Approaches to implementing Monte Carlo methods in MATLAB

Sri Krishnamurthy,CFA and Jorge Paloschi,PHD

MathWorks Consulting Services

Monte Carlo methods have long been used in computational finance to solve problems where analytical solutions are not feasible or are difficult to formulate. However, these methods are computationally intensive making it challenging to implement and adopt. In the last decade, advances in hardware, increasing processor speeds and decreasing costs have made it easier to adopt Monte Carlo methods to solve numerically intensive problems. With growing access to data and demand for quicker results, researchers are constantly looking for better ways to implement algorithms using Monte Carlo methods.

As consultants at MathWorks, we have seen many organizations use MATLAB to implement Monte Carlo solutions. Many times, MATLAB is used for rapid prototyping and when the application is ready for production, they ask us for advice on how to improve performance and scalability for their applications. In this article, we will share some of our observations and demonstrate various ways MATLAB could be used to implement Monte Carlo methods. We take a case study of pricing Asian options and show various approaches to implementing them in MATLAB.

Let us explore a sample Asian option pricing problem using Monte Carlo simulations. As discussed in [1], an Asian call option with arithmetic averaging has a payoff defined as:

$$max\left\{\frac{1}{N}\sum_{i=1}^{N} S(t_i) - K, 0\right\} \qquad (1)$$

where K is the strike price, N is the number of sampled points, and $S(t_i)$ is the asset price at time t_i. Our goal would be to simulate N sample points that describe the asset price dynamics and take the expected value to generate the option price for this scenario. In the Black-Scholes framework, the asset price dynamics for each scenario is driven by a geometric Brownian motion with:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

where S_t is the asset price at time t, μ is the riskfree compounded interest rate, σ is the instantaneous return volatilty, and W_t represents Brownian motion. From this, it can be derived that:

$$S_{t+\delta t} = S_t \exp(\vartheta \delta t + \sigma\sqrt{\delta t}\epsilon) \qquad (2)$$

$$\text{where } \vartheta = \mu - 0.5\sigma^2 \text{ and } \epsilon \sim N(0,1) \qquad (3)$$

We repeat this process M-times and take the expected value of the MX1 vector of generated option prices to be the Asian option price for this asset.

To help implement this, let's conceptualize using the grid shown in Fig 1. The x-axis represents the sampling points for each scenario and the y-axis represents the actual scenarios. Note each scenario is independent of the others in this case.

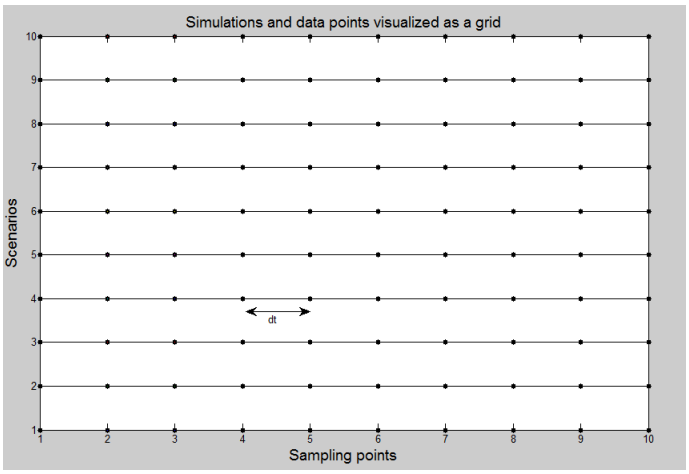

Fig 1: Visualizing the asset price generation as a grid

In order to implement this in MATLAB, let us consider various approaches. The examples illustrated are available for download at MATLAB central [3]

C/Java programmer's approach

Developers new to MATLAB, but familiar with C or Java, look for 'C or Java like syntax' in MATLAB to implement their algorithms. Example `PriceArithmeticAsianOption.m` illustrates this approach. From equations 2, 3, it is evident that we need to compute the asset prices for each step `NSteps` times and then use equation 1 to price the option for each scenario. The Statistics Toolbox provides various functions to generate random numbers and we will use the `randn` function to generate normal random numbers. We repeat this process `NPaths` times to generate `NPaths` option prices. The expected value of these `NPaths` option prices is the arithmetic Asian option price.

To see how well this algorithm performs, we run the MATLAB profiler tool which gives us two insights.

First, we haven't pre-allocated our variables though we know the size of the output beforehand. `The for` loops that incrementally grow can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve code execution time by pre-allocating the maximum amount of space required for the array ahead of time. The `mlint` messages in the MATLAB editor will highlight this in the code to warn the user.

Second, the structure of the loops indicates scope to vectorize our solution. MATLAB uses a matrix language, ideal for vector and matrix operations. You can often speed up your code by vectorizing algorithms that take advantage of this design.

MATLAB programmer's approach

Example `PriceArithmeticAsianOptionV.m` illustrates a MATLAB optimized approach to implement this algorithm. Note that we pre-allocated the variable `Path` and avoided loops using the

function `cumprod` to generate all asset prices in one line. Also note that the structure of this particular problem allows us to -benefit from vectorizing but this may not always be the case.

To see how well this algorithm performs, the profiler tool observes that, this approach takes just 0.22 seconds for 10,000 iterations with 150steps, which achieves a 215-X improvement compared to our prior approach.

Leveraging the Parallel Computing Toolbox

Monte Carlo simulation problems are typically well-suited to parallel computing. In our example, since each path is independent, we could take a task parallel approach i.e. run each task on a separate processor and improve performance further. The Parallel Computing Toolbox offers the ability to do exactly that. `PriceArithmeticAsianOptionPCT.m` illustrates how the Parallel Computing Toolbox uses the function `parfor`, the parallelized version of the traditional for loop. On our local test machine, we set up 4 "workers" and run the application. We scale up the number of trials to 1,000,000. We observe that with these changes, the application ran in 4.88 seconds, a 5-X performance improvement over the vectorized approach (with one worker) described earlier.

Fig 2 summarizes the performance of the three approaches described above.

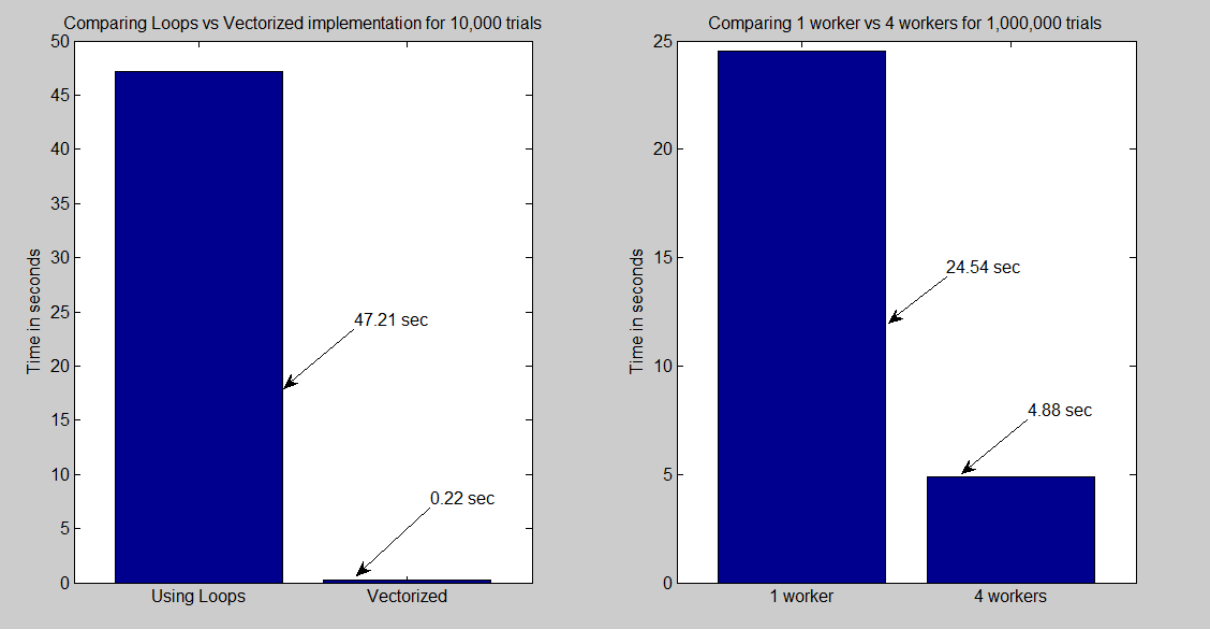

Fig 2: Performance on running the Arithmetic Average Asian Option pricing algorithm

Now, can we improve the performance of this algorithm further? We can. The Parallel Computing Toolbox can run as many as eight MATLAB workers on your local machine in addition to your MATLAB client session. Of course that also depends on how many cores/processors you have available to you on your machine. To scale up beyond eight MATLAB workers, you can easily run your applications in parallel across hundreds of computers in a cluster equipped with MATLAB Distributed Computer Server. From release 2010B, MATLAB GPU support is also available in the Parallel Computing Toolbox, allowing you to

take advantage of GPUs without low-level C programming. Furthermore, Monte Carlo problems are ideally suited to run on GPU architectures making it easier to structure and implement in MATLAB. We will address GPU computing in a future article.

Using Out-of-the box functionality in Toolboxes

The Asian Option pricing example we chose was for a single asset and we implemented the algorithm from scratch. At the heart of this algorithm was simulating Geometric Brownian motion which we implemented using multiple approaches. We could also use out-of-the box functionality. `PriceArithmeticAsianOptionFin.m` illustrates using the Financial Toolbox to simulate univariate geometric Brownian motion. The example uses the function `portsim` to simulate asset returns and the `ret2tick` function is used to derive asset prices. Though the example illustrates simulation of only one asset, it can be easily extended to simulate a portfolio. When simulating a portfolio, `portsim` simulates correlated asset returns taking the expected covariance matrix as a parameter.

A variety of models are commonly used to model asset prices with Monte Carlo methods .The Econometrics Toolbox provides a Stochastic Differential Equation (SDE) class structure that supports modeling using different structures such as GBM, CIR, HWV, Heston etc. The SDE class structure uses an object-oriented approach making it extensible and customizable. `PriceArithmeticAsianOptionSDE.m` illustrates how a `gbm` object can simulate asset prices. The example [Fig 3] illustrates the `simbyeuler` method which implements an Euler approach to approximate continuous-time stochastic processes.

In addition to providing different models to choose from and different methods for simulation, the SDE class library supports various variance reduction techniques. Variance reduction techniques, as the name suggests, reduce the variance of estimates thereby increasing the accuracy of estimation. [1] and [2] provide detailed introductions to variance reduction techniques used in Monte Carlo simulations. We discuss two methods that are commonly used in practice here.

Using antithetic variables reduces the average error in option pricing, When the `'Antithetic'` flag is set to `'true'`, the generated random numbers are used both for the primary and antithetic paths there by reducing the number of simulations required. `PriceArithmeticAsianOptionSDEAntithetic.m` illustrates using the `'Antithetic'` flag.

Another variance reduction method commonly used is stratified sampling. Stratified sampling is a variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample. `priceArithmeticAsianOptionSDEZ.m` illustrates the use of the `'z'` parameter to accomplish Stratified sampling. The `z` parameter allows you to specify a noise process directly, as a callable function of time and state or as an array of dependent random variates.

We noted earlier the use of `portsim` to simulate correlated asset returns when simulating a portfolio. We could also simulate a portfolio with correlated asset returns where the noise processes are Gaussian random draws using the `gbm` object and by specifying the correlation matrix using the

'correlation' parameter. Alternatively, noise processes driven by Gaussian and Student's t copulas can also be specified using the 'z' parameter when simulating portfolios. The Econometrics Toolbox has a demo which demonstrates all these use cases in an American Option pricing example setting [5].

The SDE framework in the Econometric toolbox eliminates the need to implement models from scratch. The availability of commonly used simulation models and the use of object oriented methodologies simplifies the task of the quantitative modeler, allowing him to focus on algorithm development rather than building artifacts from scratch.

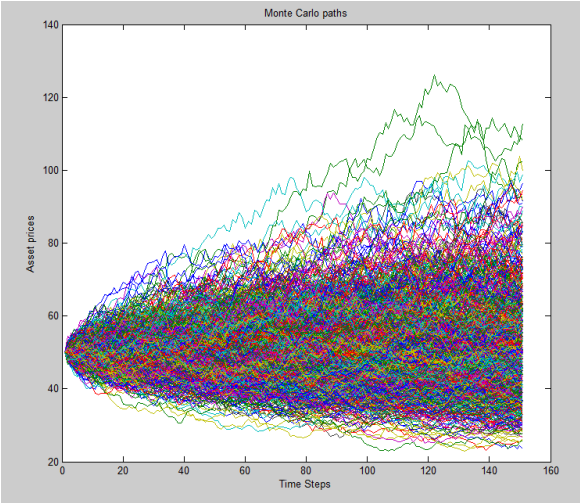

Fig 3: Simulated Monte Carlo paths

Further Optimizations using Quasi-Random Numbers

The methods we have discussed rely on random generation methods and to improve the accuracy of our computation, we have increased the number of paths and demonstrated how variance reduction methods can improve the efficiency of generating Monte Carlo paths. Recently, quasi-random number generation has become popular. Quasi-random number generators produce highly uniform samples of the unit hypercube. Quasi-random sequences are not random but seek to fill space uniformly and are supposed to perform better than generating pseudo random numbers. We refer the readers to [1] and [6] for additional coverage on these topics. In this section, we will discuss how the quasi-random sequences can be generated and used using functionality in the Statistics Toolbox.

The Statistics Toolbox supports three quasi-random sequence generators. `lhsdesign`, `sobolset`, and `haltonset`. `PriceArithmeticAsianOptionQuasi.m` illustrates use of all three generators.

In the Stratified Sampling example above, we demonstrated how we could achieve variance reduction by partitioning the hypercube into strata. Latin Hypercube Sampling (using `lhsdesign`) is another form of stratified sampling. The function `lhsdesign` produces a latin hypercube of sparse uniform samples containing `n` values on each of `p` variables. In the example, we demonstrate generating Npaths values for each of the `NSteps` variables. We then use the inverse transform method `norminv` to

obtain normal random variables used for generating Quasi-Monte Carlo Paths that are used for option pricing.

You could also use `sobolset` and `haltonset` classes for quasi-random number generation. `sobolset` and `haltonset` are built using the MATLAB object oriented technology and are extensions of the `qrandset` class. They generate the Sobol and Halton low-discrepancy sequences that are deterministic and uniformly cover the unit interval (0, 1). [1,6] provide detailed coverage of these topics. Because of how quasi-random sequences are generated, they may contain undesirable correlations, especially in their initial segments, and in higher dimensions. To address this issue, quasi-random point sets often *skip*, *leap* over, or *scramble* values in a sequence. The `Skip` and a `Leap` property can be set during the construction of the object and the `scramble` method can be invoked to apply the desired scrambling method. The `net` method is then invoked to generate the quasi-random sequence.

In our example `PriceArithmeticAsianOptionQuasi.m` we illustrate how an object of `NSteps` dimensions can be constructed. We skip the first 1000 numbers, use 100 as the Leap number and invoke the `scramble` method to scramble the quasi-random number set and then use the net method used to generate `NPaths`. Once the sequences are generated, we use the inverse transform method `norminv` to obtain normal random variables that are used for generating Quasi-Monte Carlo Paths that are used for option pricing.

Conclusion:

Our goal in this article has been to demonstrate various approaches to implementing Monte-Carlo methods using MATLAB. Using an Asian-Option pricing example, we show how MATLAB can be used to develop algorithms starting from scratch, and/or to leverage out-of-the box functionality in toolboxes. We demonstrate how simulation performance can be enhanced through following best practices and by using parallel computing. We also demonstrate how out-of-the box functionality in the Statistics Toolbox, Econometrics Toolbox and the Financial Toolbox enable easier development of algorithms. Quants and developers can choose their favored methodology and implement their algorithms optimally to build scalable applications.

Reference:

1. Numerical Methods in Finance and Economics: A MATLAB-Based Introduction by Paolo Brandimarte (J. Wiley & Sons, 2006)
2. Simulation and Optimization in Finance: Modeling with MATLAB, @RISK, or VBA by D. Pachamanova and F. Fabozzi (J. Wiley & Sons, 2010)
3. MATLAB Documentation http://www.mathworks.com/help/techdoc/
4. MATLAB central http://www.mathworks.com/matlabcentral/
5. Pricing American Basket Options by Monte Carlo Simulation demo in Econometrics Toolbox
6. Quasi-Monte Carlo: An Asian Bet by Piergiacamo Sabino in Implementing Models in Quantitative Finance: Methods and Cases (Springer Finance, 2008)