mlmc

Release 1.0.0

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

Python Module Index

e

Index

E