Question 1)b)

```
# importing the necessary packages
import numpy as np
from numpy import sqrt, exp, log
from scipy.stats import norm

##############################################################################

# variables given
S_0 = 100.
B0 = 1
mu = 0.03
sigma = 0.25
r = 0.01

# European Call, using Black Scholes formula
K = 120.
T = 1

d1 = (log(S_0/K) + (r + (sigma**2)/2)*T) / (sigma * sqrt(T))
d2 = d1 - sigma*sqrt(T)

Price1 = S_0*norm.cdf(d1) - K*exp(-r*T)*norm.cdf(d2)

print("European Call option price (using Black Scholes formula) is:")
print(round(Price1,2))

##############################################################################
# question 1)

# defining the stock generation function
def generateStock(S_0, r, sigma, T):
    return S_0 * exp((r - (sigma**2)/2)*T + sigma * sqrt(T) * np.random.standard_normal())

# defining payoff function, for this case it's regular European put
def payoff1(S, K):
    return max(S - K, 0)

# defining monte carlo integration formula, to make graphing easier
def MonteCarlo1(runs):
    # initialize array that will have payoffs of option
    payoffs = np.zeros(0)

    # looping through
    for i in xrange(runs):
        # generate future stock
        S_T = generateStock(S_0, r, sigma, T)

        # append to the payoffs list whatever the payoff is
        payoffs = np.append(payoffs,
                            exp(-r*T)*payoff1(S_T, K))

    return sum(payoffs)/runs, payoffs

# setting number of simulation runs
runs = 10000
```

```python

 # initialize array that will have payoffs of option
payoffs = np.zeros(0)

# monte carlo calculation of option price
Price2, payoffs = MonteCarlo1(runs)

var_sample = 0
payoffs_int = payoffs.astype(int)
# setting up iterating array
it = np.nditer(payoffs, flags = ['f_index'])

while not it.finished:
    var_sample = var_sample + (payoffs[it.index] - Price2)**2
    it.iternext()
var_sample = var_sample / (runs - 1)


# from analytical calculation, we know sigma = Var(X1) = 657.973

# 95% confidence, z-score = 1.96
size_estimate = (1.96**2 * 657.973) / 0.05**2

print("European Call option price (using Monte Carlo integration) is: ")
print(round(Price2,2))
print("Approximation of how many samples needed for 95% confidence (up to dime):")
print(int(size_estimate))

# running Monte Carlo calculation 100 times with the above size estimate
# counters for counting number of times the Monte Carlo estimated price
# correct up to the dime or not
correct = 0
incorrect = 0
for j in xrange(100):
    price_to_compare, dummypayoff = MonteCarlo1(int(size_estimate))

    if price_to_compare > Price1 - 0.05 and price_to_compare < Price1 + 0.05:
        correct += 1
    else:
        incorrect += 1

print("Number of times Monte Carlo estimation was within a dime:")
print(correct)
print("Number of times Monte Carlo estimation was NOT within a dime:")
print(incorrect)
```

Output:

European Call option price (using Black Scholes formula) is:
3.95
European Call option price (using Monte Carlo integration) is:
3.91
Approximation of how many samples needed for 95% confidence (up to dime):
1011067

Question 2) b)

```
# importing the necessary packages
import numpy as np

##############################################################################

alpha = 0.6
#lamba = 1

losses = np.zeros(0)

# loss random variable equation
def loss(y):
    return 100000*y - 220000

n = 0
while n < 100000:
    x = loss(np.random.weibull(alpha))
    losses = np.append(losses, x)
    n += 1

# sort losses array from largest to lowest
losses = np.sort(losses)
losses[:] = losses[::-1]

# select the loss (95% value at risk) that is the specified one
# 100000*0.05 = 5000

print("Numerical estimation of 95% Value At Risk is:")
print(losses[5000])
```

Output:

Numerical estimation of 95% Value At Risk is:
401456.414615

Question 3)c) and d)

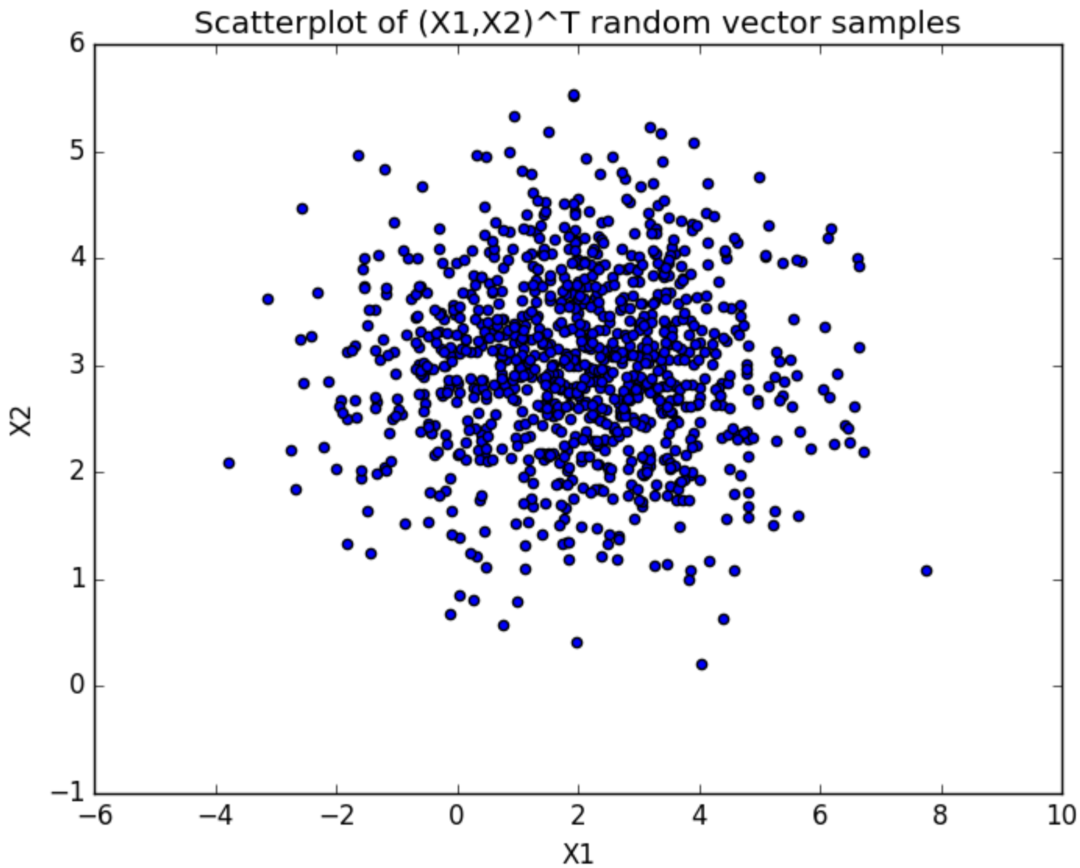
```
# importing the necessary packages
import numpy as np
from numpy import sqrt
import matplotlib.pyplot as plt

##############################################################################

# initial variables
covariance = np.zeros([2,2])
sigma1 = 3.2
sigma2 = 0.7
corr = 0.6
mu1 = 2
mu2 = 3

# setting up array for actual covariance calculation
array1 = np.array([[sqrt(sigma1), 0],
                   [0, sqrt(sigma2)]])
array2 = np.array([[1, corr],
                   [corr, 1]])

covariance = (array1.dot(array2)).dot(array1)
print("The covariance matrix is: ")
print(covariance)

##############################################################################

# hand-written result of cholesky factorization of A
# A = [[1.78885, 0]
#       [0.501996, 0.669328]]

# cholesky factorization, from built-in default function
A_actual = np.linalg.cholesky(covariance)
print("The matrix A, based on built-in cholesky function np.linalg.cholesky(): ")
print(A_actual)

##############################################################################

# cholesky factorization by python algorithm coding

# Note, need to do A_actual.tolist() to make it python list type

def cholesky(A):
    # establish output S matrix
    n = len(A)

    # setup output array
    S = [[0.0] * n for i in xrange(n)]

    # setting up calculation of individual elements for loop
    for i in xrange(n):
        for j in xrange(i + 1):
            # sigma_ij formula from class
            temp = sum(S[i][k] * S[j][k] for k in xrange(j))

            # for the diagnoal case
            if i == j:
                S[i][j]= sqrt(A[i][i] - temp)
```

```python
            # for regular case
            else:
                S[i][j] = (A[i][j] - temp) / S[j][j]

    return S

A_cholesky = np.array(cholesky(covariance))

print("The matrix A, based on python code algorithm function: ")
print(A_cholesky)

##############################################################################

# scatterplot work

plt.clf()

num = 0

# producing 1000 samples
while num < 1000:

    X1 = mu1 + A_cholesky[0,0]*np.random.standard_normal()
    X2 = mu2 + A_cholesky[1,0]*np.random.standard_normal() + A_cholesky[1,1]*np.random.standard_normal()

    plt.scatter(X1, X2)
    num += 1

plt.title("Scatterplot of (X1,X2)^T random vector samples")
plt.xlabel("X1")
plt.ylabel("X2")

plt.show()
```

Output:

```
The covariance matrix is:
[[ 3.2         0.89799777]
 [ 0.89799777  0.7       ]]
The matrix A, based on built-in cholesky function np.linalg.cholesky():
[[ 1.78885438  0.        ]
 [ 0.50199602  0.66932802]]
The matrix A, based on python code algorithm function:
[[ 1.78885438  0.        ]
 [ 0.50199602  0.66932802]]
```

Figure:


Scatterplot of (X1,X2)^T random vector samples
6
5
4
3
2
1
0
−1
X2
−6
−4
−2
0
2
4
6
8
10
X1

Question 4)

```python
# importing the necessary packages
import numpy as np

##############################################################################

# initial variables
mu = np.array([[-1000],
                      [-700],
                      [300],
                      [-200]])

covariance = np.array([[144, 72, 120, 300],
                      [72, 100, 180, 230],
                      [120, 180, 389, 880],
                      [300, 230, 880, 4469]])

weights = np.array([0.4, 0.2, 0.3, 0.1])

# setup L losses simulation
L = np.zeros(0)

# get the matrix A thanks to built-in cholesky function
A_cholesky = np.linalg.cholesky(covariance)

# simulation loop for 10,000 samples
num = 0
while num < 10000:

    # this X matrix will hold our Assets
    X = np.zeros_like(mu)

    # creating standard normal random samples matrix
    # each element is a different standard normal random sample
    Z = np.zeros_like(A_cholesky)
    for x in np.nditer(Z, op_flags = ['readwrite']):
        x[...] = np.random.standard_normal()


    # loop through to get assets vector
    for i in xrange(X.size):
        X[i] = mu[i] + sum(A_cholesky[i]*Z[i])

    # loss based on calculation
    L = np.append(L, weights.dot(X))

    num += 1

# sort L array, from largest to smallest
L = np.sort(L)
L[:] = L[::-1]

# we want 99% Value At Risk. So pick the 10000*0.01 = 100th loss element

print("Numerical estimation of 99% Value At Risk is: ")
print(L[100])
```

Output: Numerical estimation of 99% Value At Risk is: -445.8

Question 5)
Q5_scaling.py

```
# importing the necessary packages
import numpy as np
from numpy import sqrt
import matplotlib.pyplot as plt
import timeit

##############################################################################

# starting timer
start = timeit.default_timer()

# increment step method
def SimBMStep(T, N):
    # initialize W brownian motion array
    W = np.zeros(N)
    W = np.append(W, 0)    # initial 0 value

    # loop
    num = 1
    while num <= N:
        W[num] = W[num - 1] + sqrt(T/N)*np.random.standard_normal()
        num += 1


    return W

###################################

plt.clf()

plt.subplot(2,2,1)

# setting up t x-axis variable
t = np.arange(0.0, 11.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMStep(1., 10)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 10)")

ticks = np.arange(t.min(), t.max() + 1, 10)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,2)

# setting up t x-axis variable
t = np.arange(0.0, 101.0, 1.0)
```

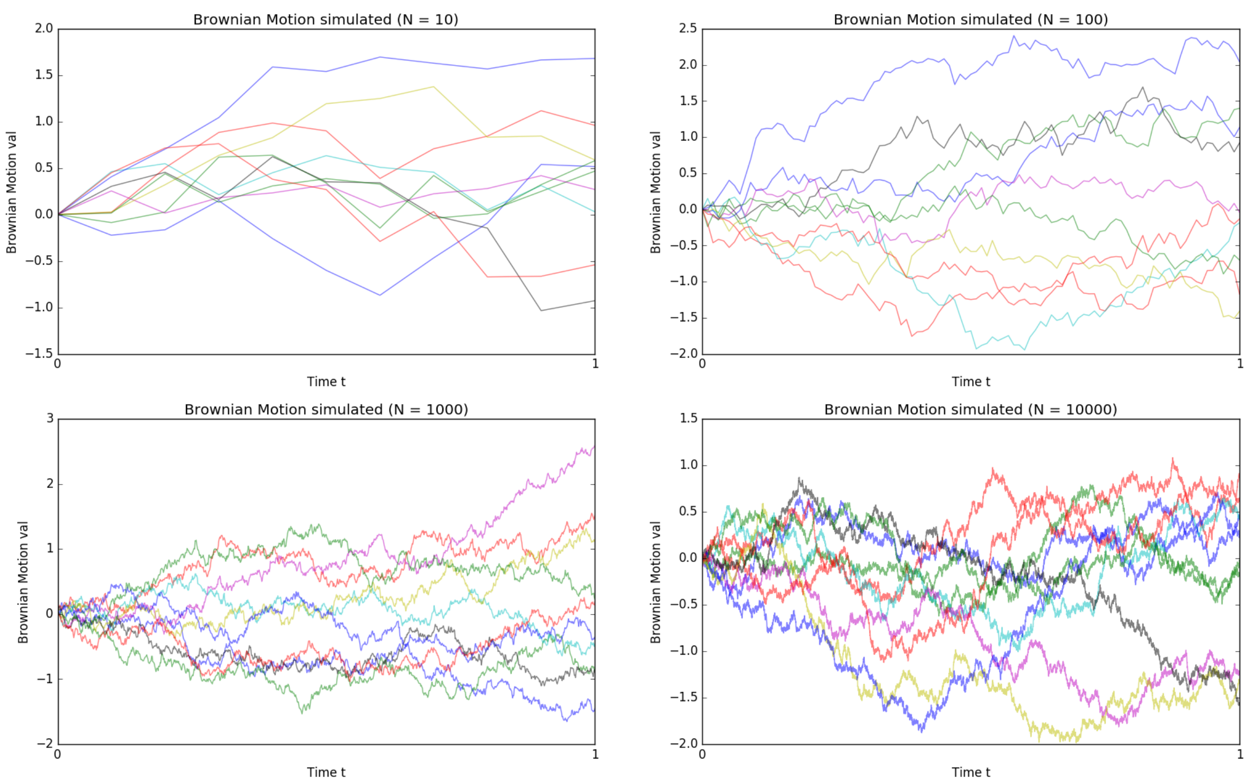
```python

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMStep(1., 100)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 100)")

ticks = np.arange(t.min(), t.max() + 1, 100)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,3)

# setting up t x-axis variable
t = np.arange(0.0, 1001.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMStep(1., 1000)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 1000)")

ticks = np.arange(t.min(), t.max() + 1, 1000)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,4)

# setting up t x-axis variable
t = np.arange(0.0, 10001.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMStep(1., 10000)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 10000)")

ticks = np.arange(t.min(), t.max() +  1, 10000)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")
```

```python

        plt.show()

        # stopping timer
        stop = timeit.default_timer()

        print "Step-size increment method runtime = " , stop - start, " seconds"
```

Output:

Step-size increment method runtime = 1.27352404594 seconds

Figure:



Q5_cholesky.py

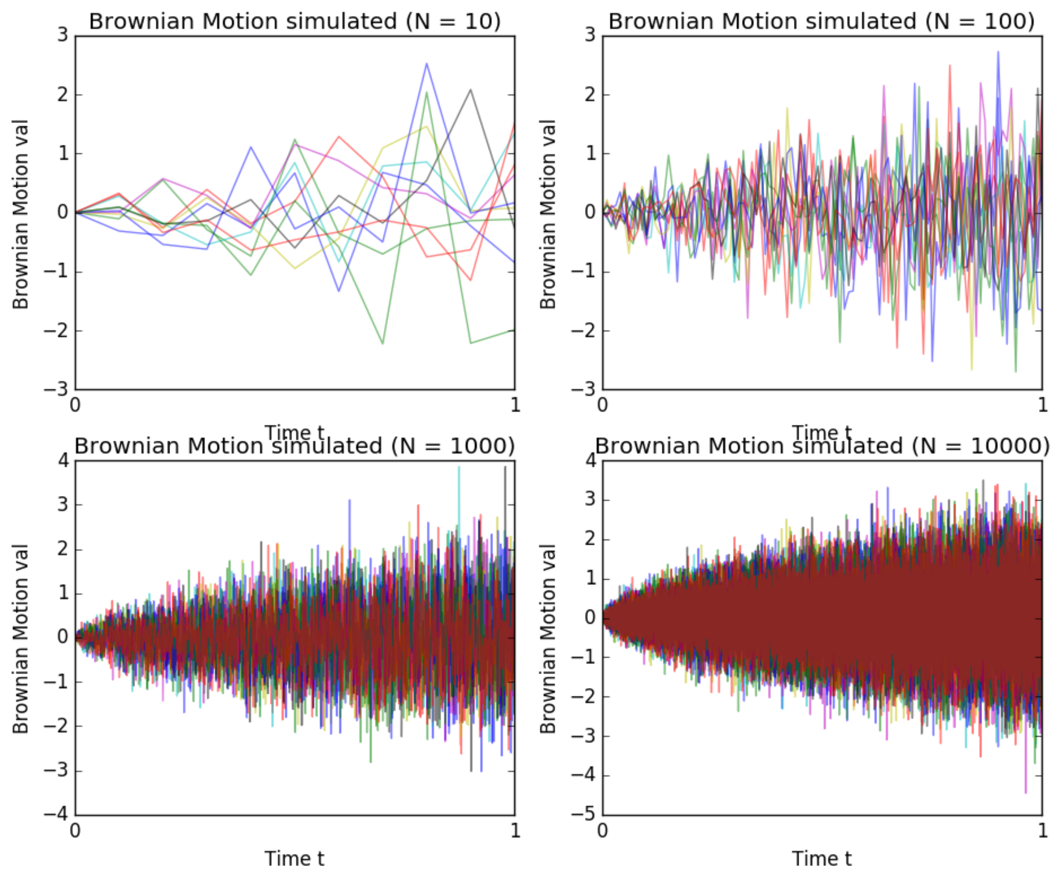
```python
# importing the necessary packages
import numpy as np
from numpy import sqrt
import matplotlib.pyplot as plt
import timeit

###############################################################################

# starting timer
start = timeit.default_timer()
```

```python

# cholesky method

def SimBMChol(T, N):
    # initialize W brownian motion array
    W = np.zeros(N)
    W = np.append(W, 0)     # initial 0 value

    # looping and iterating through
    for i in xrange(N):

        # create matrix of Z random standard normal samples
        Z = np.zeros(i+1)
        for x in np.nditer(Z, op_flags = ['readwrite']):
            x[...] = np.random.standard_normal()

        val = sum(sqrt(T/N)*Z)

        W[i+1] = val

    return W

###################################

plt.clf()

plt.subplot(2,2,1)

# setting up t x-axis variable
t = np.arange(0.0, 11.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMChol(1., 10)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 10)")

ticks = np.arange(t.min(), t.max() + 1, 10)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,2)

# setting up t x-axis variable
t = np.arange(0.0, 101.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMChol(1., 100)
    plt.plot(t, W, alpha = 0.5)
    n += 1

```

```python
plt.title("Brownian Motion simulated (N = 100)")

ticks = np.arange(t.min(), t.max() + 1, 100)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,3)

# setting up t x-axis variable
t = np.arange(0.0, 1001.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMChol(1., 1000)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 1000)")

ticks = np.arange(t.min(), t.max() + 1, 1000)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

###################################
plt.subplot(2,2,4)

# setting up t x-axis variable
t = np.arange(0.0, 10001.0, 1.0)

# loop for 10 different paths
n = 0
while n < 10:
    W = SimBMChol(1., 10000)
    plt.plot(t, W, alpha = 0.5)
    n += 1

plt.title("Brownian Motion simulated (N = 10000)")

ticks = np.arange(t.min(), t.max() +  1, 10000)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

plt.show()

# stopping timer
stop = timeit.default_timer()

print "Cholesky method runtime = " , stop - start, " seconds"
```

Output:
Cholesky method runtime = 548.683270931 seconds

Figure:



Comments:
The initial scaled simulation method gave a runtime of ~1 second, whereas the cholesky simulation method gave a runtime of ~9 minutes. There can be factors such as my algorithm implementation, but the cholesky methodology is noticeably slower than the first scaling & stepsize method.

Question 6)

6) a) and b)

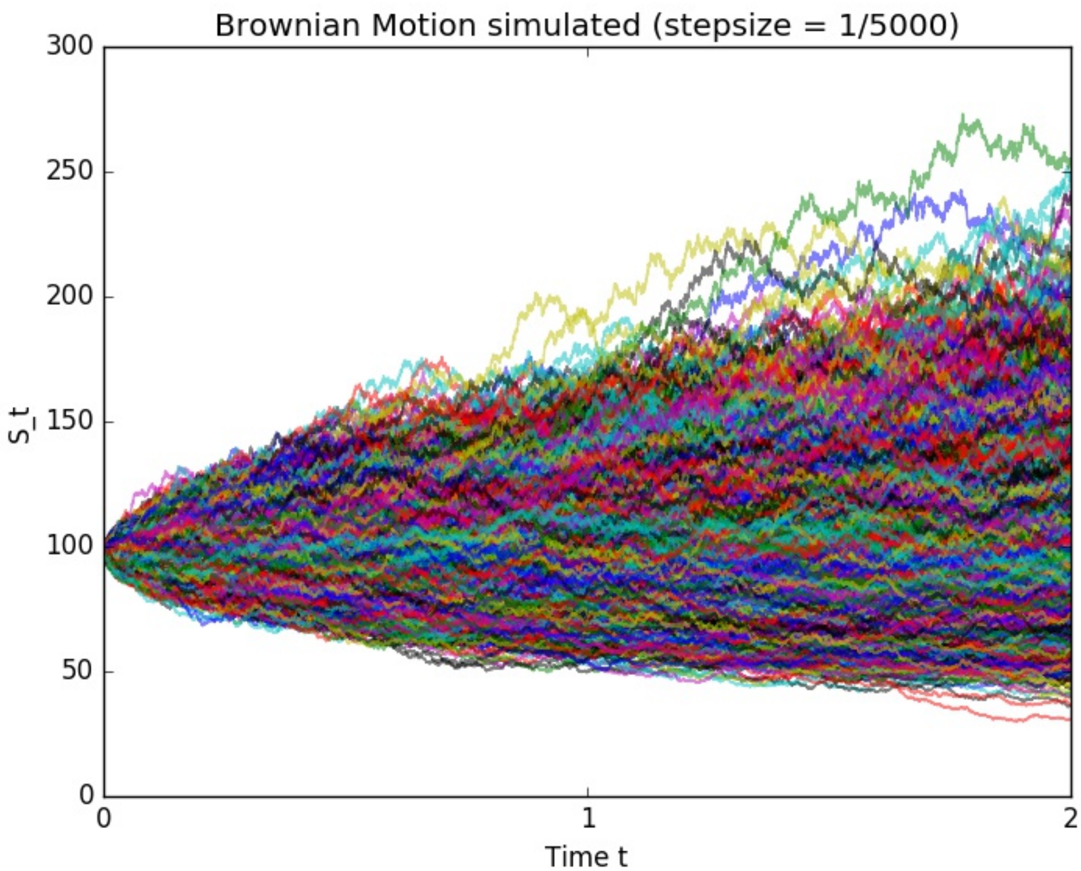
```python
# importing the necessary packages
import numpy as np
from numpy import sqrt, exp
import matplotlib.pyplot as plt
import timeit

##############################################################################

# starting timer
start = timeit.default_timer()

# initial variables
S_0 = 100.
B_0 = 1
mu = 0.17
sigma = 0.25
r = 0.02

# We want to price an up-and-out barrier put option
T = 2.
K = 105.
B = 120.

# increment step method defined function from Q5_scaling.py file
def SimBMStep(T, N):
    # initialize W brownian motion array
    W = np.zeros(N)
    W = np.append(W, 0)    # initial 0 value

    # loop
    num = 1
    while num <= N:
        W[num] = W[num - 1] + sqrt(T/N)*np.random.standard_normal()
        num += 1


    return W

# stock generation defined function from Q1.py file
# note that we're multiplying by W value, instead of sqrt(T)*random normal
def generateStock(S_0, r, sigma, T, W):
    return S_0 * exp((r - (sigma**2)/2)*T + sigma * W)

# defining payoff function, for this case it's regular European put
def payoff_func(S, K):
    return max(K - S, 0)
###################################

plt.clf()

# note, stepsize is 1/5000, but since T = 2, step size N = 10,000

# setting up t x-axis variable
t = np.arange(0.0, 10001.0, 1.0)

# keeping track of payoffs of up-and-out barrier put
```

```
payoffs = np.zeros(0)

# tally for number of times barrier was hit
countBarrier = 0

# loop for 10,000 different paths, for stepsize = 1/5000
n = 0
while n < 10000:

    # simulate brownian motion
    W = SimBMStep(1., 10000)

    # generate future stock, using the simulated W value
    S_T = generateStock(S_0, r, sigma, T, W)
    plt.plot(t, S_T, alpha = 0.5)

    # check if barrier condition was hit
    if np.any(S_T > B):
        # if past barrier at ANY point in time previously, payoff = 0
        payoffs = np.append(payoffs, 0)
        # keep track of how many times we hit barrier condition
        countBarrier += 1
    else:
        payoffs = np.append(payoffs, exp(-r*T)*payoff_func(S_T[S_T.size - 1], K))


    n += 1

# displaying the simulation brownian motion W

plt.title("Brownian Motion simulated (stepsize = 1/5000)")

ticks = np.arange(t.min(), t.max() + 1, 5000)
labels = range(ticks.size)
plt.xticks(ticks, labels)
plt.xlabel("Time t")

plt.ylabel("Brownian Motion val")

plt.show()

# calculating price
MonteCarloPrice = sum(payoffs) / 10000
print("Monte Carlo integration estimation of price is: ")
print(MonteCarloPrice)
print("And number of times barrier was hit were: ")
print(countBarrier)

# stopping timer
stop = timeit.default_timer()

print "Barrier 120 method runtime = " , stop - start, " seconds"
```

Output:
Monte Carlo integration estimation of price is:
10.8047400331

And number of times barrier was hit were:
4117
Barrier 120 method runtime = 192.690494776 seconds

Figure:



6)c)
Number of times barrier was hit for when Barrier = 120:
4117
Number of times barrier was hit for when Barrier = 105:
7769
Number of times barrier was hit for when Barrier = 180:
155

6) d)
To improve the speed of the algorithm, I would check the barrier condition upon each S_t simulation calculation. This is so that the inner simulation loop would end as soon the barrier condition gets hit.