pymcmcstat Documentation

Release 1.9.0

Paul Miles

Jul 19, 2019

Contents

The pymcmcstat package is a Python program for running Markov Chain Monte Carlo (MCMC) simulations. Included in this package is the ability to use different Metropolis based sampling techniques:

- Metropolis-Hastings (MH): Primary sampling method.
- Adaptive-Metropolis (AM): Adapts covariance matrix at specified intervals.
- Delayed-Rejection (DR): Delays rejection by sampling from a narrower distribution. Capable of n-stage delayed rejection.
- Delayed Rejection Adaptive Metropolis (DRAM): DR + AM

This package is an adaptation of the MATLAB toolbox mcmcstat. The user interface is designed to be as similar to the MATLAB version as possible, but this implementation has taken advantage of certain data structure concepts more amenable to Python.

Note, advanced plotting routines are available in the mcmcplot package. Many plotting features are directly available within pymcmcstat, but some user's may find mcmcplot useful.

CHAPTER 1

Installation

This code can be found on the Github project page. This package is available on the PyPI distribution site and the latest version can be installed via

```
pip install pymcmcstat
```

The master branch on Github typically matches the latest version on the PyPI distribution site. To install the master branch directly from Github,

```
pip install git+https://github.com/prmiles/pymcmcstat.git
```

You can also clone the repository and run `python setup.py install`.

CHAPTER 2

Getting Started

- Tutorial notebooks
- Documentation
- Release history
- Contributing guidelines

CHAPTER 3

License

MIT

CHAPTER 4

Contributors

See the GitHub contributor page

CHAPTER 5

Citing pymcmcstat

Miles, (2019). pymcmcstat: A Python Package for Bayesian Inference Using Delayed Rejection Adaptive Metropolis. Journal of Open Source Software, 4(38), 1417, https://doi.org/10.21105/joss.01417

Also, please cite the appropriate Zenodo archive for the version of *pymcmcstat* that you are using.

CHAPTER 6

Feedback

- Feature Request
- Bug Report

CHAPTER 7

Sponsor

This work was sponsored in part by the NNSA Office of Defense Nuclear Nonproliferation R&D through the Consortium for Nonproliferation Enabling Capabilities.

CHAPTER 8

Contents:

8.1 pymcmcstat package

8.1.1 pymcmcstat.MCMC module

Created on Wed Jan 17, 2018

@author: prmiles

Description: This module is intended to be the main class from which to run these Markov Chain Monte Carlo type simulations. The user will create an MCMC object, initialize data, simulation options, model settings and parameters.

class pymcmcstat.MCMC.**MCMC**(*rngseed=None*, *seterr={'over': 'ignore', 'under': 'ignore'}*)

Bases: object

Markov Chain Monte Carlo (MCMC) simulation object.

This class type is the primary feature of this Python package. All simulations are run through this class type, and for the most part the user will interact with an object of this type. The class initialization provides the option for setting the random seed, which can be very useful for testing the functionality of the code. It was found that setting the random seed at object initialization was the simplest interface.

Args:

- **rngseed** (float): Seed for numpy's random number generator.

Attributes:

- *run_simulation()*
- *display_current_mcmc_settings()*
- **data** (*DataStructure*): MCMC data structure.
- **simulation_options** (*SimulationOptions*): MCMC simulation options.
- **model_settings** (*ModelSettings*): MCMC model settings.

- **parameters** (*ModelParameters*): MCMC model parameters.

display_current_mcmc_settings()
Display model settings, simulation options, and current covariance values.

Example display:

```
model settings:
    sos_function = <function test_ssfun at 0x1c13c5d400>
    model_function = None
    sigma2 = [1.]
    N = [100.]
    N0 = [0.]
    S20 = [1.]
    nsos = 1
    nbatch = 1
simulation options:
    nsimu = 5000
    adaptint = 100
    ntry = 2
    method = dram
    printint = 100
    lastadapt = 5000
    drscale = [5. 4. 3.]
    qcov = None
covariance:
    qcov = [[0.01   0.    ]
    [0.     0.0625]]
    R = [[0.1  0.  ]
    [0.   0.25]]
    RDR = [array([[0.1 , 0.  ],
   [0.  , 0.25]]), array([[0.02, 0.  ],
   [0.  , 0.05]])]
    invR = [array([[10.,  0.],
   [ 0.,  4.]]), array([[50.,  0.],
   [ 0., 20.]])]
    last_index_since_adaptation = 0
    covchain = None
```

run_simulation(*use_previous_results=False*)
Run MCMC Simulation

Note: This is the method called by the user to run the simulation. The user must specify a data structure, setup simulation options, and define the model settings and parameters before calling this method.

Args:

- **use_previous_results** (bool): Flag to indicate whether simulation is being restarted.

pymcmcstat.MCMC.**print_rejection_statistics**(*rejected*, *isimu*, *iiadapt*, *verbosity*)
Print Rejection Statistics.

Threshold for printing is verbosity greater than or equal to 2. If the rejection counters are as follows:

- *total*: 144
- *in_adaptation_interval*: 92
- *outside_bounds*: 0

Then we would expect the following display at the 200th simulation with an adaptation interval of 100.

```
i:200 (72.0,92.0,0.0)
```

Args:

- **isimu** (int): Simulation counter
- **iiadapt** (int): Adaptation counter
- **verbosity** (int): Verbosity of display output.

8.1.2 pymcmcstat.ParallelMCMC module

Created on Tue May 1 15:58:22 2018

@author: prmiles

class pymcmcstat.ParallelMCMC.**ParallelMCMC**
Bases: object

Run Parallel MCMC Simulations.

Attributes:

- *setup_parallel_simulation()*
- *run_parallel_simulation()*
- *display_individual_chain_statistics()*

display_individual_chain_statistics()
Display chain statistics for different chains in parallel simulation.

Example display:

```
****************************************
Displaying results for chain 0
Files: <parallel_dir>/chain_0

----------------------
name      :       mean        std     MC_err        tau     geweke
m         :     1.9869     0.1295     0.0107   320.0997     0.9259
b         :     3.0076     0.2489     0.0132   138.1260     0.9413
----------------------

****************************************
Displaying results for chain 1
Files: <parallel_dir>/chain_1

----------------------
name      :       mean        std     MC_err        tau     geweke
m         :     1.8945     0.4324     0.0982  2002.6361     0.3116
b         :     3.2240     1.0484     0.2166  1734.0201     0.4161
----------------------
```

run_parallel_simulation()
Run MCMC simulations in parallel.

The code is run in parallel by using Pool. While running, you can expect a display similar to

```
Processing: <parallel_dir>/chain_1
Processing: <parallel_dir>/chain_0
Processing: <parallel_dir>/chain_2
```

The simulation is complete when you see the run time displayed.

```
Parallel simulation run time: 16.15234899520874 sec
```

setup_parallel_simulation(*mcset*, *initial_values=None*, *num_cores=1*, *num_chain=1*)

Setup simulation to run in parallel.

Settings defined in *mcset* object will be copied into different instances in order to run parallel chains.

Args:

- **mcset** (*MCMC*): Instance of MCMC object with serial simulation setup.
- **initial_values** (ndarray): Array of initial parameter values - [num_chain,npar].
- **num_cores** (int): Number of cores designated by user.
- **num_chain** (int): Number of sampling chains to be generated.

pymcmcstat.ParallelMCMC.**assign_number_of_cores**(*num_cores=1*)

Assign number of cores to use in parallel process

Args:

- **num_cores** (int): Number of cores designated by user.

Returns:

- **num_cores** (int): Number of cores designated by user or maximum number of cores available on machine.

pymcmcstat.ParallelMCMC.**check_directory**(*directory*)

Check and make sure directory exists

Args:

- **directory** (str): Folder/directory path name.

pymcmcstat.ParallelMCMC.**check_for_restart_file**(*json_restart_file*, *chain_dir*)

Check if restart directory was specified.

Args:

- **json_restart_file** (str): String specifying path to results directory.
- **chain_dir** (str): String specifying which chain is being restarted.

pymcmcstat.ParallelMCMC.**check_initial_values**(*initial_values*, *num_chain*, *npar*, *low_lim*, *upp_lim*)

Check if initial values satisfy requirements.

Args:

- **initial_values** (ndarray): Array of initial parameter values - [num_chain,npar].
- **num_chain** (int): Number of sampling chains to be generated.
- **npar** (int): Number of model parameters.
- **low_lim** (ndarray): Lower limits.
- **upp_lim** (ndarray): Upper limits.

Returns:

- **initial_values** (ndarray): Array of initial parameter values - [num_chain,npar].
- **num_chain** (int): Number of sampling chains to be generated.

pymcmcstat.ParallelMCMC.**check_options_output**(*options*)
Check output settings defined in options

Args:

- **options** (*SimulationOptions*): MCMC simulation options.

Returns:

- **options** (*SimulationOptions*): MCMC simulation options with at least binary save flag set to True.

pymcmcstat.ParallelMCMC.**check_shape_of_users_initial_values**(*initial_values*, *num_chain*, *npar*)
Check shape of users initial values

Args:

- **initial_values** (ndarray): Array of initial parameter values - expect [num_chain,npar]
- **num_chain** (int): Number of sampling chains to be generated.
- **npar** (int): Number of model parameters.

Returns:

- **num_chain** (int): Number of sampling chains to be generated - equal to number of rows in initial values array.
- **initial_values**

pymcmcstat.ParallelMCMC.**check_users_initial_values_wrt_limits**(*initial_values*, *low_lim*, *upp_lim*)
Check users initial values wrt parameter limits

Args:

- **initial_values** (ndarray): Array of initial parameter values - expect [num_chain,npar]
- **low_lim** (ndarray): Lower limits.
- **upp_lim** (ndarray): Upper limits.

Returns:

- **initial_values** (ndarray): Array of initial parameter values - expect [num_chain,npar]

Raises:

- *SystemExit* if initial values are outside parameter bounds.

pymcmcstat.ParallelMCMC.**generate_initial_values**(*num_chain*, *npar*, *low_lim*, *upp_lim*)
Generate initial values by sampling from uniform distribution between limits

Args:

- **num_chain** (int): Number of sampling chains to be generated.
- **npar** (int): Number of model parameters.
- **low_lim** (ndarray): Lower limits.

- **upp_lim** (ndarray): Upper limits.

Returns:

- **initial_values** (ndarray): Array of initial parameter values - [num_chain,npar]

pymcmcstat.ParallelMCMC.**get_parameter_features**(*parameters*)
Get features of model parameters.

Args:

- **parameters** (list): List of MCMC model parameter dictionaries.

Returns:

- **npar** (int): Number of model parameters.
- **low_lim** (ndarray): Lower limits.
- **upp_lim** (ndarray): Upper limits.

pymcmcstat.ParallelMCMC.**load_parallel_simulation_results**(*savedir*, *extension='h5'*, *chainfile='chainfile'*, *sschainfile='sschainfile'*, *s2chainfile='s2chainfile'*, *covchainfile='covchainfile'*)
Load results from parallel simulation directory json files.

Lists in json files are converted to numpy arrays.

Args:

- **savedir** (str): String indicated path to parallel directory.

Returns:

- **pres** (list): Each element of list is an MCMC result dictionary.

pymcmcstat.ParallelMCMC.**run_serial_simulation**(*mcstat*)
Run serial MCMC simulation

Args:

- **mcstat** (MCMC.MCMC): MCMC object.

Returns:

- **results** (dict): Results dictionary for serial simulation.

pymcmcstat.ParallelMCMC.**unpack_mcmc_set**(*mcset*)
Unpack attributes of MCMC object.

Args:

- **mcset** (*MCMC*): MCMC object.

Returns:

- **data** (*DataStructure*): MCMC data structure.
- **options** (*SimulationOptions*): MCMC simulation options.
- **model** (*ModelSettings*): MCMC model settings.
- **parameters** (*ModelParameters*): MCMC model parameters.

8.1.3 pymcmcstat.propagation module

Created on Wed Nov 8 12:00:11 2017

@author: prmiles

pymcmcstat.propagation.**calculate_intervals**(*chain*, *results*, *data*, *model*, *s2chain=None*, *nsample=500*, *waitbar=True*, *sstype=0*)

Calculate distribution of model response to form propagation intervals

Samples values from chain, performs forward model evaluation, and tabulates credible and prediction intervals (if obs. error var. included).

Args:

- **chain** (ndarray): Parameter chains, expect shape=(nsimu, npar).
- **results** (dict): Results dictionary generated by pymcmcstat.
- **data** (*DataStructure*): Data
- **model**: User defined function. Note, if your model outputs multiple quantities of interest (QoI) at the same time in a multi-dimensional array, then make sure it is returned as a (N, p) array where N is the number of evaluation points and p is the number of QoI.

Kwargs:

- **s2chain** (float, ndarray, or None): Observation error variance chain.
- **nsample** (int): No. of samples drawn from posteriors.
- **waitbar** (bool): Flag to display progress bar.
- **sstype** (int): Sum-of-squares type. Can be 0 (normal), 1 (sqrt), or 2 (log).

Returns:

- dict with two elements: 1) *credible* and 2) *prediction*

pymcmcstat.propagation.**check_s2chain**(*s2chain*, *nsimu*)

Check size of s2chain

Args:

- **s2chain (float, ndarray, or *None*):** Observation error variance chain or value
- **nsimu** (int): No. of elements in chain

Returns:

- **s2chain** (ndarray or *None*)

pymcmcstat.propagation.**define_sample_points**(*nsample*, *nsimu*)

Define indices to sample from posteriors.

Args:

- **nsample** (int): Number of samples to draw from posterior.
- **nsimu** (int): Number of MCMC simulations.

Returns:

- **iisample** (ndarray): Array of indices in posterior set.
- **nsample** (int): Number of samples to draw from posterior.

pymcmcstat.propagation.**generate_quantiles**(*x*, *p=array([0.25, 0.5, 0.75])*)

Calculate empirical quantiles.

Args:

- **x** (ndarray): Observations from which to generate quantile.
- **p** (ndarray): Quantile limits.

Returns:

- (ndarray): Interpolated quantiles.

pymcmcstat.propagation.**observation_sample**(*s2*, *y*, *sstype*)

Calculate model response with observation errors.

Args:

- **s2** (ndarray): Observation error(s).
- **y** (ndarray): Model responses.
- **sstype** (int): Flag to specify sstype.

Returns:

- **opred** (ndarray): Model responses with observation errors.

pymcmcstat.propagation.**plot_3d_intervals**(*intervals, time, ydata=None, xdata=None, limits=[95], adddata=False, addlegend=True, addmodel=True, figsize=None, model_display={}, data_display={}, interval_display={}, addcredible=True, addprediction=True, fig=None, legloc='upper left', ciset=None, piset=None, return_settings=False*)

Plot propagation intervals in 3-D

This routine takes the model distributions generated using the *calculate_intervals()* method and then plots specific quantiles. The user can plot just the intervals, or also include the median model response and/or observations. Specific settings for credible intervals are controlled by defining the *ciset* dictionary. Likewise, for prediction intervals, settings are defined using *piset*.

The setting options available for each interval are as follows:

- *limits*: This should be a list of numbers between 0 and 100, e.g., *limits=[50, 90]* will result in 50% and 90% intervals.
- *cmap*: The program is designed to "try" to choose colors that are visually distinct. The user can specify the colormap to choose from.
- *colors*: The user can specify the color they would like for each interval in a list, e.g., ['r', 'g', 'b']. This list should have the same number of elements as *limits* or the code will revert back to its default behavior.

Args:

- **intervals** (dict): Interval dictionary generated using *calculate_intervals()* method.
- **time** (ndarray): Independent variable, i.e., x- and y-axes of plot. Note, it must be a 2-D array with shape=(N, 2), where N is the number of evaluation points.

Kwargs:

- **ydata** (ndarray or None): Observations, expect 1-D array if defined.
- **xdata** (ndarray or None): Independent values corresponding to observations. This is required if the observations do not align with your times of generating the model response.

- **limits** (list): Quantile limits that correspond to percentage size of desired intervals. Note, this is the default limits, but specific limits can be defined using the *ciset* and *piset* dictionaries.
- **adddata** (bool): Flag to include data
- **addmodel** (bool): Flag to include median model response
- **addlegend** (bool): Flag to include legend
- **addcredible** (bool): Flag to include credible intervals
- **addprediction** (bool): Flag to include prediction intervals
- **model_display** (dict): Display settings for median model response
- **data_display** (dict): Display settings for data
- **interval_display** (dict): General display settings for intervals.
- **fig**: Handle of previously created figure object
- **figsize** (tuple): (width, height) in inches
- **legloc** (str): Legend location - matplotlib help for details.
- **ciset** (dict): Settings for credible intervals
- **piset** (dict): Settings for prediction intervals
- **return_settings** (bool): Flag to return ciset and piset along with fig and ax.

Returns:

- **(tuple) with elements**
  1) Figure handle
  2) Axes handle
  3) Dictionary with *ciset* and *piset* inside (only outputted if *return_settings=True*)

pymcmcstat.propagation.**plot_intervals**(*intervals, time, ydata=None, xdata=None, limits=[95], adddata=None, addmodel=True, addlegend=True, addcredible=True, addprediction=True, data_display={}, model_display={}, interval_display={}, fig=None, figsize=None, legloc='upper left', ciset=None, piset=None, return_settings=False*)

Plot propagation intervals in 2-D

This routine takes the model distributions generated using the *calculate_intervals()* method and then plots specific quantiles. The user can plot just the intervals, or also include the median model response and/or observations. Specific settings for credible intervals are controlled by defining the *ciset* dictionary. Likewise, for prediction intervals, settings are defined using *piset*.

The setting options available for each interval are as follows:

- *limits*: This should be a list of numbers between 0 and 100, e.g., *limits=[50, 90]* will result in 50% and 90% intervals.
- *cmap*: The program is designed to "try" to choose colors that are visually distinct. The user can specify the colormap to choose from.
- *colors*: The user can specify the color they would like for each interval in a list, e.g., ['r', 'g', 'b']. This list should have the same number of elements as *limits* or the code will revert back to its default behavior.

Args:

- **intervals** (dict): Interval dictionary generated using *calculate_intervals()* method.
- **time** (ndarray): Independent variable, i.e., x-axis of plot

Kwargs:

- **ydata** (ndarray or None): Observations, expect 1-D array if defined.
- **xdata** (ndarray or None): Independent values corresponding to observations. This is required if the observations do not align with your times of generating the model response.
- **limits** (list): Quantile limits that correspond to percentage size of desired intervals. Note, this is the default limits, but specific limits can be defined using the *ciset* and *piset* dictionaries.
- **adddata** (bool): Flag to include data
- **addmodel** (bool): Flag to include median model response
- **addlegend** (bool): Flag to include legend
- **addcredible** (bool): Flag to include credible intervals
- **addprediction** (bool): Flag to include prediction intervals
- **model_display** (dict): Display settings for median model response
- **data_display** (dict): Display settings for data
- **interval_display** (dict): General display settings for intervals.
- **fig**: Handle of previously created figure object
- **figsize** (tuple): (width, height) in inches
- **legloc** (str): Legend location - matplotlib help for details.
- **ciset** (dict): Settings for credible intervals
- **piset** (dict): Settings for prediction intervals
- **return_settings** (bool): Flag to return ciset and piset along with fig and ax.

Returns:

- **(tuple) with elements**
  1) Figure handle
  2) Axes handle
  3) Dictionary with *ciset* and *piset* inside (only outputted if *return_settings=True*)

pymcmcstat.propagation.**setup_display_settings**(*interval_display*, *model_display*, *data_display*)

Compare user defined display settings with defaults and merge.

Args:

- **interval_display** (dict): User defined settings for interval display.
- **model_display** (dict): User defined settings for model display.
- **data_display** (dict): User defined settings for data display.

Returns:

- **interval_display** (dict): Settings for interval display.
- **model_display** (dict): Settings for model display.

- **data_display** (dict): Settings for data display.

pymcmcstat.propagation.**setup_interval_colors**(*iset*, *inttype='CI'*)
Setup colors for empirical intervals

This routine attempts to distribute the color of the UQ intervals based on a normalize color map. Or, it will assign user-defined colors; however, this only happens if the correct number of colors are specified.

Args:

- **iset** (dict): This dictionary should contain the following keys - *limits*, *cmap*, and *colors*.

Kwargs:

- **inttype** (str): Type of uncertainty interval

Returns:

- **ic** (list): List containing color for each interval

8.1.4 Subpackages

pymcmcstat.chain package

pymcmcstat.chain.ChainProcessing module

Created on Tue May 1 09:12:06 2018

@author: prmiles

pymcmcstat.chain.ChainProcessing.**check_parallel_directory_contents**(*parallel_dir*, *cf_orig*)
Check that items in directory are subdirectories with name "chain_#"

Args:

- **parallel_dir** (str): Directory where parallel log files are saved.
- **cf_orig** (list): List of items contained in parallel directory.

Returns:

- **chainfolders** (list): List of items that match criteria in parallel directory.

pymcmcstat.chain.ChainProcessing.**generate_chain_list**(*pres*, *burnin_percentage=50*)
Generate list of chains.

Args:

- **pres** (list): Parallel results list.
- **burnin_percentage** (int): Percentage of chain to remove for burnin.

Returns:

- (list): Each element of list corresponds to different chain set.

pymcmcstat.chain.ChainProcessing.**generate_combined_chain_with_index**(*pres*, *burnin_percentage=50*)
Generate combined chain with index.

Args:

- **pres** (list): Parallel results list.

- **burnin_percentage** (int): Percentage of chain to remove for burnin.

Returns:

- (ndarray, list): Combined chain array, index label

pymcmcstat.chain.ChainProcessing.**load_json_object**(*filename*)
Load object stored in json file.

Note: Filename should include extension.

Args:

- **filename** (str): Load object from file with this name.

Returns:

- **results** (dict): Object loaded from file.

pymcmcstat.chain.ChainProcessing.**load_parallel_simulation_results**(*savedir*, *extension='h5'*, *chainfile='chainfile'*, *sschainfile='sschainfile'*, *s2chainfile='s2chainfile'*, *covchainfile='covchainfile'*)
Load results from parallel simulation directory json files.

Lists in json files are converted to numpy arrays.

Args:

- **savedir** (str): String indicated path to parallel directory.

Returns:

- **pres** (list): Each element of list is an MCMC result dictionary.

pymcmcstat.chain.ChainProcessing.**load_serial_simulation_results**(*savedir*, *json_file=None*, *extension='h5'*, *chainfile='chainfile'*, *sschainfile='sschainfile'*, *s2chainfile='s2chainfile'*, *covchainfile='covchainfile'*)
Load results from serial simulation directory json files.

Lists in json files are converted to numpy arrays.

Args:

- **savedir** (str): String indicated path to results directory.

Returns:

- **results** (dict): Each element of list is an MCMC result dictionary.

pymcmcstat.chain.ChainProcessing.**print_log_files**(*savedir*)

Print log files to screen.

Args:

- **savedir** (str): Directory where log files are saved.

The output display will include a date/time stamp, as well as indices of the chain that were saved during that export sequence.

Example display:

```
-------------------------
Display log file: <savedir>/binlogfile.txt
2018-05-03 14:15:54     0       999
2018-05-03 14:15:54     1000    1999
2018-05-03 14:15:55     2000    2999
2018-05-03 14:15:55     3000    3999
2018-05-03 14:15:55     4000    4999
-------------------------
```

pymcmcstat.chain.ChainProcessing.**read_in_bin_file**(*filename*)

Read in information from file containing binary data.

If file exists, it will read in the array elements. Otherwise, it will return and empty list.

Args:

- **filename** (str): Name of file to read.

Returns:

- **out** (ndarray): Array of chain elements.

pymcmcstat.chain.ChainProcessing.**read_in_parallel_json_results_files**(*parallel_dir*)

Read in json results files from directory containing results from parallel MCMC simulation.

Args:

- **parallel_dir** (str): Directory where parallel log files are saved.

pymcmcstat.chain.ChainProcessing.**read_in_parallel_savedir_files**(*parallel_dir*, *extension='h5'*, *chainfile='chainfile'*, *sschainfile='sschainfile'*, *s2chainfile='s2chainfile'*, *covchainfile='covchainfile'*)

Read in log files from directory containing results from parallel MCMC simulation.

Args:

- **parallel_dir** (str): Directory where parallel log files are saved.
- **extension** (str): Extension of files being loaded.
- **chainfile** (str): Name of chain log file.
- **sschainfile** (str): Name of sschain log file.

- **s2chainfile** (str): Name of s2chain log file.
- **covchainfile** (str): Name of covchain log file.

pymcmcstat.chain.ChainProcessing.**read_in_savedir_files**(*savedir*, *extension='h5'*, *chainfile='chainfile'*, *sschainfile='sschainfile'*, *s2chainfile='s2chainfile'*, *covchainfile='covchainfile'*)

Read in log files from directory.

Args:

- **savedir** (str): Directory where log files are saved.
- **extension** (str): Extension of files being loaded.
- **chainfile** (str): Name of chain log file.
- **sschainfile** (str): Name of sschain log file.
- **s2chainfile** (str): Name of s2chain log file.
- **covchainfile** (str): Name of covchain log file.

pymcmcstat.chain.ChainProcessing.**read_in_txt_file**(*filename*)

Read in information from file containing text data.

If file exists, it will read in the array elements. Otherwise, it will return and empty list.

Args:

- **filename** (str): Name of file to read.

Returns:

- **out** (ndarray): Array of chain elements.

pymcmcstat.chain.ChainStatistics module

Created on Thu Apr 26 10:23:51 2018

@author: prmiles

pymcmcstat.chain.ChainStatistics.**batch_mean_standard_deviation**(*chain*, *b=None*)

Standard deviation calculated from batch means

Args:

- **chain** (ndarray): Sampling chain.
- **b** (int): Step size.

Returns:

- **s** (ndarray): Batch mean standard deviation.

pymcmcstat.chain.ChainStatistics.**calculate_psrf**(*x*, *nsimu*, *nchains*)

Calculate Potential Scale Reduction Factor (PSRF)

Performs analysis of variances for set of chains corresponding to a single parameter. This code follows the MATLAB implementation found here:

https://users.aalto.fi/~ave/code/mcmcdiag/

Args:

- **x** (ndarray): Expect an [nsimu x nchains] array.
- **nsimu** (int): Number of simulations in each chain.
- **nchains** (int): Number of chains.

Returns:

- **dict**
  - R - PSRF
  - B - Between Sequence Variances
  - W - Within Sequence Variances
  - V - Mixture-of-Sequences Variances
  - neff - Effective number of samples

pymcmcstat.chain.ChainStatistics.**chainstats**(*chain=None*, *results=None*, *returnstats=False*, *display_details=False*)

Calculate chain statistics.

Args:

- **chain** (ndarray): Sampling chain.
- **results** (dict): Results from MCMC simulation.
- **returnstats** (bool): Flag to return statistics.

Returns:

- **stats** (dict): Statistical measures of chain convergence.

pymcmcstat.chain.ChainStatistics.**display_gelman_rubin**(*psrf*)

Display results of Gelman-Rubin diagnostic

Args:

- **psrf** (dict): Results from GR diagnostic

pymcmcstat.chain.ChainStatistics.**gelman_rubin**(*chains*, *names=None*, *results=None*, *display=True*)

Gelman-Rubin diagnostic for multiple chains [GR+92], [BG98].

This diagnostic technique compares the variance within a single change to the variance between multiple chains. This process serves as a method for testing whether or not the chain has converged. If the chain has converged, we would expect the variance within and the variance between to be equal. This diagnostic tool pairs well with the ParallelMCMC module, which generates a set of distinct chains that have all been initialized at different points within the parameter space.

Args:

- **chains** (list): List of arrays - each array corresponds to different chain set.
- **names** (list): List of strings - corresponds to parameter names.
- **results** (dict): Results from MCMC simulation.

Returns:

- (dict): Keywords of the dictionary correspond to the parameter names. Each keyword corresponds to a dictionary outputted from *calculate_psrf()*.

pymcmcstat.chain.ChainStatistics.**get_parameter_names**(*nparam*, *results*)
Get parameter names from results dictionary.

If no results found, then default names are generated. If some results are found, then an extended set is generated to complete the list requirement. Uses the functions: *generate_default_names()* and *extend_names_to_match_nparam()*

Args:

- **nparam** (int): Number of parameter names needed

Returns:

- **names** (list): List of length *nparam* with strings.

pymcmcstat.chain.ChainStatistics.**geweke**(*chain*, *a=0.1*, *b=0.5*)
Geweke's MCMC convergence diagnostic

Test for equality of the means of the first a% (default 10%) and last b% (50%) of a Markov chain - see [BR98].

Args:

- **chain** (ndarray): Sampling chain.
- **a** (float): First a% of chain.
- **b** (float): Last b% of chain.

Returns:

- **z** (ndarray): Convergence diagnostic prior to CDF.
- **p** (ndarray): Geweke's MCMC convergence diagnostic.

Note: The percentage of the chain should be given as a decimal between zero and one. So, for the first 10% of the chain, define `a = 0.1`. Likewise, for the last 50% of the chain, define `b = 0.5`.

---

pymcmcstat.chain.ChainStatistics.**integrated_autocorrelation_time**(*chain*)
Estimates the integrated autocorrelation time using Sokal's adaptive truncated periodogram estimator.

Args:

- **chain** (ndarray): Sampling chain.

Returns:

- **tau** (ndarray): Autocorrelation time.
- **m** (ndarray): Counter.

pymcmcstat.chain.ChainStatistics.**power_spectral_density_using_hanning_window**(*x*, *nfft=None*, *nw=None*)
Power spectral density using Hanning window.

Args:

- **x** (ndarray): Array of points - portion of chain.
- **nfft** (int): Length of Fourier transform.
- **nw** (int): Size of window.

Returns:

- **y** (ndarray): Power spectral density.

pymcmcstat.chain.ChainStatistics.**print_chain_acceptance_info**(*chain*, *results=None*)

Print chain acceptance rate(s)

If results structure is provided, it will try to print acceptance rates with respect to delayed rejection as well.

Example display (if results dictionary provided):

```
------------------------------
Acceptance rate information
---------------
Results dictionary:
Stage 1: 3.32%
Stage 2: 22.60%
Net    : 25.92% -> 1296/5000
---------------
Chain provided:
Net    : 32.10% -> 963/3000
---------------
Note, the net acceptance rate from the results dictionary
may be different if you only provided a subset of the chain,
e.g., removed the first part for burnin-in.
------------------------------
```

Args:

- **chain** (ndarray): Sampling chain.
- **results** (dict): Results from MCMC simulation. Default is *None*.

pymcmcstat.chain.ChainStatistics.**print_chain_statistics**(*names*, *meanii*, *stdii*, *mcerr*, *tau*, *p*)

Print chain statistics to terminal window.

Args:

- **names** (list): List of parameter names.
- **meanii** (list): Parameter mean values.
- **stdii** (list): Parameter standard deviation.
- **mcerr** (ndarray): Normalized batch mean standard deviation.
- **tau** (ndarray): Integrated autocorrelation time.
- **p** (ndarray): Geweke's convergence diagnostic.

Example display:

```
---------------------
name      :       mean        std     MC_err        tau     geweke
$p_{0}$   :     1.9680     0.0319     0.0013    36.3279     0.9979
$p_{1}$   :     3.0818     0.0803     0.0035    37.1669     0.9961
---------------------
```

pymcmcstat.chain.ChainStatistics.**spectral_estimate_for_variance**(*x*)

Spectral density at frequency zero.

Args:

- **x** (ndarray): Array of points - portion of chain.

Returns:

- **s** (ndarray): Spectral estimate for variance.

pymcmcstat.plotting package

pymcmcstat.plotting.MCMCPlotting module

Created on Wed Jan 31 12:54:16 2018

@author: prmiles

class pymcmcstat.plotting.MCMCPlotting.**Plot**

Bases: object

Plotting routines for analyzing sampling chains from MCMC process.

Attributes:

- *plot_density_panel()*
- *plot_chain_panel()*
- *plot_pairwise_correlation_panel()*
- *plot_histogram_panel()*
- *plot_chain_metrics()*

pymcmcstat.plotting.MCMCPlotting.**deprecation**(*message*)

pymcmcstat.plotting.MCMCPlotting.**plot_chain_metrics**(*chain*, *name=None*, *figsizeinches=None*)

Plot chain metrics for individual chain

- Scatter plot of chain
- Histogram of chain

Args:

- **chains** (ndarray): Sampling chain for specific parameter
- **names** (str): Name of each parameter
- **figsizeinches** (list): Specify figure size in inches [Width, Height]

pymcmcstat.plotting.MCMCPlotting.**plot_chain_panel**(*chains*, *names=None*, *figsizeinches=None*, *skip=1*, *maxpoints=500*)

Plot sampling chain for each parameter

Args:

- **chains** (ndarray): Sampling chain for each parameter
- **names** (list): List of strings - name of each parameter
- **figsizeinches** (list): Specify figure size in inches [Width, Height]
- **skip** (int): Indicates step size to be used when plotting elements from the chain
- **maxpoints** (int): Max number of display points - keeps scatter plot from becoming overcrowded

pymcmcstat.plotting.MCMCPlotting.**plot_density_panel**(*chains*, *names=None*, *hist_on=False*, *figsizeinches=None*, *return_kde=False*)

Plot marginal posterior densities

Args:

- **chains** (ndarray): Sampling chain for each parameter
- **names** (list): List of strings - name of each parameter
- **hist_on** (bool): Flag to include histogram on density plot
- **figsizeinches** (list): Specify figure size in inches [Width, Height]

pymcmcstat.plotting.MCMCPlotting.**plot_histogram_panel**(*chains*, *names=None*, *figsizeinches=None*)

Plot histogram from each parameter's sampling history

Args:

- **chains** (ndarray): Sampling chain for each parameter
- **names** (list): List of strings - name of each parameter
- **hist_on** (bool): Flag to include histogram on density plot
- **figsizeinches** (list): Specify figure size in inches [Width, Height]

pymcmcstat.plotting.MCMCPlotting.**plot_pairwise_correlation_panel**(*chains*, *names=None*, *figsizeinches=None*, *skip=1*, *maxpoints=500*)

Plot pairwise correlation for each parameter

Args:

- **chains** (ndarray): Sampling chain for each parameter
- **names** (list): List of strings - name of each parameter
- **figsizeinches** (list): Specify figure size in inches [Width, Height]
- **skip** (int): Indicates step size to be used when plotting elements from the chain
- **maxpoints** (py:class:*int*): Maximum allowable number of points in plot.

pymcmcstat.plotting.PredictionIntervals module

Created on Wed Nov 8 12:00:11 2017

@author: prmiles

class pymcmcstat.plotting.PredictionIntervals.**PredictionIntervals**

Bases: object

Prediction/Credible interval methods.

Attributes:

- *setup_prediction_interval_calculation()*
- *generate_prediction_intervals()*

- *plot_prediction_intervals()*

generate_prediction_intervals(*sstype=None*, *nsample=500*, *calc_pred_int=True*, *waitbar=False*)

Generate prediction/credible interval.

Args:

- **sstype** (int): Sum-of-squares type
- **nsample** (int): Number of samples to use in generating intervals.
- **calc_pred_int** (bool): Flag to turn on prediction interval calculation.
- **waitbar** (bool): Flag to turn on progress bar.

plot_prediction_intervals(*plot_pred_int=True*, *adddata=False*, *addlegend=True*, *figsizeinches=None*, *model_display={}*, *data_display={}*, *interval_display={}*)

Plot prediction/credible intervals.

Args:

- **plot_pred_int** (bool): Flag to include PI on plot.
- **adddata** (bool): Flag to include data on plot.
- **addlegend** (bool): Flag to include legend on plot.
- **figsizeinches** (list): Specify figure size in inches [Width, Height].
- **model_display** (dict): Model display settings.
- **data_display** (dict): Data display settings.
- **interval_display** (dict): Interval display settings.

Available display options (defaults in parantheses):

- **model_display**: *linestyle* (`'-'`), *marker* (`''`), *color* (`'r'`), *linewidth* (2), *markersize* (5), *label* (`model`), *alpha* (`1.0`)
- **data_display**: *linestyle* (`''`), *marker* (`'.'`), *color* (`'b'`), *linewidth* (1), *markersize* (5), *label* (`data`), *alpha* (`1.0`)
- **data_display**: *linestyle* (`':'`), *linewidth* (1), *alpha* (`1.0`), *edgecolor* (`'k'`)

setup_prediction_interval_calculation(*results*, *data*, *modelfunction*, *burnin=0*)

Setup calculation for prediction interval generation

Args:

- results (*ResultsStructure*): MCMC results structure
- data (*DataStructure*): MCMC data structure
- modelfunction: Model function handle

pymcmcstat.plotting.utilities module

Created on Mon May 14 06:24:12 2018

@author: prmiles

pymcmcstat.plotting.utilities.**append_to_nrow_ncol_based_on_shape**(*sh*, *nrow*, *ncol*)

Append to list based on shape of array

Args:

- **sh** (tuple): Shape of array.
- **nrow** (list): List of number of rows
- **ncol** (list): List of number of columns

Returns:

- **nrow** (list): List of number of rows
- **ncol** (list): List of number of columns

pymcmcstat.plotting.utilities.**check_settings**(*default_settings*, *user_settings=None*)

Check user settings with default.

Recursively checks elements of user settings against the defaults and updates settings as it goes. If a user setting does not exist in the default, then the user setting is added to the settings. If the setting is defined in both the user and default settings, then the user setting overrides the default. Otherwise, the default settings persist.

Args:

- **default_settings** (dict): Default settings for particular method.
- **user_settings** (dict): User defined settings.

Returns:

- (dict): Updated settings.

pymcmcstat.plotting.utilities.**check_symmetric**(*a*, *tol=1e-08*)

Check if array is symmetric by comparing with transpose.

Args:

- **a** (ndarray): Array to test.
- **tol** (float): Tolerance for testing equality.

Returns:

- (bool): True -> symmetric, False -> not symmetric.

pymcmcstat.plotting.utilities.**convert_flag_to_boolean**(*flag*)

Convert flag to boolean for backwards compatibility.

Args:

- **flag** (bool or int): Flag to specify something.

Returns:

- **flag** (bool): Flag to converted to boolean.

pymcmcstat.plotting.utilities.**empirical_quantiles**(*x*, *p=array([0.25*, *0.5*, *0.75])*)

Calculate empirical quantiles.

Args:

- **x** (ndarray): Observations from which to generate quantile.
- **p** (ndarray): Quantile limits.

Returns:

- (ndarray): Interpolated quantiles.

pymcmcstat.plotting.utilities.**extend_names_to_match_nparam**(*names*, *nparam*)
Append names to list using default convention until length of names matches number of parameters.

For example, if *names = ['name_1', 'name_2']* and *nparam = 4*, then two additional names will be appended to the *names* list. E.g.,:

```
names = ['name_1', 'name_2', 'p_{2}', 'p_{3}']
```

Args:

- **names** (list): Names of parameters provided by user
- **nparam** (int): Number of parameter names to generate

Returns:

- **names** (list): List of strings - extended list of parameter names

pymcmcstat.plotting.utilities.**gaussian_density_function**(*x*, *mu=0*, *sigma2=1*)
Standard normal/Gaussian density function.

Args:

- **x** (float): Value of which to calculate probability.
- **mu** (float): Mean of Gaussian distribution.
- **sigma2** (float): Variance of Gaussian distribution.

Returns:

- **y** (float): Likelihood of *x*.

pymcmcstat.plotting.utilities.**generate_default_names**(*nparam*)
Generate generic parameter name set.

For example, if *nparam* = 4, then the generated names are:

```
names = ['p_{0}', 'p_{1}', 'p_{2}', 'p_{3}']
```

Args:

- **nparam** (int): Number of parameter names to generate

Returns:

- **names** (list): List of strings - parameter names

pymcmcstat.plotting.utilities.**generate_ellipse**(*mu*, *cmat*, *ndp=100*)
Generates points for a probability contour ellipse

Args:

- **mu** (ndarray): Mean values
- **cmat** (ndarray): Covariance matrix
- **npd** (int): Number of points to generate

Returns:

- **x** (ndarray): x-points
- **y** (ndarray): y-points

pymcmcstat.plotting.utilities.**generate_names**(*nparam*, *names*)
Generate parameter name set.

For example, if *nparam* = 4, then the generated names are:

```
names = ['p_{0}', 'p_{1}', 'p_{2}', 'p_{3}']
```

Args:

- **nparam** (int): Number of parameter names to generate
- **names** (list): Names of parameters provided by user

Returns:

- **names** (list): List of strings - parameter names

pymcmcstat.plotting.utilities.**generate_subplot_grid**(*nparam=2*)
Generate subplot grid.

For example, if *nparam* = 2, then the subplot will have 2 rows and 1 column.

Args:

- **nparam** (int): Number of parameters

Returns:

- **ns1** (int): Number of rows in subplot
- **ns2** (int): Number of columns in subplot

pymcmcstat.plotting.utilities.**iqrange**(*x*)
Interquantile range of each column of x.

Args:

- **x** (ndarray): Array of points.

Returns:

- (ndarray): Interquantile range - single element array, *q3 - q1*.

pymcmcstat.plotting.utilities.**is_semi_pos_def_chol**(*x*)
Check if matrix is semi positive definite by calculating Cholesky decomposition.

Args:

- **x** (ndarray): Matrix to check

Returns:

- If matrix is *not* semi positive definite return `False, None`
- If matrix is semi positive definite return `True` and the Upper triangular form of the Cholesky decomposition matrix.

pymcmcstat.plotting.utilities.**make_x_grid**(*x*, *npts=100*)
Generate x grid based on extrema.

1. If *len(x)* > *200*, then generates grid based on difference between the max and min values in the array.

2. Otherwise, the grid is defined with respect to the array mean plus or minus four standard deviations.

Args:

- **x** (ndarray): Array of points

- **npts** (int): Number of points to use in generated grid

Returns:

- Uniformly spaced array of points with shape = (npts,1). (ndarray)

pymcmcstat.plotting.utilities.**scale_bandwidth**(*x*)
Scale bandwidth of array.

Args:

- **x** (ndarray): Array of points - column of chain.

Returns:

- **s** (ndarray): Scaled bandwidth - single element array.

pymcmcstat.plotting.utilities.**set_local_parameters**(*ii*, *local*)
Set local parameters based on tests.

Test 1

- *local == 0*

Test 2

- *local == ii*

Args:

- **ii** (int): Index.
- **local** (ndarray): Local flags.

Returns:

- **test** (ndarray): Array of Booleans indicated test results.

pymcmcstat.plotting.utilities.**setup_plot_features**(*nparam*, *names*, *figsizeinches*)
Setup plot features.

Args:

- **nparam** (int): Number of parameters
- **names** (list): Names of parameters provided by user
- **figsizeinches** (list): [Width, Height]

Returns:

- **ns1** (int): Number of rows in subplot
- **ns2** (int): Number of columns in subplot
- **names** (list): List of strings - parameter names
- **figsizeiches** (list): [Width, Height]

pymcmcstat.plotting.utilities.**setup_subsample**(*skip*, *maxpoints*, *nsimu*)
Setup subsampling from posterior.

When plotting the sampling chain, it is often beneficial to subsample in order to avoid to dense of plots. This routine determines the appropriate step size based on the size of the chain (nsimu) and maximum points allowed to plot (maxpoints). The function checks if the size of the chain exceeds the maximum number of points allowed in the plot. If yes, skip is defined such that every the max number of points are used and sampled evenly from

the start to end of the chain. Otherwise the value of skip is return as defined by the user. A subsample index is then generated based on the value of skip and the number of simulations.

Args:

- **skip** (int): User defined skip value.
- **maxpoints** (int): Maximum points allowed in each plot.

Returns:

- (int): Skip value.

pymcmcstat.procedures package

pymcmcstat.procedures.CovarianceProcedures module

Created on Thu Jan 18 07:55:46 2018

Description: Support methods for initializing and updating the covariance matrix. Additional routines associated with Cholesky Decomposition.

@author: prmiles

class pymcmcstat.procedures.CovarianceProcedures.**CovarianceProcedures**
Bases: object

Covariance matrix variables and methods.

Attributes:

- *display_covariance_settings()*
- *setup_covariance_matrix()*

display_covariance_settings(*print_these=None*)
Display subset of the covariance settings.

Args:

- **print_these** (list): List of strings corresponding to keywords. Default below.

```
print_these = ['qcov', 'R', 'RDR', 'invR', 'last_index_since_adaptation',
↪'covchain']
```

setup_covariance_matrix(*qcov*, *thetasig*, *value*)
Initialize covariance matrix.

If no proposal covariance matrix is provided, then the default is generated by squaring 5% of the initial value. This yields a diagonal covariance matrix.

$$V = diag([(0.05\theta_i)^2])$$

If the initial value was one, this would lead to zero variance. In those instances the variance is set equal to `qcov[qcov==0] = 1.0`.

Args:

- **qcov** (ndarray): Parameter covariance matrix.
- **thetasig** (ndarray): Prior variance.
- **value** (ndarray): Current parameter value.

pymcmcstat.procedures.ErrorVarianceEstimator module

Created on Thu Jan 18 13:12:50 2018

@author: prmiles

class pymcmcstat.procedures.ErrorVarianceEstimator.**ErrorVarianceEstimator**
Bases: object

Estimate observation errors.

Attributes:

- *update_error_variance()*
- *gammar()*
- *gammar_mt()*

gammar(*m*, *n*, *a*, *b=1*)
Random deviates from gamma distribution.

Returns a m x n matrix of random deviates from the Gamma distribution with shape parameter A and scale parameter B:

$$p(x|A,B) = \frac{B^{-A}}{\Gamma(A)} * x^{A-1} * \exp(-x/B)$$

Args:

- **m** (int): Number of rows in return
- **n** (int): Number of columns in return
- **a** (float): Shape parameter
- **b** (float): Scaling parameter

gammar_mt(*m*, *n*, *a*, *b=1*)
Wrapper routine for calculating random deviates from gamma distribution using method of Marsaglia and Tsang (2000) [MT00].

Args:

- **m** (int): Number of rows in return
- **n** (int): Number of columns in return
- **a** (float): Shape parameter
- **b** (float): Scaling parameter

update_error_variance(*sos*, *model*)
Update observation error variance.

Strategy: Treat error variance σ^2 as parameter to be sampled.

Definition: The property that the prior and posterior distributions have the same parametric form is termed conjugacy.

Starting from the likelihood function, it can be shown

$$\sigma^2|(\nu, q) \sim \text{Inv-Gamma}\left(\frac{N_s + N}{2}, \frac{N_s\sigma_s^2 + SS_q}{2}\right)$$

where N_s and σ_s^2 are shape and scaling parameters, N is the number of observations, and SS_q is the sum-of-squares error. For more details regarding the interpretation of N_s and σ_s^2, please refer to [Smi14] page 163.

Note: The variables N_s and σ_s^2 correspond to `N0` and `S20` in the *`ModelSettings`* class, respectively.

Args:

- **sos** (`ndarray`): Return argument from evaluation of sum-of-squares function.
- **model** (*`ModelSettings`*): MCMC model settings.

pymcmcstat.procedures.PriorFunction module

Created on Thu Jan 18 09:10:21 2018

Description: Prior function

@author: prmiles

class pymcmcstat.procedures.PriorFunction.**PriorFunction**(*priorfun=None*, *mu=array([0])*, *sigma=array([inf])*)

Bases: `object`

Prior distribution functions.

Attributes:

- *`default_priorfun()`*
- *`evaluate_prior()`*

classmethod default_priorfun(*theta*, *mu*, *sigma*)

Default prior function - Gaussian.

$$\pi_0(q) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big[-\frac{1}{2}\Big(\frac{q-\mu}{\sigma}\Big)^2\Big]$$

Args:

- **theta** (`ndarray`): Current parameter values.
- **mu** (`ndarray`): Prior mean.
- **sigma** (`ndarray`): Prior standard deviation.

evaluate_prior(*theta*)

Evaluate the prior function.

Args:

- **theta** (`ndarray`): Current parameter values.

pymcmcstat.procedures.SumOfSquares module

Created on Wed Jan 17 16:21:48 2018

@author: prmiles

class pymcmcstat.procedures.SumOfSquares.**SumOfSquares**(*model*, *data*, *parameters*)
Bases: object

Sum-of-squares function evaluation.

Description: Sum-of-squares (sos) class intended for used in MCMC simulator. Each instance will contain the sos function. If the user did not specify a sos-function, then the user supplied model function will be used in the default mcmc sos-function.

Attributes:

- *evaluate_sos_function()*
- *mcmc_sos_function()*

evaluate_sos_function(*theta*, *custom=None*)
Evaluate sum-of-squares function.

Args:

- **theta** (ndarray): Parameter values.

Returns:

- **ss** (ndarray): Sum-of-squares error(s)

classmethod mcmc_sos_function(*theta*, *data*, *nbatch*, *model_function*)
Default sum-of-squares function.

Note: This method requires specifying a model function instead of a sum of squares function. Not recommended for most applications.

Basic formulation:

$$SS_{q,i} = \sum [w_i (y_i^{data} - y_i^{model})^2]$$

where w_i is the weight of a particular data set, and $SS_{q,i}$ is the sum-of-squares error for the i-th data set.

Args:

- **theta** (ndarray): Parameter values.

Returns:

- **ss** (ndarray): Sum-of-squares error(s)

pymcmcstat.samplers package

pymcmcstat.samplers.Adaptation module

Created on Thu Jan 18 11:14:11 2018

@author: prmiles

class pymcmcstat.samplers.Adaptation.**Adaptation**
Bases: object

Adaptive Metropolis (AM) algorithm based on [HST+01].

Attributes:

- *run_adaptation()*

run_adaptation(*covariance*, *options*, *isimu*, *iiadapt*, *rejected*, *chain*, *chainind*, *u*, *npar*, *alpha*)
Run adaptation step

Args:

- **covariance** (*CovarianceProcedures*): Covariance methods and variables
- **options** (*SimulationOptions*): Options for MCMC simulation
- **isimu** (int): Simulation counter
- **iiadapt** (int): Adaptation counter
- **rejected** (dict): Rejection counter
- **chain** (ndarray): Sampling chain
- **chainind** (ind): Relative point in chain
- **u** (ndarray): Latest random sample points
- **npar** (int): Number of parameters being sampled
- **alpha** (float): Latest Likelihood evaluation

Returns:

- **covariance** (*CovarianceProcedures*): Updated covariance object

pymcmcstat.samplers.Adaptation.**adjust_cov_matrix**(*upcov*, *R*, *npar*, *qcov_adjust*, *qcov_scale*, *rejected*, *iiadapt*, *verbosity*)
Adjust covariance matrix if found to be singular.

Args:

- **upcov** (ndarray): Parameter covariance matrix.
- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **npar** (int): Number of parameters.
- **qcov_adjust** (float): Covariance adjustment factor.
- **qcov_scale** (float): Scale parameter
- **rejected** (dict): Rejection counters.
- **iiadapt** (int): Adaptation counter.
- **verbosity** (int): Verbosity of display output.

Returns:

- **R** (ndarray): Cholesky decomposition of covariance matrix.

pymcmcstat.samplers.Adaptation.**below_burnin_threshold**(*rejected*, *iiadapt*, *R*, *burnin_scale*, *verbosity*)
Update Cholesky Matrix using below burnin thershold

Args:

- **rejected** (dict): Rejection counters.
- **iiadapt** (int): Adaptation counter
- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **burnin_scale** (float): Scale for burnin.

- **verbosity** (int): Verbosity of display output.

Returns:

- **R** (ndarray): Scaled Cholesky matrix.

pymcmcstat.samplers.Adaptation.**check_for_singular_cov_matrix**(*upcov*, *R*, *npar*, *qcov_adjust*, *qcov_scale*, *rejected*, *iiadapt*, *verbosity*)

Check if singular covariance matrix

Args:

- **upcov** (ndarray): Parameter covariance matrix.
- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **npar** (int): Number of parameters.
- **qcov_adjust** (float): Covariance adjustment factor.
- **qcov_scale** (float): Scale parameter
- **rejected** (dict): Rejection counters.
- **iiadapt** (int): Adaptation counter.
- **verbosity** (int): Verbosity of display output.

Returns:

- **R** (ndarray): Adjusted Cholesky decomposition of covariance matrix.

pymcmcstat.samplers.Adaptation.**cholupdate**(*R*, *x*)

Update Cholesky decomposition

Args:

- **R** (ndarray): Weighted Cholesky decomposition
- **x** (ndarray): Weighted sum based on local chain update

Returns:

- **R1** (ndarray): Updated Cholesky decomposition

pymcmcstat.samplers.Adaptation.**initialize_covariance_mean_sum**(*x*, *w*)

Initialize covariance chain, local mean, local sum

Args:

- **x** (ndarray): Chain segment
- **w** (ndarray): Weights

Returns:

- **xcov** (ndarray): Initial covariance matrix
- **xmean** (ndarray): Initial mean chain values
- **wsum** (ndarray): Initial weighted sum

pymcmcstat.samplers.Adaptation.**is_semi_pos_def_chol**(*x*)

Check if matrix is semi-positive definite using Cholesky Decomposition

Args:

- **x** (ndarray): Covariance matrix

Returns:

- *Boolean*
- **c** (ndarray): Cholesky decomposition (upper triangular form) or *None*

pymcmcstat.samplers.Adaptation.**scale_cholesky_decomposition**(*Ra*, *qcov_scale*)
Scale Cholesky decomposition

Args:

- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **qcov_scale** (float): Scale factor

Returns:

- **R** (ndarray): Scaled Cholesky decomposition of covariance matrix.

pymcmcstat.samplers.Adaptation.**setup_cholupdate**(*R*, *oldwsum*, *wsum*, *oldmean*, *xi*)
Setup input arguments to the Cholesky update function

Args:

- **R** (ndarray): Previous Cholesky decomposition matrix
- **oldwsum** (ndarray): Previous weighted sum
- **w** (ndarray): Current Weights
- **oldmean** (ndarray): Previous mean chain values
- **xi** (ndarray): Row of chain segment

Returns:

- **Rin** (ndarray): Scaled Cholesky decomposition matrix
- **xin** (ndarray): Updated mean chain values for Cholesky function

pymcmcstat.samplers.Adaptation.**setup_w_R**(*w*, *oldR*, *n*)
Setup weights and Cholesky matrix

Args:

- **x** (ndarray): Chain segment
- **w** (ndarray): Weights
- **oldcov** (ndarray or *None*): Previous covariance matrix
- **oldmean** (ndarray): Previous mean chain values
- **oldwsum** (ndarray): Previous weighted sum
- **oldR** (ndarray): Previous Cholesky decomposition matrix

Returns:

- **w** (ndarray): Weights
- **R** (ndarray): Previous Cholesky decomposition matrix

pymcmcstat.samplers.Adaptation.**unpack_covariance_settings**(*covariance*)
Unpack covariance settings

Args:

- **covariance** (*CovarianceProcedures*): Covariance methods and variables

Returns:

- **last_index_since_adaptation** (int): Last index since adaptation occured.
- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **oldcovchain** (ndarray): Covariance matrix history.
- **oldmeanchain** (ndarray): Current mean chain values.
- **oldwsum** (ndarray): Weights
- **no_adapt_index** (numpy.ndarray): Indices of parameters not being adapted.
- **qcov_scale** (float): Scale parameter
- **qcov** (ndarray): Covariance matrix

pymcmcstat.samplers.Adaptation.**unpack_simulation_options**(*options*)

Unpack simulation options

Args:

- **options** (*SimulationOptions*): Options for MCMC simulation

Returns:

- **burnintime** (int):
- **burnin_scale** (float): Scale for burnin.
- **ntry** (int): Number of tries to take before rejection. Default is method dependent.
- **drscale** (ndarray): Reduced scale for sampling in DR algorithm. Default is [5,4,3].
- **alphatarget** (float): Acceptance ratio target.
- **etaparam** (float):
- **qcov_adjust** (float): Adjustment scale for covariance matrix.
- **doram** (int): Flag to perform 'ram' algorithm (Obsolete).
- **verbosity** (int): Verbosity of display output.

pymcmcstat.samplers.Adaptation.**update_cov_from_covchain**(*covchain*, *qcov*, *no_adapt_index*)

Update covariance matrix from covariance matrix chain

Args:

- **covchain** (ndarray): Covariance matrix history.
- **qcov** (ndarray): Parameter covariance matrix
- **no_adapt_index** (numpy.ndarray): Indices of parameters not being adapted.

Returns:

- **upcov** (ndarray): Updated covariance matrix

pymcmcstat.samplers.Adaptation.**update_cov_via_ram**(*u*, *isimu*, *etaparam*, *npar*, *alphatarget*, *alpha*, *R*)

Update covariance matrix via RAM

Args:

- **u** (ndarray): Latest random sample points

- **isimu** (int): Simulation counter
- **alphatarget** (float): Acceptance ratio target.
- **npar** (int): Number of parameters.
- **etaparam** (float):
- **alpha** (float): Latest Likelihood evaluation
- **R** (ndarray): Cholesky decomposition of covariance matrix.

Returns:

- **upcov** (ndarray): Updated parameter covariance matrix.

pymcmcstat.samplers.Adaptation.**update_covariance_mean_sum**(*x*, *w*, *oldcov*, *oldmean*, *oldwsum*, *oldR=None*)

Update covariance chain, local mean, local sum

Args:

- **x** (ndarray): Chain segment
- **w** (ndarray): Weights
- **oldcov** (ndarray or *None*): Previous covariance matrix
- **oldmean** (ndarray): Previous mean chain values
- **oldwsum** (ndarray): Previous weighted sum
- **oldR** (ndarray): Previous Cholesky decomposition matrix

Returns:

- **xcov** (ndarray): Updated covariance matrix
- **xmean** (ndarray): Updated mean chain values
- **wsum** (ndarray): Updated weighted sum

pymcmcstat.samplers.Adaptation.**update_delayed_rejection**(*R*, *npar*, *ntry*, *drscale*)

Update Cholesky/Inverse matrix for Delayed Rejection

Args:

- **R** (ndarray): Cholesky decomposition of covariance matrix.
- **npar** (int): Number of parameters.
- **ntry** (int): Number of tries to take before rejection. Default is method dependent.
- **drscale** (ndarray): Reduced scale for sampling in DR algorithm. Default is [5,4,3].

Returns:

- **RDR** (list): List of Cholesky decomposition of covariance matrices for each stage of DR.
- **InvR** (list): List of Inverse Cholesky decomposition of covariance matrices for each stage of DR.

pymcmcstat.samplers.DelayedRejection module

Created on Thu Jan 18 10:42:07 2018

@author: prmiles

class pymcmcstat.samplers.DelayedRejection.**DelayedRejection**
Bases: object

Delayed Rejection (DR) algorithm based on [HLMS06].

Attributes:

- *run_delayed_rejection()*
- *initialize_next_metropolis_step()*
- *alphafun()*

alphafun(*trypath*, *invR*)
Calculate likelihood according to DR

Args:

- **trypath** (list): Sequence of DR steps
- **invR** (ndarray): Inverse Cholesky decomposition matrix

Returns:

- **alpha** (float): Result of likelihood function according to delayed rejection

classmethod initialize_next_metropolis_step(*npar*, *old_theta*, *sigma2*, *RDR*)
Take metropolis step according to DR

Args:

- **npar** (int): Number of parameters
- **old_theta** (ndarray): *q^{k-1}*
- **sigma2** (float): Observation error variance
- **RDR** (ndarray): Cholesky decomposition of parameter covariance matrix for DR steps
- **itry** (int): DR step counter

Returns:

- **next_set** (*ParameterSet*): New proposal set
- **u** (numpy.ndarray): Numbers sampled from standard normal distributions (`u.shape = (1,npar)`)

run_delayed_rejection(*old_set*, *new_set*, *RDR*, *ntry*, *parameters*, *invR*, *sosobj*, *priorobj*, *custom=None*)
Perform delayed rejection step - occurs in standard metropolis is not accepted.

Args:

- **old_set** (*ParameterSet*): Features of q^{k-1}
- **new_set** (*ParameterSet*): Features of q^{*}
- **RDR** (ndarray): Cholesky decomposition of parameter covariance matrix for DR steps
- **ntry** (int): Number of DR steps to perform until rejection
- **parameters** (*ModelParameters*): Model parameters
- **invR** (ndarray): Inverse Cholesky decomposition matrix
- **sosobj** (*SumOfSquares*): Sum-of-Squares function
- **priorobj** (*PriorFunction*): Prior function

Returns:

- **accept** (int): 0 - reject, 1 - accept
- **out_set** (*ParameterSet*): If accept == 1, then latest DR set; Else, $q^k = q^{k-1}$
- **outbound** (int): 1 - rejected due to sampling outside of parameter bounds

pymcmcstat.samplers.DelayedRejection.**extract_state_elements**(*iq*, *stage*, *trypath*)

Extract elements from tried paths.

Args:

- **iq** (int): Stage number.
- **stage** (int): Number of stages - 2
- **trypath** (list): Sequence of DR steps

pymcmcstat.samplers.DelayedRejection.**log_posterior_ratio**(*x1*, *x2*)

Calculate the logarithm of the posterior ratio.

Args:

- **x1** (*ParameterSet*): Old set - q^{k-1}
- **x2** (*ParameterSet*): New set - q^*

Returns:

- **zq** (float): Logarithm of posterior ratio.

pymcmcstat.samplers.DelayedRejection.**nth_stage_log_proposal_ratio**(*iq*, *trypath*, *invR*)

Gaussian nth stage log proposal ratio.

Logarithm of $q_i(y_n, ..., y_{n-j})/q_i(x, y_1, ..., y_j)$

Args:

- **iq** (int): Stage number.
- **trypath** (list): Sequence of DR steps
- **invR** (ndarray): Inverse Cholesky decomposition matrix

Returns:

- **zq** (float): Logarithm of Gaussian nth stage proposal ratio.

pymcmcstat.samplers.DelayedRejection.**update_set_based_on_acceptance**(*accept*, *old_set*, *next_set*)

Define output set based on acceptance

$$
\begin{aligned}
&\text{If } u_\alpha < \alpha, \\
&\quad \text{Set } q^k = q^*,\ SS_{q^k} = SS_{q^*} \\
&\text{Else} \\
&\quad \text{Set } q^k = q^{k-1},\ SS_{q^k} = SS_{q^{k-1}}
\end{aligned}
$$

Args:

- **accept** (int): 0 - reject, 1 - accept
- **old_set** (*ParameterSet*): Features of q^{k-1}
- **next_set** (*ParameterSet*): New proposal set

Returns:

- **out_set** (*ParameterSet*): If accept == 1, then latest DR set; Else, $q^k = q^{k-1}$

pymcmcstat.samplers.Metropolis module

Created on Thu Jan 18 10:30:29 2018

@author: prmiles

class pymcmcstat.samplers.Metropolis.**Metropolis**

Bases: object

Pseudo-Algorithm:

1. Sample $z_k \sim N(0,1)$
2. Construct candidate $q^* = q^{k-1} + Rz_k$
3. Compute $SS_{q^*} = \sum_{i=1}^{N}[v_i - f_i(q^*)]^2$
4. Compute $\alpha = \min\left(1, e^{[SS_{q^*} - SS_{q^{k-1}}]/(2\sigma_{k-1}^2)}\right)$
5. **If** $u_\alpha < \alpha$, Set $q^k = q^*$, $SS_{q^k} = SS_{q^*}$

   Else Set $q^k = q^{k-1}$, $SS_{q^k} = SS_{q^{k-1}}$

Attributes:

- acceptance_test()
- *run_metropolis_step()*
- *unpack_set()*

classmethod calculate_posterior_ratio(*ss1*, *ss2*, *sigma2*, *newprior*, *oldprior*)

Calculate acceptance ratio

$$\alpha = \min\left[1, \frac{\mathcal{L}(\nu_{obs}|q^*, \sigma_{k-1}^2)\pi_0(q^*)}{\mathcal{L}(\nu_{obs}|q^{k-1}, \sigma_{k-1}^2)\pi_0(q^{k-1})}\right]$$

where the Gaussian likelihood function is

$$\mathcal{L}(\nu_{obs}|q, \sigma) = \exp\Big(-\frac{SS_q}{2\sigma}\Big)$$

and Gaussian prior function is

$$\pi_0(q) = \exp\Big[-\frac{1}{2}\Big(\frac{q-\mu_0}{\sigma_0}\Big)^2\Big].$$

For the Gaussian likelihood and prior, this yields the acceptance ratio

$$\alpha = \exp\Big[-0.5\Big(\sum\Big(\frac{SS_{q^*} - SS_{q^{k-1}}}{\sigma_{k-1}^2}\Big) + p_1 - p_2\Big)\Big].$$

For more details regarding the prior function, please refer to the *PriorFunction* class.

Note: The default behavior of the package is to use Gaussian likelihood and prior functions (as of v1.8.0). Future releases will expand the functionality to allow for alternative likelihood and prior definitions.

Args:

- **ss1** (ndarray): SS error from proposed candidate, q^*
- **ss2** (ndarray): SS error from previous sample point, q^{k-1}
- **sigma2** (ndarray): Error variance estimate from previous sample point, σ^2_{k-1}
- **newprior** (ndarray): Prior for proposal candidate
- **oldprior** (ndarray): Prior for previous sample

Returns:

- **alpha** (float): Result of likelihood function

run_metropolis_step(*old_set*, *parameters*, *R*, *prior_object*, *sos_object*, *custom=None*)
Run Metropolis step.

Args:

- **old_set** (*ParameterSet*): Features of q^{k-1}
- **parameters** (*ModelParameters*): Model parameters
- **R** (ndarray): Cholesky decomposition of parameter covariance matrix
- **priorobj** (*PriorFunction*): Prior function
- **sosobj** (*SumOfSquares*): Sum-of-Squares function

Returns:

- **accept** (int): 0 - reject, 1 - accept
- **newset** (*ParameterSet*): Features of q^*
- **outbound** (int): 1 - rejected due to sampling outside of parameter bounds
- **npar_sample_from_normal** (ndarray): Latet random sample points

classmethod unpack_set(*parset*)
Unpack parameter set

Args:

- **parset** (*ParameterSet*): Parameter set to unpack

Returns:

- **theta** (ndarray): Value of sampled model parameters
- **ss** (ndarray): Sum-of-squares error using sampled value
- **prior** (ndarray): Value of prior
- **sigma2** (ndarray): Error variance

pymcmcstat.samplers.SamplingMethods module

Created on Thu Jan 18 10:11:40 2018

@author: prmiles

class pymcmcstat.samplers.SamplingMethods.**SamplingMethods**
Bases: object

Metropolis sampling methods.

Attributes:

- *Metropolis*
- *DelayedRejection*
- *Adaptation*

pymcmcstat.samplers.utilities module

Created on Wed Jun 27 12:07:11 2018

Utility functions used by different samplers

@author: prmiles

pymcmcstat.samplers.utilities.**acceptance_test**(*alpha*)
Run standard acceptance test

$$\begin{aligned}&\text{If } u_\alpha < \alpha,\\ &\quad \text{Set } q^k = q^*,\ SS_{q^k} = SS_{q^*}\\ &\text{Else}\\ &\quad \text{Set } q^k = q^{k-1},\ SS_{q^k} = SS_{q^{k-1}}\end{aligned}$$

Args:

- **alpha** (float): Result of likelihood function

Returns:

- **accept** (bool): False - reject, True - accept

pymcmcstat.samplers.utilities.**calculate_log_posterior_ratio**(*loglikestar*, *loglike*, *logpriorstar*, *logprior*)
Calculate log posterior ratio:

$$\log(\alpha) = \min\left[0, \log(\mathcal{L}(\nu_{obs}|q^*)) + \log(\pi_0(q^*)) - \log(\mathcal{L}(\nu_{obs}|q^{k-1})) - \log(\pi_0(q^{k-1}))\right]$$

For more details regarding the prior and likelihood functions, please refer to the *PriorFunction* and LikelihoodFunction class, respectively.

Note: The default behavior of the package is to use Gaussian likelihood and prior functions (as of v1.8.0). Future releases will expand the functionality to allow for alternative likelihood and prior definitions.

Args:

- **likestar** (float): Likelihood from proposed candidate, q^*
- **like** (float): Likelihood from previous sample point, q^{k-1}
- **priorstar** (float): Prior for proposal candidate
- **prior** (float): Prior for previous sample

Returns:

- **alpha** (float): Acceptance ratio

pymcmcstat.samplers.utilities.**is_sample_outside_bounds**(*theta*, *lower_limits*, *upper_limits*)

Check whether proposal value is outside parameter limits

Args:

- **theta** (ndarray): Value of sampled model parameters
- **lower_limits** (ndarray): Lower limits
- **upper_limits** (ndarray): Upper limits

Returns:

- **outsidebounds** (bool): True -> Outside of parameter limits

pymcmcstat.samplers.utilities.**log_posterior_ratio_acceptance_test**(*alpha*)

Run log posterior ratio acceptance test

Args:

- **alpha** (float): Log posterior ratio

Returns:

- **accept** (bool): False - reject, True - accept

pymcmcstat.samplers.utilities.**posterior_ratio_acceptance_test**(*alpha*)

Run posterior ratio acceptance test

Args:

- **alpha** (float): Posterior ratio

Returns:

- **accept** (bool): False - reject, True - accept

pymcmcstat.samplers.utilities.**sample_candidate_from_gaussian_proposal**(*npar*, *oldpar*, *R*)

Sample candidate from Gaussian proposal distribution

Args:

- **npar** (int): Number of parameters being samples
- **oldpar** (ndarray): q^{k-1} Old parameter set.
- **R** (ndarray): Cholesky decomposition of parameter covariance matrix.

Returns:

- **newpar** (ndarray): q^* - candidate
- **npar_sample_from_sample** (ndarray): Sampled values from normal distibution: $N(0, 1)$.

pymcmcstat.samplers.utilities.**set_outside_bounds**(*next_set*)

Assign set features based on being outside bounds

Args:

- **next_set** (*ParameterSet*): q^*

Returns:

- **next_set** (*ParameterSet*): q^* with updated features
- **outbound** (bool): True

pymcmcstat.settings package

pymcmcstat.settings.DataStructure module

Created on Wed Jan 17 09:03:37 2018

@author: prmiles

class pymcmcstat.settings.DataStructure.**DataStructure**
Bases: object

Structure for storing data in MCMC object. The following random data sets will be referenced in examples for the different class methods:

```
x1 = np.random.random_sample(size = (5, 1))
y1 = np.random.random_sample(size = (5, 2))

x2 = np.random.random_sample(size = (10, 1))
y2 = np.random.random_sample(size = (10, 3))
```

Attributes:

- *add_data_set()*
- *get_number_of_batches()*
- *get_number_of_data_sets()*
- *get_number_of_observations()*

add_data_set(*x*, *y*, *n=None*, *weight=1*, *user_defined_object=0*)
Add data set to MCMC object.

This method must be called first before using any of the other methods within *DataStructure*.

```
mcstat = MCMC()
mcstat.data.add_data_set(x = x1, y = y1)
mcstat.data.add_data_set(x = x2, y = y2)
```

This yields the following variables in the data structure.

- ***xdata* (list): List of numpy arrays**
  - xdata[0] = x1, xdata[0].shape = (5,1)
  - xdata[1] = x2, xdata[1].shape = (10,1)
- ***ydata* (list): List of numpy arrays**
  - ydata[0] = y1, ydata[0].shape = (5,2)
  - ydata[1] = y2, ydata[1].shape = (10,3)
- *n* (list): List of integers. n = [5, 10]
- *shape* (list): List of *y.shape*. shape = [(5,2),(10,3)]

- *weight* (list): List of weights. `weight = [1, 1]`
- *user_defined_object* (list): List of objects. `user_defined_object = [0,0]`

Args:

- **x** (ndarray): Independent data. Recommend input as column vectors.
- **y** (ndarray): Dependent data. Recommend input as column vectors.
- **n** (list): List of integers denoting number of data points.
- **weight** (list): Weight of each data set.
- **user_defined_object** (*User Defined*): Any object can be stored in this variable.

Note: In general, it is recommended that user's format their data as a column vector. So, if you have *nds* independent data points, *x* and *y* should be *[nds,1]* or *[nds,]* numpy arrays. Note if a list is sent, the code will convert it to a numpy array.

---

get_number_of_batches()

Get number of batches in data structure. Essentially, each time you call the *add_data_set()* method you are adding another batch. It is also equivalent to say the number of batches is equal to the length of the list *ydata*. For example,

```
nb = mcstat.data.get_number_of_batches()
```

should return `nb = 2` because `len(mcstat.data.ydata) = 2`.

Returns:

- **nbatch** (int): Number of batches.

get_number_of_data_sets()

Get number of data sets in data structure. A data set is strictly speaking defined as the total number of columns in each element of the *ydata* list. For example,

```
nds = mcstat.data.get_number_of_data_sets()
```

should return `nds = 2 + 3 = 5` because the number of columns in *y1* is 2 and the number of columns in *y2* is 3.

Returns:

- Number of columns in *ydata* (int)

get_number_of_observations()

Get number of observations in data structure. An observation is essentially the total number of rows from each element of the *ydata* list. For example,

```
nobs = mcstat.data.get_number_of_observations()
```

should return `nobs = 5 + 10 = 15` because the number of rows in *y1* is 5 and the number of rows in *y2* is 10.

Returns:

- Number of rows in *ydata* (ndarray)

pymcmcstat.settings.ModelParameters module

Created on Wed Jan 17 09:13:03 2018

@author: prmiles

class pymcmcstat.settings.ModelParameters.**ModelParameters**

Bases: object

MCMC Model Parameters.

Example:

```
mcstat = MCMC()

mcstat.parameters.add_model_parameter(name = 'm', theta0 = 1., minimum = -10,␣
↪maximum = 10)
mcstat.parameters.add_model_parameter(name = 'b', theta0 = -5., minimum = -10,␣
↪maximum = 100)
mcstat.parameters.display_model_parameter_settings()
```

This will display to screen:

```
Sampling these parameters:
name          start [   min,    max] N(  mu, sigma^2)
m         :   1.00 [-10.00,  10.00] N(0.00, inf)
b         :  -5.00 [-10.00, 100.00] N(0.00, inf)
```

Attributes:

- *add_model_parameter()*
- *display_parameter_settings()*

add_model_parameter(*name=None*, *theta0=None*, *minimum=-inf*, *maximum=inf*, *prior_mu=array([0.])*, *prior_sigma=inf*, *sample=True*, *local=0*, *adapt=True*)

Add model parameter to MCMC simulation.

Args:

- name (str): Parameter name
- theta0 (float): Initial value
- minimum (float): Lower parameter bound
- maximum (float): Upper parameter bound
- prior_mu (float): Mean value of prior distribution
- prior_sigma (float): Standard deviation of prior distribution
- sample (bool): Flag to turn sampling on (True) or off (False)
- local (int): Local flag - still testing.

The default prior is a uniform distribution from minimum to maximum parameter value.

display_parameter_settings(*verbosity=None*, *no_adapt=None*)

Display parameter settings

Args:

- **verbosity** (int): Verbosity of display output. 0
- **no_adapt** (ndarray): Boolean array of indices not to be adapted.

classmethod scan_for_local_variables(*nbatch*, *parameters*)
Scan for local variables

Args:

- **nbatch** (int): Number of data batches
- **parameters** (list): List of model parameters.

Returns:

- **local** (ndarray): Array with local flag indices.

classmethod setup_adaptation_indices(*parind*, *adapt*)
Setup adaptation parameter indices.

Args:

- **parind** (ndarray): Array of boolean flags from parameter structure.
- **adapt** (ndarray): Array of boolean flags from parameter structure.

Returns:

- **parind** (ndarray): Array of indices corresponding to sampling parameters.
- **adapt** (ndarray): Array of indices corresponding to adaptating parameters.
- **no_adapt** (ndarray): Boolean array of indices not being adapted.

..note:

```
The size of the returned arrays will equal the number of parameters being␣
↪sampled.
```

classmethod setup_adapting(*adapt*, *sample*)
Setup parameters being adapted.

All parameters that are not being sampled will automatically be thrown out of adaptation. This method checks that the default adaptation status is consistent.

Args:

- **adapt** (bool): Flag from parameter structure.
- **sample** (bool): Flag from parameter structure.

Returns:

- bool

classmethod setup_prior_mu(*mu*, *value*)
Setup prior mean.

Args:

- **mu** (float): defined mean
- **value** (float): default value

Returns:

- Prior mean

classmethod setup_prior_sigma(*sigma*)
Setup prior variance.

Args:

- **sigma** (float): defined variance

Returns:

- Prior mean

pymcmcstat.settings.ModelParameters.**check_noadaptind**(*no_adapt*, *npar*)
Check if noadaptind is None -> Empty List

Args:

- **no_adapt** (ndarray): Boolean array of indices not to be adapted.
- **npar** (int): Number of parameters.

Returns:

- **no_adapt** (ndarray): Boolean array of indices not to be adapted.

pymcmcstat.settings.ModelParameters.**check_verbosity**(*verbosity*)
Check if verbosity is None -> 0

Args:

- **verbosity** (int): Verbosity level

Returns:

- **verbosity** (int): Returns 0 if verbosity was initially *None*

pymcmcstat.settings.ModelParameters.**format_number_to_str**(*number*)
Format number for display

Args:

- **number** (float): Number to be formatted

Returns:

- (str): Formatted string display

pymcmcstat.settings.ModelParameters.**generate_default_name**(*nparam*)
Generate generic parameter name. For example, if `nparam = 4`, then the generated name is:

```
names = 'p_{3}'
```

Args:

- **nparam** (int): Number of parameter names to generate

Returns:

- **name** (str): Name based on size of parameter list

pymcmcstat.settings.ModelParameters.**less_than_or_equal_to_zero**(*x*)
Return result of test on number based on less than or equal to

Args:

- **x** (float): Number to be tested

Returns:

- (bool): Result of test: *x<=0*

pymcmcstat.settings.ModelParameters.**noadapt_display_setting**(*no_adapt*)
Define display settins if index not being adapted.

Args:

- **no_adapt** (bool): Flag to determine whether or not it is to be adapted..

Returns:

- **st** (str): String to be displayed.

pymcmcstat.settings.ModelParameters.**prior_display_setting**(*x*)
Define display string for prior.

Args:

- **x** (float): Prior mean

Returns:

- **h2** (str): String to be displayed, depending on if *x* is infinity.

pymcmcstat.settings.ModelParameters.**replace_list_elements**(*x*, *testfunction*, *value*)
Replace list elements based on results from testfunction.

Args:

- **x** (list): List of numbers to be tested
- **testfunction** (testfunction()): Test function
- **value** (float): Value to assign if test function return True

Returns:

- **x** (list): Updated list

pymcmcstat.settings.ModelSettings module

Created on Wed Jan 17 09:06:51 2018

@author: prmiles

class pymcmcstat.settings.ModelSettings.**ModelSettings**
Bases: object

MCMC Model Settings

Attributes:

- *define_model_settings()*
- *display_model_settings()*

define_model_settings(*sos_function=None*, *prior_function=None*, *prior_type=1*, *prior_update_function=None*, *prior_pars=None*, *model_function=None*, *sigma2=None*, *N=None*, *S20=nan*, *N0=None*, *nbatch=None*)
Define model settings.

Args:

- **sos_function**: Handle for sum-of-squares function
- **prior_function**: Handle for prior function

- **prior_type**: Pending...
- **prior_update_function**: Pending...
- **prior_pars**: Pending...
- **model_function**: Handle for model function (needed if `sos_function` not specified)
- **sigma2** (`float`): List of initial error observations.
- **N** (`int`): Number of observations - see *DataStructure*.
- **S20** (`float`): List of scaling parameter in observation error estimate.
- **N0** (`float`): List of scaling parameter in observation error estimate.
- **nbatch** (`int`): Number of batch data sets - see *get_number_of_batches()*.

Note: Variables `sigma2, N, S20, N0`, and `nbatch` converted to `ndarray` for subsequent processing.

---

display_model_settings(*print_these=None*)

Display subset of the simulation options.

Args:

- **print_these** (`list`): List of strings corresponding to keywords. Default below.

```
print_these = ['sos_function', 'model_function', 'sigma2', 'N', 'N0', 'S20',
↪'nsos', 'nbatch']
```

pymcmcstat.settings.SimulationOptions module

Created on Wed Jan 17 09:08:13 2018

@author: prmiles

class pymcmcstat.settings.SimulationOptions.**SimulationOptions**

Bases: `object`

MCMC simulation options.

Attributes:

- *define_simulation_options()*
- *display_simulation_options()*

define_simulation_options(*nsimu=10000*, *adaptint=None*, *ntry=None*, *method='dram'*, *printint=None*, *adaptend=0*, *lastadapt=0*, *burnintime=0*, *waitbar=True*, *debug=0*, *qcov=None*, *updatesigma=False*, *stats=0*, *drscale=array([5., 4., 3.])*, *adascale=None*, *savesize=0*, *maxmem=0*, *chainfile='chainfile'*, *s2chainfile='s2chainfile'*, *sschainfile='sschainfile'*, *covchainfile='covchainfile'*, *savedir=None*, *save_to_bin=False*, *skip=1*, *label=None*, *RDR=None*, *verbosity=1*, *maxiter=None*, *priorupdatestart=0*, *qcov_adjust=1e-08*, *burnin_scale=10*, *alphatarget=0.234*, *etaparam=0.7*, *initqcovn=None*, *doram=None*, *rndseq=None*, *results_filename=None*, *save_to_json=False*, *save_to_txt=False*, *json_restart_file=None*, *save_lightly=False*)

Define simulation options.

Args:

- **nsimu** (int): Number of parameter samples to simulate. Default is 1e4.
- **adaptint** (int): Number of interates between adaptation. Default is method dependent.
- **ntry** (int): Number of tries to take before rejection. Default is method dependent.
- **method** (str): Sampling method ('mh', 'am', 'dr', 'dram'). Default is 'dram'.
- **printint** (int): Printing interval.
- **adaptend** (int): Obsolete.
- **lastadapt** (int): Last adaptation iteration (i.e., no more adaptation beyond this point).
- **burnintime** (int):
- **waitbar** (int): Flag to use progress bar. Default is 1 -> on (otherwise -> off).
- **debug** (int): Flag to perform debug. Default is 0 -> off.
- **qcov** (ndarray): Proposal parameter covariance matrix.
- **updatesigma** (bool): Flag for updating measurement error variance. Default is 0 -> off (1 -> on).
- **stats** (int): Calculate convergence statistics. Default is 0 -> off (1 -> on).
- **drscale** (ndarray): Reduced scale for sampling in DR algorithm. Default is [5,4,3].
- **adascale** (float): User defined covariance scale. Default is method dependent (untested).
- **savesize** (int): Size of chain segments when saving to log files. Default is 0.
- **maxmem** (int): Maximum memory available in mega bytes (Obsolete).
- **chainfile** (str): File name for chain log file.
- **sschainfile** (str): File name for sschain log file.
- **s2chainfile** (str): File name for s2chain log file.
- **covchainfile** (str): File name for qcov log file.
- **savedir** (str): Output directory of log files. Default is current directory.
- **save_to_bin** (bool): Save log files to binary. Default is False.
- **save_to_txt** (bool): Save log files to text. Default is False.
- **skip** (int):
- **label** (str):
- **RDR** (ndarray): R matrix for each stage of DR.
- **verbosity** (int): Verbosity of display output.
- **maxiter** (int): Obsolete.
- **priorupdatestart**
- **qcov_adjust** (float): Adjustment scale for covariance matrix.
- **burnin_scale** (float): Scale for burnin.
- **alphatarget** (float): Acceptance ratio target.
- **etaparam** (float):

- **initqcovn** (float): Proposal covariance weight in update.
- **doram** (int): Flag to perform 'ram' algorithm (Obsolete).
- **rndseq** (ndarray): Random number sequence (Obsolete).
- **results_filename** (str): Output file name when saving results structure with json.
- **save_to_json** (bool): Save results structure to json file. Default is False.
- **json_restart_file** (str): Extract parameter covariance and last sample value from saved json file.
- **save_lightly** (bool): Flag to indicate whether results json file should include chains.

Note: For the log file names chainfile, sschainfile, s2chainfile and covchainfile do not include the extension. By specifying whether to save to text or to binary, the appropriate extension will be added.

---

display_simulation_options(*print_these=None*)

Display subset of the simulation options.

Args:

- **print_these** (list): List of strings corresponding to keywords. Default below.

```
print_these = ['nsimu', 'adaptint', 'ntry', 'method', 'printint', 'lastadapt',
↪ 'drscale', 'qcov']
```

pymcmcstat.settings.SimulationOptions.**check_lightly_save**(*save_lightly*, *save_to_json*, *save_to_bin*, *save_to_txt*)

Check settings for lightly save

If saving to json, chains will be removed if already being stored via one of the logging methods, binary or text. If logging methods not being used, then chains will be included in json file.

Args:

- **save_lightly** (bool): Flag to save results w/out arrays
- **save_to_json** (bool): Flag to save to json
- **save_to_bin** (bool): Flag to save to binary
- **save_to_txt** (bool): Flag to save to text

pymcmcstat.structures package

pymcmcstat.structures.ParameterSet module

Created on Thu Jan 18 10:15:37 2018

@author: prmiles

class pymcmcstat.structures.ParameterSet.**ParameterSet**(*theta=None*, *ss=None*, *prior=None*, *sigma2=None*, *alpha=None*)

Bases: object

Basic MCMC parameter set.

Description: Storage device for passing parameter sets back and forth between sampling methods.

Args:

- **theta** (ndarray): Sampled values.
- **ss** (ndarray): Sum-of-squares error(s).
- **prior** (ndarray): Result from prior function.
- **sigma2** (ndarray): Observation errors.
- **alpha** (float): Result from evaluating likelihood function.

pymcmcstat.structures.ResultsStructure module

Created on Wed Jan 17 09:18:19 2018

@author: prmiles

class pymcmcstat.structures.ResultsStructure.**ResultsStructure**
Bases: object

Results from MCMC simulation.

Description: Class used to organize results of MCMC simulation.

Attributes:

- *export_simulation_results_to_json_file()*
- *determine_filename()*
- *save_json_object()*
- *load_json_object()*
- *add_basic()*
- *add_updatesigma()*
- *add_dram()*
- *add_prior()*
- *add_options()*
- *add_model()*
- *add_chain()*
- *add_s2chain()*
- *add_sschain()*
- *add_time_stats()*
- *add_random_number_sequence()*

add_basic(*nsimu*, *covariance*, *parameters*, *rejected*, *simutime*, *theta*)
Add basic results from MCMC simulation to structure.

Args:

- **nsimu** (int): Number of MCMC simulations.
- **model** (*ModelSettings*): MCMC model settings.

- **covariance** (*CovarianceProcedures*): Covariance variables.
- **parameters** (*ModelParameters*): Model parameters.
- **rejected** (dict): Dictionary of rejection stats.
- **simutime** (float): Simulation run time in seconds.
- **theta** (ndarray): Last sampled values.

add_chain(*chain=None*)
Add chain to results structure.

Args:

- **chain** (ndarray): Model parameter sampling chain.

add_dram(*drscale*, *RDR*, *total_rejected*, *drsettings*)
Add results specific to performing DR algorithm.

Args:

- **drscale** (ndarray): Reduced scale for sampling in DR algorithm. Default is [5,4,3].
- **RDR** (ndarray): Cholesky decomposition of covariance matrix based on DR.
- **total_rejected** (int): Number of rejected samples.
- **drsettings** (*DelayedRejection*): Need access to counters within DR class.

add_model(*model=None*)
Saves subset of features of the model settings in a nested dictionary.

Args:

- **model** (*ModelSettings*): MCMC model settings.

add_options(*options=None*)
Saves subset of features of the simulation options in a nested dictionary.

Args:

- **options** (*SimulationOptions*): MCMC simulation options.

add_prior(*mu*, *sigma*, *priortype*)
Add results specific to prior function.

Args:

- **mu** (ndarray): Prior mean.
- **sigma** (ndarray): Prior standard deviation.
- **priortype** (int): Flag identifying type of prior.

Note: This feature is not currently implemented.

---

add_random_number_sequence(*rndseq*)
Add random number sequence to results structure.

Args:

- **rndseq** (ndarray): Sequence of sampled random numbers.

Note: This feature is not currently implemented.

add_s2chain(*s2chain=None*)
Add observation error chain to results structure.

Args:

- **s2chain** (ndarray): Sampling chain of observation errors.

add_sschain(*sschain=None*)
Add sum-of-squares chain to results structure.

Args:

- **sschain** (ndarray): Calculated sum-of-squares error for each parameter chains set.

add_time_stats(*mtime*, *drtime*, *adtime*)
Add time spend using each sampling algorithm.

Args:

- **mtime** (float): Time spent performing standard Metropolis.
- **drtime** (float): Time spent performing Delayed Rejection.
- **adtime** (float): Time spent performing Adaptation.

Note: This feature is not currently implemented.

add_updatesigma(*updatesigma*, *sigma2*, *S20*, *N0*)
Add information to results structure related to observation error.

Args:

- **updatesigma** (bool): Flag to update error variance(s).
- **sigma2** (ndarray): Latest estimate of error variance(s).
- **S20** (ndarray): Scaling parameter(s).
- **N0** (ndarray): Shape parameter(s).

If `updatesigma is True`, then

```
results['sigma2'] = np.nan
results['S20'] = S20
results['N0'] = N0
```

Otherwise

```
results['sigma2'] = sigma2
results['S20'] = np.nan
results['N0'] = np.nan
```

classmethod determine_filename(*options*)
Determine results filename.

If not specified by *results_filename* in the simulation options, then a default naming format is generated using the date string associated with the initialization of the simulation.

Args:

- **options** (*SimulationOptions*): MCMC simulation options.

Returns:

- **filename** (str): Filename string.

export_lightly(*results*)
Export minimal simulation results to a json file.

This will save the key terms in the results dict, excluding arrays. Ideally, this is used in conjunction with one of the chain saving methods. The goal is to provide a results dict to simplify post- processing and reduces storage overhead.

Args:

- **results** (*ResultsStructure*): Dictionary of MCMC simulation results/settings.

export_simulation_results_to_json_file(*results*)
Export simulation results to a json file.

Args:

- **results** (*ResultsStructure*): Dictionary of MCMC simulation results/settings.

classmethod load_json_object(*filename*)
Load object stored in json file.

Note: Filename should include extension.

Args:

- **filename** (str): Load object from file with this name.

Returns:

- **results** (dict): Object loaded from file.

classmethod save_json_object(*results*, *filename*)
Save object to json file.

Note: Filename should include extension.

Args:

- **results** (dict): Object to save.
- **filename** (str): Write object into file with this name.

pymcmcstat.structures.ResultsStructure.**lighten_results**(*results*)
Saves subset of features of the simulation options in a nested dictionary.

Args:

- **options** (*SimulationOptions*): MCMC simulation options.

pymcmcstat.utilities package

pymcmcstat.utilities.NumpyEncoder module

Created on Mon Apr 2 08:42:56 2018

@author: prmiles

class pymcmcstat.utilities.NumpyEncoder.**NumpyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

Encoder used for storing numpy arrays in json files.

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

pymcmcstat.utilities.general module

Created on Tue Jun 26 06:16:02 2018

General functions used throughout package

@author: prmiles

pymcmcstat.utilities.general.**check_settings**(*default_settings*, *user_settings=None*)

Check user settings with default.

Recursively checks elements of user settings against the defaults and updates settings as it goes. If a user setting does not exist in the default, then the user setting is added to the settings. If the setting is defined in both the user and default settings, then the user setting overrides the default. Otherwise, the default settings persist.

Args:

- **default_settings** (dict): Default settings for particular method.
- **user_settings** (dict): User defined settings.

Returns:

- (dict): Updated settings.

pymcmcstat.utilities.general.**message**(*verbosity*, *level*, *printthis*)
Display message

Args:

- **verbosity** (int): Verbosity of display output.
- **level** (int): Print level relative to verbosity.
- **printthis** (str): String to be printed.

pymcmcstat.utilities.general.**removekey**(*d*, *key*)
Removed elements from dictionary and return the remainder.

Args:

- **d** (dict): Original dictionary.
- **key** (str): Keyword to be removed.

Returns:

- **r** (dict): Updated dictionary without the keyword, value pair.

pymcmcstat.utilities.progressbar module

pymcmcstat.utilities.progressbar.**progress_bar**(*iters*)
Simulation progress bar.

A simple progress bar to monitor MCMC sampling progress. Modified from original code by Corey Goldberg (2010).

Args:

- **iters** (int): Number of iterations in simulation.

Example display:

```
[--------          21%                   ] 2109 of 10000 complete in 0.5 sec
```

Note: Will display a progress bar as simulation runs, providing feedback as to the status of the simulation. Depending on the available resources, the appearance of the progress bar may differ.

CHAPTER 9

References

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[BG98] Stephen P Brooks and Andrew Gelman. General methods for monitoring convergence of iterative simulations. *Journal of computational and graphical statistics*, 7(4):434–455, 1998.

[BR98] Stephen P Brooks and Gareth O Roberts. Assessing convergence of markov chain monte carlo algorithms. *Statistics and Computing*, 8(4):319–335, 1998. URL: http://www.math.pitt.edu/~swigon/Homework/brooks97assessing.pdf.

[GR+92] Andrew Gelman, Donald B Rubin, and others. Inference from iterative simulation using multiple sequences. *Statistical science*, 7(4):457–472, 1992.

[HLMS06] Heikki Haario, Marko Laine, Antonietta Mira, and Eero Saksman. Dram: efficient adaptive mcmc. *Statistics and Computing*, 16(4):339–354, 2006. URL: https://link.springer.com/article/10.1007/s11222-006-9438-0.

[HST+01] Heikki Haario, Eero Saksman, Johanna Tamminen, and others. An adaptive metropolis algorithm. *Bernoulli*, 7(2):223–242, 2001. URL: https://projecteuclid.org/euclid.bj/1080222083.

[MT00] George Marsaglia and Wai Wan Tsang. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):363–372, 2000. URL: https://dl.acm.org/citation.cfm?id=358414.

[Smi14] Ralph C. Smith. *Uncertainty quantification: theory, implementation, and applications*. Volume 12. SIAM, 2014.

Python Module Index

p

Index

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

U