMENTS] [-o OUTPUT] [-k KINETICS] [--disable-bootstrap] [--
↪disable-correl]
                      [--alpha ALPHA] [--autocorrel-alpha ACALPHA] [--nsets NSETS]
                      [-e {cumulative,blocked,none}] [--window-frac WINDOW_FRAC] [--
↪disable-averages]
```

Calculate average rates/fluxes and associated errors from weighted ensemble data. Bin assignments (usually "assign.h5") and kinetics data (usually "direct.h5") data files must have been previously generated (see "w_assign –help" and "w_direct init –help" for information on generating these files).

The evolution of all datasets may be calculated, with or without confidence intervals.

6.19.20.1 Output format

The output file (-o/–output, usually "direct.h5") contains the following dataset:

```
/avg_rates [state,state]
  (Structured -- see below) State-to-state rates based on entire window of
  iterations selected.

/avg_total_fluxes [state]
  (Structured -- see below) Total fluxes into each state based on entire
  window of iterations selected.

/avg_conditional_fluxes [state,state]
  (Structured -- see below) State-to-state fluxes based on entire window of
  iterations selected.
```

If –evolution-mode is specified, then the following additional datasets are available:

```
/rate_evolution [window][state][state]
  (Structured -- see below). State-to-state rates based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.

/target_flux_evolution [window,state]
  (Structured -- see below). Total flux into a given macro state based on
  windows of iterations of varying width, as in /rate_evolution.

/conditional_flux_evolution [window,state,state]
  (Structured -- see below). State-to-state fluxes based on windows of
  varying width, as in /rate_evolution.
```

The structure of these datasets is as follows:

```
iter_start
  (Integer) Iteration at which the averaging window begins (inclusive).

iter_stop
  (Integer) Iteration at which the averaging window ends (exclusive).

expected
  (Floating-point) Expected (mean) value of the observable as evaluated within
  this window, in units of inverse tau.

ci_lbound
  (Floating-point) Lower bound of the confidence interval of the observable
  within this window, in units of inverse tau.

ci_ubound
  (Floating-point) Upper bound of the confidence interval of the observable
  within this window, in units of inverse tau.

stderr
  (Floating-point) The standard error of the mean of the observable
  within this window, in units of inverse tau.

corr_len
  (Integer) Correlation length of the observable within this window, in units
  of tau.
```

Each of these datasets is also stamped with a number of attributes:

```
mcbs_alpha
  (Floating-point) Alpha value of confidence intervals. (For example,
  *alpha=0.05* corresponds to a 95% confidence interval.)

mcbs_nsets
  (Integer) Number of bootstrap data sets used in generating confidence
  intervals.

mcbs_acalpha
  (Floating-point) Alpha value for determining correlation lengths.
```

6.19.20.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: kinavg.h5).
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates are stored in KINETICS (default:␣
↪kintrace.h5).
```

confidence interval calculation options:

```
--disable-bootstrap, -db
                      Enable the use of Monte Carlo Block Bootstrapping.
--disable-correl, -dc
                      Disable the correlation analysis.
--alpha ALPHA         Calculate a (1-ALPHA) confidence interval' (default: 0.05)
--autocorrel-alpha ACALPHA
                      Evaluate autocorrelation to (1-ACALPHA) significance. Note that␣
↪too small an
                      ACALPHA will result in failure to detect autocorrelation in a␣
↪noisy flux signal.
                      (Default: same as ALPHA.)
--nsets NSETS         Use NSETS samples for bootstrapping (default: chosen based on␣
↪ALPHA)
```

calculation options:

```
-e {cumulative,blocked,none}, --evolution-mode {cumulative,blocked,none}
                      How to calculate time evolution of rate estimates. ``cumulative``␣
↪evaluates rates
                      over windows starting with --start-iter and getting progressively␣
↪wider to --stop-
                      iter by steps of --step-iter. ``blocked`` evaluates rates over␣
↪windows of width
                      --step-iter, the first of which begins at --start-iter. ``none``␣
↪(the default)
                      disables calculation of the time evolution of rate estimates.
--window-frac WINDOW_FRAC
                      Fraction of iterations to use in each window when running in␣
↪``cumulative`` mode.
                      The (1 - frac) fraction of iterations will be discarded from the␣
```

(continues on next page)

(continued from previous page)

```
↪start of each
                        window.
```

misc options:

```
--disable-averages, -da
                        Whether or not the averages should be printed to the console (set␣
↪to FALSE if flag
                        is used).
```

6.19.21 w_kinetics

WARNING: w_kinetics is being deprecated. Please use w_direct instead.

usage:

```
w_kinetics trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER]
                       [--step-iter STEP] [-a ASSIGNMENTS] [-o OUTPUT]
```

Calculate state-to-state rates and transition event durations by tracing trajectories.

A bin assignment file (usually "assign.h5") including trajectory labeling is required (see "w_assign –help" for information on generating this file).

This subcommand for w_direct is used as input for all other w_direct subcommands, which will convert the flux data in the output file into average rates/fluxes/populations with confidence intervals.

6.19.21.1 Output format

The output file (-o/–output, by default "direct.h5") contains the following datasets:

```
``/conditional_fluxes`` [iteration][state][state]
  *(Floating-point)* Macrostate-to-macrostate fluxes. These are **not**
  normalized by the population of the initial macrostate.

``/conditional_arrivals`` [iteration][stateA][stateB]
  *(Integer)* Number of trajectories arriving at state *stateB* in a given
  iteration, given that they departed from *stateA*.

``/total_fluxes`` [iteration][state]
  *(Floating-point)* Total flux into a given macrostate.

``/arrivals`` [iteration][state]
  *(Integer)* Number of trajectories arriving at a given state in a given
  iteration, regardless of where they originated.

``/duration_count`` [iteration]
  *(Integer)* The number of event durations recorded in each iteration.

``/durations`` [iteration][event duration]
  *(Structured -- see below)*  Event durations for transition events ending
  during a given iteration. These are stored as follows:
```

(continues on next page)

(continued from previous page)

```
    istate
      *(Integer)* Initial state of transition event.
    fstate
      *(Integer)* Final state of transition event.
    duration
      *(Floating-point)* Duration of transition, in units of tau.
    weight
      *(Floating-point)* Weight of trajectory at end of transition, **not**
      normalized by initial state population.
```

Because state-to-state fluxes stored in this file are not normalized by initial macrostate population, they cannot be used as rates without further processing. The `w_direct kinetics` command is used to perform this normalization while taking statistical fluctuation and correlation into account. See `w_direct kinetics --help` for more information. Target fluxes (total flux into a given state) require no such normalization.

6.19.21.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: kintrace.h5).
```

6.19.22 w_stateprobs

WARNING: w_stateprobs is being deprecated. Please use w_direct instead.

usage:

```
w_stateprobs trace [-h] [-W WEST_H5FILE] [--first-iter N_ITER] [--last-iter N_ITER]
                       [--step-iter STEP] [-a ASSIGNMENTS] [-o OUTPUT] [-k KINETICS]
                       [--disable-bootstrap] [--disable-correl] [--alpha ALPHA]
                       [--autocorrel-alpha ACALPHA] [--nsets NSETS] [-e {cumulative,
↪blocked,none}]
                       [--window-frac WINDOW_FRAC] [--disable-averages]
```

Calculate average populations and associated errors in state populations from weighted ensemble data. Bin assignments, including macrostate definitions, are required. (See "w_assign –help" for more information).

6.19.22.1 Output format

The output file (-o/–output, usually "direct.h5") contains the following dataset:

```
/avg_state_probs [state]
  (Structured -- see below) Population of each state across entire
  range specified.

/avg_color_probs [state]
  (Structured -- see below) Population of each ensemble across entire
  range specified.
```

If –evolution-mode is specified, then the following additional datasets are available:

```
/state_pop_evolution [window][state]
  (Structured -- see below). State populations based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.

/color_prob_evolution [window][state]
  (Structured -- see below). Ensemble populations based on windows of
  iterations of varying width.  If --evolution-mode=cumulative, then
  these windows all begin at the iteration specified with
  --start-iter and grow in length by --step-iter for each successive
  element. If --evolution-mode=blocked, then these windows are all of
  width --step-iter (excluding the last, which may be shorter), the first
  of which begins at iteration --start-iter.
```

The structure of these datasets is as follows:

```
iter_start
  (Integer) Iteration at which the averaging window begins (inclusive).

iter_stop
```

(continues on next page)

(continued from previous page)

```
  (Integer) Iteration at which the averaging window ends (exclusive).

expected
  (Floating-point) Expected (mean) value of the observable as evaluated within
  this window, in units of inverse tau.

ci_lbound
  (Floating-point) Lower bound of the confidence interval of the observable
  within this window, in units of inverse tau.

ci_ubound
  (Floating-point) Upper bound of the confidence interval of the observable
  within this window, in units of inverse tau.

stderr
  (Floating-point) The standard error of the mean of the observable
  within this window, in units of inverse tau.

corr_len
  (Integer) Correlation length of the observable within this window, in units
  of tau.
```

Each of these datasets is also stamped with a number of attributes:

```
mcbs_alpha
  (Floating-point) Alpha value of confidence intervals. (For example,
  *alpha=0.05* corresponds to a 95% confidence interval.)

mcbs_nsets
  (Integer) Number of bootstrap data sets used in generating confidence
  intervals.

mcbs_acalpha
  (Floating-point) Alpha value for determining correlation lengths.
```

6.19.22.2 Command-line options

optional arguments:

```
-h, --help            show this help message and exit
```

WEST input data options:

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

iteration range:

```
--first-iter N_ITER   Begin analysis at iteration N_ITER (default: 1).
--last-iter N_ITER    Conclude analysis with N_ITER, inclusive (default: last completed␣
```

(continues on next page)

(continued from previous page)

```
↪iteration).
--step-iter STEP      Analyze/report in blocks of STEP iterations.
```

input/output options:

```
-a ASSIGNMENTS, --assignments ASSIGNMENTS
                      Bin assignments and macrostate definitions are in ASSIGNMENTS␣
↪(default:
                      assign.h5).
-o OUTPUT, --output OUTPUT
                      Store results in OUTPUT (default: stateprobs.h5).
```

input/output options:

```
-k KINETICS, --kinetics KINETICS
                      Populations and transition rates are stored in KINETICS (default:␣
↪assign.h5).
```

confidence interval calculation options:

```
--disable-bootstrap, -db
                      Enable the use of Monte Carlo Block Bootstrapping.
--disable-correl, -dc
                      Disable the correlation analysis.
--alpha ALPHA         Calculate a (1-ALPHA) confidence interval' (default: 0.05)
--autocorrel-alpha ACALPHA
                      Evaluate autocorrelation to (1-ACALPHA) significance. Note that␣
↪too small an
                      ACALPHA will result in failure to detect autocorrelation in a␣
↪noisy flux signal.
                      (Default: same as ALPHA.)
--nsets NSETS         Use NSETS samples for bootstrapping (default: chosen based on␣
↪ALPHA)
```

calculation options:

```
-e {cumulative,blocked,none}, --evolution-mode {cumulative,blocked,none}
                      How to calculate time evolution of rate estimates. ``cumulative``␣
↪evaluates rates
                      over windows starting with --start-iter and getting progressively␣
↪wider to --stop-
                      iter by steps of --step-iter. ``blocked`` evaluates rates over␣
↪windows of width
                      --step-iter, the first of which begins at --start-iter. ``none``␣
↪(the default)
                      disables calculation of the time evolution of rate estimates.
--window-frac WINDOW_FRAC
                      Fraction of iterations to use in each window when running in␣
↪``cumulative`` mode.
                      The (1 - frac) fraction of iterations will be discarded from the␣
↪start of each
                      window.
```

misc options:

```
--disable-averages, -da
                      Whether or not the averages should be printed to the console (set␣
↪to FALSE if flag
                      is used).
```

6.20 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.

6.20.1 Overall structure

```
/
    #ibstates/
        index
        naming
            bstate_index
            bstate_pcoord
            istate_index
            istate_pcoord
    #tstates/
        index
    bin_topologies/
        index
        pickles
    iterations/
        iter_XXXXXXXX/\|iter_XXXXXXXX/
            auxdata/
            bin_target_counts
            ibstates/
                bstate_index
                bstate_pcoord
                istate_index
                istate_pcoord
            pcoord
            seg_index
            wtgraph
        ...
    summary
```

6.20.2 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

6.20.2.1 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

6.20.3 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_co	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)	

6.20.3.1 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	

6.20.4 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

6.21 Checklist

6.21.1 Configuring a WESTPA Simulation

- Files for dynamics propagation
  - Have you set up all of the files for propagating the dynamics (e.g. for GROMACS, the .top, .gro, .mdp, and .ndx files)?
- System implementation (`system.py`)
  - Is `self.pcoord_len` set to the number of data points that corresponds to the frequency with which the dynamics engine outputs the progress coordinate? Note: Many MD engines (e.g. GROMACS) output the initial point (i.e. zero).
  - Are the bins in the expected positions? You can easily view the positions of the bins using a Python interpreter.
- Initializing the simulation (`init.sh`)
  - Is the directory structure for the trajectory output files consistent with specifications in the master configuration file (`west.cfg`)?
  - Are the basis (bstate) states, and if applicable, target states (tstate), specified correctly?
- Calculating the progress coordinate for initial states (`get_pcoord.sh`)
  - Ensure that the procedure to extract the progress coordinate works by manually checking the procedure on one (or more) basis state files.
  - If your initialization (`init.sh`) gives an error message indicating the "incorrect shape" of the progress coordinate, check that get_pcoord.sh is not writing to a single file. If this is the case, w_init will crash

since multiple threads will be simultaneously writing to a single file. To fix this issue, you can add $$ to the file name (e.g. change `OUT=dist.xvg` to `OUT=dist_$$.xvg`) in `get_pcoord.sh`.

- Segment implementation (`runseg.sh`)
  - Ensure that the progress coordinate is being calculated correctly. If necessary, manually run a single dynamics segment () for a single trajectory walker to do so (e.g. for GROMACS, run the .tpr file for a length of). Double check that if any analysis programs are being run that their input is correct.
  - Are you feeding the velocities and state information required for the thermostat and barostat from one dynamics segment to the next? In GROMACS, this information is stored in the .edr and .trr files.
- Log of simulation progress (`west.h5`)
  - Check that the first iteration has been initialized, i.e. typing:

```
h5ls west.h5/iterations
```

    at the command line gives:

```
iter_00000001            Group
```

  - In addition, the progress coordinate should be initialized as well, i.e. using the command:

```
h5ls -d west.h5/iterations/iter_00000001/pcoord
```

    shows that the array is populated by zeros and the first point is the value calculated by get_pcoord.sh:

```
pcoord                   Dataset {10, 21, 1}
   Data:
       (0,0,0) 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       (2,15,0) 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 8, 0, 0, 0, 0, 0, 0, 0,
       (5,8,0) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 8, 0,
       (8,2,0) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

6.21.2 Running a WESTPA simulation

- If you encounter an issue while running the simulation
  - Use the `--debug` option on the servers w_run and save the output to a file. (note that this will generate a very detailed log of the process, try searching for "ERROR" for any errors and "iteration" to look at every iteration)
  - Use a program like hdfview, h5ls or Python with h5py library to open the `west.h5` file and ensure that the progress coordinate is being passed around correctly.
  - Use hdfview, h5ls or Python with h5py library to ensure that the number of trajectory walkers is correct.
- Is your simulation failing while the progress coordinate is being calculated?

  - One of the most error prone part during an iteration is the progress coordinate extraction. Programs that are not designed for quick execution have a lot of trouble during this step (VMD is a very commonly encountered one for example). Probably the best way to deal with this issue is to hard code a script to do the progress coordinate extraction. If you are doing molecular dynamics simulations multiple libraries for Python and C/C++ that deal with most output formats for MD packages exist and they usually come with a lot of convenience functions that can help you extract the progress coordinate. AMBER tools and GROMACS tools seems to work adequately for this purpose as well.

- Is your progress coordinate what you think it is?
  - Once your simulation it is running, it is well worth your time to ensure that the progress coordinate being reported is what you think it is. This can be done in a number of ways:
  - Check the `seg_log` output. This captures the standard error/output from the terminal session that your segment ran in, assuming you are running the executable propagator, and can be useful to ensure that everything is being done as you believe it should be (GROMACS tools, such as `g_dist`, for instance, report what groups have their distance being calculated here).
  - Look at a structure! Do so in a program such as VMD or pyMOL, and calculate your progress coordinate manually and check it visually, if feasible. Does it look correct, and seem to match what's being reported in the .h5 file? This is well worth your time before the simulation has proceeded very far, and can save a significant amount of wallclock and computational time.

6.21.3 Analyzing a WESTPA simulation

- If you are running the analysis on shared computing resources
  - Be sure to use the `--serial` flag (see the individual tool documentation). Otherwise, many of the included tools default to parallel mode (w_assign, for instance), which will create as many Python threads as there are CPU cores available.

6.22 Frequently Asked Questions (FAQ)

This page may be outdated, the most recent list of FAQs are available here:

6.22.1 Simulation

- How can I cleanly shutdown a simulation (without corrupting the h5 file)?

It is generally safe to shutdown a WESTPA simulation by simply canceling the job through your queue management. However, to ensure data integrity in the h5 file, you should wait until the WESTPA log indicates that an iteration has begun or is occurring; canceling a job too quickly after submission can result in the absolute corruption of the h5 file and should be avoided.

- Storage of Large Files

During a normal WESTPA run, many small files are created and it is convenient to tar these into a larger file (one tarball per iteration, for instance). It is generally best to do this 'offline'. An important aspect to consider is that some disk systems, such as LUSTRE, will suffer impaired performance if very large files are created. On Stampede, for instance, any file larger than 200 GB must be 'striped' properly (such that its individual bits are spread across numerous disks).

Within the user guide for such systems, there is generally a section on how to handle large files. Some computers have special versions of tar which stripe appropriately; others do not (such as Stampede). For those that do not, it may be necessary to contact the sysadmin, and/or create a directory where you can place your tarball with a different stripe level than the default.

- H5py Inflate() Failed error

While running or analyzing a simulation, you may run into an error such as `IOError: Can't write data (Inflate() failed)`. These errors may be related to an open bug in H5py. However, the following tips may help you to find a workaround.

WESTPA may present you with such an error when unable to read or write a data set. In the case that a simulation gives this error when you attempt to run it, it may be helpful to check if a data set may be read or written to using an interactive Python session. Restarting the simulation may require deleting and remaking the data set. Also, this error may be related to compression and other storage options. Thus, it may be helpful to disable compression and chunked storage. Note that existing datasets will retain compression and other options given to them at the time of their creation, so it may be necessary to truncate an iteration (for example, using `w_truncate`) in order for changes to take effect.

This error may also occur during repeated opening (e.g., 1000s of times) of an HDF5 data set. Thus, this error may occur while running analysis scripts. In this case, it may be helpful to cache data sets in physical memory (RAM) as `numpy` arrays when they are read, so that the script loads the dataset a minimal number of times.

- Dynamics Packages

WESTPA was designed to work cleanly with any dynamics package available (using the executable propagator); however, many of the tips and tricks available on the web or the user manual for these packages make the (reasonable) assumption that you will be running a set of brute force trajectories. As such, some of their guidelines for handling periodic boundary conditions may not be applicable.

- How can I restart a WESTPA simulation?

In general restarting a westpa simulation will restart an incomplete iteration, retaining data from segments that have completed and re-running segments that were incomplete (or never started).

In case that the iteration data got corrupted or you want to go back to an specific iteration and change something, you need to delete all the trajectory segments and other files related to that iteration and run w_truncate on that iteration. This will delete westpa's information about the nth iteration, which includes which segments have run and which have not. Then restarting your westpa simulation will restart that iteration afresh.

6.22.2 GROMACS

- Periodic Boundary Conditions

While many of the built in tools now handle periodic boundary conditions cleanly (such as g_dist) with relatively little user interaction, others, such as g_rms, do not. If your simulation analysis protocol requires you to run such a tool, you must correct for the periodic boundary conditions before running it. While there are guidelines available to help you correct for whatever conditions your system may have here, there is an implicit assumption that you have one long running trajectory.

It will be necessary, within your executable propagator (usually runseg.sh) to run trjconv (typically, two or three times, depending on your needs: once to remove the periodic boundary conditions, then to make molecules whole, then to remove any jumps). If no extra input is supplied (the -s flag in GROMACS 4.X), GROMACS uses the first frame of your segment trajectory as a reference state to remove jumps. If your segment's parent ended the previous iteration having jumped across the box barrier, trjconv will erroneously assume this is the correct state and 'correct' any jump back across the barrier. **This can result in unusually high RMSD values for one segment for one or more iterations,** and can show as discontinuities on the probability distribution. It is important to note that a lack of discontinuities does not imply a lack of imaging problems.

To fix this, simply pass in the last frame of the imaged parent trajectory and use that as the reference structure for trjconv. This will ensure that trjconv is aware if your segment has crossed the barrier at time 0 and will make the appropriate corrections.

6.22.3 Development

- **I'm trying to profile a parallel script using the –profile**
  option of bin/west. I get a PicklingError. What gives?

When executing a script using –profile, the following error may crop up:

```
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function␣
↪failed
```

The cProfile module used by the –profile option modifies function definitions such that they are no longer pickleable, meaning that they cannot be passed through the work manager to other processes. If you absolutely must profile a parallel script, use the threads work manager.

PYTHON MODULE INDEX

W

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z