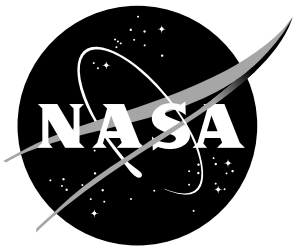
NASA/TM–2020–220585



Multi Model Monte Carlo with Python (MXMCPy)

Geoffrey F. Bomarito
Langley Research Center, Hampton, Virginia

James E. Warner
Langley Research Center, Hampton, Virginia

Patrick E. Leser
Langley Research Center, Hampton, Virginia

William P. Leser
Langley Research Center, Hampton, Virginia

Luke Morrill
Langley Research Center, Hampton, Virginia

April 2020

NASA STI Program. . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

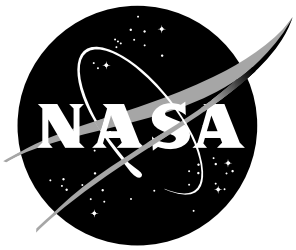
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at `http://www.sti.nasa.gov`
- E-mail your question to `help@sti.nasa.gov`
- Phone the NASA STI Information Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2020–220585



Multi Model Monte Carlo with Python (MXMCPy)

Geoffrey F. Bomarito
Langley Research Center, Hampton, Virginia

James E. Warner
Langley Research Center, Hampton, Virginia

Patrick E. Leser
Langley Research Center, Hampton, Virginia

William P. Leser
Langley Research Center, Hampton, Virginia

Luke Morrill
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2020

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

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. Motivated by uncertainty propagation problems where classical Monte Carlo (MC) simulation is computationally intractable, various multi-model MC approaches have recently emerged that yield unbiased estimators with significantly reduced variance relative to MC for the same cost. These existing methods include multi-level Monte Carlo (MLMC), multi-fidelity Monte Carlo (MFMC), and approximate control variates (ACV). Given a fixed computational budget and a collection of models with varying cost/accuracy, each method seeks a sample allocation strategy across the models that results in an estimator with optimal variance reduction.

`MXMCPy` is a versatile tool that enables convenient access to many existing multi-model MC approaches within one modular and extensible package. 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. This report introduces the `MXMCPy` software, providing a summary of the problem solving workflow for users as well as a brief overview of the code layout for developers.

1 Motivation

Uncertainty quantification (UQ) is one of the fastest growing paradigms within the field of computational science and engineering. In many practical applications, UQ is performed in conjunction with high-fidelity simulations arising from the numerical solution of partial differential equations governing the problem. One of the key topics within UQ is the propagation of uncertainties through these potentially expensive computational models from random input parameters to quantities of interest. This report introduces software that offers an efficient approach for solving this class of UQ problems.

Within the context of uncertainty propagation, one is often interested in computing estimators for the statistics of the model outputs. Arguably the most well known and general purpose approach for calculating these estimators is Monte Carlo (MC) simulation. MC estimators are robust, unbiased, and have a rate of convergence that is independent of the dimensionality (number of random input variables) of the problem. However, the convergence rate is still relatively slow, often requiring thousands or millions of samples (and model evaluations), and hence MC estimators are impractical for expensive models. To reduce this computational expense, lower fidelity surrogate models are commonly constructed to replace the high-fidelity model for MC simulation, but will generally yield biased and potentially inaccurate estimators.

Multi-model MC methods have recently emerged to bridge this gap, combining information from two or more models of varying fidelity and computational cost to produce efficient, accurate predictions. By retaining the high-fidelity model, un-

biased estimators of a quantity of interest can be constructed, while introducing an ensemble of low-fidelity models provides computational speedup. Given a fixed computational budget, the crux of multi-model approaches is the determination of a sample allocation strategy across the available models that results in an estimator with the minimum variance. This is posed as an optimization problem where the estimator variance is the objective function and the resulting optimal sample allocation is dependent on the relative costs of, and correlations between, the models. Similar to traditional MC simulation, multi model methods have convergence rates independent of the number of input parameters, avoiding the *curse of dimensionality* as an advantage over many alternative approaches.

Existing multi-model methods are distinguished by both the types of models that are used and the approach for solving the variance minimization optimization problem. Multilevel MC (MLMC) [1, 2] restricted low-fidelity models to those arising from coarsened discretizations of the same governing equations in space/time, while multifidelity MC (MFMC) [3] generalized the types of models permitted to have different forms (e.g., data-driven, reduced-order, analytical, etc.). Both MLMC and MFMC introduce assumptions (on model dependency structure) that simplify the variance minimization problem to yield analytical expressions for optimal sample allocation. More recently, a generalized approximate control variates (ACV) [4] approach was proposed that unified and improved upon MLMC-based and MFMC methods. It was shown that by considering a general model dependency structure, orders of magnitude improvement in variance reduction was possible with a newly proposed set of estimators whose sample allocations are optimized numerically. However, only a preliminary sketch of possible numerical optimization schemes was offered, leaving the door open for more robust sample allocation strategies to be developed. Additionally, the best multi-model estimator (including MLMC/MFMC) for a given problem was shown to be highly dependent on the types, relative costs, and correlations of the available models.

Motivated by the breakthroughs and findings of the work of Gorodetsky et al. [4], Multi Model Monte Carlo with Python (`MXMCPy`)[1] is a software package being developed as a general capability for computing statistics of expensive, high-fidelity models. To the developers' knowledge, it is the first publicly available library to offer convenient access to many of the existing multi-model MC approaches, including MLMC, MFMC, and ACV estimators. `MXMCPy` users can easily compare these existing methods to determine the best choice for their particular problem given the nature of their available models. The software also includes utilities to assist a non-expert user in completing an end-to-end uncertainty propagation analysis with a particular algorithm. For researchers in the field, `MXMCPy` provides an open-source basis to develop and share new variance reduction methods by exploring improved sample allocation optimization schemes and easily testing against the current state-of-the-art.

This report introduces the `MXMCPy` software, providing a summary of the problem solving workflow for users as well as a brief overview of the code layout for developers. While the functionality of `MXMCPy` is summarized within, this report is not meant to

[1] Available open source at https://github.com/nasa/MXMCPy

serve as the package's user documentation, which can be found with the source code itself. The user documentation will contain more detailed descriptions of `MXMCPy` classes and methods as well as *getting started* instructions that include information on installation and dependencies. The following section provides an overview of how a user could leverage `MXMCPy` for an end-to-end uncertainty propagation analysis, including code snippets demonstrating a handful of key `MXMCPy` classes and methods. Then, a brief introduction to the software layout of `MXMCPy` is provided for potential developers, highlighting the particular sections of code that require modification to implement new multi-model optimization strategies. A brief verification study is then covered. Finally, the key points of the report are summarized in the last section.

2 Introduction for Users

This section provides a high-level overview of how an `MXMCPy` user could leverage the software for an end-to-end uncertainty propagation analysis. As a starting point, the `MXMCPy` workflow assumes a user has access to a high-fidelity model that predicts a certain quantity of interest (output) for a given realization of random inputs, along with an ensemble of (one or more) lower cost/fidelity models that approximate the same output. Furthermore, the user must have a mechanism for producing an arbitrary number of random input samples to their models (*e.g.*, via a random number generator or more advanced sampling method [5]). With these prerequisites, `MXMCPy` can be leveraged to compute an accurate (low variance) and efficient estimator for the user's quantity of interest.

Figure 1 shows a diagram of the `MXMCPy` workflow for the case of three computational models (one high fidelity and two lower fidelity). For example, in an application such as additive manufacturing (AM) via selective laser melting (SLM), the quantity of interest might be the expected value of melt pool width resulting from a specific set of processing parameters (*e.g.*, laser power and velocity). Here, there is uncertainty in these parameters and others, such as the absorptivity of the powderbed, and a researcher is interested in constructing an estimator for the expected melt pool width. The high fidelity model (`model 1`) is a thermomechanical model implemented with the finite element method that is accurate but expensive. The user also has access to faster, but cruder, data-driven (`model 2`) and analytical (`model 3`) models.

Within the overall workflow, `MXMCPy` was developed primarily to address the problem of determining the optimal sample allocation across available models to minimize estimator variance, as it is the key technical challenge in multi-model MC approaches. Other tasks such as the generation of random input parameters and the execution of the computational models themselves are highly problem-specific and are thus the responsibility of the user. For completeness, an overview of both the `MXMCPy`- and user-focused steps is provided.

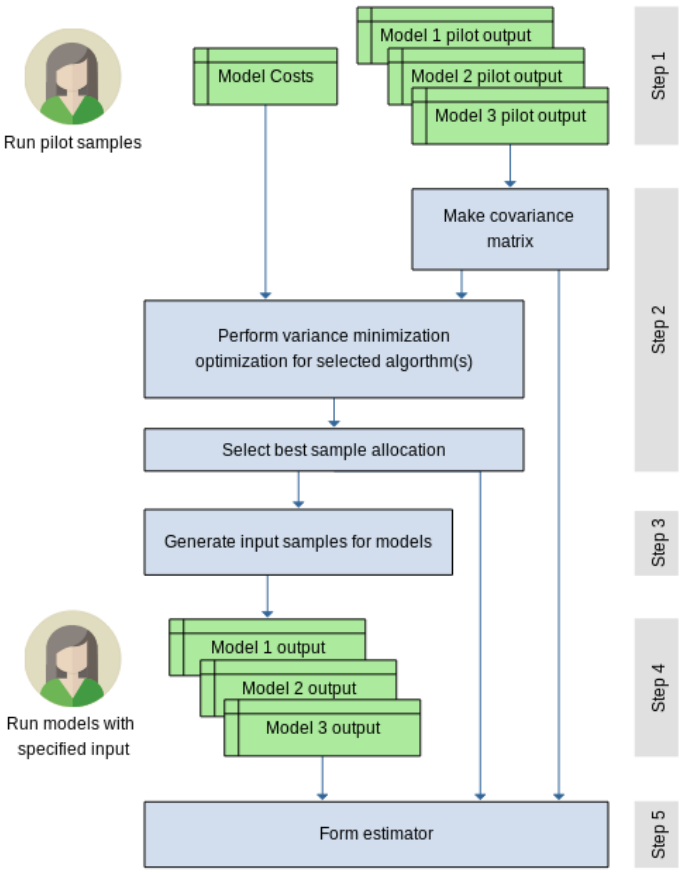


Figure 1: Example `MXMCPy` workflow

Step 1: Compute model outputs for pilot samples (User)

In order to determine optimal sample allocations, `MXMCPy` needs information regarding the computational cost of the available models as well as the relationship between the predicted outputs of each model. The former is simply a list of estimated average run times for each model while the latter is quantified via the covariance matrix of the outputs from each model. Unless a user has an a priori estimate of the covariance between each of the available models, the covariance matrix can be readily calculated based on an initial set of outputs ("pilot outputs", Figure 1). Here, each model is executed for the same set of pilot inputs[2] and the resulting outputs are collected for use in the next step below. Model costs can also be measured during the execution of the pilot samples.

Note that the number of pilot input samples used is a balance of cost versus accuracy; the more samples used results in a more accurate covariance estimate but at the expense of longer computation time to execute the models. At the end of this step, it is assumed that two Python lists, `model_costs` and `pilot_outputs` are available in the workflow to follow:

```
model_costs = [cost_m1, cost_m2, cost_m3]
pilot_outputs = [pilot_outputs_m1, pilot_outputs_m2, pilot_outputs_m3]
```

with the costs (estimated run times) and arrays of pilot outputs for each model, respectively. Costs are represented by floating point values, which can be either absolute measures or normalized in some fashion. Outputs are represented as arrays of floating point values.

Step 2: Perform sample allocation optimization

With estimated costs and pilot outputs from all available models, `MXMCPy` can now calculate the optimal sample allocation that minimizes estimator variance for a given computational budget. Furthermore, a range of multi-model MC methods can be tested on the solution to this optimization problem and compared to determine the best one (*i.e.* , the one that yields the smallest variance) for the available models. This functionality is available through the `Optimizer` class in `MXMCPy`, which encapsulates all the details of the sample allocation optimization and provides access to all available allocation strategies.

For example, a multi-model MC algorithm comparison could be performed using `MXMCPy` as follows

```
from mxmc import Optimizer
from mxmc import OutputProcessor

covariance_matrix = OutputProcessor.compute_covariance_matrix(
                                    pilot_outputs)
mxmc_optimizer = Optimizer(model_costs, covariance_matrix)

min_variance = 1e9
for algorithm in ["mfmc", "mlmc", "acvkl"]:
```

[2]This is not strictly necessary using `MXMCPy` functionality, but there must be at least two overlapping samples between each model pair to calculate covariance.

```
  opt_result = mxmc_optimizer.optimize(algorithm, target_cost)
  if opt_result.variance < min_variance:
    min_variance = opt_result.variance
    sample_allocation = opt_result.allocation
```

yielding the sample allocation across available models that minimizes estimator variance for the computational budget (`target_cost`). Here, the `OutputProcessor` utility in `MXMCPy` is used to compute the covariance matrix for the models based on the pilot outputs generated in the previous step. After the `Optimizer` class is initialized using model costs and covariance matrix, the `optimize` function is used to find the optimal sample allocation and minimized estimator variance for a given algorithm and target cost. Here, three common multi-model methods are tested for illustration. For more information about `MXMCPy` functionality and all available algorithms, see the user documentation that accompanies the source code.

Step 3: Generate input samples for models

Once the optimal sample allocation problem has been solved with `MXMCPy`, a user will need to use the results to generate input samples to evaluate each of their models. All information regarding sample allocation is encapsulated in the `SampleAllocation` class in `MXMCPy`, an object of which is returned following optimization with the `Optimizer` class (*e.g.*, the `sample_allocation` in the previous step).

One straightforward approach for generating input samples for each model using the `SampleAllocation` object is outlined in the code snippet below:

```
num_total_samples = sample_allocation.num_total_samples
all_samples = ... #user-defined random sample generation
model_input_samples = sample_allocation.allocation_samples_to_models(
                                    all_samples)
```

Here, a user first determines the total number of samples (across all models) that will be required for their problem. Then, the user will use an external, problem-specific code to generate an array of random samples (with size `num_total_samples`). Finally, `sample_allocation` is used to partition the input samples into subsets to be run with each model. Note that `model_input_samples` is a list containing input sample arrays for each model (three arrays in this case):

```
model_input_samples = [inputs_m1, inputs_m2, inputs_m3]
```

Step 4: Compute model outputs for prescribed inputs

With input samples randomly generated and allocated to the available models, outputs must be computed for each model. Similar to generating pilot model outputs (Step 1), this step is the responsibility of the user and does not require `MXMCPy` functionality. The user will need to produce a list containing arrays of outputs for each model, *e.g.*,

```
model_outputs = [outputs_m1, outputs_m2, outputs_m3]
```

. Here, the sizes of the output arrays must match those of the input arrays in `model_input_samples` from the previous step and the ordering between inputs and outputs should be consistent.

Step 5: Form estimator

The final step in the end-to-end `MXMCPy` workflow is to calculate the final estimator for a user's quantity of interest. This is easily done using the `Estimator` class in `MXMCPy`, *e.g.*,

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

Here, the optimal `SampleAllocation` object found previously and the covariance matrix among outputs are used to initialize the `Estimator` class. Then, the model outputs computed in the previous step are used to calculate the estimator. The estimated variance associated with this estimator is available in the `min_variance` from Step 2 above.

3 Introduction for Developers

`MXMCPy` was designed with the intent of being modular and easily extensible. This extensibility offers two benefits: when researchers implement new extensions they are (1) easily accessible to the broader research community and (2) easier to compare to existing methods. This section provides a more in depth look at the code within the `MXMCPy` package in order to orient future developers and encourage future contributions.

3.1 Code Organization

The `MXMCPy` package is organized – see Figure 2 – with the purpose of making the code most relevant to *users* up front while storing the code which is more relevant to *developers*. The code with which *users* interact is located in the `mxmc` directory. For instance, all the classes and functions used in Section 2 are all present here. The `optimizers` subdirectory contains the code that performs the various sample allocation optimizations (the bulk of the code associated with `MXMCPy`). The `optimizers` directory is organized with analytical optimizers (*i.e.* MFMC and MLMC) directly within this directory and numerically based optimizers (*e.g.*ACVMF and ACVIS) within the `approximate_control_variates` sub directory. The `util` directory contains utility classes and functions that may be useful in development and testing of `MXMCPy` code.

3.2 The Sample Allocation Optimization

Before describing what is likely to be the most common extension of `MXMCPy`, the definition of custom optimizers, it is first necessary to more formally define the sample allocation optimization. Without loss of generality, a multi-model MC estimator

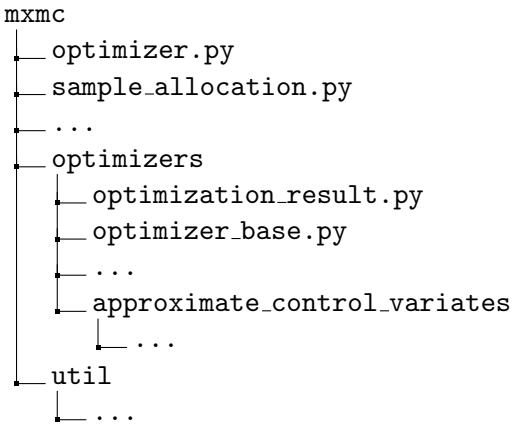


Figure 2: Directory structure of `MXMCPy`

with approximate control means can be written:

$$\tilde{Q}(\boldsymbol{\alpha}, \mathbf{z}) = \hat{Q}_0(z_0) + \sum_{i=1}^{M} \alpha_i \left(\hat{Q}_i(z_{i+}) - \hat{Q}_i(z_{i-}) \right),$$

where M is the number of lower fidelity models, $\boldsymbol{\alpha}$ is the set of control variate weights $\{\alpha_i\}_{i=1}^{M}$, and $\mathbf{z}$ is the sample allocation associated with the potentially overlapping sets of samples $\{z_0, z_{1+}, z_{1-}, ..., z_{M+}, z_{M-}\}$ [4]. $\hat{Q}_i(z)$ is an MC estimator based on samples z: *i.e.* $\hat{Q}_i(z)$ is the mean of the outputs of model i given inputs z.

Because $\tilde{Q}$ is an unbiased estimator of $\mathbb{E}[Q_0]$, error of the estimate is associated solely with the variance of the estimator: $\mathbb{V}\text{ar}[\tilde{Q}]$. Thus, an optimal estimator is for which the variance is minimized:

$$\tilde{Q}^{opt} = \min_{\boldsymbol{\alpha}, \mathbf{z}} \mathbb{V}\text{ar}[\tilde{Q}(\boldsymbol{\alpha}, \mathbf{z})].$$

The optimal values of $\boldsymbol{\alpha}$ can be calculated for arbitrary $\mathbf{z}$ (see [4]), *i.e.*

$$\boldsymbol{\alpha}^{opt}(\mathbf{z}) = \arg\min_{\boldsymbol{\alpha}} \mathbb{V}\text{ar}[\tilde{Q}(\boldsymbol{\alpha}, \mathbf{z})].$$

This reduces the estimator optimization problem to one based solely on the sample allocation,

$$\tilde{Q}^{opt} = \min_{\mathbf{z}} \mathbb{V}\text{ar}[\tilde{Q}(\boldsymbol{\alpha}^{opt}(\mathbf{z}), \mathbf{z})].$$

Thus the primary result of a sample allocation optimization is a `SampleAllocation` object, which is written formally as

$$\mathbf{z}^{opt} = \arg\min_{\mathbf{z}} \mathbb{V}\text{ar}[\tilde{Q}(\boldsymbol{\alpha}^{opt}(\mathbf{z}), \mathbf{z})].$$

Note that typically this optimization is constrained by the total cost a user is willing to invest in getting the estimate.

3.3 Defining a Custom Optimizer

The implementation of a custom optimizer in `MXMCPy` contains two steps: (1) implement the optimizer and (2) connect the optimizer to the front end.

3.3.1 Implementing an Optimizer

An implementation of the abstract base class `OptimizerBase` (`optimizer_base.py`) is needed. The only requirement of the implementation is the definition of the function:

```
def optimize(self, target_cost)
    # perform the optimization
    return optimization_result
```

The return value of the `optimize` function is expected to be an `OptimizationResult` (`optimization_result.py`) object. It contains the actual cost of the estimator, the variance of the estimator, and a `SampleAllocation` (`sample_allocation.py`) object. The creation of the `SampleAllocation` object is handled by the constructor of the `OptimizationResult` class. The inputs to the constructor are straight forward and described in the code's documentation; however, the format of the `sample_array` parameter requires special attention.

The `sample_array` is a two dimensional `numpy` array of integers which fully characterizes a sample allocation $\mathbf{z}$. Table 1 illustrates the format for a case of 4 models (1 high fidelity model and M=3 lower fidelity models) using an MFMC-type sampling strategy. The rows of the array are each associated with a group of samples, the union of which includes all the samples in the allocation. The first column indicates the number of samples in the sample group. The remaining columns indicate whether the sample group is included (1) or excluded (0) in each z_i. An illustration and further explanation of this format can be found in Figure 2.1 of reference [4].

3.3.2 Registering the Optimizer

In order to make the custom optimizer usable from the main `Optimizer` interface, it must be added to the `ALGORITHM_MAP` in `optimizer.py`. The `ALGORITHM_MAP` is a map from the string name of an optimizer to its optimizer class. For instance, a new optimizer could be registered using

Table 1: Diagram of a `sample_array` representing an MFMC-type sample allocation with 4 models.

	number of samples	z_0	z_{1+}	z_{1-}	z_{2+}	z_{2-}	z_{3+}	z_{3-}
sample group 1	10	1	1	1	1	1	1	1
sample group 2	50	0	0	1	1	1	1	1
sample group 3	100	0	0	0	0	1	1	1
sample group 4	1000	0	0	0	0	0	0	1

```
from my_code import CustomOptimizerClass
ALGORITHM_MAP['custom_optimizer'] = CustomOptimizerClass
```

The optimizer could then be used in the same manner as others.

```
from mxmc import Optimizer
mxmc_optimizer = Optimizer(model_costs, covariance_matrix)
opt_result = mxmc_optimizer.optimize('custom_optimizer', target_cost)
```

4 Verification

In this section, `MXMCPy` is used in a simple verification problem to illustrate that the variance of `MXMCPy` estimators is accurately implemented in sample allocation optimization. Here, the predicted estimator variance for a given method, $\mathbb{V}\text{ar}[\tilde{Q}(\boldsymbol{\alpha}^{opt}, \mathbf{z}^{opt})]$ from Section 3.2, is compared to the actual variance of the final estimator obtained from the end-to-end procedure described in Section 2. The code that was used to produce these results is included as an example in the code's repository.

4.1 Model definitions

Let $[\Sigma]$ be an MxM covariance matrix and $\{X\}$ be an M-dimensional random input vector with independent and identically distributed components according to a standard Normal distribution. If L is the lower triangular matrix obtained by Cholesky decomposition of $[\Sigma] = [L][L]^T$, then outputs $\{Y\} = [L]\{X\}$ have covariance $[\Sigma]$.

In this verification example, $M = 3$ and the covariance matrix was

$$[\Sigma] = \begin{bmatrix} 1.0 & 0.9 & 0.8 \\ 0.9 & 1.0 & 0.9 \\ 8.0 & 0.9 & 1.0 \end{bmatrix},$$

representing a reasonable covariance matrix from a set of models with decreasing fidelity. The Cholesky decomposition of $[\Sigma]$ is

$$[L] = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.9 & 0.43588989 & 0.0 \\ 0.8 & 0.41294832 & 0.43528575 \end{bmatrix}.$$

Given the 3-dimensional random inputs X, three models for high-, med-, and low-fidelity are defined as

$$\begin{aligned} Y_{high} &= X_1 \\ Y_{med} &= 0.9X_1 + 0.43588989X_2 \\ Y_{low} &= 0.8X_1 + 0.41294832X_2 + 0.43528575X_3, \end{aligned}$$

which have the desired covariance. To facilitate the use of `MXMCPy` in this simple setting, the models are assumed to have costs of 1, 0.1, and 0.01, respectively.

4.2 Estimator Variance

Using the exact covariance matrix and the assumed model costs from the previous section, an `MXMCPy` sample allocation optimization (Section 2, step 2) was performed for three algorithms with a target cost of 80. The result of the optimization includes both a `SampleAllocation` and a prediction of estimator variance. The `SampleAllocation` is used to create an `Estimator` following the remainder of the workflow in Section 2. The resulting `Estimator` should make estimates which have a variance close to the predicted estimator variance. To quantify the actual variance of the `Estimator`, steps 3-5 were repeated 10,000 times with the above defined models resulting in 10,000 independent estimates. The variance of these estimates is compared to the predicted variance in Table 2.
In all three of the algorithms the relative error is small, illustrating the accuracy of the predicted estimator variance calculated by `MXMCPy`.

4.3 Effect of Pilot Samples

The verification problem above used the exact covariance matrix to perform the sample allocation optimization. In practice, however, pilot samples will generally need to be used to approximate the covariance matrix. This approximation can lead to an increased error in predicted estimator variance.

The workflow of the previous section was repeated using differing numbers of pilot samples to approximate the covariance matrix to illustrate the impact of this error. For a given number of pilot samples, ten calculations of the covariance matrix were made (step 1, 2) and ten corresponding evaluations of the estimator variance were made (again using 10,000 random estimates for each). The mean relative error is calculated for each algorithm as a function of number of pilot samples. The results are illustrated in Table 3. The last row of Table 3 uses the exact covariance matrix, and so defines the approximate error floor for this study. A clear trend is seen where an increased accuracy in the covariance matrix (increased number of pilot samples) corresponds to more accurate predictions of estimator variance. It is reiterated that the magnitude of this approximation error is problem dependent and the number of pilot samples chosen reflects a balance of accuracy versus computational cost.

5 Summary

This report provides an introduction to the `MXMCPy` Python package that enables efficient uncertainty propagation using multi-model MC methods. Specifically, multi-model MC aim to compute statistics of outputs from expensive, high-fidelity models

Table 2: Verification of variance predictions using the true covariance matrix.

Algorithm	Predicted Variance	Actual Variance	Relative Error (%)
MFMC	5.287e-03	5.360e-03	1.38
ACVIS	5.998e-03	6.069e-03	1.18
GRDMR	5.332e-03	5.367e-03	0.66

Table 3: The effect of pilot samples on accuracy of variance predictions.

Number of Pilot Samples	Relative Error (%) MFMC	ACVIS	GRDMR
10	102.1	106.2	107.9
20	27.2	36.2	33.2
40	16.9	18.6	16.4
80	12.4	10.8	13.4
160	7.5	7.7	7.8
320	4.0	3.8	3.8
∞	1.1	0.8	1.2

by leveraging faster, low-fidelity models for speedup. Several variations of this concept have emerged in recent years (MLMC, MFMC, ACV, etc.), and it remains an active area of research. `MXMCPy` is a general tool that provides easy access to these existing multi-model MC approaches within one modular and extensible package. 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. This report provided both an overview of how `MXMCPy` can be applied to an end-to-end analysis for users and an introduction to the layout of the software to help facilitate modification and extension of the code for interested developers and researchers.

References

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

2. Giles, M. B.: Multilevel Monte Carlo Methods. *Monte Carlo and Quasi-Monte Carlo Methods 2012*, J. Dick, F. Y. Kuo, G. W. Peters, and I. H. Sloan, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 83–103.

3. 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.

4. 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. URL `http://www.sciencedirect.com/science/article/pii/S0021999120300310`.

5. Gamerman, D.; and Lopes, H. F.: *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference*. Chapman and Hall/CRC, Boca Raton, Florida, Second ed., 2006.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-04-2020	Technical Memorandum	

4. TITLE AND SUBTITLE
Multi Model Monte Carlo with Python (MXMCPy)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Geoffrey F. Bomarito and James E. Warner and Patrick E. Leser and William P. Leser and Luke Morrill

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
295670.01.20.23.30

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, Virginia 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L–21133

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2020–220585

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT

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. Motivated by uncertainty propagation problems where classical Monte Carlo (MC) simulation is computationally intractable, various multi-model MC approaches have recently emerged that yield unbiased estimators with significantly reduced variance relative to MC for the same cost. These existing methods include multi-level Monte Carlo (MLMC), multi-fidelity Monte Carlo (MFMC), and approximate control variates (ACV). Given a fixed computational budget and a collection of models with varying cost/accuracy, each method seeks a sample allocation strategy across the models that results in an estimator with optimal variance reduction.

`MXMCPy` is a versatile tool that enables convenient access to many existing multi-model MC approaches within one modular and extensible package. 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. This report introduces the `MXMCPy` software, providing a summary of the problem solving workflow for users as well as a brief overview of the code layout for developers.

15. SUBJECT TERMS
Uncertainty Quantification, Monte Carlo, Surrogate Modeling, Model Fusion

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (help@sti.nasa.gov)
U	U	U	UU	18	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658