MLMCPy Documentation

Release 1.0

James Warner

Feb 06, 2019

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

MLMCPy is an implementation of the Multi-Level Monte Carlo (MLMC) method in Python. It is a software package developed to enable user-friendly utilization of the Multi-Level Monte Carlo (MLMC) approach for uncertainty quantification.

MLMCPy's primary class, MLMCSimulator, is initialized with an instance of a descendant of the Input abstract base class and a list of instances of descendants of the Model abstract base class.

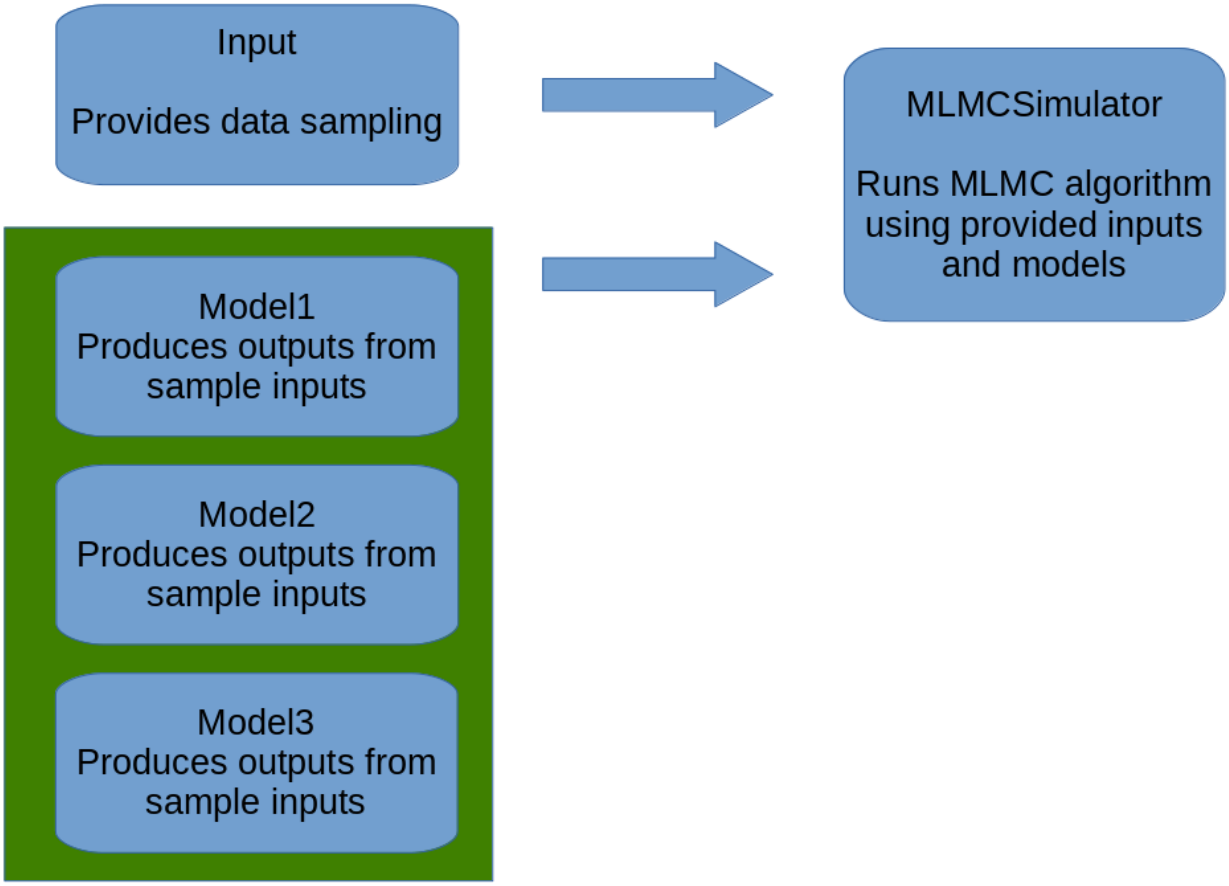
The Input class provides input data sampling that can be provided to the models. Some of the included Input classes include InputFromData, which loads sample inputs from saved data files. Also included is the RandomInput class, which allows sampling from a distribution.

The Model class receives sample inputs and produces outputs from those samples. The ModelFromData class, for example, is to be provided with precomputed input and output data in data files along with associated model costs. Running the evaluate function with a sample input parameter will return the corresponding output.

MLMCSimulator's simulate() function proceeds in two phases. First, it determines the number of samples that should be passed through each model. This is determined based on either the epsilon parameter, which specifies the target precision of the estimate, or by the target_cost parameter, which specifies the desired total cost of performing the simulation.

Once the number of samples to be taken at each level has been determined, the simulator enters its second phase, in which each model processes input samples as determined by the first phase.

Once processing is complete, simulate() returns an estimate for each quantity of interest, the number of samples taken at each level, and the variances of model outputs for each quantity of interest.

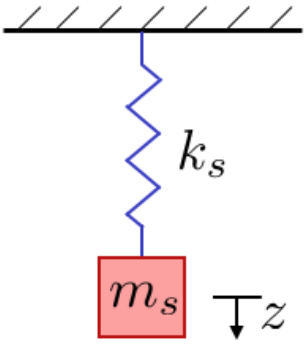

Input
Provides data sampling
Model1
Produces outputs from sample inputs
Model2
Produces outputs from sample inputs
Model3
Produces outputs from sample inputs
MLMCSimulator
Runs MLMC algorithm using provided inputs and models

CHAPTER

TWO

EXAMPLE - SPRING MASS SYSTEM

This example provides a simple demonstration of MLMCPy functionality. The goal is to estimate the maximum displacement of a spring-mass system with random stiffness using Multi-Level Monte Carlo (MLMC) and compare to standard Monte Carlo simulation. The example covers all steps for computing MLMC estimators using MLMCPy, including defining a random input parameter (spring stiffness) using a MLMCPy random input, creating a user-defined computational model (spring mass numerical integrator) that uses the standardized MLMCPy interface, and running MLMC with a hierarchy of these models (according to time step size) to obtain an estimator for a quantity of interest (max. displacement) within a prescribed precision. The full source code for this example can be found in the MLMCPy repository: `/MLMCPy/examples/spring_mass/from_model/run_mlmc_from_model.py`



2.1 Problem Specification

The governing equation of motion for the system is given by

$$m_s \ddot{z} = -k_s z + m_s g \tag{2.1}$$

where m_s is the mass, k_s is the spring stiffness, g is the acceleration due to gravity, z is the vertical displacement of the mass, and $\ddot{z}$ is the acceleration of the mass. The source of uncertainty in the system will be the spring stiffness, which is modeled as a random variable of the following form:

$$K_s = \gamma + \eta B \tag{2.2}$$

where γ and η are shift and scale parameters, respectively, and $B = \text{Beta}(\alpha, \beta)$ is a standard Beta random variable with shape parameters α and β. Let these parameters take the following values: $\gamma = 1.0N/m$, $\eta = 2.5N/m$, $\alpha = 3.0$, and $\beta = 2.0$. The mass is assumed to be deterministic, $m_s = 1.5kg$, and the acceleration due to gravity is $g = 9.8m^2/s$.

With uncertainty in an input parameter, the resulting displacement, Z, is a random variable as well. The quantity of interest in this example with be the maximum displacement over a specified time window, $Z_{max} = max_t(Z)$. The

equation of motion in Equation (1) can be numerically integrated over the time window with a specified time step, and the maximum of the resulting displacement time series can be taken to obtain Z_{max}.

The goal of this example will be to estimate the expected value of the maximum displacement, $E[Z_{max}]$, using the MLMC approach with MLMCPy and compare it to a Monte Carlo simulation solution. The MLMC expected value estimate of a random quantity, X, is as follows:

$$E[X] \approx E[X_0] + \sum_{l=1}^{L} E[X_l - X_{l-1}] \tag{2.3}$$

where L is the number of levels (and number of models of varying fidelity) used. Each expected value is approximated by it's Monte Carlo estimator:

$$E[X_l] \approx \frac{1}{N_l} \sum_{i=1}^{N_l} X_l^{(i)} \tag{2.4}$$

The MLMC method prescribes the number of samples to be taken on each level, N_l, based on a user-specified precision and the variance of the output on each level.

For this example, three levels will be employed, where each level corresponds to a maximum displacement predicted using a spring mass simulator model with varying time step. First, the random spring stiffness is represented using a MLMCPy random input, then a spring mass model is created and three instantiations of it are made with different time steps, then MLMC is used to estimate the expected maximum displacement to a prescibed precision.

2.2 Step 1: Initialization; define the random input parameter

Begin by importing the needed Python modules, including MLMCPy classes and the SpringMassModel class that defines the spring mass numerical integrator:

```
import numpy as np
import timeit

from spring_mass import SpringMassModel
from MLMCPy.input import RandomInput
from MLMCPy.mlmc import MLMCSimulator
```

Below is a snippet of the SpringMassModel class, the entire class can be found in the MLMCPy repo (`/MLMCPy/examples/spring_mass/from_model/spring_mass_model.py`):

```
from MLMCPy.model import Model

class SpringMassModel(Model):
    """
    Defines Spring Mass model with 1 free param (stiffness of spring, k). The
    quantity of interest that is returned by the evaluate() function is the
    maximum displacement over the specified time interval
    """

    def __init__(self, mass=1.5, gravity=9.8, state0=None, time_step=None,
                 cost=None):
```

Note that user-defined models in MLMCPy must inherit from the MLMCPy abstract class `Model` and implement an `evaluate` function that accepts and returns numpy arrays for inputs and outputs, respectively. Here, the `time_step` argument governs numerical integration and will define the three levels used for MLMC.

The first step in an analysis is to define the random variable representing the model inputs. Here, the spring stiffness K_s is defined by a Beta random variable and created with MLMCPy as follows:

```
# Step 1 - Define random variable for spring stiffness:
# Need to provide a sampleable function to create RandomInput instance.
def beta_distribution(shift, scale, alpha, beta, size):

    return shift + scale*np.random.beta(alpha, beta, size)

stiffness_distribution = RandomInput(distribution_function=beta_distribution,
                                     shift=1.0, scale=2.5, alpha=3., beta=2.)
```

The `RandomInput` class is initialized with a function that produces random samples and any parameters it requires. See the *Input Module Documentation* for more details about specifying random input parameters with MLMCPy.

2.3 Step 2: Generate reference solution using standard Monte Carlo simulation

Here a reference solution is generated using standard Monte Carlo simulation and a prescribed number of samples. This is done by instantiating a spring mass model (time step = 0.01) and evaluating the model for random samples of the stiffness random variable defined in Step 1. The code to do so is as follows:

```
# Step 2: Run standard Monte Carlo to generate a reference solution and target␣
↪precision
num_samples = 5000
model = SpringMassModel(mass=1.5, time_step=0.01)
input_samples = stiffness_distribution.draw_samples(num_samples)
output_samples_mc = np.zeros(num_samples)

start_mc = timeit.default_timer()

for i, sample in enumerate(input_samples):
    output_samples_mc[i] = model.evaluate([sample])

mc_total_cost = timeit.default_timer() - start_mc
mean_mc = np.mean(output_samples_mc)
precision_mc = (np.var(output_samples_mc) / float(num_samples))
```

The total time to compute the solution, the mean estimate, and the resulting precision in the estimate are stored for comparison to MLMC later. Note that this precision will be used as the target threshold for MLMC in Step 4.

2.4 Step 3: Initialize a hierarchy (3 levels) of models for MLMC

In order to apply the MLMC method (Equation (3)), multiple levels of models (defined by cost/accuracy) must be defined. The following code initializes three separate spring mass models defined by varying time step (the smaller the time step, the higher the cost and accuracy):

```
# Step 3 - Initialize spring-mass models for MLMC. Here using three levels
# with MLMC defined by different time steps

model_level1 = SpringMassModel(mass=1.5, time_step=1.0)
model_level2 = SpringMassModel(mass=1.5, time_step=0.1)
model_level3 = SpringMassModel(mass=1.5, time_step=0.01)

models = [model_level1, model_level2, model_level3]
```

2.5 Step 4: Run MLMC to estimate the expected maximum displacement

With a random input defined in Step 1 and multiple fidelity models defined in Step 3, MLMC can now be used to estimate the maximum displacement using the `MLMCSimulator` class. Here, MLMC is used to obtain an estimate with the same level of precision that was calculated using Monte Carlo in Step 2. The following code executes the MLMC algorithm and times it for comparison later:

```
# Step 4 - Initialize MLMC & predict max displacement to specified error.
mlmc_simulator = MLMCSimulator(stiffness_distribution, models)

start_mlmc = timeit.default_timer()

[estimates, sample_sizes, variances] = \
    mlmc_simulator.simulate(epsilon=np.sqrt(precision_mc),
                            initial_sample_sizes=100,
                            verbose=True)

mlmc_total_cost = timeit.default_timer() - start_mlmc
```

Note that `MLMCSimulator` uses an initial setup to estimate output variances, where the provided models are executed a number of times equal to the `initial_sample_sizes` argument. See *Multi-Level Monte Carlo Simulator Documentation* for more details about the `MLMCSimulator` API.

2.6 Step 5: Compare the MLMC and Monte Carlo results

Finally, the MLMC estimate is compared to the Monte Carlo reference solution.

```
print 'MLMC estimate: %s' % estimates[0]
print 'MLMC precision: %s' % variances[0]
print 'MLMC total cost: %s' % mlmc_total_cost

print "MC # samples: %s" % num_samples
print "MC estimate: %s" % mean_mc
print "MC precision: %s" % precision_mc
print "MC total cost: %s" % mc_total_cost
print "MLMC computational speedup: %s" %  (mc_total_cost / mlmc_total_cost)
```

For one particular execution of this script in a single-core environment, the following results were obtained. Note that MLMC used 5553, 386, and 3 samples (model evaluations) on levels 1, 2, and 3, respectively, compared with 5000 samples of level 3 used by Monte Carlo simulation. The resulting computational speed up was 2.62

Description	MLMC Value	MC Value
Estimate	12.2739151773	12.390705590117555
Error	0.045171289	0.071619124
Precision	0.009916230329196151	0.010780941000560835
Total cost (seconds)	0.63	1.14

CHAPTER

THREE

SOURCE CODE DOCUMENTATION

Documentation for the core MLMCPy classes.

3.1 Multi-Level Monte Carlo Simulator Documentation

class MLMCSimulator.**MLMCSimulator**(*data*, *models*)

Computes an estimate based on the Multi-Level Monte Carlo algorithm.

__init__(*data*, *models*)

Requires a data object that provides input samples and a list of models of increasing fidelity.

Parameters

- **data** (Input) – Provides a data sampling function.
- **models** (*list (*Model*)*) – Each model Produces outputs from sample data input.

simulate(*epsilon*, *initial_sample_sizes=100*, *target_cost=None*, *sample_sizes=None*, *verbose=False*)

Perform MLMC simulation. Computes number of samples per level before running simulations to determine estimates. Can be specified based on target precision to achieve (epsilon), total target cost (in seconds), or on number of sample to run on each level directly.

Parameters

- **epsilon** (*float, list of floats, or ndarray.*) – Desired accuracy to be achieved for each quantity of interest.
- **initial_sample_sizes** (*ndarray, int, list*) – Sample sizes used when computing cost and variance for each model in simulation.
- **target_cost** (*float or int*) – Target cost to run simulation (optional). If specified, overrides any epsilon value provided.
- **sample_sizes** (*ndarray*) – Number of samples to compute at each level
- **verbose** (*bool*) – Whether to print useful diagnostic information.
- **only_collect_sample_sizes** (*bool*) – indicates whether to bypass simulation phase and simply return prescribed number of samples for each model. Return value is changed to one dimensional ndarray.

Returns Tuple of ndarrays (estimates, sample count per level, variances)

3.2 Input Module Documentation

class Input.**Input**
Abstract base class defining data inputs from which samples can be drawn.

draw_samples(*num_samples*)
Draws requested number of samples from a data source.

Parameters **num_samples** (*int*) – Number of sample rows to be returned.

Returns A ndarray with num_samples rows. Can be one or two dimensional.

reset_sampling()
Used to reset the sample index to 0 when drawing from an indexed data set such as an ndarray extracted from a data file. Does not need to perform any actions for some data sources, for example random distributions as in the RandomInput class.

class InputFromData.**InputFromData**(*input_filename*, *delimiter=' '*, *skip_header=0*, *shuffle_data=True*)
Used to draw random samples from a data file.

__init__(*input_filename*, *delimiter=' '*, *skip_header=0*, *shuffle_data=True*)

Parameters

- **input_filename** (*string*) – path of file containing data to be sampled.
- **delimiter** (*str or int*) – Character used to separate data in data file. Can also be an integer to specify width of each entry.
- **skip_header** (*int*) – Number of header rows to skip in data file.
- **shuffle_data** (*bool*) – Whether or not to randomly shuffle data during initialization.

draw_samples(*num_samples*)
Returns an array of samples from the previously loaded file data.

Parameters **num_samples** (*int*) – Number of samples to be returned.

Returns 2d ndarray of samples, each row being one sample. For one dimensional input data, this will have shape (num_samples, 1)

reset_sampling()
Used to restart sampling from beginning of data set.

class RandomInput.**RandomInput**(*distribution_function*, *random_seed=None*, ***distribution_function_args*)
Used to draw samples from a specified distribution . Any distribution function provided must accept a "size" parameter that determines the sample size.

__init__(*distribution_function*, *random_seed=None*, ***distribution_function_args*)

Parameters

- **distribution_function** (*function*) – Returns a sample of a distribution with the sample sized determined by a "size" parameter. Typically, a numpy function such as numpy.random.uniform() is used.
- **distribution_function_args** – Any arguments required by the distribution function, with the exception of "size", which will be provided to the function when draw_samples is called.

draw_samples(*num_samples*)
Returns num_samples samples from a distribution in the form of a numpy array.

Parameters **num_samples** (*int*) – Total number of samples to take across all CPUs.

Returns A ndarray of distribution sample. If multiple CPUs are available, will return a subset of sample determined by number of CPUs.

reset_sampling()
Used to reset the sample index to 0 when drawing from an indexed data set such as an ndarray extracted from a data file. Does not need to perform any actions for some data sources, for example random distributions as in the RandomInput class.

3.3 Model Documentation

class Model.**Model**
Abstract base class for Models which should evaluate sample inputs to produce outputs.

Parameters **inputs** – one dimensional ndarray

Returns two dimensional ndarray

class ModelFromData.**ModelFromData**(*input_filename*, *output_filename*, *cost*, *delimiter=None*, *skip_header=0*, *wait_cost_duration=False*)
Used to produce outputs from inputs based on data provided in text files.

__init__(*input_filename*, *output_filename*, *cost*, *delimiter=None*, *skip_header=0*, *wait_cost_duration=False*)

Parameters

- **input_filename** (*string*) – Path to file containing input data.
- **output_filename** (*string*) – Path to file containing output data.
- **cost** (*float or ndarray*) – The average cost of computing a sample output. If multiple quantities of interest are provided in the data, an ndarray of costs can specify cost for each quantity of interest.
- **delimiter** (*string, int, list(int)*) – Delimiter used to separate data in data files, or size of each entry in the case of fixed width data.
- **wait_cost_duration** (*bool*) – Whether to sleep for the duration of the cost in order to simulate real time model evaluation.

evaluate(*input_data*)
Returns outputs corresponding to provided input_data. input_data will be searched for within the stored input data and the index of the match will be used to extract and return output data.

Parameters **input_data** (*ndarray*) – Scalar or vector to be searched for in input data.

Returns A ndarray of matched output_data.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

D

E

I

M

R

S