mlmc

Release 1.0.1

Jan 28, 2022

Contents

CHAPTER 1

MLMC

MLMC provides tools for the multilevel Monte Carlo method, which is theoretically described by M. Giles.

mlmc package includes:

- samples scheduling
- estimation of generalized moment functions
- probability density function approximation
- advanced post-processing with our Quantity structure

1.1 Installation

mlmc can be installed via pip

```
pip install mlmc
```

CHAPTER 2

Tutorials

The following tutorials illustrates how to use mlmc package.

2.1 Sampler creation

Sampler controls the execution of MLMC samples.

First, import mlmc package and define basic MLMC parameters.

```
import mlmc
n_levels = 3 # number of MLMC levels
step_range = [0.5, 0.005] # simulation steps at the coarsest and finest levels
level_parameters = mlmc.estimator.determine_level_parameters(n_levels, step_range)
# level_parameters determine each level simulation steps
# level_parameters can be manually prescribed as a list of lists
```

Prepare a simulation, it must be instance of class that inherits from *mlmc.sim.simulation.Simulation*.

```
simulation_factory = mlmc.SynthSimulation()
```

Create a sampling pool.

```
sampling_pool = mlmc.OneProcessPool()
```

You can also use `mlmc.sampling_pool.ProcessPool` which supports parallel execution of MLMC samples. In order to use PBS (portable batch system), employ `mlmc.sampling_pool_pbs.SamplingPoolPBS`.

Create a sample storage. It contains sample's related data e.g. simulation result.

```
# Memory() storage keeps samples in the computer main memory
sample_storage = mlmc.Memory()
```

We support also HDF5 file storage `mlmc.sample_storage_hdf.SampleStorageHDF`.

Finally, create a sampler that manages scheduling MLMC samples and also saves the results.

```
sampler = mlmc.Sampler(sample_storage=sample_storage,
                               sampling_pool=sampling_pool,
                               sim_factory=simulation_factory,
                               level_parameters=level_parameters)
```

Samples scheduling

2.2 Samples scheduling

Once you create a sampler you can schedule samples.

2.2.1 1. Prescribe the exact number of samples

```
n_samples = [100, 75, 50]
sampler.set_initial_n_samples(n_samples)
```

Schedule set samples.

```
sampler.schedule_samples()
```

You can wait until all samples are finished.

```
running = 1
while running > 0:
    running = 0
    running += sampler.ask_sampling_pool_for_samples()
```

2.2.2 2. Prescribe a target variance

Set target variance and number of random variable moments that must meet this variance.

```
target_var = 1e-4
n_moments = 10
```

The first phase is the same as the first approach, but the initial samples are automatically determined as a sequence from 100 samples at the coarsest level to 10 samples at the finest level.

```
sampler.set_initial_n_samples()
sampler.schedule_samples()
running = 1
while running > 0:
    running = 0
    running += sampler.ask_sampling_pool_for_samples()
```

The *mlmc.quantity.quantity.Quantity* instance is created, for details see *Quantity tutorial*

```
root_quantity = mlmc.make_root_quantity(storage=sampler.sample_storage,
                               q_specs=sampler.sample_storage.load_result_format())
```

`root_quantity` contains the structure of sample results and also allows access to their values.

In order to estimate moment values including variance, moment functions class (in this case Legendre polynomials) instance and `mlmc.estimator.Estimate` instance are created.

```
true_domain = mlmc.Estimate.estimate_domain(root_quantity, sample_storage)
moments_fn = mlmc.Legendre(n_moments, true_domain)

estimate_obj = mlmc.Estimate(root_quantity, sample_storage=sampler.sample_storage,
                                        moments_fn=moments_fn)
```

At first, the variance of moments and average execution time per sample at each level are estimated from already finished samples.

```
variances, n_ops = estimate_obj.estimate_diff_vars_regression(sampler.n_finished_
↪samples)
```

Then, an initial estimate of the number of MLMC samples that should meet prescribed target variance is conducted.

```
from mlmc.estimator import estimate_n_samples_for_target_variance
n_estimated = estimate_n_samples_for_target_variance(target_var, variances, n_ops,
                                                     n_levels=sampler.n_levels)
```

Now it is time for our sampling algorithm that gradually schedules samples and refines the total number of samples until the number of estimated samples is greater than the number of scheduled samples.

```
while not sampler.process_adding_samples(n_estimated):
    # New estimation according to already finished samples
    variances, n_ops = estimate_obj.estimate_diff_vars_regression(sampler._n_
↪scheduled_samples)
    n_estimated = estimate_n_samples_for_target_variance(target_var, variances, n_ops,
                                                         n_levels=sampler.n_levels)
```

Finally, wait until all samples are finished.

```
running = 1
while running > 0:
    running = 0
    running += sampler.ask_sampling_pool_for_samples()
```

Since our sampling algorithm determines the number of samples according to moment variances, the type of moment functions (Legendre by default) might affect total number of MLMC samples.

2.3 Quantity tutorial

An overview of basic *mlmc.quantity.quantity.Quantity* operations. Quantity related classes and functions allow estimate mean and variance of MLMC samples results, derive other quantities from original ones and much more.

```
import numpy as np
import mlmc.quantity.quantity_estimate
from examples.synthetic_quantity import create_sampler
```

First, the synthetic Quantity with the following `result_format` is created

```
# result_format = [
#     mlmc.QuantitySpec(name="length", unit="m", shape=(2, 1), times=[1, 2, 3],␣
↪locations=['10', '20']),
#     mlmc.QuantitySpec(name="width", unit="mm", shape=(2, 1), times=[1, 2, 3],␣
↪locations=['30', '40']),
```

(continues on next page)

(continued from previous page)

```
# ]
# Meaning: sample results contain data on two quantities in three time steps [1, 2,␣
↪3] and in two locations,
#          each quantity can have different shape

sampler, simulation_factory, moments_fn = create_sampler()
root_quantity = mlmc.make_root_quantity(sampler.sample_storage, simulation_factory.
↪result_format())
```

root_quantity is *mlmc.quantity.quantity.Quantity* instance and represents the whole result data. According to result_format it contains two sub-quantities named "length" and "width".

2.3.1 Mean estimates

To get estimated mean of a quantity:

```
root_quantity_mean = mlmc.quantity.quantity_estimate.estimate_mean(root_quantity)
```

root_quantity_mean is an instance of *mlmc.quantity.quantity.QuantityMean*

To get the total mean value:

```
root_quantity_mean.mean
```

To get the total variance value:

```
root_quantity_mean.var
```

To get means at each level:

```
root_quantity_mean.l_means
```

To get variances at each level:

```
root_quantity_mean.l_vars
```

2.3.2 Estimate moments and covariance matrix

Create a quantity representing moments and get their estimates

```
moments_quantity = mlmc.quantity.quantity_estimate.moments(root_quantity, moments_
↪fn=moments_fn)
moments_mean = mlmc.quantity.quantity_estimate.estimate_mean(moments_quantity)
```

To obtain central moments, use:

```
central_root_quantity = root_quantity - root_quantity_mean.mean
central_moments_quantity = mlmc.quantity.quantity_estimate.moments(central_root_
↪quantity,
                                                                   moments_
↪fn=moments_fn)
central_moments_mean = mlmc.quantity.quantity_estimate.estimate_mean(central_moments_
↪quantity)
```

Create a quantity representing a covariance matrix

```
covariance_quantity = mlmc.quantity.quantity_estimate.covariance(root_quantity,␣
↪moments_fn=moments_fn)
cov_mean = mlmc.quantity.quantity_estimate.estimate_mean(covariance_quantity)
```

2.3.3 Quantity selection

According to the result_format, it is possible to select items from a quantity

```
length = root_quantity["length"]  # Get quantity with name="length"
width = root_quantity["width"]  # Get quantity with name="width"
```

`length` and `width` are still *mlmc.quantity.quantity.Quantity* instances

To get a quantity at particular time:

```
length_locations = length.time_interpolation(2.5)
```

`length_locations` represents results for all locations of quantity named "length" at the time 2.5

To get quantity at particular location:

```
length_result = length_locations['10']
```

`length_result` represents results shape=(2, 1) of quantity named "length" at the time 2,5 and location '10'

Now it is possible to slice Quantity `length_result` the same way as `np.ndarray`. For example:

```
length_result[1, 0]
length_result[:, 0]
length_result[:, :]
length_result[:1, :1]
length_result[:2, ...]
```

Keep in mind:

- all derived quantities such as `length_locations` and `length_result`, ... are still *mlmc.quantity.quantity.Quantity* instances
- selecting location before time is not supported!

2.3.4 Binary operations

Following operations are supported

- Addition, subtraction, ... of compatible quantities

```
quantity = root_quantity + root_quantity
quantity = root_quantity + root_quantity + root_quantity
```

- Operations with Quantity and a constant

```
const = 5
quantity_const_add = root_quantity + const
quantity_const_sub = root_quantity - const
quantity_const_mult = root_quantity * const
```

(continues on next page)

(continued from previous page)

```
quantity_const_div = root_quantity / const
quantity_const_mod = root_quantity % const
quantity_add_mult = root_quantity + root_quantity * const
```

2.3.5 NumPy universal functions

Examples of tested NumPy universal functions:

```
quantity_np_add = np.add(root_quantity, root_quantity)
quantity_np_max = np.max(root_quantity, axis=0, keepdims=True)
quantity_np_sin = np.sin(root_quantity)
quantity_np_sum = np.sum(root_quantity, axis=0, keepdims=True)
quantity_np_maximum = np.maximum(root_quantity, root_quantity)

x = np.ones(24)
quantity_np_divide_const = np.divide(x, root_quantity)
quantity_np_add_const = np.add(x, root_quantity)
quantity_np_arctan2_cosnt = np.arctan2(x, root_quantity)
```

2.3.6 Quantity selection by conditions

Method `select` returns *mlmc.quantity.quantity.Quantity* instance

```
selected_quantity = root_quantity.select(0 < root_quantity)
```

```
quantity_add = root_quantity + root_quantity
quantity_add_select = quantity_add.select(root_quantity < quantity_add)
root_quantity_selected = root_quantity.select(-1 != root_quantity)
```

Logical operation among more provided conditions is AND

```
quantity_add.select(root_quantity < quantity_add, root_quantity < 10)
```

User can use one of the logical NumPy universal functions

```
selected_quantity_or = root_quantity.select(np.logical_or(0 < root_quantity, root_
↪quantity < 10))
```

It is possible to explicitly define the selection condition of one quantity by another quantity

```
mask = np.logical_and(0 < root_quantity, root_quantity < 10)  # mask is Quantity␣
↪instance
q_bounded = root_quantity.select(mask)
```

2.4 Results postprocessing

If you already know how to create a sampler, schedule samples and handle quantities, postprocessing will be easy for you. Otherwise, see the previous tutorials before.

First, schedule samples and estimate moments for a particular quantity

```
import mlmc
n_levels = 3 # number of MLMC levels
step_range = [0.5, 0.005] # simulation steps at the coarsest and finest levels
target_var = 1e-4
n_moments = 10
level_parameters = mlmc.estimator.determine_level_parameters(n_levels, step_range)
# level_parameters determine each level simulation steps
# level_parameters can be manually prescribed as a list of lists

simulation_factory = mlmc.SynthSimulation()
sampling_pool = mlmc.OneProcessPool()
# Memory() storage keeps samples in the computer main memory
sample_storage = mlmc.Memory()

sampler = mlmc.Sampler(sample_storage=sample_storage,
                               sampling_pool=sampling_pool,
                               sim_factory=simulation_factory,
                               level_parameters=level_parameters)

sampler.set_initial_n_samples()
sampler.schedule_samples()
running = 1
while running > 0:
    running = 0
    running += sampler.ask_sampling_pool_for_samples()

# Get particular quantity
root_quantity = mlmc.make_root_quantity(sampler.sample_storage, simulation_factory.
↪result_format())
length = root_quantity['length']
time = length[1]
location = time['10']
q_value = location[0]


true_domain = mlmc.Estimate.estimate_domain(q_value, sample_storage)
moments_fn = mlmc.Legendre(n_moments, true_domain)
estimate_obj = mlmc.Estimate(q_value, sample_storage=sampler.sample_storage,
                                      moments_fn=moments_fn)

variances, n_ops = estimate_obj.estimate_diff_vars_regression(sampler.n_finished_
↪samples)

from mlmc.estimator import estimate_n_samples_for_target_variance
n_estimated = estimate_n_samples_for_target_variance(target_var, variances, n_ops,
                                                     n_levels=sampler.n_levels)

while not sampler.process_adding_samples(n_estimated):
    # New estimation according to already finished samples
    variances, n_ops = estimate_obj.estimate_diff_vars_regression(sampler._n_
↪scheduled_samples)
    n_estimated = estimate_n_samples_for_target_variance(target_var, variances, n_ops,
                                                         n_levels=sampler.n_levels)

running = 1
while running > 0:
    running = 0
    running += sampler.ask_sampling_pool_for_samples()
```

2.4.1 Probability density function approximation

```
from mlmc.plot.plots import Distribution
distr_obj, result, _, _ = estimate_obj.construct_density()
distr_plot = Distribution(title="distributions", error_plot=None)
distr_plot.add_distribution(distr_obj)

if n_levels == 1:
    samples = estimate_obj.get_level_samples(level_id=0)[..., 0]
    distr_plot.add_raw_samples(np.squeeze(samples)) # add histogram
distr_plot.show()
```

You can find more complex examples in `examples.shooting`

CHAPTER 3

MLMC package

The mlmc package provides tools to realize the Multilevel Monte Carlo method.

3.1 Subpackages

plot	Subpackage provides plot functions to display pdf, violinplot, ...
quantity	Subpackage provides methods to represent and handle a quantity of interest.
random	Subpackage provides random field generation and GSTools library interface
sim	Contains a parent simulation class and a specific synthetic simulation
tool	Contains classes that provide an interface to other resources such as HDF5, Gmsh, PBS, ...

3.2 Classes

3.2.1 Sampler

`Sampler`(sample_storage, sampling_pool, ...)	Manages samples scheduling, results collection, and result storage.

3.2.2 SamplingPool

SamplingPool([work_dir, debug])	Determining the runtime environment of samples, eg single process, multiple processes, running PBS, ...
OneProcessPool([work_dir, debug])	
ProcessPool(n_processes[, work_dir, debug])	Suitable for local parallel sampling for simulations WITHOUT external program call

3.2.3 SamplingPoolPBS

SamplingPoolPBS(work_dir[, debug])	Sampling pool PBS (Portable batch system) runtime environment

3.2.4 SampleStorage

SampleStorage	Provides methods to store and retrieve sample's data
Memory()	Sample's data are stored in the main memory

3.2.5 SampleStorageHDF

SampleStorageHDF(file_path)	Sample's data are stored in a HDF5 file

3.2.6 Estimate

Estimate(quantity, sample_storage[, moments_fn])	Provides wrapper methods for moments estimation, pdf approximation, ...

3.2.7 Moments

Moments(size, domain[, log, safe_eval])	Class for calculating moments of a random variable
Monomial(size[, domain, ref_domain, log, ...])	Monomials generalized moments
Fourier(size[, domain, ref_domain, log, ...])	Fourier functions generalized moments
Legendre(size, domain[, ref_domain, log, ...])	Legendre polynomials generalized moments

3.2.8 LevelSimulation

LevelSimulation(config_dict, Any], ...)	This class is used to pass simulation data at a given level between a Sampler and a SamplingPool User shouldn't change this class

3.3 mlmc.plot

Subpackage provides plot functions to display pdf, violinplot, ...

3.3.1 Submodules

3.3.2 mlmc.plot.plots module

class mlmc.plot.plots.**Aux**
Bases: object

plot_bootstrap_variance_compare()
Plot fraction (MLMC var est) / (BS var set) for the total variance and level variances. :param moments_fn: :return:

plot_bs_level_variances_error()
Plot error of estimates of V_l. Scaled as V_l^2 / N_l

plot_bs_var_error_contributions()
MSE of total variance and contribution of individual levels.

plot_bs_var_log_var()
Test that MSE of log V_l scales as variance of log chi^2_{N-1}, that is approx. 2 / (n_samples-1).

plot_bs_variances(*variances*, *y_label=None*, *log=True*, *y_lim=None*)
Plot BS estimate of error of variances of other related quantities. :param variances: Data, shape: (n_levels + 1, n_moments). :return:

plot_means_and_vars(*moments_mean*, *moments_var*, *n_levels*, *exact_moments*)
Plot means with variance whiskers to given axes. :param moments_mean: array, moments mean :param moments_var: array, moments variance :param n_levels: array, number of levels :param exact_moments: array, moments from distribution :param ex_moments: array, moments from distribution samples :return:

plot_var_regression(*i_moments=None*)
Plot total and level variances and their regression and errors of regression. :param i_moments: List of moment indices to plot. If it is an int M, the range(M) is used.

If None, self.moments.size is used.

class mlmc.plot.plots.**BSplots**(*n_samples*, *bs_n_samples*, *n_moments*, *ref_level_var*)
Bases: object

plot_bootstrap_variance_compare()
Plot fraction (MLMC var est) / (BS var set) for the total variance and level variances. :return:

plot_bs_level_variances_error()
Plot error of estimates of V_l. Scaled as V_l^2 / N_l

plot_bs_var_error_contributions()
MSE of total variance and contribution of individual levels.

plot_bs_var_log_var()
Test that MSE of log V_l scales as variance of log chi^2_{N-1}, that is approx. 2 / (n_samples-1).

plot_bs_variances(*variances*, *y_label=None*, *log=True*, *y_lim=None*)
Plot BS estimate of error of variances of other related quantities. :param variances: Data, shape: (n_levels + 1, n_moments). :return:

plot_means_and_vars(*moments_mean*, *moments_var*, *n_levels*, *exact_moments=None*)
Plot means with variance whiskers to given axes. :param moments_mean: array, moments mean :param moments_var: array, moments variance :param n_levels: array, number of levels :param exact_moments: array, moments from distribution :return:

plot_var_regression(*estimator*, *n_levels*, *moments_fn*, *i_moments=None*)
Plot total and level variances and their regression and errors of regression. :param i_moments: List of moment indices to plot. If it is an int M, the range(M) is used.

If None, self.moments_fn.size is used.

set_moments_color_bar(*range*, *label*, *ax=None*)
Create colorbar for a variable with given range and add it to given axes. :param range: single value as high bound or tuple (low bound, high bound) :param label: Label of the colorbar. :param ax: :return: Function to map values to colors. (normalize + cmap)

class mlmc.plot.plots.**Distribution**(*exact_distr=None*, *title=''*, *quantity_name='X'*, *legend_title=''*, *log_density=False*, *cdf_plot=True*, *log_x=False*, *error_plot='l2'*)
Bases: object

Class for plotting distribution approximation: PDF and CDF (optional) Provides methods to: add more plots, add exact PDF, add ECDF/histogram from single level MC

add_distribution(*distr_object*, *label=None*)
Add plot for distribution 'distr_object' with given label. :param distr_object: Instance of Distribution, we use methods: density, cdf and attribute domain :param label: string label for legend :return:

add_raw_samples(*samples*)
Add histogram and ecdf for raw samples. :param samples:

adjust_domain(*domain*)
Enlarge common domain by given bounds. :param value: [lower_bound, upper_bound]

reset()

show(*file=''*)
Set colors according to the number of added plots. Set domain from all plots. Plot exact distribution. show, possibly save to file. :param file: None, or filename, default name is same as plot title.

class mlmc.plot.plots.**Eigenvalues**(*log_y=True*, *title='Eigenvalues'*)
Bases: object

Plot of eigenvalues (of the covariance matrix), several sets of eigenvalues can be added together with error bars and cut-tresholds. Colors are chosen automatically. Slight X shift is used to avoid point overlapping. For log Y scale only positive values are plotted.

add_linear_fit(*values*)

add_values(*values*, *errors=None*, *threshold=None*, *label=''*)
Add set of eigenvalues into the plot. :param values: array (n,); eigen values in increasing or decreasing ordred, automatically flipped to decreasing. :param errors: array (n,); corresponding std errors :param threshold: horizontal line marking noise level or cut-off eigen value :return:

adjust_ylim(*ylim*)
Enlarge common domain by given bounds. :param value: [lower_bound, upper_bound]

show(*file=''*)
Show the plot or save to file. :param file: filename base, None for show. :return:

class mlmc.plot.plots.**Variance**(*moments=None*)
Bases: object

Plot level variances, i.e. Var X^l as a function of the mesh step. Selected moments are plotted.

add_level_variances(*steps*, *variances*)
Add variances for single MLMC instance. :param steps, variances : as returned by Estimate.estimate_level_vars :param n_levels:

show(*file=''*)

class mlmc.plot.plots.**VarianceBreakdown**(*moments=None*)
Bases: object

Plot total variance average over moments and variances of individual moments, Brake down to contribution of individual levels and optionally comparison to the reference level variances using error bars for the (signed) difference: ref_level_vars - level_vars

add_variances(*level_vars*, *n_samples*, *ref_level_vars=None*)
Add plot of variances for single MLMC instance.

Parameters

- **level_vars** – Array (n_levels, n_moments) of level variances.
- **n_samples** – Array (n_levels,) of numberf of samples on levels
- **ref_level_vars** – reference level vars (e.g. from bootstrapping)

Returns

show(*file=''*)
Show the plot or save to file. :param filename: filename base, None for show. :return:

mlmc.plot.plots.**create_color_bar**(*range*, *label*, *ax=None*)
Create colorbar for a variable with given range and add it to given axes. :param range: single value as high bound or tuple (low bound, high bound) :param label: Label of the colorbar. :param ax: :return: Function to map values to colors. (normalize + cmap)

mlmc.plot.plots.**make_monotone**(*X*, *Y*)

mlmc.plot.plots.**moments**(*moments_fn*, *size=None*, *title=''*, *file=''*)
Plot moment functions. :param moments_fn: :param size: :param title: :param file: :return:

mlmc.plot.plots.**moments_subset**(*n_moments*, *moments=None*)
Return subset of range(n_moments) for ploting. :param n_moments: Actual number of moments. :param moments: Type of subset:

None - all moments int - size of subset, formed by geometrical sequence

Returns

mlmc.plot.plots.**plot_convergence**(*quantiles*, *conv_val*, *title*)
Plot convergence with moment size for various quantiles. :param quantiles: iterable with quantiles :param conv_val: matrix of ConvResult, n_quantiles x n_moments :param title: plot title and filename used to save :return:

mlmc.plot.plots.**plot_diff_var**(*ref_mc_diff_vars*, *n_moments*, *steps*)
Plot level diff vars

mlmc.plot.plots.**plot_diff_var_subsample**(*level_variance_diff*, *n_levels*)
Plot diff between V* and V :param level_variance_diff: array of moments sqrt(V/V*) :param n_levels: array, number of levels :return: None

mlmc.plot.plots.**plot_error**(*arr*, *ax*, *label*)

mlmc.plot.plots.**plot_mlmc_conv**(*n_moments*, *vars_est*, *exact_mean*, *means_est*, *target_var*)

mlmc.plot.plots.**plot_n_sample_est_distributions**(*title*, *cost*, *total_std*, *n_samples*, *rel_moments*)

mlmc.plot.plots.**plot_pbs_flow_job_time**()

mlmc.plot.plots.**plot_regression_diffs**(*all_diffs*, *n_moments*)
Plot level variance difference regression :param all_diffs: list, difference between Estimate._variance_regression result and Estimate.estimate_diff_var result :param n_moments: number of moments :return:

mlmc.plot.plots.**plot_var_regression**(*ref_level_vars*, *reg_vars*, *n_levels*, *n_moments*)
Plot levels variance regression

mlmc.plot.plots.**plot_vars**(*moments_mean*, *moments_var*, *n_levels*, *exact_moments=None*, *ex_moments=None*)
Plot means with variance whiskers :param moments_mean: array, moments mean :param moments_var: array, moments variance :param n_levels: array, number of levels :param exact_moments: array, moments from distribution :param ex_moments: array, moments from distribution samples :return: None

3.3.3 mlmc.plot.violinplot module

3.3.4 Module contents

Subpackage provides plot functions to display pdf, violinplot, ...

3.4 mlmc.quantity

Subpackage provides methods to represent and handle a quantity of interest.

3.4.1 Submodules

3.4.2 mlmc.quantity.quantity module

class mlmc.quantity.quantity.**Quantity**(*quantity_type*, *operation*, *input_quantities=[]*)
Bases: object

static QArray(*quantities*)

static QDict(*key_quantity*)

static QField(*key_quantity*)

static QTimeSeries(*time_quantity*)

static add_op(*x*, *y*)

static create_quantity(*quantities*, *operation*)
Create new quantity (Quantity or QuantityConst) based on given quantities and operation. There are two scenarios: 1. At least one of quantities is Quantity instance then all quantities are considered to be input_quantities

of new Quantity

2. All of quantities are QuantityConst instances then new QuantityConst is created :param quantities: List[Quantity] :param operation: function which is run with given quantities :return: Quantity

get_cache_key(*chunk_spec*)
Create cache key

get_quantity_storage()
Get QuantityStorage instance :return: None, QuantityStorage

static mod_op(*x*, *y*)

static mult_op(*x*, *y*)

static pick_samples(*chunk*, *subsample_params*)
Pick samples some samples from chunk in order to have 'k' samples from 'n' after all chunks are processed Inspired by https://dl.acm.org/doi/10.1145/23002.23003 method S

Parameters

- **chunk** – np.ndarray, shape M, N, 2, where N denotes number of samples in chunk
- **subsample_params** – instance of SubsampleParams class, it has two parameters: k: number of samples which we want to get from all chunks n: number of all samples among all chunks

Returns np.ndarray

samples(*chunk_spec*)
Return list of sample chunks for individual levels. Possibly calls underlying quantities. :param chunk_spec: object containing chunk identifier level identifier and chunk_slice - slice() object :return: np.ndarray or None

select(**args*)
Performs sample selection based on conditions :param args: Quantity :return: Quantity

selection_id()
Get storage ids of all input quantities :return: List[int]

set_selection_id()
Set selection id selection id is None by default,

but if we create new quantity from quantities that are result of selection we need to pass selection id

size() → int
Quantity size from qtype :return: int

static sub_op(*x*, *y*)

subsample(*sample_vec*)
Subsampling :param sample_vec: list of number of samples at each level :return: Quantity

static truediv_op(*x*, *y*)

static wrap(*value*)
Convert flat, bool or array (list) to Quantity :param value: flat, bool, array (list) or Quantity :return: Quantity

class mlmc.quantity.quantity.**QuantityConst**(*quantity_type*, *value*)
Bases: *mlmc.quantity.quantity.Quantity*

samples(*chunk_spec*)
Get constant values with an enlarged number of axes :param chunk_spec: object containing chunk identifier level identifier and chunk_slice - slice() object :return: np.ndarray

selection_id()
Get storage ids of all input quantities :return: List[int]

class mlmc.quantity.quantity.**QuantityMean**(*quantity_type*, *l_means*, *l_vars*, *n_samples*, *n_rm_samples*)
Bases: object

l_means

l_vars

mean

n_rm_samples

n_samples

var

class mlmc.quantity.quantity.**QuantityStorage**(*storage*, *qtype*)
Bases: *mlmc.quantity.quantity.Quantity*

chunks(*level_id=None*)

get_quantity_storage()
Get QuantityStorage instance :return: None, QuantityStorage

level_ids()
Number of levels :return: List[int]

n_collected()

samples(*chunk_spec*)
Get results for given level id and chunk id :param chunk_spec: object containing chunk identifier level identifier and chunk_slice - slice() object :return: Array[M, chunk size, 2]

selection_id()
Identity of QuantityStorage instance :return: int

mlmc.quantity.quantity.**make_root_quantity**(*storage: mlmc.sample_storage.SampleStorage*, *q_specs: List[mlmc.quantity.quantity_spec.QuantitySpec]*)
Create a root quantity that has QuantityStorage as the input quantity, QuantityStorage is the only class that directly accesses the stored data. Quantity type is created based on the q_spec parameter :param storage: SampleStorage :param q_specs: same as result format in simulation class :return: QuantityStorage

3.4.3 mlmc.quantity.quantity_estimate module

mlmc.quantity.quantity_estimate.**cache_clear**()

mlmc.quantity.quantity_estimate.**covariance**(*quantity*, *moments_fn*, *cov_at_bottom=True*)
Create quantity with operation that evaluates covariance matrix :param quantity: Quantity :param moments_fn: mlmc.moments.Moments child :param cov_at_bottom: bool, if True cov matrices are underneath,

a scalar is substituted with a matrix of moments of that scalar

Returns Quantity

mlmc.quantity.quantity_estimate.**estimate_mean**(*quantity*)
MLMC mean estimator. The MLMC method is used to compute the mean estimate to the Quantity dependent on the collected samples. The squared error of the estimate (the estimator variance) is estimated using the central limit theorem. Data is processed by chunks, so that it also supports big data processing :param quantity: Quantity :return: QuantityMean which holds both mean and variance

mlmc.quantity.quantity_estimate.**mask_nan_samples**(*chunk*)
Mask out samples that contain NaN in either fine or coarse part of the result :param chunk: np.ndarray [M, chunk_size, 2] :return: chunk: np.ndarray, number of masked samples: int

mlmc.quantity.quantity_estimate.**moment**(*quantity*, *moments_fn*, *i=0*)
Create quantity with operation that evaluates particular moment :param quantity: Quantity instance :param moments_fn: mlmc.moments.Moments child :param i: index of moment :return: Quantity

mlmc.quantity.quantity_estimate.**moments**(*quantity*, *moments_fn*, *mom_at_bottom=True*)
Create quantity with operation that evaluates moments_fn :param quantity: Quantity :param moments_fn: mlmc.moments.Moments child :param mom_at_bottom: bool, if True moments are underneath,

a scalar is substituted with an array of moments of that scalar

Returns Quantity

3.4.4 mlmc.quantity.quantity_spec module

class mlmc.quantity.quantity_spec.**ChunkSpec**(*chunk_id: int = None*, *chunk_slice: slice = None*, *level_id: int = None*)

Bases: object

class mlmc.quantity.quantity_spec.**QuantitySpec**(*name: str*, *unit: str*, *shape: Tuple[int, int]*, *times: List[float]*, *locations: Union[List[str], List[Tuple[float, float, float]]]*)

Bases: object

3.4.5 mlmc.quantity.quantity_types module

class mlmc.quantity.quantity_types.**ArrayType**(*shape*, *qtype: mlmc.quantity.quantity_types.QType*)

Bases: *mlmc.quantity.quantity_types.QType*

get_key(*key*)
ArrayType indexing :param key: int, tuple of ints or slice objects :return: QuantityType - ArrayType or self._qtype

reshape(*data*)

size() → int
Size of type :return: int

class mlmc.quantity.quantity_types.**BoolType**(*qtype=<class 'float'>*)

Bases: *mlmc.quantity.quantity_types.ScalarType*

class mlmc.quantity.quantity_types.**DictType**(*args: List[Tuple[str, mlmc.quantity.quantity_types.QType]]*)

Bases: *mlmc.quantity.quantity_types.QType*

base_qtype()

get_key(*key*)

get_qtypes()

replace_scalar(*substitute_qtype*)
Find ScalarType and replace it with substitute_qtype :param substitute_qtype: QType, replaces ScalarType :return: DictType

size() → int
Size of type :return: int

class mlmc.quantity.quantity_types.**FieldType**(*args: List[Tuple[str, mlmc.quantity.quantity_types.QType]]*)

Bases: *mlmc.quantity.quantity_types.QType*

get_key(*key*)

size() → int
Size of type :return: int

class mlmc.quantity.quantity_types.**QType**(*qtype*)
Bases: object

base_qtype()

static keep_dims(*chunk*)
Always keep chunk shape to be [M, chunk size, 2]! For scalar quantities, the input block can have the shape (chunk size, 2) Sometimes we need to 'flatten' first few shape to have desired chunk shape :param chunk: list :return: list

replace_scalar(*substitute_qtype*)
Find ScalarType and replace it with substitute_qtype :param substitute_qtype: QType, replaces ScalarType :return: QType

reshape(*data*)

size() → int
Size of type :return: int

class mlmc.quantity.quantity_types.**ScalarType**(*qtype=<class 'float'>*)
Bases: *mlmc.quantity.quantity_types.QType*

base_qtype()

replace_scalar(*substitute_qtype*)
Find ScalarType and replace it with substitute_qtype :param substitute_qtype: QType, replaces ScalarType :return: QType

size() → int
Size of type :return: int

class mlmc.quantity.quantity_types.**TimeSeriesType**(*times*, *qtype*)
Bases: *mlmc.quantity.quantity_types.QType*

get_key(*key*)

size() → int
Size of type :return: int

static time_interpolation(*quantity*, *value*)
Interpolation in time :param quantity: Quantity instance :param value: point where to interpolate :return: Quantity

3.4.6 Module contents

Subpackage provides methods to represent and handle a quantity of interest.

3.5 mlmc.random

Subpackage provides random field generation and GSTools library interface

3.5.1 Submodules

3.5.2 mlmc.random.correlated_field module

class mlmc.random.correlated_field.**Field**(*name*, *field=None*, *param_fields=[]*, *regions=[]*)
Bases: object

sample()
Internal method to generate/compute new sample. :return:

set_points(*points*)
Internal method to set evaluation points. See Fields.set_points.

class mlmc.random.correlated_field.**Fields**(*fields*)
Bases: object

sample()
Return dictionary of sampled fields. :return: { 'field_name': sample, ... }

set_outer_fields(*outer*)
Set fields that will be in a dictionary produced by FieldSet.sample() call. :param outer: A list of names of fields that are sampled. :return:

set_points(*points*, *region_ids=[]*, *region_map={}*)
Set mesh related data to fields. - set points for sample evaluation - translate region names to region ids in fields - create maps from region constraned point sets of fields to full point set :param points: np array of points for field evaluation :param regions: regions of the points;

empty means no points for fields restricted to regions and all points for unrestricted fields

Returns

names

class mlmc.random.correlated_field.**FourierSpatialCorrelatedField**(*corr_exp='gauss'*, *dim=2*, *corr_length=1.0*, *aniso_correlation=None*, *mu=0.0*, *sigma=1.0*, *log=False*, ***kwargs*)
Bases: *mlmc.random.correlated_field.RandomFieldBase*

Generate spatial random fields

exp(*mode_no=1000*)
Compute an exponential spectrum :param mode_no: int, Number of Fourier modes :return: numpy.ndarray

gau(*mode_no=1000*)
Compute a gaussian spectrum :param mode_no: int, Number of Fourier modes :return: numpy.ndarray

get_normal_distr()
Normal distributed arrays :return: np.ndarray

random_field()
Calculates the random modes for the randomization method.

class mlmc.random.correlated_field.**GSToolsSpatialCorrelatedField**(*model*, *mode_no=1000*, *log=False*, *sigma=1*)
Bases: *mlmc.random.correlated_field.RandomFieldBase*

change_srf(*seed*)
Spatial random field with new seed :param seed: int, random number generator seed :return: None

random_field()
Generate the spatial random field :return: field, np.ndarray

sample()

Returns Random field evaluated in points given by 'set_points'

class mlmc.random.correlated_field.**RandomFieldBase**(*corr_exp='gauss'*, *dim=2*, *corr_length=1.0*, *aniso_correlation=None*, *mu=0.0*, *sigma=1.0*, *log=False*, ***kwargs*)

Bases: object

Base class for various methods for generating random fields.

Generating realizations of a spatially correlated random field F for a fixed set of points at X. E[F(x)] = mu(x) Cov_ij = Cov[x_i,x_j] = E[(F(x_i) - mu(x))(F(x_j) - mu(x))]

We assume stationary random field with covariance matrix Cov_ij: Cov_i,j = c(x_i - x_j)

where c(X) is the "stationary covariance" function. We assume: c(X) = sigma^2 exp(-|X^t K X|^(alpha/2))

for spatially heterogeneous sigma(X) we consider particular non-stationary generalization: Cov_i,i = sigma(x_i)*sigma(x_j) exp(-|X^t K X|^(alpha/2)); X = x_i - x_j

where:

- sigma(X) is the standard deviance of the single uncorrelated value
- K is a positive definite tensor with eigen vectors corresponding to main directions and eigen values equal to (1/l_i)^2, where l_i is correlation length in singel main direction.
- alpha is =1 for "exponential" and =2 for "Gauss" correlation

SVD decomposition: Considering first m vectors, such that lam(m)/lam(0) <0.1

Example: ‘‘‘

field = SpatialCorrelatedField(corr_exp='exp', corr_length=1.5) X, Y = np.mgrid[0:1:10j, 0:1:10j] points = np.vstack([X.ravel(), Y.ravel()]) field.set_points(points) sample = field.sample()

’’’

sample()

Parameters **uncorelated** – Random samples from standard normal distribution. Removed as the spectral method do not support it.

Returns Random field evaluated in points given by 'set_points'.

set_points(*points*, *mu=None*, *sigma=None*)

Parameters

- **points** – N x d array. Points X_i where the field will be evaluated. d is the dimension.
- **mu** – Scalar or N array. Mean value of uncorrelated field: E(F(X_i)).
- **sigma** – Scalar or N array. Standard deviance of uncorrelated field: sqrt(E (F(X_i) - mu_i)^2)

Returns None

class mlmc.random.correlated_field.**SpatialCorrelatedField**(*corr_exp='gauss'*, *dim=2*, *corr_length=1.0*, *aniso_correlation=None*, *mu=0.0*, *sigma=1.0*, *log=False*, ***kwargs*)

Bases: *mlmc.random.correlated_field.RandomFieldBase*

cov_matrix()
Setup dense covariance matrix for given set of points. :return: None.

svd_dcmp(*precision=0.01*, *n_terms_range=(1*, *inf)*)
Does decomposition of covariance matrix defined by set of points :param precision: Desired accuracy of the KL approximation, smaller eigen values are dropped. :param n_terms_range: (min, max) number of terms in KL expansion to use. The number of terms estimated from given precision is snapped to the given interval.

truncated SVD: cov_mat = U*diag(ev) * V, _cov_l_factor = U[:,0:m]*sqrt(ev[0:m])

Note on number of terms: According to: C. Schwab and R. A. Todor: KL Approximation of Random Fields by Generalized Fast Multiploe Method the eigen values should decay as (Proposition 2.18):

lambda_m ~ sigma^2 * (1/gamma) **(m**(1/d) + alpha) / Gamma(0.5 * m**(1/d))

where gamma = correlation length / domain diameter ans alpha is the correlation exponent. Gamma is the gamma function. ... should be checked experimantaly and generalized for sigma(X)

Returns

mlmc.random.correlated_field.**kozeny_carman**(*porosity*, *m*, *factor*, *viscosity*)
Kozeny-Carman law. Empirical relationship between porosity and conductivity. :param porosity: Porosity value. :param m: Power. Suitable values are 1 < m < 4 :param factor: [m^2]

E.g. 1e-7 , m = 3.48; juta fibers 2.2e-8 , 1.46; glass fibers 1.8e-13, 2.89; erruptive material 1e-12 2.76; erruptive material 1.8e-12 1.99; basalt

Parameters **viscosity** – [Pa . s], water: 8.90e-4

Returns

mlmc.random.correlated_field.**positive_to_range**(*exp*, *a*, *b*)
Mapping a positive parameter 'exp' from the interval <0, infty) to the interval <a,b). Suitable e.g. to generate meaningful porosity from a variable with lognormal distribution. :param exp: A positive parameter. (LogNormal distribution.) :param a, b: Range interval.

3.5.3 mlmc.random.frac_geom module

3.5.4 mlmc.random.gstools_wrapper module

3.5.5 Module contents

Subpackage provides random field generation and GSTools library interface

3.6 mlmc.sim

Contains a parent simulation class and a specific synthetic simulation

3.6.1 Submodules

3.6.2 mlmc.sim.simulation module

class mlmc.sim.simulation.**Simulation**
Bases: abc.ABC

static calculate(*config_dict*, *seed*)
Method that actually run the calculation, calculate fine and coarse sample and also extract their results :param config_dict: dictionary containing simulation configuration, LevelSimulation.config_dict (set in level_instance) :param seed: random seed, int :return: List[fine result, coarse result], both flatten arrays (see mlmc.sim.synth_simulation._calculate())

level_instance(*fine_level_params: List[float]*, *coarse_level_params: List[float]*) → mlmc.level_simulation.LevelSimulation
Create LevelSimulation object which is farther used for calculation etc. :param fine_level_params: :param coarse_level_params: :return: LevelSimulation

result_format() → List[mlmc.quantity.quantity_spec.QuantitySpec]
Define simulation result format :return: List[QuantitySpec, ...]

3.6.3 mlmc.sim.synth_simulation module

class mlmc.sim.synth_simulation.**SynthSimulation**(*config=None*)
Bases: *mlmc.sim.simulation.Simulation*

static calculate(*config*, *seed*)
Calculate fine and coarse sample and also extract their results :param config: dictionary containing simulation configuration :param seed: random number generator seed :return: np.ndarray, np.ndarray

static generate_random_samples(*distr*, *seed*, *size*)
Generate random samples from given distribution :param distr: scipy distribution :param seed: uint32 :param size: size of result :return: fine sample, coarse sample

level_instance(*fine_level_params: List[float]*, *coarse_level_params: List[float]*) → mlmc.level_simulation.LevelSimulation

Parameters

- **fine_level_params** –
- **coarse_level_params** –

Returns

n_ops_estimate(*step*)

result_format() → List[mlmc.quantity.quantity_spec.QuantitySpec]
Result format :return:

static sample_fn(*x*, *h*)
Calculates the simulation sample :param x: Distribution sample :param h: Simluation step :return: sample

static sample_fn_no_error(*x*, *h*)
Calculates the simulation sample :param x: Distribution sample :param h: Simluation step :return: sample

len_results = 0

n_nans = 0

nan_fraction = 0

result_dict = {}

class mlmc.sim.synth_simulation.**SynthSimulationWorkspace**(*config*)
Bases: *mlmc.sim.synth_simulation.SynthSimulation*

static calculate(*config*, *seed*)
Calculate fine and coarse sample and also extract their results :param config: dictionary containing simulation configuration :param seed: random number generator seed :return: np.ndarray, np.ndarray

static generate_random_samples(*distr*, *seed*, *size*)
Generate random samples from given distribution :param distr: scipy distribution :param seed: uint32 :param size: size of result :return: fine sample, coarse sample

level_instance(*fine_level_params: List[float], coarse_level_params: List[float]*) → mlmc.level_simulation.LevelSimulation

Parameters

- **fine_level_params** –
- **coarse_level_params** –

Returns

n_ops_estimate(*step*)

static sample_fn(*x*, *h*)
Calculates the simulation sample :param x: Distribution sample :param h: Simluation step :return: sample

static sample_fn_no_error(*x*, *h*)
Calculates the simulation sample :param x: Distribution sample :param h: Simluation step :return: sample

CONFIG_FILE = 'synth_sim_config.yaml'

len_results = 0

n_nans = 0

nan_fraction = 0

result_dict = {}

3.6.4 Module contents

Contains a parent simulation class and a specific synthetic simulation

3.7 mlmc.tool

Contains classes that provide an interface to other resources such as HDF5, Gmsh, PBS, ...

3.7.1 Submodules

3.7.2 mlmc.tool.context_statprof module

3.7.3 mlmc.tool.distribution module

class mlmc.tool.distribution.**Distribution**(*moments_obj*, *moment_data*, *domain=None*, *force_decay=(True, True)*, *monitor=False*)
Bases: object

Calculation of the distribution

cdf(*values*)

density(*value*, *moments_fn=None*)

Parameters

- **value** – float or np.array
- **moments_fn** – counting moments function

Returns density for passed value

end_point_derivatives()
Compute approximation of moment derivatives at endpoints of the domain. :return: array (2, n_moments)

estimate_density(*tol=None*)
Run nonlinear iterative solver to estimate density, use previous solution as initial guess. Faster, but worse stability. :return: None

estimate_density_minimize(*tol=1e-05*, *reg_param=0.01*)
Optimize density estimation :param tol: Tolerance for the nonlinear system residual, after division by std errors for individual moment means, i.e. res = ‖ (F_i - mu_i) / sigma_i ‖_2 :return: None

eval_moments(*x*)

extend_size(*new_size*)

mlmc.tool.distribution.**KL_divergence**(*prior_density*, *posterior_density*, *a*, *b*)
Compute D_KL(P | Q) = int_R P(x) log(P(X)/Q(x)) dx :param prior_density: P :param posterior_density: Q :return: KL divergence value

mlmc.tool.distribution.**L2_distance**(*prior_density*, *posterior_density*, *a*, *b*)

mlmc.tool.distribution.**compute_exact_moments**(*moments_fn*, *density*, *tol=0.0001*)
Compute approximation of moments using exact density. :param moments_fn: Moments function. :param n_moments: Number of mements to compute. :param density: Density function (must accept np vectors). :param a, b: Integral bounds, approximate integration over R. :param tol: Tolerance of integration. :return: np.array, moment values

3.7.4 mlmc.tool.flow_mc module

class mlmc.tool.flow_mc.**FlowSim**(*config=None*, *clean=None*)
Bases: *mlmc.sim.simulation.Simulation*

static calculate(*config*, *seed*)
Method that actually run the calculation, it's called from mlmc.tool.pbs_job.PbsJob.calculate_samples() Calculate fine and coarse sample and also extract their results :param config: dictionary containing simulation configuration, LevelSimulation.config_dict (set in level_instance) :param seed: random seed, int :return: List[fine result, coarse result], both flatten arrays (see mlmc.sim.synth_simulation.calculate())

static extract_mesh(*mesh_file*)
Extract mesh from file :param mesh_file: Mesh file path :return: Dict

static generate_random_sample(*fields*, *coarse_step*, *n_fine_elements*)
Generate random field, both fine and coarse part. Store them separeted. :return: Dict, Dict

level_instance(*fine_level_params: List[float], coarse_level_params: List[float]*) → mlmc.level_simulation.LevelSimulation
Called from mlmc.Sampler, it creates single instance of LevelSimulation (mlmc.) :param fine_level_params: in this version, it is just fine simulation step :param coarse_level_params: in this

version, it is just coarse simulation step :return: mlmc.LevelSimulation object, this object is serialized in SamplingPoolPbs and deserialized in PbsJob,

so it allows pass simulation data from main process to PBS process

static make_fields(*fields*, *fine_mesh_data*, *coarse_mesh_data*)
Create random fields that are used by both coarse and fine simulation :param fields: correlated_field.Fields instance :param fine_mesh_data: Dict contains data extracted from fine mesh file (points, point_region_ids, region_map) :param coarse_mesh_data: Dict contains data extracted from coarse mesh file (points, point_region_ids, region_map) :return: correlated_field.Fields

static result_format() → List[mlmc.quantity.quantity_spec.QuantitySpec]
Define simulation result format :return: List[QuantitySpec, ...]

FIELDS_FILE = 'fields_sample.msh'
Gather data for single flow call (coarse/fine)

Usage: mlmc.sampler.Sampler uses instance of FlowSim, it calls once level_instance() for each level step (The level_instance() method

is called as many times as the number of levels), it takes place in main process

mlmc.tool.pbs_job.PbsJob uses static methods in FlowSim, it calls calculate(). That's where the calculation actually runs, it takes place in PBS process

It also extracts results and passes them back to PbsJob, which handles the rest

GEO_FILE = 'mesh.geo'

MESH_FILE = 'mesh.msh'

MESH_FILE_VAR = 'mesh_file'

TIMESTEP_H1_VAR = 'timestep_h1'

TIMESTEP_H2_VAR = 'timestep_h2'

YAML_FILE = 'flow_input.yaml'

YAML_TEMPLATE = 'flow_input.yaml.tmpl'

total_sim_id = 0

mlmc.tool.flow_mc.**create_corr_field**(*model='gauss'*, *corr_length=0.125*, *dim=2*, *log=True*, *sigma=1*, *mode_no=1000*)
Create random fields :return:

mlmc.tool.flow_mc.**force_mkdir**(*path*, *force=False*)
Make directory 'path' with all parents, remove the leaf dir recursively if it already exists. :param path: path to directory :param force: if dir already exists then remove it and create new one :return: None

mlmc.tool.flow_mc.**substitute_placeholders**(*file_in*, *file_out*, *params*)
Substitute for placeholders of format '<name>' from the dict 'params'. :param file_in: Template file. :param file_out: Values substituted. :param params: { 'name': value, ... }

3.7.5 mlmc.tool.gmsh_io module

Module containing an expanded python gmsh class

class mlmc.tool.gmsh_io.**GmshIO**(*filename=None*)
Bases: object

This is a class for storing nodes and elements. Based on Gmsh.py

Members: nodes – A dict of the form { nodeID: [xcoord, ycoord, zcoord] } elements – A dict of the form { elemID: (type, [tags], [nodeIDs]) } physical – A dict of the form { name: (id, dim) }

Methods: read([file]) – Parse a Gmsh version 1.0 or 2.0 mesh file write([file]) – Output a Gmsh version 2.0 mesh file

read(*mshfile=None*)
Read a Gmsh .msh file.

Reads Gmsh format 1.0 and 2.0 mesh files, storing the nodes and elements in the appropriate dicts.

read_element_data()
Write given element data to the MSH file. Write only a single '$ElementData' section. :param f: Output file stream. :param ele_ids: Iterable giving element ids of N value rows given in 'values' :param name: Field name. :param values: np.array (N, L); N number of elements, L values per element (components) :return:

TODO: Generalize to time dependent fields.

read_element_data_head(*mshfile*)

reset()
Reinitialise Gmsh data structure

write_ascii(*mshfile=None*)
Dump the mesh out to a Gmsh 2.0 msh file.

write_binary(*filename=None*)
Dump the mesh out to a Gmsh 2.0 msh file.

write_element_data(*f*, *ele_ids*, *name*, *values*)
Write given element data to the MSH file. Write only a single '$ElementData' section. :param f: Output file stream. :param ele_ids: Iterable giving element ids of N value rows given in 'values' :param name: Field name. :param values: np.array (N, L); N number of elements, L values per element (components) :return:

TODO: Generalize to time dependent fields.

write_fields(*msh_file*, *ele_ids*, *fields*)
Creates input data msh file for Flow model. :param msh_file: Target file (or None for current mesh file) :param ele_ids: Element IDs in computational mesh corrsponding to order of field values in element's barycenter. :param fields: {'field_name' : values_array, ..}

3.7.6 mlmc.tool.hdf5 module

class mlmc.tool.hdf5.**HDF5**(*file_path*, *load_from_file=False*)
Bases: object

HDF5 file is organized into groups (h5py.Group objects) which is somewhat like dictionaries in python terminology - 'keys' are names of group members 'values' are members (groups (h5py.Group objects) and datasets (h5py.Dataset objects - similar to NumPy arrays)). Each group and dataset (including root group) can store metadata in 'attributes' (h5py.AttributeManager objects) HDF5 files (h5py.File) work generally like standard Python file objects

Our HDF5 file strucutre: Main Group: Keys:

Levels: h5py.Group

Attributes: level_parameters: [[a], [b], [], . . .]

Keys:

<N>: h5py.Group (N - level id, start with 0)

Attributes: id: str n_ops_estimate: float

Keys:

scheduled: h5py.Dataset dtype: S100 shape: (N,), N - number of scheduled values maxshape: (None,) chunks: True

collected_values: h5py.Dataset dtype: numpy.float64 shape: (Nc, 2, M) dtype structure is defined in simulation class maxshape: (None, 2, None) chunks: True

collected_ids: h5py.Dataset dtype: numpy.int16 index into scheduled shape: (Nc, 1) maxshape: (None, 1) chunks: True

failed: h5py.Dataset dtype: ('S100', 'S1000') shape: (Nf, 1) mashape: (None, 1) chunks: True

add_level_group(*level_id*)
Create group for particular level, parent group is 'Levels' :param level_id: str, mlmc.Level identifier :return: LevelGroup instance, it is container for h5py.Group instance

clear_groups()
Remove HDF5 group Levels, it allows run same mlmc object more times :return: None

create_file_structure(*level_parameters*)
Create hdf structure :param level_parameters: List[float] :return: None

init_header(*level_parameters*)
Add h5py.File metadata to .attrs (attrs objects are of class h5py.AttributeManager) :param level_parameters: MLMC level range of steps :return: None

load_from_file()
Load root group attributes from existing HDF5 file :return: None

load_level_parameters()

load_result_format()
Load format result, it just read dataset :return:

save_result_format(*result_format*, *res_dtype*)
Save result format to dataset :param result_format: List[QuantitySpec] :param res_dtype: result numpy dtype :return: None

result_format_dset_name
Result format dataset name :return: str

class mlmc.tool.hdf5.**LevelGroup**(*file_name*, *hdf_group_path*, *level_id*, *loaded_from_file=False*)
Bases: object

append_failed(*failed_samples*)
Save level failed sample ids (not append samples) :param failed_samples: set; Level sample ids :return: None

append_scheduled(*scheduled_samples*)
Save scheduled samples to dataset (h5py.Dataset) :param scheduled_samples: list of sample ids :return: None

append_successful(*samples: numpy.array*)
Save level samples to datasets (h5py.Dataset), save ids of collected samples and their results :param samples: np.ndarray :return: None

chunks(*n_samples=None*)

clear_failed_dataset()
Clear failed_ids dataset :return: None

collected(*chunk_slice*)
Read collected data by chunks, number of items in chunk is determined by LevelGroup.chunk_size (number of bytes) :param chunk_slice: slice() object :return: np.ndarray

collected_n_items()
Number of collected samples :return: int

get_failed_ids()
Failed samples ids :return: list of failed sample ids

get_finished_ids()
Get collected and failed samples ids :return: NumPy array

get_unfinished_ids()
Get unfinished sample ids as difference between scheduled ids and finished ids :return: list

scheduled()
Read level dataset with scheduled samples :return:

COLLECTED_ATTRS = {'sample_id': {'default_shape': (0,), 'dtype': {'formats': ['S10

FAILED_DTYPE = {'formats': ('S100', 'S1000'), 'names': ('sample_id', 'message')}

SCHEDULED_DTYPE = {'formats': ['S100'], 'names': ['sample_id']}

collected_ids_dset
Collected ids dataset :return: Dataset name

failed_dset
Dataset of ids of failed samples :return: Dataset name

n_ops_estimate
Get number of operations estimate :return: float

scheduled_dset
Dataset with scheduled samples :return: Dataset name

3.7.7 mlmc.tool.pbs_job module

class mlmc.tool.pbs_job.**PbsJob**(*output_dir*, *jobs_dir*, *job_id*, *level_sim_file*, *debug*)
Bases: object

calculate_samples()
Calculate scheduled samples :return:

static command_params()
Read command parameters - job identifier and file with necessary files :return: None

classmethod create_job(*output_dir*, *jobs_dir*, *job_id*, *level_sim_file*, *debug*)
Create PbsProcess instance from SamplingPoolPBS :param output_dir: str :param jobs_dir: str :param job_id: str :param level_sim_file: str, file name format of LevelSimulation serialization :param debug: bool, if True keep sample directories :return: PbsProcess instance

classmethod create_process()
Create PbsProcess via PBS :return:

static get_job_n_running(*job_id*, *jobs_dir*)
Get number of running (scheduled) samples for given unfinished jobs :param job_id: str :param jobs_dir: str, path to jobs directory :return: int

static get_scheduled_sample_ids(*job_id*, *jobs_dir*)
Get scheduled samples :param job_id: str :param jobs_dir: str :return:

static job_id_from_sample_id(*sample_id*, *jobs_dir*)
Get job ID for given sample ID :param sample_id: str :param jobs_dir: jobs directory with results :return: str, job id

static read_results(*job_id*, *jobs_dir*)
Read result file for given job id :param job_id: str :param jobs_dir: path to jobs directory :return: successful: Dict[level_id, List[Tuple[sample_id:str, Tuple[ndarray, ndarray]]]]

failed: Dict[level_id, List[Tuple[sample_id: str, error message: str]]] time: Dict[level_id: int, List[total time: float, number of success samples: int]]

save_sample_id_job_id(*job_id*, *sample_ids*)
Store the sample ID associated with the job ID :param job_id: str :param sample_ids: list of str

save_scheduled(*scheduled*)
Save scheduled samples to yaml file format: List[Tuple[level_id, sample_id]] :return: None

write_pbs_id(*pbs_job_id*)
Create empty file name contains pbs jobID and our jobID :param pbs_job_id: str :return: None

CLASS_FILE = 'pbs_process_serialized.txt'

FAILED_RESULTS = '{}_failed_results.yaml'

PBS_ID = '{}_'

SAMPLE_ID_JOB_ID = 'sample_id_job_id.json'

SCHEDULED = '{}_scheduled.yaml'

SUCCESSFUL_RESULTS = '{}_successful_results.yaml'

TIME = '{}_times.yaml'

3.7.8 mlmc.tool.process_base module

class mlmc.tool.process_base.**ProcessBase**
Bases: object

Parent class for particular simulation processes

all_collect(*sampler_list*)
Collect samples :param mlmc_list: List of mlmc.MLMC objects :return: None

analyze_error_of_level_variances(*cl*, *mlmc_level*)
Analyze error of level variances :param cl: mlmc.estimate.CompareLevels instance :param mlmc_level: selected MC method :return: None

analyze_error_of_log_variance(*cl*, *mlmc_level*)
Analyze error of level variances :param cl: mlmc.estimate.CompareLevels instance :param mlmc_level: selected MC method :return: None

analyze_error_of_regression_level_variances(*cl*, *mlmc_level*)
Analyze error of level variances :param cl: mlmc.estimate.CompareLevels instance :param mlmc_level: selected MC method :return: None

analyze_error_of_regression_variance(*cl*, *mlmc_level*)
Analyze error of regression variance :param cl: CompareLevels :param mlmc_level: selected MC method :return:

analyze_error_of_variance(*cl*, *mlmc_level*)
Analyze error of variance for particular mlmc method or for all collected methods :param cl: mlmc.estimate.CompareLevels instance :param mlmc_level: selected MC method :return: None

analyze_pdf_approx(*cl*)
Plot densities :param cl: mlmc.estimate.CompareLevels :return: None

analyze_regression_of_variance(*cl*, *mlmc_level*)
Analyze regression of variance :param cl: mlmc.estimate.CompareLevels instance :param mlmc_level: selected MC method :return: None

create_pbs_object(*output_dir*, *clean*)
Initialize object for PBS execution :param output_dir: Output directory :param clean: bool, if True remove existing files :return: None

generate_jobs(*mlmc*, *n_samples=None*)
Generate level samples :param n_samples: None or list, number of samples for each level :return: None

static get_arguments(*arguments*)
Getting arguments from console :param arguments: list of arguments :return: namespace

n_sample_estimate(*mlmc*, *target_variance=0.001*)
Estimate number of level samples considering target variance :param mlmc: MLMC object :param target_variance: float, target variance of moments :return: None

process_analysis(*cl*)
Main analysis function. Particular types of analysis called from here. :param cl: Instance of CompareLevels - list of Estimate objects :return:

rm_files(*output_dir*)
Rm files and dirs :param output_dir: Output directory path :return:

run(*renew=True*)
Run mlmc :return: None

set_environment_variables()
Set pbs config, flow123d, gmsh :return: None

set_moments(*n_moments*, *log=False*)
Create moments function instance :param n_moments: int, number of moments :param log: bool, If true then apply log transform :return:

setup_config(*n_levels*, *clean*)
Set simulation configuration depends on particular task :param n_levels: Number of levels :param clean: bool, if False use existing files :return: mlmc.MLMC

3.7.9 mlmc.tool.simple_distribution module

class mlmc.tool.simple_distribution.**SimpleDistribution**(*moments_obj*, *moment_data*, *domain=None*, *force_decay=(True, True)*, *verbose=False*)

Bases: object

Calculation of the distribution

cdf(*values*)

density(*value*)

Parameters

- **value** – float or np.array
- **moments_fn** – counting moments function

Returns density for passed value

end_point_derivatives()
Compute approximation of moment derivatives at endpoints of the domain. :return: array (2, n_moments)

estimate_density_minimize(*tol=1e-05*, *reg_param=0.01*)
Optimize density estimation :param tol: Tolerance for the nonlinear system residual, after division by std errors for individual moment means, i.e. res = ‖ (F_i - mu_i) / sigma_i ‖_2 :return: None

eval_moments(*x*)

mlmc.tool.simple_distribution.**KL_divergence**(*prior_density*, *posterior_density*, *a*, *b*)
Compute D_KL(P | Q) = int_R P(x) log(P(X)/Q(x)) dx :param prior_density: P :param posterior_density: Q :return: KL divergence value

mlmc.tool.simple_distribution.**L2_distance**(*prior_density*, *posterior_density*, *a*, *b*)

mlmc.tool.simple_distribution.**best_fit_all**(*values*, *range_a*, *range_b*)

mlmc.tool.simple_distribution.**best_p1_fit**(*values*)
Find indices a < b such that linear fit for values[a:b] have smallest residual / (b - a)** alpha alpha is fixed parameter. This should find longest fit with reasonably small residual. :return: (a, b)

mlmc.tool.simple_distribution.**compute_exact_cov**(*moments_fn*, *density*, *tol=1e-10*)
Compute approximation of covariance matrix using exact density. :param moments_fn: Moments function. :param density: Density function (must accept np vectors). :param tol: Tolerance of integration. :return: np.array, moment values

mlmc.tool.simple_distribution.**compute_exact_moments**(*moments_fn*, *density*, *tol=1e-10*)
Compute approximation of moments using exact density. :param moments_fn: Moments function. :param density: Density function (must accept np vectors). :param tol: Tolerance of integration. :return: np.array, moment values

mlmc.tool.simple_distribution.**compute_semiexact_cov**(*moments_fn*, *density*, *tol=1e-10*)
Compute approximation of covariance matrix using exact density. :param moments_fn: Moments function. :param density: Density function (must accept np vectors). :param tol: Tolerance of integration. :return: np.array, moment values

mlmc.tool.simple_distribution.**compute_semiexact_moments**(*moments_fn*, *density*, *tol=1e-10*)

mlmc.tool.simple_distribution.**construct_ortogonal_moments**(*moments*, *cov*, *tol=None*)
For given moments find the basis orthogonal with respect to the covariance matrix, estimated from samples. :param moments: moments object :return: orthogonal moments object of the same size.

mlmc.tool.simple_distribution.**detect_treshold_slope_change**(*values*, *log=True*)
Find a longest subsequence with linear fit residual X% higher then the best at least 4 point fit. Extrapolate this fit to the left.

Parameters

- **values** – Increassing sequence.
- **log** – Use logarithm of the sequence.

Returns Index K for which K: should have same slope.

mlmc.tool.simple_distribution.**lsq_reconstruct**(*cov*, *eval*, *evec*, *treshold*)

3.7.10 mlmc.tool.stats_tests module

mlmc.tool.stats_tests.**anova**(*level_moments*)
Analysis of variance :param level_moments: moments values per level :return: bool

mlmc.tool.stats_tests.**chi2_test**(*var_0*, *samples*, *max_p_val=0.01*, *tag=''*)
Test that variance of samples is sigma_0, false failures with probability max_p_val. :param sigma_0: Exact mean. :param samples: Samples to test. :param max_p_val: Probability of failed t-test for correct samples.

mlmc.tool.stats_tests.**t_test**(*mu_0*, *samples*, *max_p_val=0.01*)
Test that mean of samples is mu_0, false failures with probability max_p_val.

Perform the two-tailed t-test and Assert that p-val is smaller then given value. :param mu_0: Exact mean. :param samples: Samples to test. :param max_p_val: Probability of failed t-test for correct samples.

3.7.11 Module contents

Contains classes that provide an interface to other resources such as HDF5, Gmsh, PBS, ...

Python Module Index

e

m

Index

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

V

W

Y