MXMCPy Documentation

Release 1.0

UQCoE

Mar 28, 2022

Contents

Contents:

CHAPTER 1

Introduction

Multi Model Monte Carlo with Python (`MXMCPy`) is a software package developed as a general capability for computing the statistics of outputs from an expensive, high-fidelity model by leveraging faster, low-fidelity models for speedup. Several existing methods are currently implemented, including multi-level Monte Carlo (MLMC) [1], multi-fidelity Monte Carlo (MFMC) [2], and approximate control variates (ACV) [3, 4]. Given a fixed computational budget and a collection of models with varying cost/accuracy, `MXMCPy` will determine an sample allocation strategy across the models that results in an estimator with optimal variance reduction using any of the available algorithms.

With `MXMCPy`, users can easily compare existing methods to determine the best choice for their particular problem, while developers have a basis for implementing and sharing new variance reduction approaches. See the remainder of the documentation for more details of using the code. For additional information, see the report that accompanied the release of `MXMCPy` [5].

[1] Giles, M. B.: Multi-level Monte Carlo path simulation. OPERATIONS RESEARCH , vol. 56, no. 3, 2008, pp. 607–617.

[2] Peherstorfer, B.; Willcox, K.; and Gunzburger, M.: Optimal Model Management for Multifidelity Monte Carlo Estimation. SIAM Journal on Scientific Computing, vol. 38, 01 2016, pp. A3163–A3194

[3] Gorodetsky, A.; Geraci, G.; Eldred, M.; and Jakeman, J. D.: A generalized approximate control variate framework for multifidelity uncertainty quantification. Journal of Computational Physics, 2020, p. 109257

[4] Bomarito, G. F., Leser, P. E., Warner, J. E., and Leser, W. P: On the Optimization of Approximate Control Variates with Parametrically-Defined Estimators. In Preparation.

[5] Bomarito, G. F., Warner, J. E., Leser, P. E., Leser, W. P., and Morrill, L.: Multi Model Monte Carlo with Python (MXMCPy). NASA/TM–2020–220585. 2020.

CHAPTER 2

Available Algorithms

There are currently 13 optimization algorithms implemented in `MXMCPy`. The algorithms are listed below along with some general information.

2.1 Multi-level Monte Carlo

Algorithm Name	Optimization	Sampling Strategy
MLMC	Analytic	Recursive Difference

Ref: Giles, M. B.: Multi-level Monte Carlo path simulation. Operations Research , vol. 56, no. 3, 2008, pp. 607–617.

2.2 Multifidelity Monte Carlo

Algorithm Name	Optimization	Sampling Strategy
MFMC	Analytic	Multifidelity

Ref: Peherstorfer, B.; Willcox, K.; and Gunzburger, M.: Optimal Model Management for Multifidelity Monte Carlo Estimation. SIAM Journal on Scientific Computing, vol. 38, 01 2016, pp. A3163–A3194

2.3 Approximate Control Variates

Algorithm Name	Optimization	Sampling Strategy
WRDIFF	Numerical	Recursive Difference
ACVIS	Numerical	Independent Samples
ACVMF	Numerical	Multifidelity
ACVKL	Numerical	Multifidelity

Ref: Gorodetsky, A.; Geraci, G.; Eldred, M.; and Jakeman, J. D.: A generalized approximate control variate framework for multifidelity uncertainty quantification. Journal of Computational Physics, 2020, p. 109257

2.4 Parametrically-defined Approximate Control Variates

Algorithm Name	Optimization	Sampling Strategy
GRDSR	Numerical	Recursive Difference
GRDMR	Numerical	Recursive Difference
GISSR	Numerical	Independent Samples
GISMR	Numerical	Independent Samples
ACVMFU	Numerical	Multifidelity
GMFSR	Numerical	Multifidelity
GMFMR	Numerical	Multifidelity

Ref: Bomarito, G. F., Leser, P. E., Warner, J. E., and Leser, W. P: On the Optimization of Approximate Control Variates with Parametrically-Defined Estimators. In Preparation.

CHAPTER 3

End-to-End Example

This example provides a simple demonstration of MXMCPy functionality. Here, the high-fidelity model considered is the Ishigami function, which is frequently used to test methods for uncertainty quantification:

$$f^{(1)} \;=\; sin(z_1) \;+\; 5\, sin^2(z_2) \;+\; \frac{1}{10}\, z_3^4 sin(z_1)$$

$$with \;\; z_i \sim U(-\pi,\, \pi)$$

The following two correlated functions [1] are treated as low-fidelity models to facilitate the use of multi-model estimators:

$$f^{(2)} \;=\; sin(z_1) \;+\; 4.75\, sin^2(z_2) \;+\; \frac{1}{10}\, z_3^4 sin(z_1)$$

$$f^{(3)} \;=\; sin(z_1) \;+\; 3\, sin^2(z_2) \;+\; \frac{9}{10}\, z_3^2 sin(z_1)$$

Furthermore, the costs associated with evaluating the functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ are assumed to be 1.0, 0.05, 0.001 seconds, respectively. The goal is to estimate $E[f^{(1)}]$ using MXMCPy by leveraging all three models and comparing with the analytical solution, $E[f^{(1)}] = 2.5$.

This example covers each step for utilizing MXMCPy: estimating the covariance of model outputs from pilot samples, performing the sample allocation optimization, and forming an estimator from outputs generated from each model according to the optimal sample allocation.

The full source code for this example can be found in the MXMCPy repository:

`/examples/ishigami/run_ishigami.py`

3.1 Step 1: Compute model outputs for pilot samples

In order to determine the optimal sample allocation across the available models, MXMCPy needs the cost of each model as well as the covariance matrix of the outputs from each model. If the covariance matrix is unknown, it can be estimated from pilot samples, demonstrated here.

Starting from the beginning, the necessary Python modules are imported, including MXMCPy classes and Numpy:

```
import numpy as np

from mxmc import Optimizer
from mxmc import OutputProcessor
from mxmc import Estimator
```

It is assumed that a user-defined class (`IshigamiModel`) implementing the Ishigami models and function (`get_uniform_sample_distribution`) for sampling the uniform random inputs are available. Three IshigamiModels are then instantiated per the parameters in the equations above and stored in a list variable named "models":

```
from ishigami_model import IshigamiModel

num_pilot_samples = 10
model_costs = np.array([1, .05, .001])

high_fidelity_model =   IshigamiModel(a=5.,   b=.1, c=4.)
medium_fidelity_model = IshigamiModel(a=4.75, b=.1, c=4.)
low_fidelity_model =    IshigamiModel(a=3.,   b=.9, c=2.)

models = [high_fidelity_model, medium_fidelity_model, low_fidelity_model]
```

Ten pilot samples are computed with each model as follows:

```
pilot_inputs = get_uniform_sample_distribution(num_pilot_samples)
pilot_outputs = list()
for model in models:
    pilot_outputs.append(model.evaluate(pilot_inputs))
```

The covariance matrix is then computed using the `MXMCPy` `OutputProcessor` class:

```
covariance_matrix = OutputProcessor.compute_covariance_matrix(pilot_outputs)
```

At this point the necessary elements for computing optimal sample allocation are now available: model costs, pilot outputs, and a covariance matrix.

3.2 Step 2: Perform sample allocation optimization

The sample allocation optimization with `MXMCPy` is now performed assuming a computational budget or target cost of 10000 seconds.

In the below snippet taken from the example code, all available optimization algorithms are individually tested to find the method that produces the lowest estimator variance. The Optimizer.optimize() method returns an instance of the `OptimizationResult` class with attributes for estimatore variance and optimal sample allocation. The sample allocation with the lowest variance will be used to generate an estimator in the subsequent steps.

```
target_cost = 10000
variance_results = dict()
sample_allocation_results = dict()

mxmc_optimizer = Optimizer(model_costs, covariance_matrix)

algorithms = Optimizer.get_algorithm_names()
for algorithm in algorithms:

```

(continues on next page)

(continued from previous page)

```
    opt_result = mxmc_optimizer.optimize(algorithm, target_cost)
    variance_results[algorithm] = opt_result.variance
    sample_allocation_results[algorithm] = opt_result.allocation

    print("{} method variance: {}".format(algorithm, opt_result.variance))

best_method = min(variance_results, key=variance_results.get)
sample_allocation = sample_allocation_results[best_method]

print("Best method: ", best_method)
```

The `Optimizer` class also provides functionality for determining an optimal subset of the models via the boolean parameter auto_model_selection of the Optimizer.optimize() method. By default, all provided models are used. Note that enabling this option could take considerably longer as every combination of the models will be tested.

```
mxmc_optimizer = Optimizer(model_costs,
                           covariance_matrix,
                           auto_model_selection=True)

opt_result = mxmc_optimizer.optimize(algorithm, target_cost)
variance_results = opt_result.variance
sample_allocation_results = opt_result.allocation
```

3.3 Step 3: Generate input samples for models

The `SampleAllocation` class provides functionality for determining how many random input samples are needed and how they are to be allocated across the available models. This is demonstrated below for the optimal sample allocation object found in the previous step:

```
num_total_samples = sample_allocation.num_total_samples
all_samples = get_uniform_sample_distribution(num_total_samples) # User code.
model_input_samples = sample_allocation.allocate_samples_to_models(all_samples)
```

The num_total_samples property can be referenced for creation of an ndarray of inputs, which can then be provided to the allocate_samples_to_models() method. This method will redistribute the ndarray of input samples into a list of ndarrays, each containing the prescribed number of samples for each model.

3.4 Step 4: Compute model outputs for prescribed inputs

Now that the input samples for each model have been generated and allocated, outputs are generated by evaluating the Ishigami models as follows:

```
model_outputs = list()
for input_sample, model in zip(model_input_samples, models):
    model_outputs.append(model.evaluate(input_sample))
```

The outputs are stored in a list of ndarrays corresponding to each model.

Note that for the practical case where evaluating the available models is time consuming and must be done in an *offline* fashion, the `SampleAllocation` class has functionality to save and load to/from disc. This way, the optimal sample allocation can be saved after Step 2 above and then loaded to generate an estimator in Step 5 next.

3.5 Step 5: Form estimator

Finally, an estimator for $E[f^{(1)}]$ is computed using the `Estimator` class and the model outputs from the previous step:

```
estimator = Estimator(sample_allocation, covariance_matrix)
estimate = estimator.get_estimate(model_outputs)

print("Estimate = ", estimate)
```

Note that the sample allocation object from Step 2 and the covariance matrix from Step 1 are required here. The covariance matrix could also be updated at this point using the model outputs generated in the previous step. The `MXMCPy` estimator is close to the true value of 2.5.

[1] "High-dimensional and higher-order multifidelity Monte Carlo estimators" (Quaglino, 2018)

CHAPTER 4

Source Code Documentation

Documentation for the core MXMC classes.

4.1 Optimizer Module

class optimizer.**Optimizer**(**args*, ***kwargs*)

User interface for accessing all MXMC variance minimization optimizers for computing optimal sample allocations.

Parameters

- **model_costs** (*list of floats*) – cost (run time) of all models
- **covariance** (*2D np.array*) – Covariance matrix defining covariance (of outputs) among all available models. Size MxM where M is # models.

static get_algorithm(*algorithm_name*)

Returns a reference to a class indicated by the provided name.

static get_algorithm_names()

Returns a list of strings containing the names of all available optimization algorithms.

optimize(*algorithm*, *target_cost*, *auto_model_selection=False*)

Performs variance minimization optimization to determine the optimal sample allocation across available models within a specified target computational cost.

Parameters

- **algorithm** (*string*) – name of method to use for optimization (e.g., "mlmc", "mfmc", "acvkl").
- **target_cost** (*float*) – total target cost constraint so that optimizer finds sample allocation that requires less than or equal computation time.
- **auto_model_selection** (*Boolean*) – flag to use automatic model selection in optimization to test all subsets of models for best set.

Returns An OptimizationResult namedtuple with entries for cost, variance, and sample_array. cost (float) is expected cost of all model evaluations prescribed in sample_array (np.array). variance is the minimized variance from the optimization.

4.2 Sample Allocation Module

class sample_allocations.**SampleAllocation**

Base class for managing the allocations of random input samples (model evaluations) across available models. Provides a user with number of model evaluations required to generate an estimator and how to partition input samples to do so, after the sample allocation optimization problem is solved.

Parameters **compressed_allocation** (*2D np.array*) – a two dimensional array completely describing a MXMC sample allocation; returned by each optimizer class as the result of sample allocation optimization. See docs for a description and example of the format.

Variables

- **num_total_samples** – number of total input samples needed across all available models
- **_utilized_models** – list of indices corresponding to models with samples allocated to them
- **num_models** – total number of available models

get_number_of_samples_per_model()

Returns the total number of samples allocated to each available model :type: (list of integers)

get_sample_indices_for_model()

Parameters **model_index** (*int*) – index of model to return indices for (from 0 to #models-1)

Returns binary array with indices of samples required by specified model

Type (np.array with length of num_total_samples)

allocate_samples_to_models()

Allocates a given array of all input samples across all available models according to the sample allocation determined by an MXMX optimizer.

Parameters **all_samples** (*2D np.array*) – array of user-generated random input samples with length equal to num_total_samples

Returns individual arrays of input samples for all available models :type: (list of np.arrays with length equal to num_models)

4.3 Output Processor Module

class output_processor.**OutputProcessor**

Class to estimate covariance matrix from collection of output samples from multiple models. Handles the general case where each models' outputs were not computed from the same set of random inputs and pairwise overlaps between samples are identified using a SampleAllocation object.

static compute_covariance_matrix(*model_outputs*, *sample_allocation=None*)

Estimate covariance matrix from collection of output samples from multiple models. In the simple case, the outputs for each model were generated from the same collection of inputs and covariance can be straightforwardly computed. In the general case, the outputs were not computed from the same inputs and a SampleAllocation object must be supplied to identify overlap between samples to compute covariance.

Parameters

- **model_outputs** (*list of np.array*) – list of arrays of outputs for each model. Each array must be the same size unless a sample allocation is provided.
- **sample_allocation** (*SampleAllocation object.*) – An MXMC sample allocation object defining the indices of samples that each model output was generated for, if applicable. Default is None indicating that all supplied model output arrays are the same size and were generated for the same inputs.

Returns covariance matrix among all model outputs (2D np.array with size equal to the number of models).

4.4 Estimator Module

class estimator.**Estimator**

Class to create MXMC estimators given an optimal sample allocation and outputs from high & low fidelity models.

Parameters

- **allocation** (*SampleAllocation object*) – SampleAllocation object defining the optimal sample allocation using an MXMC optimizer.
- **covariance** (*2D np.array*) – Covariance matrix defining covariance among all models being used for estimator. Size MxM where M is # models.

4.5 Utilities Module

generic_numerical_optimization.**perform_slsqp_then_nelder_mead**(*constraints*, *initial_guess*, *obj_func*, *obj_func_and_grad*)

generic_numerical_optimization.**perform_slsqp**(*constraints*, *initial_guess*, *obj_func_and_grad*)

generic_numerical_optimization.**perform_nelder_mead**(*constraints*, *initial_guess*, *obj_func*)

read_sample_allocation.**read_sample_allocation**()

Read sample allocation from file

Parameters **filename** (*string*) – name of hdf5 sample allocation file

Returns appropriate child of the SampleAllocationBase class based on the optimization method stored in the hdf5 file.

sample_modification.**adjust_sample_allocation_to_cost**(*target_cost*, *model_costs*, *covariance*)

Tests all possible increases to sample counts per group and returns sample allocation with the lowest variance while limiting total cost to be within the specified target_cost.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index