Assignment 2: Option Pricing and the Black-Scholes formula

The University of British Columbia
Science One CS 2015-2016
Instructor: Michael Gelbart

Due Thursday, November 12th at 11:59pm

Last updated October 23rd at 3:20 pm

Overview

In this assignment you will simulate the price of a stock changing over time, and estimate the value of a stock option. You will then compare your estimate with the famous Black-Scholes formula.

Learning Goals

The skills/concepts you will practice in this assignment are:

- arrays
- functions
- the `numpy` library
- plotting with the `matplotlib` library
- performing a Monte Carlo simulation with (pseudo-)random numbers.

Background

Stock options. A *stock option* or *call option* is a promise that you will be allowed (but not obligated) to buy a certain stock for a certain price at a certain time. For example, I might have a stock option for 1 Google share with a *strike price* of \$500 and an *expiration date* of one year from now. This means that, if I want to, I have the option to buy 1 Google share for \$500 in one year.

So, under what circumstances would I want to do that? Well, if the Google stock price one year from now is \$400, then I could just buy a share for \$400 on the market; I certainly would not prefer to exercise my option to pay \$500 when I could be paying \$400 for the same thing! Therefore, we say that the option turned out to be worthless (zero value) if the share price is below the strike price at the expiration date of the option.

On the other hand, if the Google share price turned out to be \$600 one year from now, then I'm in business: by exercising my option to buy a share for \$500, and then immediately selling it at the market price of \$600, I have made a profit of \$100. Thus, we say the option value turned out to be \$100. A representation of what we just said in mathematical symbols is that, at the expiration time T, the option price P_T is a function of the strike price X and the stock price at the expiration date S_T given by

$$P_T = \max(0, S_T - X)\,. \tag{1}$$

Option pricing. An important question that arises is, what is the fair price of an option at the time it is issued ($t = 0$), *before* we know what is going to happen to the stock price? (This initial option price is not to be confused with the value of the option once the expiration date is reached, which was discussed above; I will refer to these as P_0 and P_T respectively.) We can begin to answer this question by thinking about what variables the option price might depend on. Some reasonable candidates are

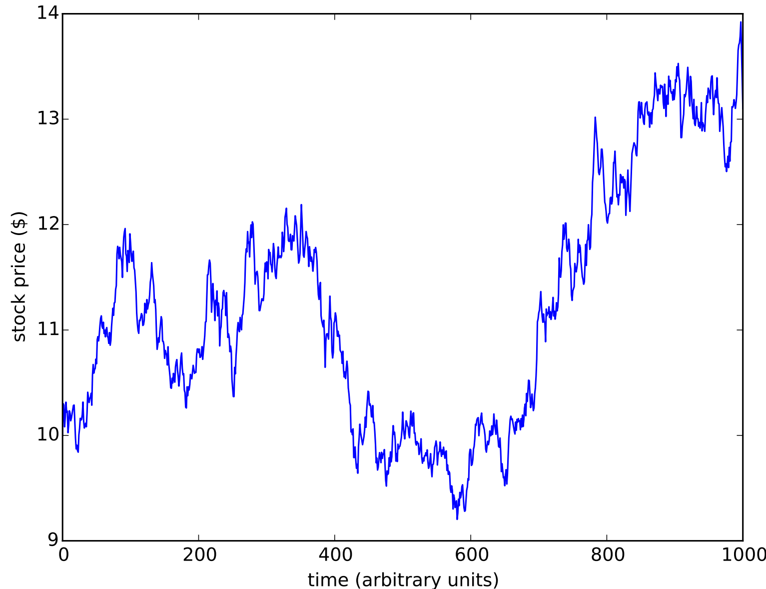

Figure 1: Simulating a stock price using geometric Brownian motion with $S_0 = \$10$, $T = 1000$, and $\sigma = 0.01$. In this particular run of the simulation, S_T is just under \$12.

the initial (current) stock price, which we will call S_0, the strike price X, and expiration time T. One other parameter is relevant here, namely the *volatility* of the stock, which we denote as σ. The volatility is analogous to the step size in a random walk: a larger volatility means more movement in a given amount of time.

The Monte Carlo approach. Although the problem of option pricing has been addressed theoretically, we will tackle it by *simulating* the stock price as it changes over time, and then computing the option value P_T with eq. (1). Sometimes the simulation will yield a worthless option, and other times it will be worth something. By repeating the simulation many times and taking the average, we can compute an "expected" or average value of the option, which we will take to be its fair price (P_0). This general approach of simulating a random process in order to understand its characteristics is called *Monte Carlo* (after the famous Monte Carlo casino, where there is randomness aplenty).

Geometric Brownian motion. Our particular approach for simulating the stock price over time will be to assume the stock price follows a *geometric Brownian motion*. In Assignment 1 you simulated a random walk which is essentially Brownian motion: at each time step you add a small random amount to your current position. In geometric Brownian motion, at each time step you *multiply* your current state by a (positive) random amount. Note that in regular Brownian motion the position can be both positive or negative, but in geometric Brownian motion the position is always positive; this is good because stock prices are never negative. (If this helps, you can also think of geometric Brownian motion as an exponentiated Brownian motion.) An example is shown in fig. 1.

The update rule for our stock price model is as follows:

$$S_t = S_{t-1} \exp\left(-0.5\sigma^2 + \sigma Z\right) , \tag{2}$$

where Z is a random number drawn from a Gaussian distribution (also called a Normal distribution or a "bell curve"). You can generate these random draws with `numpy.random.randn()` and you can

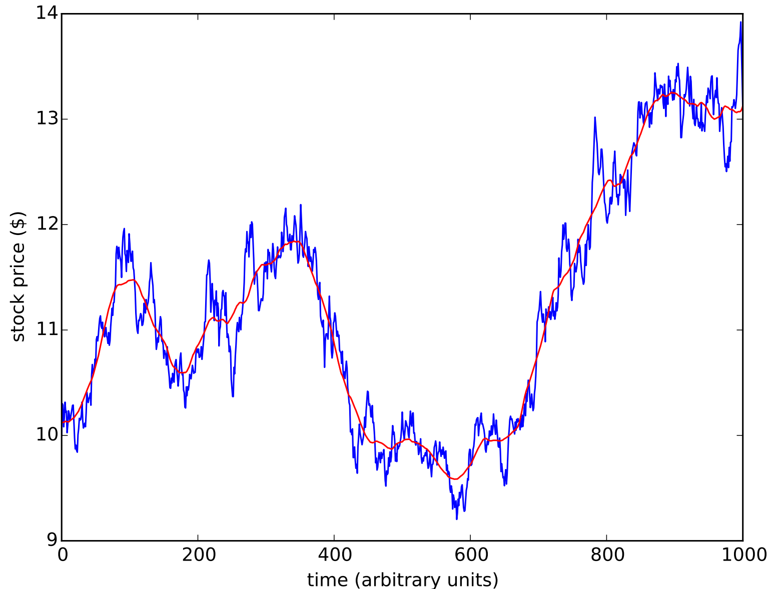

Figure 2: The same simulation as in Figure 1 (blue), but with an overlay (red) showing a smoothed version generated using a moving average window with $W = 25$.

perform most of the math operations with `numpy` as well; for example, `numpy.exp(x)` computes $\exp(x)$, etc. Note that if $\sigma = 0$ in eq. (2), then the stock price never changes, whereas if σ is large the stock price can change wildly; this matches our intuition that σ represents the volatility of the stock. Note: the mathematical notation $\exp(x)$ means e^x.

Note: what we are doing is similar to numerically solving a differential equation (in fact, we are numerically solving something called a *stochastic differential equation*). When we numerically solve differential equations there is usually a time step Δt, but to keep things simpler I have scaled the units of time so that $\Delta t = 1$. Also for the sake of simplicity, I have neglected the interest rate, which usually plays an important role in financial models.

Smoothing a curve. Figure 1 shows the stock price at every time step, and is therefore very "bumpy". Investors are often more interested in a *smoothed* depiction of the stock price over time, in which small ups and downs are removed, leaving only the salient trends. Figure 2 shows a smoothed version of fig. 1. The smoothing is achieved by replacing the price at time t with an average of all the prices from times $t - W$ to $t + W$ (inclusive), for some positive integer W (thus, the total width of the window is $2W + 1$). Near the edges of the plot, where we do not have enough data for the entire window, the average computed only over the portion of the window that is within the data range. For example, if $T = 1000$ and $W = 5$ then the smoothed curve at $t = 0$ is the average of the prices from $t = 0$ to $t = 5$, the smoothed curve at $t = 1$ is the average of the prices from $t = 0$ to $t = 5$, the smoothed curve at $t = 10$ is the average of the prices from $t = 5$ to $t = 15$, and the smoothed curve at $t = 997$ is the average of the prices from $t = 992$ to $t = 1000$.

The Black-Scholes formula. The famous Black-Scholes formula was derived by a theoretical analysis of the geometric Brownian motion model discussed above. It turns out that (in our oversimplified

case of zero interest rate) the expected option price, P_0, is given by:

$$P_0 = \Phi(d_1)S_0 - \Phi(d_2)X\,, \tag{3}$$

where

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left[\log\left(\frac{S_0}{X}\right) + \frac{\sigma^2}{2}T\right],$$
$$d_2 = \frac{1}{\sigma\sqrt{T}}\left[\log\left(\frac{S_0}{X}\right) - \frac{\sigma^2}{2}T\right],$$

and $\Phi(\cdot)$ is the cumulative distribution function of the Gaussian distribution, which can be computed with `scipy.stats.norm.cdf(x)` (after importing the module with `import scipy.stats`). Note: unless otherwise specified, log refers to the base-e logarithm, also known as the natural logarithm and also sometimes written as $\ln(x)$.

Your task

Write a program `OptionPrice.py` that takes in six command-line arguments: the initial stock price S_0, the volatility σ, the number of time steps T, the option's strike price X, the smoothing window size W, and the number of repeated experiments N. Your program should contain the following functions:

1. A function `simulate` that takes as arguments S_0, σ, and T. This function should simulate the geometric Brownian motion for T steps using eq. (2) and then return an array of size $T+1$ containing the stock price at each time step.
2. A function `smooth` that takes in an array of values and the window width W. This function should smooth the input array using the sliding window averaging scheme described above with window width W, and then return the smoothed result. The returned array should be the same size as the input array; you can access the length of an array with `len(array)`.
3. A function `blackScholes` that takes as arguments S_0, X, σ, and T and returns the Black-Scholes option price given by eq. (3).

Making use of the functions described above, your program should perform the following tasks:

I. Simulate the stock price and plot the results (S_t vs. t) using `matplotlib`. Make sure to include the axis labels as in figs. 1 and 2; you can set these with `plt.xlabel` and `plt.ylabel` (assuming you defined `plt` by typing `import matplotlib.pyplot as plt` at the top of your file.
IMPORTANT NOTE: now that we have learned about arrays, you can use `plt.plot(x,y)` where `x` and `y` are arrays; this means you do not need to create your plot one point at a time as you did earlier(!!!), but rather you can create the entire curve with one call to `plt.plot`. You also no longer want the `'.b'` because you are no longer plotting individual dots. You can still include `'b'` if you want, but it will not do anything since blue is the default colour for plotting.

II. Produce a smoothed version of the simulated results using the sliding window averaging scheme described above and window width W. Plot this on top of your original plot in red, using, for example, `plt.plot(x,smooth,'r')`. Save the plot to the file `stockprice.pdf` using the command `plt.savefig('stockprice.pdf')`.
IMPORTANT NOTE: You do **not** need to generate a figure like fig. 1, just one like fig. 2.

III. Call the `simulate` function N times and compute the option value P_T in each case; see eq. (1). Print out the option value averaged over the N trials as an estimate of P_0.

IV. Print the Black-Scholes option price, which is the theoretical or "correct" value of P_0.

Sample Output

An example run would produce a plot like Figure 2 (saved to the file `stockprice.pdf`) and, in addition, produce output like this:

```
>> python OptionPrice.py 10.0 0.01 1000 10.0 25 1000
Estimated option price:     $1.280859
Black-Scholes option price: $1.256329

>> python OptionPrice.py 10.0 0.01 1000 20.0 25 1000
Estimated option price:     $0.032621
Black-Scholes option price: $0.022139

>> python OptionPrice.py 10.0 0.01 1000 100.0 25 1000
Estimated option price:     $0.000000
Black-Scholes option price: $0.000000
```

Testing Hints

Testing large programs can be very difficult. The best way to test your code is to test each function individually. I recommend crafting some test cases that are simple enough for you to figure out what the output should be by hand. For example, you could test your `smooth` function by passing it an array of all zeros and an arbitrary W. Since the average of a bunch of zeros is just zero, you expect the smoothed version to be the same as the input (all zeros) regardless of W. Then, if you run your `smooth` function on and array of all zeros and you get back something other than an array of all zeros, then you know you have a bug and you can focus on finding it. (But, if you get the right result, that doesn't prove that your code is correct!) Another good example would be an arbitrary input array and $W = 0$ ... what should this return?

The difficulty of testing your code is increased further because it involves randomness and therefore gives different results every time it is executed. One sanity check is that your estimated option price should approach the Black-Scholes price as N becomes large (this means they will become close to each other, as in the examples above; you should not expect an exact match).

Submission

Following the instructions on the course webiste, submit your program `OptionPrice.py`, along with the plot `stockprice.pdf` produced when running your program with $S_0 = 10$, $\sigma = 0.01$, $T = 1000$, and $W = 25$.

Grading

In this assignment you will be graded both on whether your program works and also on your code itself. In order to get full marks, you need to meet the following criteria:

1. Your program prints the correct output to the screen, formatted exactly as shown above.
2. Your program produces the file `stockprice.pdf` in the current directory (i.e., the same directory as your program), depicting the stock price with proper axis labeling as in fig. 1.
3. Your program uses well-chosen variable names that describe the variable clearly and concisely. For the volatility, examples of reasonable names are `sigma` or `volatility`, whereas examples of poorly chosen names are `x` or `THE_VOLATILITY_OF_THE_STOCK_PRICE`.

4. All `import` statements appear at the top of your program, just below your initial comment.
5. Your program contains the functions `simulate`, `smooth`, and `blackScholes`, and:
   (a) These functions conform exactly to the interfaces described above.
   (b) These functions are located at the top of your file, just below your `import` statements but above the rest of your code.
   (c) Each of these three functions is preceded with a comment describing the inputs and outputs of the function (variable names, variable types, descriptions), as well as a brief explanation of what the function does. As an example, here is an acceptable comment for the `simulate` function (which you are free to copy if you wish):

   ```
   # This function simulates geometric Brownian motion.
   # Inputs: S_0   (float), the initial stock price
   #         sigma (float), the stock volatility
   #         T     (int),   the number of time steps to simulate
   # Output: the simulated stock price over time (array of size T+1)
   ```

   Note that I specified the names of the inputs but not the output. This is because only the names of the inputs are specified when you define a function with `def`. However, I still specified the type of the output.

Sample Syntax

To help you remember the Python syntax for functions, arrays, and loops, here is an example function:

```
# This function counts the number of positive elements in an array
# Inputs  : a   (array), the input array
# Outputs : num (int)  , the number of positive elements
def count_positive(a):
   num = 0
   for a_i in a:
      if a_i > 0:
         num = num + 1
   return num
```

Here is the same function written with a `while` loop instead of a `for` loop:

```
# This function counts the number of positive elements in an array
# Inputs  : a   (array), the input array
# Outputs : num (int)  , the number of positive elements
def count_positive(a):
   num = 0
   i = 0
   while i < len(a):
      if a[i] > 0:
         num = num + 1
      i = i + 1
   return num
```

In general, anything you can do with a `for` loop can also be done with a `while` loop.