firedrake - mlmc

User Guide

A. Gregory

Imperial College London

Contents

Preface

This is the documentation manual for the **firedrake-mlmc** package that comprises of a toolbox of Python algorithms that fit under the multilevel Monte Carlo uncertainty quantification framework for random PDEs using **firedrake**, an automated finite element solver.

The repository can be downloaded from:
`https://github.com/firedrakeproject/firedrake-mlmc`

ALASTAIR GREGORY
Imperial College London
2016

Forming Mesh and Function Space Hierarchies

The first step to using this package is understanding how to use hierarchies of meshes and function spaces alongside **firedrake**. Suppose we build a **Mesh** on a unit square, and want to use this as the coarsest mesh in a refined hierarchy of $L = 4$ other meshes (so a hierarchy of length $L + 1$). We can do this via the example below, by using a **GeneralisedMeshHierarchy**. Note the **GeneralisedMeshHierarchy** differs from the traditional **firedrake** multigrid **MeshHierarchy** by allowing the additional option of refining with a factor $M \geq 2$ where this is required to be a factor of 2. Each **GeneralisedMeshHierarchy** has an attribute **GeneralisedMeshHierarchy._hierarchy** corresponding to the M-refined hierarchy of meshes, and **GeneralisedMeshHierarchy._full_hierarchy** corresponding to the full hierarchy of intermediate meshes (with $M = 2$).
One can then build **GeneralisedFunctionSpaceHierarchy**s via the following commands:

```
import firedrake_mlmc as *

# mesh on coarsest level
coarsest_mesh = UnitIntervalMesh(5)

# L and refinement factor
L = 4; M = 2

# define mesh and function space hierarchies
m = GeneralisedMeshHierarchy(coarsest_mesh, L, M)
fs = GeneralisedFunctionSpaceHierarchy(m, 'DG', 1)
```

Furthermore these generalised hierarchies are also invariant to the prolong and injection methods that exist in multigrid **firedrake** currently. For example, given `F = Function(fs[0]).assign(2)` and `G = Function(fs[2])`, one can prolong F to G via `prolong(F, G)`.
Just as with their **Mesh** counterparts, a **GeneralisedFunctionSpaceHierarchy** has attributes, **GeneralisedFunctionSpaceHierarchy._hierarchy** corresponding to the M-refined hierarchy of functionspaces, and **GeneralisedFunctionSpaceHierarchy._full_hierarchy** corresponding to the full hierarchy of intermediate function spaces (with $M = 2$).

Creating States of Coarse and Fine Solutions

An important aspect of multilevel Monte Carlo is being able to have coupled realisations of random variables (in our case, random discretized fields) from 2 consecutive levels, for example `coarse_lvl` and `fine_lvl`. These are typically coupled due to positive correlation one is required to have between the two of them, such as is usually imparted by using the same random input (e.g. initial condition). To this end, it is important that this package keeps each pair of coupled realisations together. It does this via `State(coarse, fine)`. An example of how to create a **State** is below.

```
# define the coarse and fine levels
coarse_lvl = 0; fine_lvl = 1

# define the coarse and fine functions
coarse = Function(fs[coarse_lvl])
fine = Function(fs[fine_lvl])
```

At this point, one should use `coarse` and `fine` as the fields to solve the coarse and finer discretization of the random system on. As a demostration of this on an extremely simple random system, we interpolate `sin((x + p) * 2 * pi)` on to `coarse` and `fine` where $p \sim N(0, .1)$. As each couple of realisations have to be positively correlated in the multilevel Monte Carlo framework, one should use the same draw of p.

Example 1

```
import numpy as np

def example_system(coarse, fine,
                   coarse_lvl, fine_lvl,
                   m):
    # randomly draw shift
    p = np.random.normal(0, .1, 1)[0]

    # interpolate
    x = SpatialCoordinate(m[coarse_lvl])
    coarse.interpolate(sin((x[0] + p) * 2 * pi))
    x = SpatialCoordinate(m[fine_lvl])
    fine.interpolate(sin((x[0] + p) * 2 * pi))
    return coarse, fine
```

Now that we have two realisations of our (all be it in this case incredibly simple) 'system', we can combine them in a **State**.

```
coarse, fine = example_system(coarse, fine,
                              coarse_lvl, fine_lvl,
                              m)

# build state
S= State(coarse, fine)

# check the levels of the state contents
print S.levels
```

This object requires `coarse_lvl` and `fine_lvl` to be consecutive levels in the **GeneralisedMeshHierarchy** / **GeneralisedFunctionSpaceHierarchy**. Here, the coarse and fine inputs can be a **Function** or one can use input scalar quantities of interest from functions which have been computed on different levels of the **GeneralisedFunctionSpaceHierarchy** using **Constant**s.

```
# define the scalar quantities of interest
coarse = Constant(0, domain=m[coarse_lvl])
fine = Constant(0, domain=m[fine_lvl])

# build state
S= State(coarse, fine)
```

These **Constant**s contain their level information from the specified domain (**GeneralisedMeshHierarchy**) whereas **Function**s in a **State** exist on a **GeneralisedFunctionSpaceHierarchy**.

Building an Ensemble Hierarchy

Now that we can generate a **State** for every realisation in an ensemble corresponding to two consecutive levels in a hierarchy of discretizations, a hierarchy of ensembles need to be able to store these **State**s. Another object, a **EnsembleHierarchy**, is used to do exactly this. It stores the **State**s from each realisation on different levels so that statistics and the outputs of multilevel Monte Carlo can be computed. One can store a state in the **EnsembleHierarchy** using the following commands:

```
# initialize a hierarchy
hierarchy = EnsembleHierarchy(fs, state_type=Function)

# append state to hierarchy
hierarchy.AppendToEnsemble(S)
```

Given this logic, one can store N random realisations of a system on one level of the hierarchy via a simple loop. Continuing the earlier example,

Example 1

```
# initialize a hierarchy
hierarchy = EnsembleHierarchy(fs, state_type=Function)

# define coarse and fine level
coarse_lvl = 0; fine_lvl = 1

# define the coarse and fine functions
coarse = Function(fs[coarse_lvl])
fine = Function(fs[fine_lvl])

# store N random realisations of system in ensemble
# hierarchy
for i in range(N):
    coarse, fine = example_system(coarse, fine,
                                  coarse_lvl, fine_lvl,
                                  m)
    S = State(coarse, fine)
    hierarchy.AppendToEnsemble(S)
```

Once initialized, the **EnsembleHierarchy** object itself can be indexed to access the stored **State**s. The n'th state that is appended to the l'th ($l \in [0, L]$) level of the hierarchy will be indexed by `hierarchy[l][n]`.

```
# display the finer input of the n'th state on
# the coarsest level in the hierarchy
print hierarchy[0][n][1]
```

This object also carries a lot of useful information about the contents on itself, from the refinement factor of **Mesh**s (`hierarchy.M`) used in the hierarchy to the minimum cell edge lengths in each of those **Mesh**s (`hierarchy.dxl`).
For storage purposes, it may be benficial to keep the statistical information from an **EnsembleHierarchy** whilst clearing the actual stored **Function**s or **Constant**s, and one can do that via `hierarchy.ClearEnsemble()`. Using this directly however is not reccomended and should always be done as part of a statistics update (see next Chapter).

Computing Sample Statistics

The entire purpose of carrying out a multilevel Monte Carlo implementation is the computation of sample statistics, such as a mean of a random field. These statistics can be computed via this package and the **EnsembleHierarchy** generated in the previous Chapter. Via the command `hierarchy.UpdateStatistics()` can find sample mean and variance differences (between consecutive levels) on each level as well as the overall multilevel Monte Carlo estimator. The first index of the sample means and variances are simply the mean and variance of the finest level in the first pair of consecutive levels as in the standard multilevel Monte Carlo telescoping sum framework.

```
# update statistics
hierarchy.UpdateStatistics()

# display sample means on each level
print hierarchy.Mean

# display sample variance on each level
print hierarchy.Variance

# display multilevel monte carlo estimator
print hierarchy.MultilevelExpectation
```

Alongside updating the statistics of solutions to random systems using the **EnsembleHierarchy**, one can also clear the hierarchy at this point for storage purposes. This is much safer than using the method in the last Chapter as all statistics will be updated and stored online before clearing the hierarchy. One can do this via the following command:

```
# update statistics and clear ensemble
hierarchy.UpdateStatistics(clear_ensemble=True)
```

There is also a class **CumulativeDistributionFunction** that can find pointwise and field evaluations of the multilevel Monte Carlo approximation to the distribution function of a solution to a random discretization system.

```
# generate the cumulative distribution function object
CDF = CumulativeDistributionFunction(hierarchy)
```

To compute pointwise evaluations of the CDF for a spatial coordinate (x, y) at a **Numpy** array of points `yp`, use the following command:

```
import numpy as np

# define the spatial coordinate
coord = np.array([0.5, 0.5])

# evaluate CDF at points xp for spatial coordinate (x, y)
point_evaluations = CDF.Density(yp, coord)
```

To compute an evaluation of the CDF for the entire domain at a single field `y`, use the following command:

```
# evaluate CDF at field
field_evaluation = CDF.Compute(y)
```

Bounding Error

In the multilevel Monte Carlo framework, it is a common practice to reduce the computational cost of an uncertainty quantification simulation from the standard Monte Carlo approach by bounding the Mean Square Error of the estimator in question. For the estimator that is found via `hierarchy.MultilevelExpectation`, a simple estimator for the expectation of a field / scalar quantity of interest, one can express this Mean Square Error by a combination of the squared discretization bias and the variance of the estimator. For each component of this error to be bound by a user-defined tolerance $\frac{\epsilon^2}{2}$, one can use the following methods to find the optimal sample size on each level, and the finest level to bound both of these components. These are derived using the sample statistics of the **EnsembleHierarchy** via the analysis in [Gil08].

```
# find out the optimal number of samples on each level
# to bound variance of the multilevel monte
# carlo estimator by ((eps ** 2) / 2)
OptimalNl(hierarchy, epsilon)

# find out whether the bias of the multilevel
# monte carlo estimator is below (eps / sqrt(2))
Convergence(hierarchy, epsilon)
```

Plotting

Plotting results common-place in multilevel Monte Carlo literature is made simple with a **Plot** class. The results that can be plotted are the sample mean and variance differences for each level in the **EnsembleHierarchy** and the overall multilevel Monte Carlo estimator (this can only be done in one-dimensional cases).

```
# in one dimension, plot multilevel monte carlo
# estimator (produces Figure 1)
P = Plot(hierarchy)
P.plot_mlmc_estimator()
```

```
# plot the sample statistics (asymptotic decay
# in mean and variance of 'level' differences)
# (produces Figure 2)
P.plot_sample_stats()
```

An example of the output from both of the above plots for the system in Example 1 is now shown. The mlmc estimator provides a good approximation of the true mean of the system, which is simply $sin(2\pi x)$, $x \in [0, 1]$.

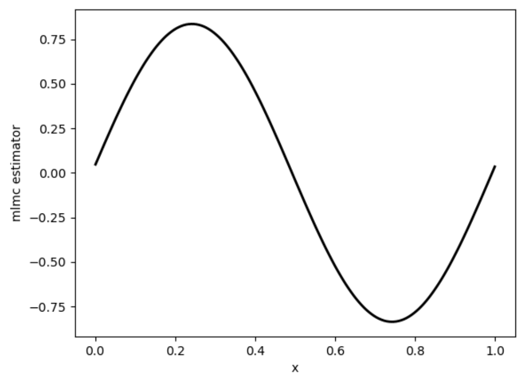

Figure 1: *Multilevel Monte Carlo estimator of the random system in Example 1.*

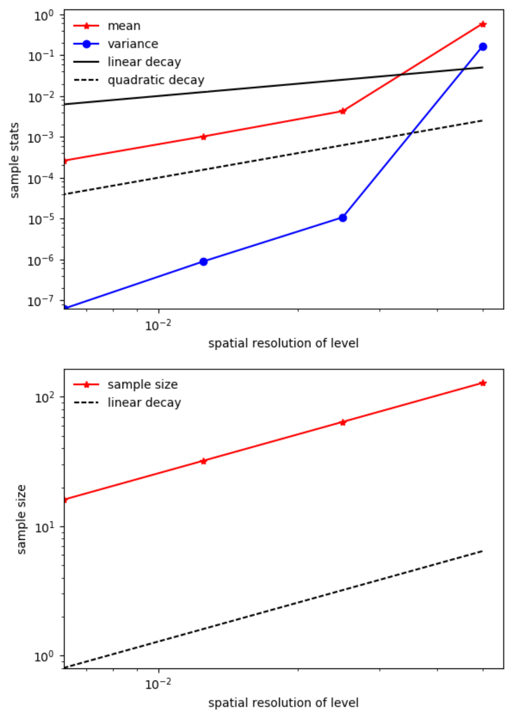

Figure 2: *Sample statistics and sample sizes of each level in the multilevel Monte Carlo estimator of the random system in Example 1.*

Bibliography

[Gil08] M. B. Giles. Multilevel Monte Carlo Path Simulation. *Oper. Res.*, 56(3):607–617, 2008.