

Question 1)b)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import sqrt, exp, log
4. from scipy.stats import norm
5.
6. #####
7.
8. # variables given
9. S_0 = 100.
10. B0 = 1
11. mu = 0.03
12. sigma = 0.25
13. r = 0.01
14.
15. # European Call, using Black Scholes formula
16. K = 120.
17. T = 1
18.
19. d1 = (log(S_0/K) + (r + (sigma**2)/2)*T) / (sigma * sqrt(T))
20. d2 = d1 - sigma*sqrt(T)
21.
22. Price1 = S_0*norm.cdf(d1) - K*exp(-r*T)*norm.cdf(d2)
23.
24. print("European Call option price (using Black Scholes formula) is:")
25. print(round(Price1,2))
26.
27. #####
28. # question 1)
29.
30. # defining the stock generation function
31. def generateStock(S_0, r, sigma, T):
32.     return S_0 * exp((r - (sigma**2)/2)*T + sigma * sqrt(T) * np.random.standard_normal
33.     ())
34.
35. # defining payoff function, for this case it's regular European put
36. def payoff1(S, K):
37.     return max(S - K, 0)
38.
39. # defining monte carlo integration formula, to make graphing easier
40. def MonteCarlo1(runs):
41.     # initialize array that will have payoffs of option
42.     payoffs = np.zeros(0)
43.
44.     # looping through
45.     for i in xrange(runs):
46.         # generate future stock
47.         S_T = generateStock(S_0, r, sigma, T)
48.
49.         # append to the payoffs list whatever the payoff is
50.         payoffs = np.append(payoffs,
51.                             exp(-r*T)*payoff1(S_T, K))
52.
53.     return sum(payoffs)/runs, payoffs
54.
55. # setting number of simulation runs
56. runs = 10000
```

```
56.  
57. # initialize array that will have payoffs of option  
58. payoffs = np.zeros(0)  
59.  
60. # monte carlo calculation of option price  
61. Price2, payoffs = MonteCarlo1(runs)  
62.  
63. var_sample = 0  
64. payoffs_int = payoffs.astype(int)  
65. # setting up iterating array  
66. it = np.nditer(payoffs, flags = ['f_index'])  
67.  
68. while not it.finished:  
69.     var_sample = var_sample + (payoffs[it.index] - Price2)**2  
70.     it.iternext()  
71. var_sample = var_sample / (runs - 1)  
72.  
73.  
74. # from analytical calculation, we know sigma = Var(X1) = 657.973  
75.  
76. # 95% confidence, z-score = 1.96  
77. size_estimate = (1.96**2 * 657.973) / 0.05**2  
78.  
79. print("European Call option price (using Monte Carlo integration) is: ")  
80. print(round(Price2,2))  
81. print("Approximation of how many samples needed for 95% confidence (up to dime):")  
82. print(int(size_estimate))  
83.  
84. # running Monte Carlo calculation 100 times with the above size estimate  
85. # counters for counting number of times the Monte Carlo estimated price  
86. # correct up to the dime or not  
87. correct = 0  
88. incorrect = 0  
89. for j in xrange(100):  
90.     price_to_compare, dummypayoff = MonteCarlo1(int(size_estimate))  
91.  
92.     if price_to_compare > Price1 - 0.05 and price_to_compare < Price1 + 0.05:  
93.         correct += 1  
94.     else:  
95.         incorrect += 1  
96.  
97. print("Number of times Monte Carlo estimation was within a dime:")  
98. print(correct)  
99. print("Number of times Monte Carlo estimation was NOT within a dime:")  
100. 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)

```
1. # importing the necessary packages
2. import numpy as np
3.
4. #####
5.
6. alpha = 0.6
7. #lambda = 1
8.
9. losses = np.zeros(0)
10.
11. # loss random variable equation
12. def loss(y):
13.     return 100000*y - 220000
14.
15. n = 0
16. while n < 100000:
17.     x = loss(np.random.weibull(alpha))
18.     losses = np.append(losses, x)
19.     n += 1
20.
21. # sort losses array from largest to lowest
22. losses = np.sort(losses)
23. losses[:] = losses[::-1]
24.
25. # select the loss (95% value at risk) that is the specified one
26. # 100000*0.05 = 5000
27.
28. print("Numerical estimation of 95% Value At Risk is:")
29. print(losses[5000])
```

**Output:**

Numerical estimation of 95% Value At Risk is:  
401456.414615

Question 3)c) and d)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import sqrt
4. import matplotlib.pyplot as plt
5.
6. #####
7.
8. # initial variables
9. covariance = np.zeros([2,2])
10. sigma1 = 3.2
11. sigma2 = 0.7
12. corr = 0.6
13. mu1 = 2
14. mu2 = 3
15.
16. # setting up array for actual covariance calculation
17. array1 = np.array([[sqrt(sigma1), 0],
18.                    [0, sqrt(sigma2)]])
19. array2 = np.array([[1, corr],
20.                    [corr, 1]])
21.
22. covariance = (array1.dot(array2)).dot(array1)
23. print("The covariance matrix is: ")
24. print(covariance)
25.
26. #####
27.
28. # hand-written result of cholesky factorization of A
29. # A = [[1.7885, 0]
30. #       [0.501996, 0.669328]]
31.
32. # cholesky factorization, from built-in default function
33. A_actual = np.linalg.cholesky(covariance)
34. print("The matrix A, based on built-in cholesky function np.linalg.cholesky(): ")
35. print(A_actual)
36.
37. #####
38.
39. # cholesky factorization by python algorithm coding
40.
41. # Note, need to do A_actual.tolist() to make it python list type
42.
43. def cholesky(A):
44.     # establish output S matrix
45.     n = len(A)
46.
47.     # setup output array
48.     S = [[0.0] * n for i in xrange(n)]
49.
50.     # setting up calculation of individual elements for loop
51.     for i in xrange(n):
52.         for j in xrange(i + 1):
53.             # sigma_ij formula from class
54.             temp = sum(S[i][k] * S[j][k] for k in xrange(j))
55.
56.             # for the diagonal case
57.             if i == j:
58.                 S[i][j] = sqrt(A[i][i] - temp)
```

```
59.         # for regular case
60.     else:
61.         S[i][j] = (A[i][j] - temp) / S[j][j]
62.
63.     return S
64.
65. A_cholesky = np.array(cholesky(covariance))
66.
67. print("The matrix A, based on python code algorithm function: ")
68. print(A_cholesky)
69.
70. #####
71.
72. # scatterplot work
73.
74. plt.clf()
75.
76. num = 0
77.
78. # producing 1000 samples
79. while num < 1000:
80.
81.     X1 = mu1 + A_cholesky[0,0]*np.random.standard_normal()
82.     X2 = mu2 + A_cholesky[1,0]*np.random.standard_normal() + A_cholesky[1,1]*np.random.
        standard_normal()
83.
84.     plt.scatter(X1, X2)
85.     num += 1
86.
87. plt.title("Scatterplot of (X1,X2)^T random vector samples")
88. plt.xlabel("X1")
89. plt.ylabel("X2")
90.
91. plt.show()
```

### Output:

The covariance matrix is:

```
[[ 3.2    0.89799777]
 [ 0.89799777 0.7    ]]
```

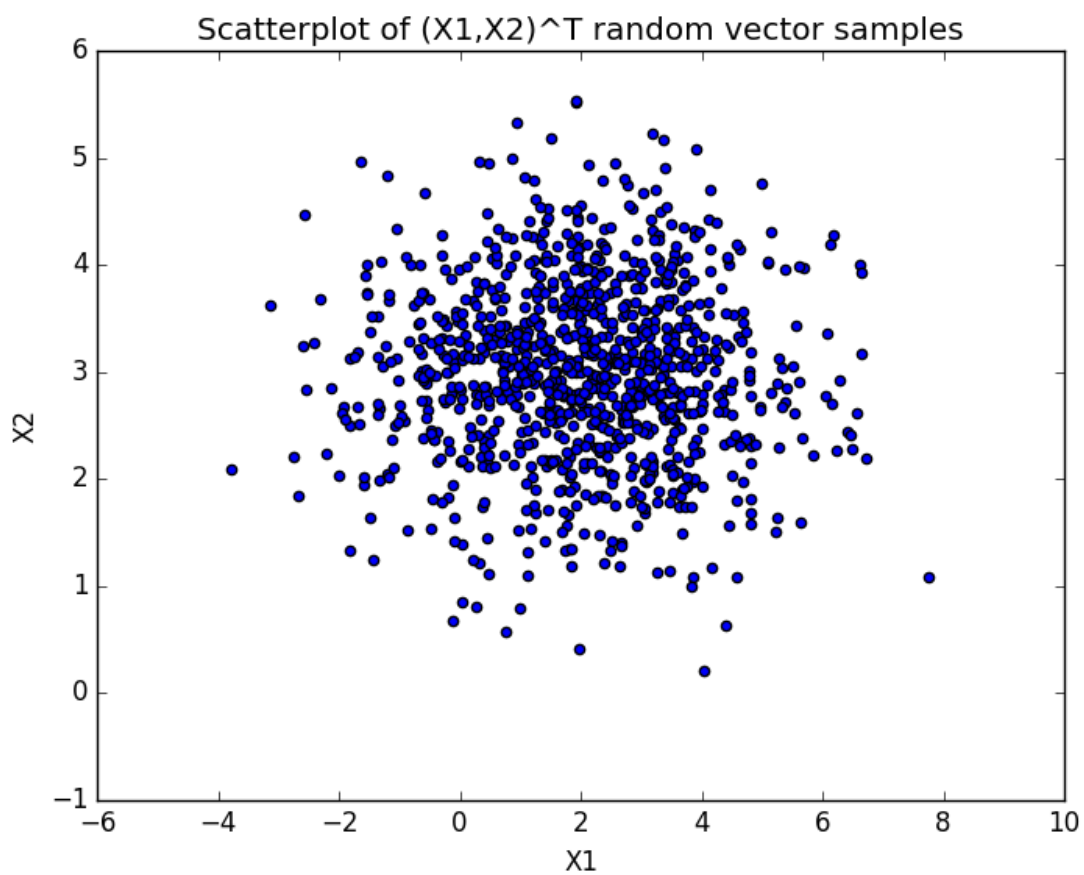
The matrix A, based on built-in cholesky function np.linalg.cholesky():

```
[[ 1.78885438 0.    ]
 [ 0.50199602 0.66932802]]
```

The matrix A, based on python code algorithm function:

```
[[ 1.78885438 0.    ]
 [ 0.50199602 0.66932802]]
```

Figure:



Question 4)

```
1. # importing the necessary packages
2. import numpy as np
3.
4. #####
5.
6. # initial variables
7. mu = np.array([-1000],
8.                [-700],
9.                [300],
10.               [-200]])
11.
12. covariance = np.array([[144, 72, 120, 300],
13.                        [72, 100, 180, 230],
14.                        [120, 180, 389, 880],
15.                        [300, 230, 880, 4469]])
16.
17. weights = np.array([0.4, 0.2, 0.3, 0.1])
18.
19. # setup L losses simulation
20. L = np.zeros(0)
21.
22. # get the matrix A thanks to built-in cholesky function
23. A_cholesky = np.linalg.cholesky(covariance)
24.
25. # simulation loop for 10,000 samples
26. num = 0
27. while num < 10000:
28.
29.     # this X matrix will hold our Assets
30.     X = np.zeros_like(mu)
31.
32.     # creating standard normal random samples matrix
33.     # each element is a different standard normal random sample
34.     Z = np.zeros_like(A_cholesky)
35.     for x in np.nditer(Z, op_flags = ['readwrite']):
36.         x[...] = np.random.standard_normal()
37.
38.
39.     # loop through to get assets vector
40.     for i in xrange(X.size):
41.         X[i] = mu[i] + sum(A_cholesky[i]*Z[i])
42.
43.     # loss based on calculation
44.     L = np.append(L, weights.dot(X))
45.
46.     num += 1
47.
48. # sort L array, from largest to smallest
49. L = np.sort(L)
50. L[:] = L[::-1]
51.
52. # we want 99% Value At Risk. So pick the 10000*0.01 = 100th loss element
53.
54. print("Numerical estimation of 99% Value At Risk is: ")
55. print(L[100])
```

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

Question 5)  
Q5\_scaling.py

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import sqrt
4. import matplotlib.pyplot as plt
5. import timeit
6.
7. #####
8.
9. # starting timer
10. start = timeit.default_timer()
11.
12. # increment step method
13. def SimBMStep(T, N):
14.     # initialize W brownian motion array
15.     W = np.zeros(N)
16.     W = np.append(W, 0)    # initial 0 value
17.
18.     # loop
19.     num = 1
20.     while num <= N:
21.         W[num] = W[num - 1] + sqrt(T/N)*np.random.standard_normal()
22.         num += 1
23.
24.
25.     return W
26.
27. #####
28.
29. plt.clf()
30.
31. plt.subplot(2,2,1)
32.
33. # setting up t x-axis variable
34. t = np.arange(0.0, 11.0, 1.0)
35.
36. # loop for 10 different paths
37. n = 0
38. while n < 10:
39.     W = SimBMStep(1., 10)
40.     plt.plot(t, W, alpha = 0.5)
41.     n += 1
42.
43. plt.title("Brownian Motion simulated (N = 10)")
44.
45. ticks = np.arange(t.min(), t.max() + 1, 10)
46. labels = range(ticks.size)
47. plt.xticks(ticks, labels)
48. plt.xlabel("Time t")
49.
50. plt.ylabel("Brownian Motion val")
51.
52. #####
53. plt.subplot(2,2,2)
54.
55. # setting up t x-axis variable
56. t = np.arange(0.0, 101.0, 1.0)
```



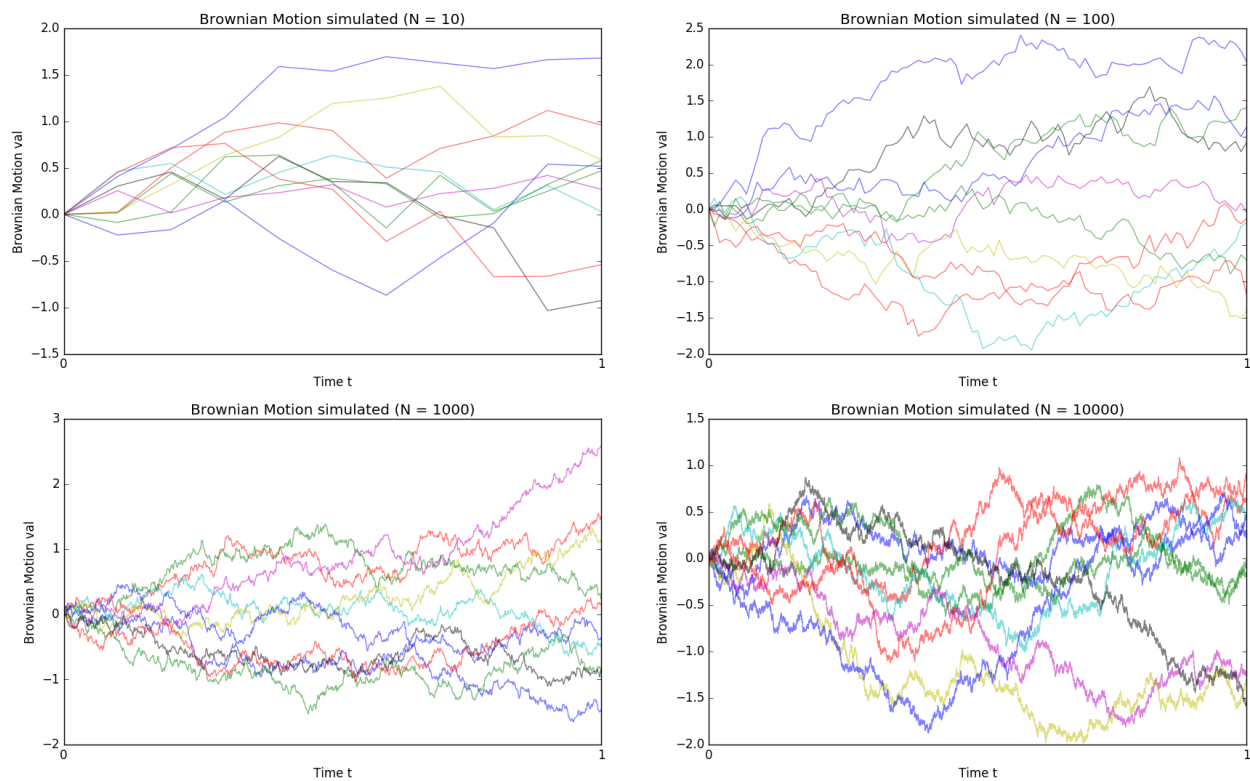
```
57.
58. # loop for 10 different paths
59. n = 0
60. while n < 10:
61.     W = SimBMStep(1., 100)
62.     plt.plot(t, W, alpha = 0.5)
63.     n += 1
64.
65. plt.title("Brownian Motion simulated (N = 100)")
66.
67. ticks = np.arange(t.min(), t.max() + 1, 100)
68. labels = range(ticks.size)
69. plt.xticks(ticks, labels)
70. plt.xlabel("Time t")
71.
72. plt.ylabel("Brownian Motion val")
73.
74. #####
75. plt.subplot(2,2,3)
76.
77. # setting up t x-axis variable
78. t = np.arange(0.0, 1001.0, 1.0)
79.
80. # loop for 10 different paths
81. n = 0
82. while n < 10:
83.     W = SimBMStep(1., 1000)
84.     plt.plot(t, W, alpha = 0.5)
85.     n += 1
86.
87. plt.title("Brownian Motion simulated (N = 1000)")
88.
89. ticks = np.arange(t.min(), t.max() + 1, 1000)
90. labels = range(ticks.size)
91. plt.xticks(ticks, labels)
92. plt.xlabel("Time t")
93.
94. plt.ylabel("Brownian Motion val")
95.
96. #####
97. plt.subplot(2,2,4)
98.
99. # setting up t x-axis variable
100.     t = np.arange(0.0, 10001.0, 1.0)
101.
102.     # loop for 10 different paths
103.     n = 0
104.     while n < 10:
105.         W = SimBMStep(1., 10000)
106.         plt.plot(t, W, alpha = 0.5)
107.         n += 1
108.
109.     plt.title("Brownian Motion simulated (N = 10000)")
110.
111.     ticks = np.arange(t.min(), t.max() + 1, 10000)
112.     labels = range(ticks.size)
113.     plt.xticks(ticks, labels)
114.     plt.xlabel("Time t")
115.
116.     plt.ylabel("Brownian Motion val")
```

```
117.  
118.     plt.show()  
119.  
120.     # stopping timer  
121.     stop = timeit.default_timer()  
122.  
123.     print "Step-size increment method runtime = " , stop - start, " seconds"
```

**Output:**

Step-size increment method runtime = 1.27352404594 seconds

**Figure:**



**Q5\_cholesky.py**

```
1. # importing the necessary packages  
2. import numpy as np  
3. from numpy import sqrt  
4. import matplotlib.pyplot as plt  
5. import timeit  
6.  
7. #####  
8.  
9. # starting timer  
10. start = timeit.default_timer()
```

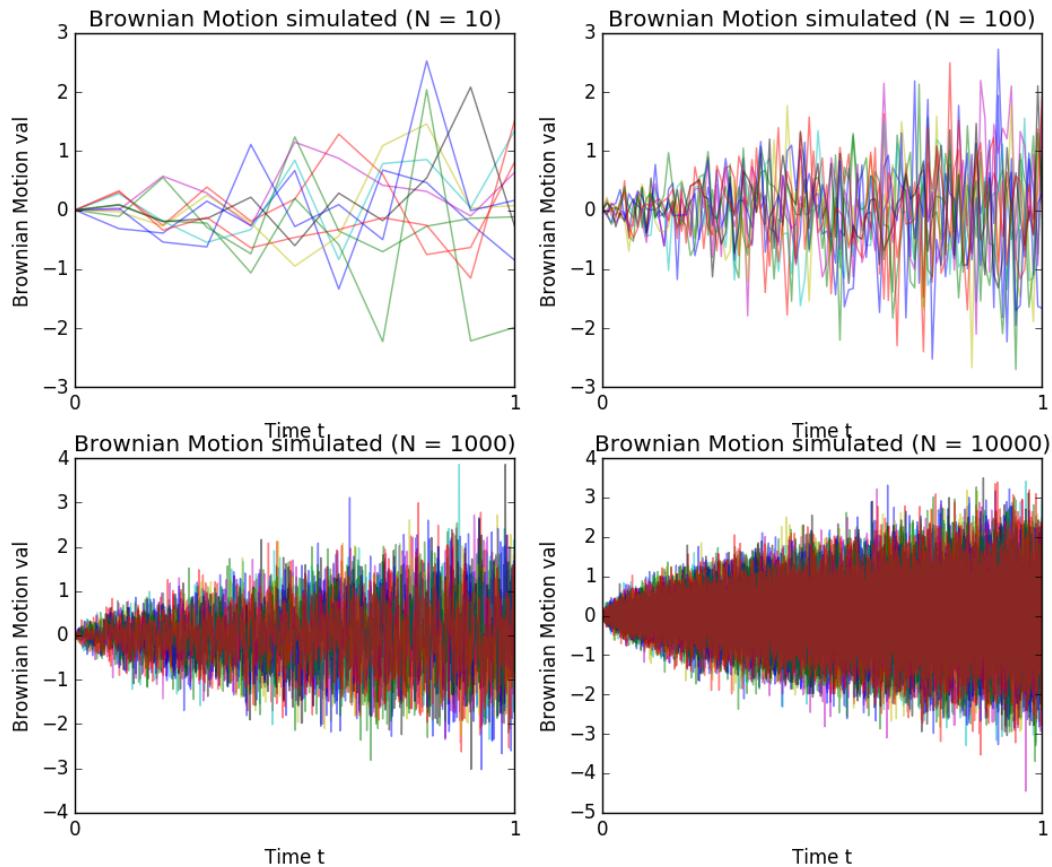
```
11.
12. # cholesky method
13.
14. def SimBMChol(T, N):
15.     # initialize W brownian motion array
16.     W = np.zeros(N)
17.     W = np.append(W, 0)    # initial 0 value
18.
19.     # looping and iterating through
20.     for i in xrange(N):
21.
22.         # create matrix of Z random standard normal samples
23.         Z = np.zeros(i+1)
24.         for x in np.nditer(Z, op_flags = ['readwrite']):
25.             x[...] = np.random.standard_normal()
26.
27.         val = sum(sqrt(T/N)*Z)
28.
29.         W[i+1] = val
30.
31.     return W
32.
33. #####
34.
35. plt.clf()
36.
37. plt.subplot(2,2,1)
38.
39. # setting up t x-axis variable
40. t = np.arange(0.0, 11.0, 1.0)
41.
42. # loop for 10 different paths
43. n = 0
44. while n < 10:
45.     W = SimBMChol(1., 10)
46.     plt.plot(t, W, alpha = 0.5)
47.     n += 1
48.
49. plt.title("Brownian Motion simulated (N = 10)")
50.
51. ticks = np.arange(t.min(), t.max() + 1, 10)
52. labels = range(ticks.size)
53. plt.xticks(ticks, labels)
54. plt.xlabel("Time t")
55.
56. plt.ylabel("Brownian Motion val")
57.
58. #####
59. plt.subplot(2,2,2)
60.
61. # setting up t x-axis variable
62. t = np.arange(0.0, 101.0, 1.0)
63.
64. # loop for 10 different paths
65. n = 0
66. while n < 10:
67.     W = SimBMChol(1., 100)
68.     plt.plot(t, W, alpha = 0.5)
69.     n += 1
70.
```

```
71. plt.title("Brownian Motion simulated (N = 100)")
72.
73. ticks = np.arange(t.min(), t.max() + 1, 100)
74. labels = range(ticks.size)
75. plt.xticks(ticks, labels)
76. plt.xlabel("Time t")
77.
78. plt.ylabel("Brownian Motion val")
79.
80. #####
81. plt.subplot(2,2,3)
82.
83. # setting up t x-axis variable
84. t = np.arange(0.0, 1001.0, 1.0)
85.
86. # loop for 10 different paths
87. n = 0
88. while n < 10:
89.     W = SimBMChol(1., 1000)
90.     plt.plot(t, W, alpha = 0.5)
91.     n += 1
92.
93. plt.title("Brownian Motion simulated (N = 1000)")
94.
95. ticks = np.arange(t.min(), t.max() + 1, 1000)
96. labels = range(ticks.size)
97. plt.xticks(ticks, labels)
98. plt.xlabel("Time t")
99.
100.     plt.ylabel("Brownian Motion val")
101.
102. #####
103.     plt.subplot(2,2,4)
104.
105.     # setting up t x-axis variable
106.     t = np.arange(0.0, 10001.0, 1.0)
107.
108.     # loop for 10 different paths
109.     n = 0
110.     while n < 10:
111.         W = SimBMChol(1., 10000)
112.         plt.plot(t, W, alpha = 0.5)
113.         n += 1
114.
115.     plt.title("Brownian Motion simulated (N = 10000)")
116.
117.     ticks = np.arange(t.min(), t.max() + 1, 10000)
118.     labels = range(ticks.size)
119.     plt.xticks(ticks, labels)
120.     plt.xlabel("Time t")
121.
122.     plt.ylabel("Brownian Motion val")
123.
124.     plt.show()
125.
126.     # stopping timer
127.     stop = timeit.default_timer()
128.
129.     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)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import sqrt, exp
4. import matplotlib.pyplot as plt
5. import timeit
6.
7. #####
8.
9. # starting timer
10. start = timeit.default_timer()
11.
12. # initial variables
13. S_0 = 100.
14. B_0 = 1
15. mu = 0.17
16. sigma = 0.25
17. r = 0.02
18.
19. # We want to price an up-and-out barrier put option
20. T = 2.
21. K = 105.
22. B = 120.
23.
24. # increment step method defined function from Q5_scaling.py file
25. def SimBMStep(T, N):
26.     # initialize W brownian motion array
27.     W = np.zeros(N)
28.     W = np.append(W, 0)    # initial 0 value
29.
30.     # loop
31.     num = 1
32.     while num <= N:
33.         W[num] = W[num - 1] + sqrt(T/N)*np.random.standard_normal()
34.         num += 1
35.
36.
37.     return W
38.
39. # stock generation defined function from Q1.py file
40. # note that we're multiplying by W value, instead of sqrt(T)*random normal
41. def generateStock(S_0, r, sigma, T, W):
42.     return S_0 * exp((r - (sigma**2)/2)*T + sigma * W)
43.
44. # defining payoff function, for this case it's regular European put
45. def payoff_func(S, K):
46.     return max(K - S, 0)
47. #####
48.
49. plt.clf()
50.
51. # note, stepsize is 1/5000, but since T = 2, step size N = 10,000
52.
53. # setting up t x-axis variable
54. t = np.arange(0.0, 10001.0, 1.0)
55.
56. # keeping track of payoffs of up-and-out barrier put
```

```
57. payoffs = np.zeros(0)
58.
59. # tally for number of times barrier was hit
60. countBarrier = 0
61.
62. # loop for 10,000 different paths, for stepsize = 1/5000
63. n = 0
64. while n < 10000:
65.
66.     # simulate brownian motion
67.     W = SimBMStep(1., 10000)
68.
69.     # generate future stock, using the simulated W value
70.     S_T = generateStock(S_0, r, sigma, T, W)
71.     plt.plot(t, S_T, alpha = 0.5)
72.
73.     # check if barrier condition was hit
74.     if np.any(S_T > B):
75.         # if past barrier at ANY point in time previously, payoff = 0
76.         payoffs = np.append(payoffs, 0)
77.         # keep track of how many times we hit barrier condition
78.         countBarrier += 1
79.     else:
80.         payoffs = np.append(payoffs, exp(-r*T)*payoff_func(S_T[S_T.size - 1], K))
81.
82.
83.     n += 1
84.
85. # displaying the simulation brownian motion W
86.
87. plt.title("Brownian Motion simulated (stepsize = 1/5000)")
88.
89. ticks = np.arange(t.min(), t.max() + 1, 5000)
90. labels = range(ticks.size)
91. plt.xticks(ticks, labels)
92. plt.xlabel("Time t")
93.
94. plt.ylabel("Brownian Motion val")
95.
96. plt.show()
97.
98. # calculating price
99. MonteCarloPrice = sum(payoffs) / 10000
100.     print("Monte Carlo integration estimation of price is: ")
101.     print(MonteCarloPrice)
102.     print("And number of times barrier was hit were: ")
103.     print(countBarrier)
104.
105.     # stopping timer
106.     stop = timeit.default_timer()
107.
108.     print "Barrier 120 method runtime = " , stop - start, " seconds"
```

**Output:**

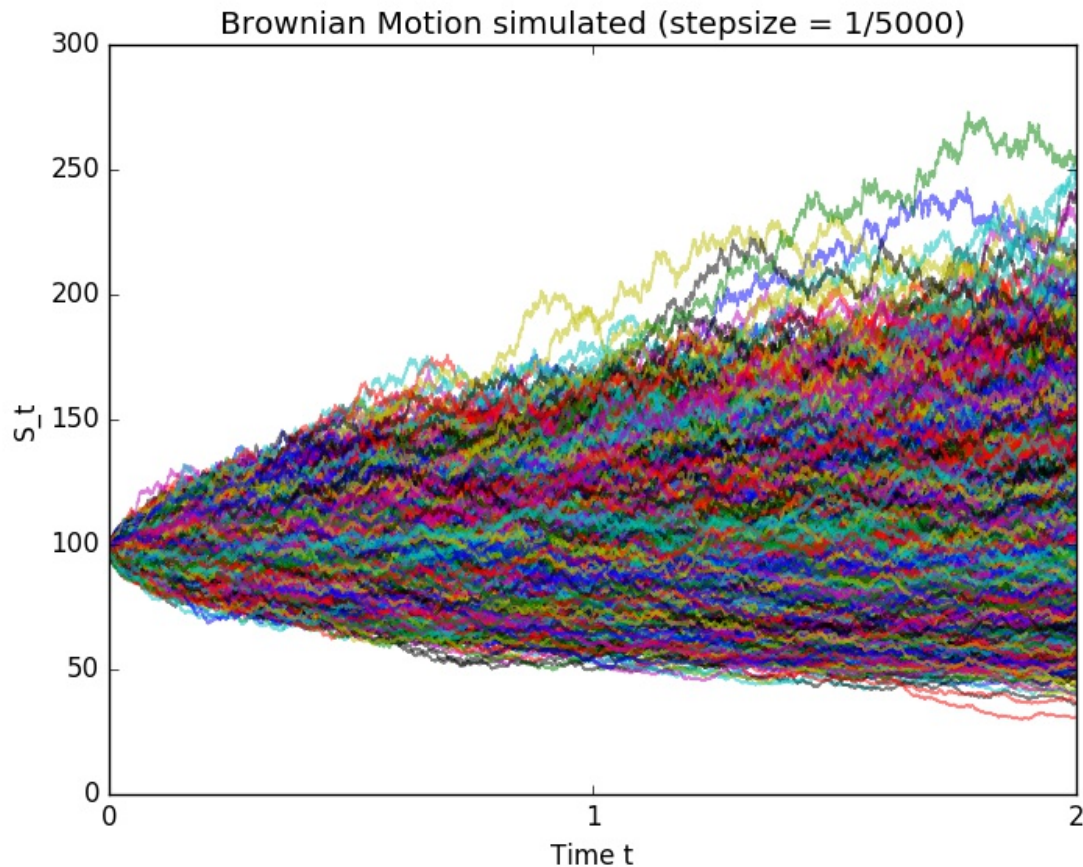
Monte Carlo integration estimation of price is:  
10.8047400331

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.