Monte Carlo Methods for Uncertainty Quantification: Practical 2

This practical is all about the use of MLMC (multilevel Monte Carlo) for uncertainty quantification in various settings.

A number of exercises are suggested, but if you have particular application interests you may want to try your own application once you have a solid understanding of the way in which MLMC works.

1. Download from the course webpage the Matlab routines `mlmc_test.m`, `mlmc.m`, `gbm.m`, `ellip.m`, `para.m`, `hyp.m`.

   - `mlmc_test.m` performs a number of tests for an MLMC application.
   - `mlmc.m` does the main MLMC computation, working out the optimal number of samples to use on each level of approximation.
   - `gbm.m` is an application for a financial option based on an underyling stock represented by a Geometric Brownian Motion model.
   - `ellip.m` is a very simple 1D elliptic solver with random forcing.
   - `para.m` is a very simple 1D parabolic solver with random initial data.

2. Start with the `gbm.m` application. Look carefully at the routine `gbm_l` which computes N_ℓ fine path samples S^f and coarse path samples S^c, and the corresponding payoff functions P_ℓ^f and $P_{\ell-1}^c$, for a given level ℓ.

   Run the code and see the results it produces. Look at the code and see how the fine and coarse paths are computed; check that this matches the explanation given in the lectures.

   Note that in `gbm_l` the sample paths are computed in groups of 10,000. This is a trick to minimise the overheads in MATLAB. If programmed in C / C++ / FORTRAN you would typically do one path at a time.

   There are a number of ways in which you can modify this example:

   (a) Change the payoff function to the "digital" option:

   $$P = \exp(-r\,T) \times \begin{cases} 10, & S > K \\ 0, & S \le K \end{cases}$$

   which is currently commented out in the code.

   Re-run the example. What do you see? How does it affect the multilevel parameters α and β?

(b) Change the payoff function to a "lookback" call option based on the maximum path value:

$$P = \exp(-r\,T)\max(0, \max_{[0,T]} S(t) - K)$$

You can approximate the maximum by using the maximum of the values at the discrete timesteps.

(c) As written currently, the number of timesteps on level ℓ is 2^ℓ. Change this so that it is instead 4^ℓ.
Note: this changes the way in which the computational cost grows with level, so you will have to change the parameter M accordingly.

(d) The Euler discretisation is not very accurate. Modify the code to instead use the Milstein approximation:

$$S_{n+1} = S_n + r\,S_n\,\Delta t + \sigma S_n \Delta W_n + \tfrac{1}{2}\sigma^2 S_n(\Delta W_n^2 - \Delta t)$$

which has a first order strong error.

3. `ellip.m` solves the 1D elliptic PDE:

$$u''(x) = 100\,Z\sin(\pi x), \quad 0 < x < 1$$

where Z is a standard Normal random variable (zero mean and unit variance), and the boundary conditions are $u(0) = u(1) = 0$.

The output of interest is taken to be

$$P = \int_0^1 u^2(x)\;dx.$$

A simple finite difference approximation is used for the PDE, and the integral is approximated by trapezoidal integration.

Since the coefficients of the tri-diagonal matrix do not vary, the matrix is precomputed. This then allows us to compute samples 100 at a time to minimise the MATLAB overhead.

(a) Modify the code so that it is solving

$$(a\,u')' = 100, \quad 0 < x < 1$$

with $a(x) = \exp(-Z\sin(\pi x))$, where Z is again a standard Normal random variable, and the boundary conditions are still $u(0) = u(1) = 0$.
Note that in this case you will need to generate a separate matrix for each random sample, and so you will need to process all of the samples one by one.

(b) Modify the code so that the boundary conditions are $u(0) = 0$, $u(1) = Z_2$, where Z_2 is a second independent standard Normal random variable.

(c) Experiment with other combinations of uncertain forcing, coefficients and boundary conditions, using multiple independent random variables.

(d) Experiment also with different output functions.

4. `para.m` solves the 1D parabolic PDE:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad 0 < x < 1, \quad 0 < t < \tfrac{1}{4}$$

subject to boundary conditions $u(0,t) = u(1,t) = 0$, and random initial data

$$u(x,0) = 100\, Z \sin(\pi x).$$

The output of interest is taken to be

$$P = \int_0^1 u^2(x, \tfrac{1}{4})\ dx.$$

The numerical approximation uses simple explicit time-marching and central spatial finite differences. To maintain numerical stability, the timestep is proportional to the square of the grid spacing, so doubling the number of points in space increases the number of timesteps by factor 4, and hence increases the computational cost by factor 8. That is why M is set equal to 8 in the code.

(a) Modify the code to use implicit time-marching. You will then be able keep the timestep proportional to the grid spacing as you refine the grid.

(b) Using the implicit time-marching, modify the code again to solve

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(a(x)\frac{\partial u}{\partial x}\right), \quad 0 < x < 1, \quad 0 < t < \tfrac{1}{4}$$

where $a(x) = \exp(-Z \sin(\pi x))$. with Z being a standard Normal random variable.

(c) Experiment with various combinations of uncertain forcing, coefficients and boundary conditions, using multiple independent random variables, and also with different output functions.

5. `hyp.m` solves Burgers nonlinear hyperbolic PDE

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(\tfrac{1}{2}u^2) = 0, \quad 0 < x < 1$$

using explicit time-marching and central space differences.

It uses random initial conditions

$$u(x,0) = \begin{cases} U_1, & 0 < x \le 0.5 \\ -U_2, & 0.5 < x < 1, \end{cases}$$

where U_1, U_2 are two independent uniformly distributed random numbers in the interval $[0,1]$.

The boundary conditions are $u(0,t) = 1, u(1,t) = -1$, and the output is

$$P = 10 \int_0^1 u^2(x, \tfrac{1}{4})\ dx.$$

(a) Again experiment with various modifications, but you will need to be careful to maintain the numerical stability. That's why I switched to uniform random numbers in this example, so I could control the maximum and minimum values of $u(x,0)$ to ensure the timestep and smoothing were chosen in a way which would be stable.