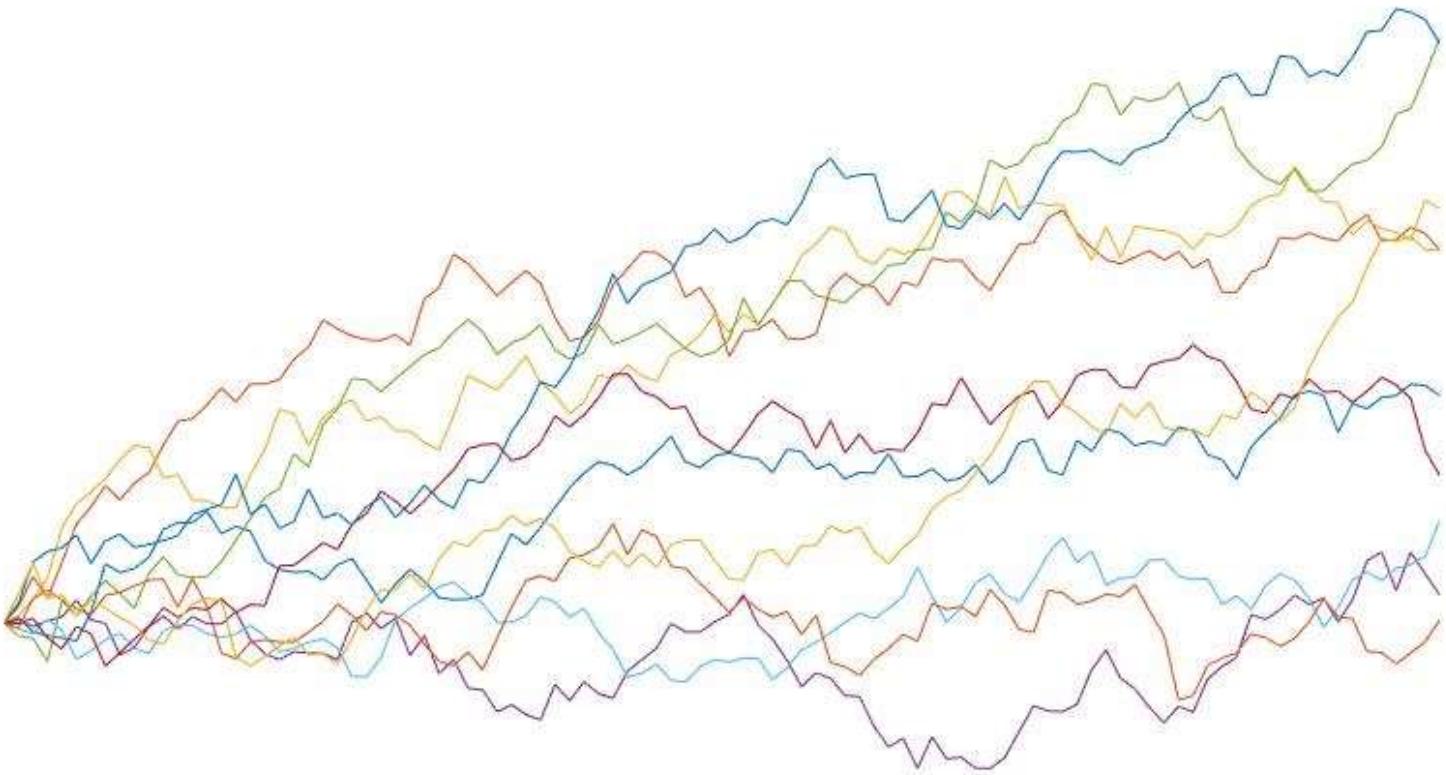
SHOW

Paws Development



European Vanilla Option Pricing with Monte Carlo in Python

November 22, 2016November 28, 2016

In mathematical finance stock prices are often assumed to follow a Wiener process or Brownian motion. Thus, by simulating these stochastic processes we can determine the price of financial instruments whose value depends on the current and future price of an underlying stock. In this post we are going to consider European vanilla options, these give the owner the right to buy (a call option) or sell (a put option) the underlying asset at an agreed price (the strike price) at some time in the future (the maturity).

The price of the underlying stock $S(t)$ is assumed to follow a stochastic differential equation (SDE) of the form

$$dS(t) = \mu S(t)dt + \sigma S(t)dW$$

where μ is the drift, σ denotes volatility and the dW is a Wiener process. We use Ito's Lemma to obtain the logarithm of $S(t)$ which ensures the stock price remains greater than or equal to zero

$$d\log S(t) = \left(\mu - \tfrac{1}{2}\sigma^2\right)dt + \sigma dW.$$

This is a constant coefficient SDE with solution

$$\log S(t) = \log S(0) + \left(\mu - \tfrac{1}{2}\sigma^2\right)t + \sigma \int_0^t dW.$$

The final term denotes a Wiener process with variance t and mean zero, and can therefore be rewritten as $N(0,t)$ or $\sqrt{t}N(0,1)$ where $N(0,1)$ is a normally distributed random variable with mean zero and unit variance. If we now make a risk-neutral assumption the drift μ becomes the risk-free interest rate r. Thus, we can determine the stock price at maturity T

$$S(T) = S(0) \times \exp\left[\left(r - \tfrac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}N(0,1)\right].$$

In a Monte Carlo simulation we generate a large number of stock price estimates using the above expression which we then use to estimate the option price. The option price is determined by calculating the expected value (denoted by $\mathbb{E}$) of some pay-off function f and then discounting by the increase in value due to the risk-free interest rate

$$e^{-rT}\mathbb{E}\left[f(S(t), K)\right].$$

where the pay-off function depends on the stock price and the strike price K. A call option is only exercised if the asset price at maturity is greater than the strike price, whereas a put option is only exercised if the asset price is less than the strike price, therefore, the pay-off function is defined as $\max(S(T) - K, 0)$ and $\max(K - S(T), 0)$ respectively.

Now we can begin implementing our pricing class in Python, initially we define a constructor which takes one argument class containing all the required parameters for the Monte Carlo simulation.

```
1  import math
2  import random
3
4  class EuropeanVanillaPricing:
5      def __init__(self, param):
6          self.pc = param.pc
7          self.S = param.S
8          self.K = param.K
9          self.T = param.T
10         self.r = param.r
11         self.sigma = param.sigma
12         self.iterations = param.iterations
```

Note: The number of approximations to generate for the underlying stock is denoted by the variable iterations and the variable `pc` takes values 'call' and 'put'.

The option price can then be calculated by following a simple procedure: 1) Generate a large number of approximations for the stock price at maturity. 2) Determine the average pay-off from the stock prices. 3) Finally we take the risk-free interest rate discount to obtain the option price.

The Monte Carlo pricing function using only built-in Python functions is given by:

```
def getMCPrice(self):
    total = 0.0
    mul = self.S*math.exp(self.T*(self.r - 0.5*self.sigma**2))
    for i in range(0, self.iterations):
        rand = random.gauss(0,1)
        if self.pc == 'call':
            total += max(mul*math.exp(math.sqrt(self.T*self.sigma**2)*rand) -
        elif self.pc == 'put':
            total += max(self.K - mul*math.exp(math.sqrt(self.T*self.sigma**2)

    return math.exp(-1.0*self.r*self.T)*(total/self.iterations)
```

The above function however performs poorly because of the for-loop used to generate pay-offs. We can achieve a significant speed-up by making use of the NumPy library which implements most vector operations in compiled code for performance. To begin we import the NumPy library:

```
import numpy as np
```

Then we construct a $N \times 2$ matrix of zeros where N is the number of iterations and generate a vector containing N normally distributed random numbers with mean zero and unit variance.

```
calc = np.zeros([self.iterations, 2])
rand = np.random.normal(0, 1, [1, self.iterations])
```

The second column of the $N \times 2$ matrix is then assigned to $S(T) - K$ or $K - S(T)$ for a call or put option respectively.

```
mult = self.S*np.exp(self.T*(self.r - 0.5*self.sigma*self.sigma))

if self.pc == 'call':
    calc[:,1] = mult*np.exp(np.sqrt((self.sigma**2)*self.T)*rand) - self.K
elif self.pc == 'put':
    calc[:,1] = self.K - mult*np.exp(np.sqrt((self.sigma**2)*self.T)*rand)
```

We now determine the maximum value in each row of the $N \times 2$ matrix using the NumPy `amax` command. The average pay-off is then calculated by summing together the returned vector and dividing by the number of iterations. The average pay-off is then multiplied by $\exp(-rT)$. Hence, the NumPy optimised pricing function takes the form:

```
    def getMCPrice(self):
        'Determine the option price using a Monte Carlo approach'
        calc = np.zeros([self.iterations, 2])
        rand = np.random.normal(0, 1, [1, self.iterations])
        mult = self.S*np.exp(self.T*(self.r - 0.5*self.sigma**2))

        if self.pc == 'call':
            calc[:,1] = mult*np.exp(np.sqrt((self.sigma**2)*self.T)*rand) - se
        elif self.pc == 'put':
            calc[:,1] = self.K - mult*np.exp(np.sqrt((self.sigma**2)*self.T)*r

        avg_po = np.sum(np.amax(calc, axis=1))/float(self.iterations)

        return np.exp(-1.0*self.r*self.T)*avg_po
```

To validate our method we also include the exact Black-Scholes solution for European option prices in our class (see `getBlackScholesPrice()`). Thus, the complete code for our pricing class is as follows:

```
import numpy as np
import math
import time

class EuropeanVanillaPricing:
    def __init__(self, param):
        self.method = param.method
        self.pc = param.pc
        self.S = param.S
        self.K = param.K
        self.T = param.T
        self.r = param.r
        self.sigma = param.sigma
        if self.method == 'mc':
            self.iterations = param.iterations

    def getPrice(self):
        if self.method == 'mc':
            return self.getMCPrice()
        elif self.method == 'exact':
            return self.getBlackScholesPrice()

    def getMCPrice(self):
        'Determine the option price using a Monte Carlo approach'
        calc = np.zeros([self.iterations, 2])
        rand = np.random.normal(0, 1, [1, self.iterations])
        mult = self.S*np.exp(self.T*(self.r - 0.5*self.sigma**2))

        if self.pc == 'call':
            calc[:,1] = mult*np.exp(np.sqrt((self.sigma**2)*self.T)*rand) - se
        elif self.pc == 'put':
            calc[:,1] = self.K - mult*np.exp(np.sqrt((self.sigma**2)*self.T)*r

        avg_po = np.sum(np.amax(calc, axis=1))/float(self.iterations)

        return np.exp(-1.0*self.r*self.T)*avg_po

    def getBlackScholesPrice(self):
```

```
        'Determine the option price using the exact Black-Scholes expression.'
        d1 =  np.log(self.S/self.K) + (self.r + 0.5*self.sigma**2)*self.T
        d1 /= self.sigma*np.sqrt(self.T)

        d2 = d1 - self.sigma*np.sqrt(self.T)

        call = self.S*self.ncdf(d1)
        call -= self.K*np.exp(-1.0*self.r*self.T)*self.ncdf(d2)

        if self.pc == 'call':
            return call
        elif self.pc == 'put':
            return self.applyPCParity(call)

    def ncdf(self, x):
        'Cumulative distribution function for the standard normal distributior
        return (1.0 + math.erf(x / math.sqrt(2.0))) / 2.0

    def applyPCParity(self, call):
        'Make use of put-call parity to determine put price.'
        return self.K*np.exp(-1.0*self.r*self.T) - self.S + call
```

To test our implementation (vanilla.py) we use the following parameters:

- Risk-free interest rate $r = 0.05$
- Maturity $T = 1.0$
- Strike price $K = 100.0$
- Volatility $\sigma = 0.15$
- Initial stock price $S(0) = 90.0$

Using these parameters we run the following Python code:

```
from vanilla import *
import time

class Parameters:
  pass

testParam = Parameters()
testParam.T = 1.0
testParam.S = 90.0
testParam.K = 100.0
testParam.sigma = 0.15
testParam.r = 0.05;
testParam.method = 'mc'
testParam.pc = 'call'
testParam.iterations = 1000000

option = EuropeanVanillaPricing(testParam)
print 'Method: Monte Carlo'
t0 = time.time()
print 'Price: '+ str(option.getPrice())
t1 = time.time()
print 'Iterations: ' + str(testParam.iterations)
print 'Time Taken: ' + str(t1-t0) + 's'
```

This code produces the output:

```
Method: Monte Carlo
Price: 3.34867238038
Iterations: 1000000
Time Taken: 0.212198019028s
```

A call option with the above parameters has price 3.344 (to 3 d.p.) determined using the exact Black-Scholes expression (where `method = 'exact'`). The above number of iterations produces a solution which is approximately accurate to three significant figures. To test the performance of our pricing class we compare it with the initial implementation without NumPy which is found to be 8 times slower on the same machine with the above parameters.

In the next post in this series we will use CUDA and C++ to efficiently estimate prices for exotic options.

Published by Dipesh Amin

Mathematician & Developer View all posts by Dipesh Amin