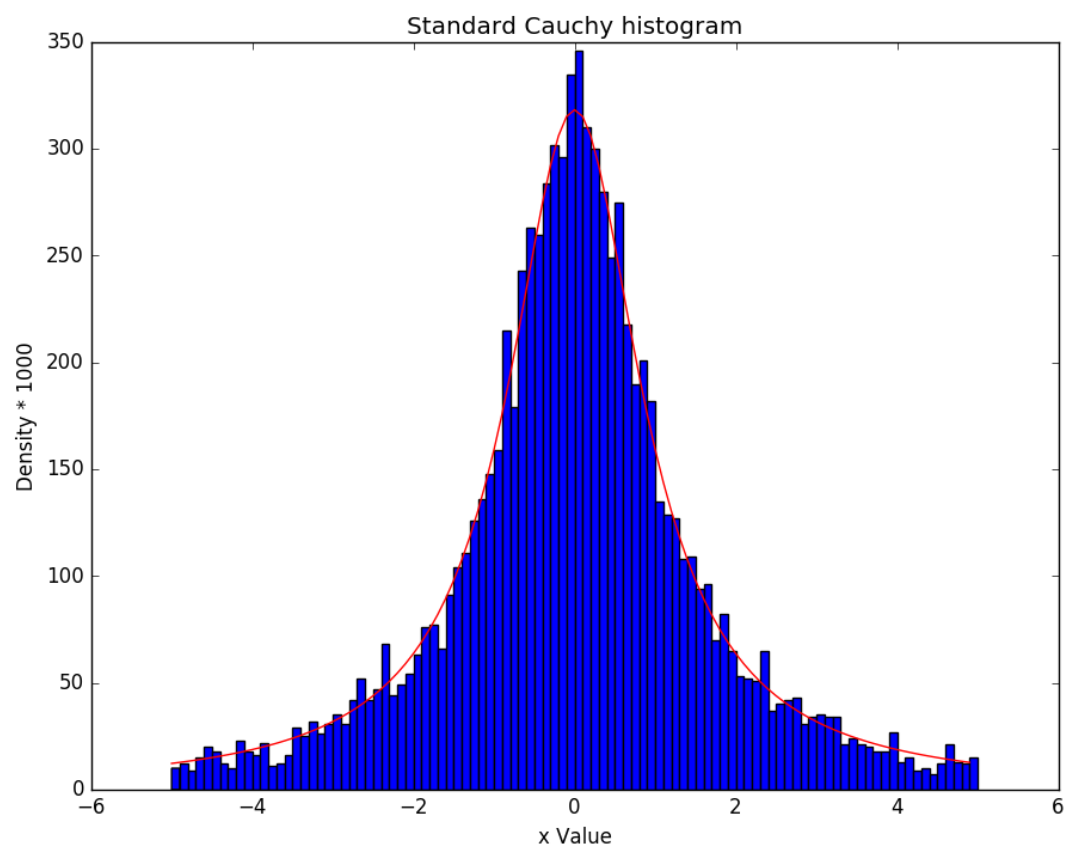


Question 1)b)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import pi, tan
4. import matplotlib.pyplot as plt
5.
6. #####
7. # CAUCHY
8.
9. # density function
10. def f(x):
11.     return (1/pi) * 1/(1 + x**2)
12.
13. # standard cauchy distribution
14. X = np.random.standard_cauchy(10000)
15.
16. # generalized inverse
17. # there is no built in function for cotangent.
18. # so I used the identity: cot(x) = 1/tan(x)
19. Y = -(1./tan(pi*X))
20.
21. # setting up variables to plot original density
22. Z = np.arange(-5, 5, 0.1)
23. W = f(Z)*1000
24.
25. # standard cauchy distribution plot
26. plt.plot(Z, W, color = 'red')
27. # histogram sample plot
28. plt.hist(Y, range = (-5, 5), bins = 100, color = 'blue')
29. plt.title ("Standard Cauchy histogram")
30. plt.xlabel ("x Value")
31. plt.ylabel ("Density * 1000")
32.
33. plt.show()
```

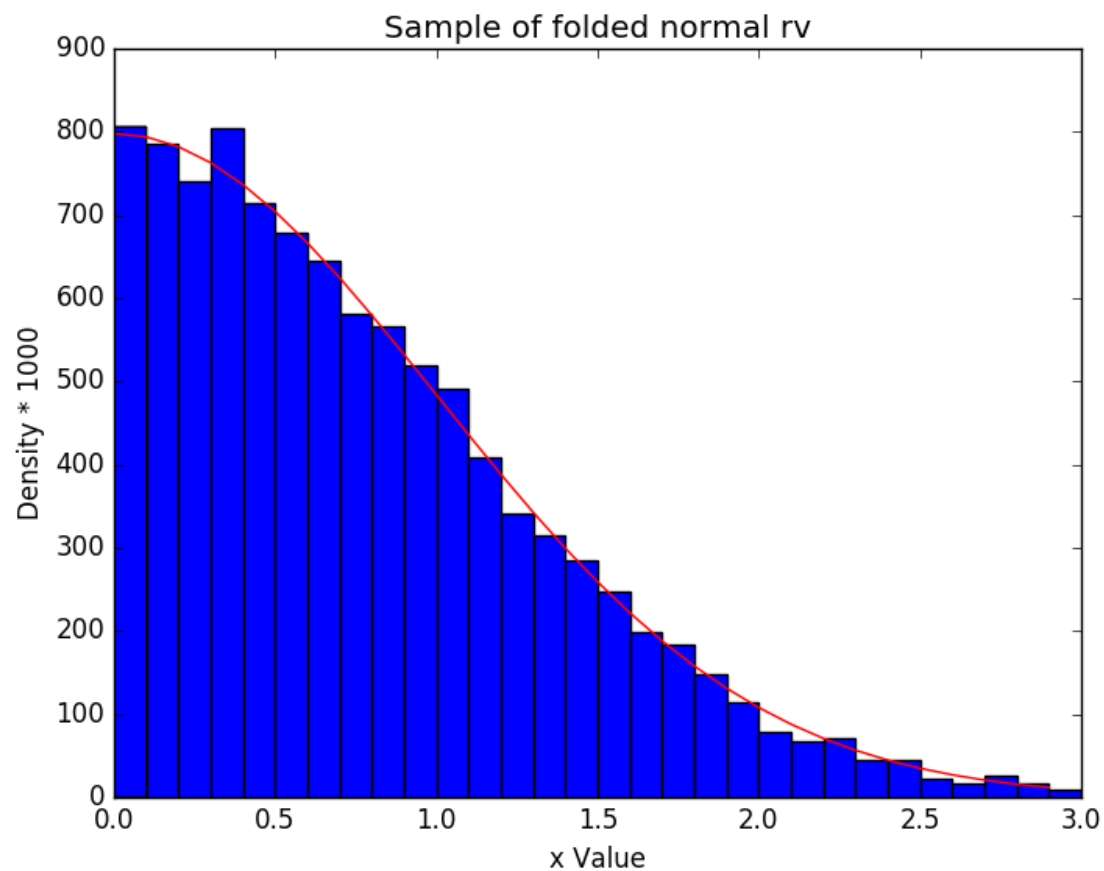
Figure:



Question 3)b)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import pi, sqrt, exp
4. import matplotlib.pyplot as plt
5.
6. #####
7. # acceptance rejection method
8.
9. # original density function
10. def f(x):
11.     return 2./sqrt(2.*pi) * exp(-(x**2) / 2.)
12. # density function of dominant exponential distribution
13. def g(x):
14.     return exp(-x)
15.
16. # array that will be filled with accepted variables
17. a = np.zeros(0)
18.
19. # constant c for method
20. c = 2./sqrt(2.*pi) * exp(0.5)
21.
22. n = 0
23.
24. plt.clf()
25.
26. # want 10000 samples
27. while n < 10000:
28.     U1 = np.random.rand()
29.     U2 = np.random.rand()
30.     Y = -np.log(1 - U1)
31.
32.     if c * g(Y) * U2 <= f(Y):
33.         n += 1
34.         # accept
35.         a = np.append(a, Y)
36.
37.
38.
39. plt.hist(a, range = (0, 3), bins = 30, color = 'blue')
40.
41. # setting up variables to plot original density
42. Z = np.arange(0, 3, 0.1)
43. W = f(Z)*1000
44.
45. # original distribution plot
46. plt.plot(Z, W, color = 'red')
47. plt.title ("Sample of folded normal rv")
48. plt.xlabel ("x Value")
49. plt.ylabel ("Density * 1000")
50.
51. plt.show()
```

Figure:



Question 5)  
Box Muller .py

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import pi, sin, cos, sqrt, exp, log
4. import matplotlib.pyplot as plt
5. import timeit
6.
7. #####
8. # Box Muller method
9.
10. # starting timer
11. start = timeit.default_timer()
12.
13. plt.clf()
14.
15. # exact normal density function
16. def f(x):
17.     return 1/(sqrt(2 * pi)) * exp(-(x)**2 / 2 )
18.
19. # initialize array of X rv's
20. X1 = np.zeros(0)
21. X2 = np.zeros(0)
22.
23. # defining Box Muller function producting std normal variables
24. def BoxMull(X1, X2):
25.     n = 0
26.     while n < 50000:
27.
28.         U1 = np.random.rand()
29.         U2 = np.random.rand()
30.
31.         X1 = np.append(X1, cos(2*pi*U1) * sqrt(-2*log(U2)))
32.         X2 = np.append(X2, sin(2*pi*U1) * sqrt(-2*log(U2)))
33.
34.         n += 1
35.
36.     return X1, X2
37.
38. # get function values
39. X1, X2 = BoxMull(X1, X2)
40.
41. # setting up variables to plot original density
42. Z = np.arange(-3, 3, 0.1)
43. W = f(Z)*5000
44.
45. # standard normal distribution plot
46. plt.plot(Z, W, color = 'red')
47.
48. # histogram sample plot
49. plt.hist([X1, X2], range = (-3, 3), bins = 60, alpha = 0.5)
50. plt.title ("Box-Muller method sampling")
51. plt.xlabel ("x Value")
52. plt.ylabel ("Density * 5000")
53.
54. plt.show()
55.
```

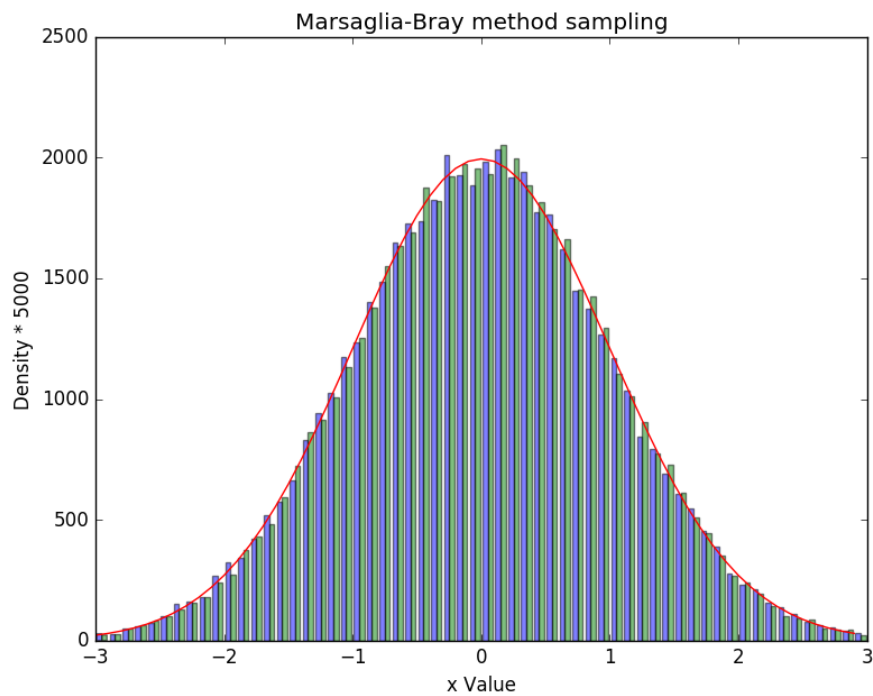
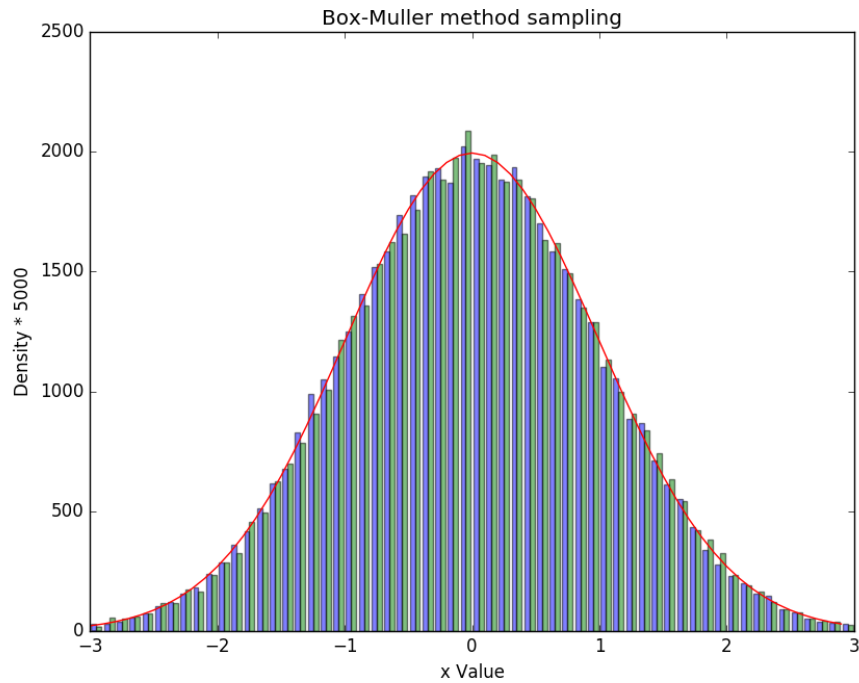
```
56. # stopping timer
57. stop = timeit.default_timer()
58.
59. print "Box Muller method runtime = " , stop - start, " seconds"
```

### Marsaglia Bray .py

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import pi, sqrt, exp, log
4. import matplotlib.pyplot as plt
5. import timeit
6.
7. #####
8. # Marsaglia-Bray method
9.
10. # starting timer
11. start = timeit.default_timer()
12.
13. plt.clf()
14.
15. # exact normal density function
16. def f(x):
17.     return 1/(sqrt(2 * pi)) * exp(-(x)**2 / 2 )
18.
19. # initialize array of accepted rv's
20. X1 = np.zeros(0)
21. X2 = np.zeros(0)
22.
23. # defining Marsaglia Bray function producing std normal variables
24. def MarsBray(X1, X2):
25.     n = 0
26.     while n < 50000:
27.         # setting dummy initial value
28.         S = 2
29.
30.         # this while loop makes sure that we are using an accepted S
31.         while S > 1:
32.             V1 = 2 * np.random.rand() - 1
33.             V2 = 2 * np.random.rand() - 1
34.             S = (V1 ** 2) + (V2 ** 2)
35.
36.             X1 = np.append(X1, V1 * sqrt(-2 * log(S) / S))
37.             X2 = np.append(X2, V2 * sqrt(-2 * log(S) / S))
38.
39.             n += 1
40.
41.     return X1, X2
42.
43. # get function values
44. X1, X2 = MarsBray(X1, X2)
45.
46. # setting up variables to plot original density
47. Z = np.arange(-3, 3, 0.1)
48. W = f(Z)*5000
49.
50. # standard normal distribution plot
51. plt.plot(Z, W, color = 'red')
52.
```

```
53. # histogram sample plot
54. plt.hist([X1, X2], range = (-3, 3), bins = 60, alpha = 0.5)
55. plt.title ("Marsaglia-Bray method sampling")
56. plt.xlabel ("x Value")
57. plt.ylabel ("Density * 5000")
58.
59. plt.show()
60.
61. # stopping timer
62. stop = timeit.default_timer()
63.
64. print "Marsaglia-Bray method runtime = " , stop - start, " seconds"
```

Figures:



Output of code:

Box Muller method runtime = 2.83222103119 seconds

Marsaglia-Bray method runtime = 2.75477385521 seconds



Question 6)

```
1. # importing the necessary packages
2. import numpy as np
3. from numpy import sqrt, exp, log
4. from scipy.stats import norm
5. import matplotlib.pyplot as plt
6.
7. #####
8. # question 6)a)
9.
10. # variables given
11. S_0 = 100.
12. B0 = 1
13. mu = 0.05
14. sigma = 0.2
15. r = 0.03
16.
17. # European Put, using Black Scholes formula
18. K = 110.
19. T = 0.5
20.
21. d1 = (log(S_0/K) + (r + (sigma**2)/2)*T) / (sigma * sqrt(T))
22. d2 = d1 - sigma*sqrt(T)
23.
24. Price1 = K*exp(-r*T)*norm.cdf(-d2) - S_0*norm.cdf(-d1)
25.
26. print("European Put option price (using Black Scholes formula) is:")
27. print(round(Price1,2))
28.
29. #####
30. # question 6)b)
31.
32. # defining the stock generation function
33. def generateStock(S_0, r, sigma, T):
34.     return S_0 * exp((r - (sigma**2)/2)*T + sigma * sqrt(T) * np.random.standard_normal
35.         ())
36.
37. # defining payoff function, for this case it's regular European put
38. def payoff1(S, K):
39.     return max(0, K - S)
40.
41. # setting number of simulation runs
42. runs = 10000
43. # initialize array that will have payoffs of option
44. payoffs = np.zeros(0)
45.
46. # looping through
47. for i in xrange(runs):
48.     # generate future stock
49.     S_T = generateStock(S_0, r, sigma, T)
50.     # append to the payoffs list whatever the payoff is
51.     payoffs = np.append(payoffs,
52.                         payoff1(S_T, K))
53.
54. Price2 = exp(-r*T) * sum(payoffs)/runs
55.
56. print("European Put option price (using Monte Carlo integration) is: ")
57. print(round(Price2,2))
```

```
58.
59. #####
60. # question 6)c)
61.
62. # defining monte carlo integration formula, to make graphing easier
63. def MonteCarlo1(runs):
64.     # initialize array that will have payoffs of option
65.     payoffs = np.zeros(0)
66.
67.     # looping through
68.     for i in xrange(runs):
69.         # generate future stock
70.         S_T = generateStock(S_0, r, sigma, T)
71.
72.         # append to the payoffs list whatever the payoff is
73.         payoffs = np.append(payoffs,
74.                             payoff1(S_T, K))
75.
76.     return exp(-r*T) * sum(payoffs)/runs
77.
78. #####
79. plt.subplot(2, 2, 1)
80.
81. n = 0
82.
83. while n < 50:
84.     y = MonteCarlo1(n)
85.
86.     plt.plot(n, y, color = 'red', marker = '.')
87.
88.     n += 1
89.
90. plt.plot(n, y, color = 'red', marker = '.',
91.          label = "samples n = 50");
92.
93. plt.title ("Approximation of European Put")
94. plt.xlabel ("Trials")
95. plt.ylabel ("Value")
96. plt.legend()
97.
98. #####
99. plt.subplot(2, 2, 2)
100.
101.     n = 0
102.
103.     while n < 100:
104.         y = MonteCarlo1(n)
105.
106.         plt.plot(n, y, color = 'red', marker = '.')
107.
108.         n += 1
109.
110.     plt.plot(n, y, color = 'red', marker = '.',
111.             label = "samples n = 100");
112.
113.     plt.title ("Approximation of European Put")
114.     plt.xlabel ("Trials")
115.     plt.ylabel ("Value")
116.     plt.legend()
117.
```

```
118. #####
119. plt.subplot(2, 2, 3)
120.
121.     n = 0
122.
123.     while n < 500:
124.         y = MonteCarlo1(n)
125.
126.         plt.plot(n, y, color = 'red', marker = '.')
127.
128.         n += 1
129.
130.     plt.plot(n, y, color = 'red', marker = '.',
131.             label = "samples n = 500");
132.
133.     plt.title ("Approximation of European Put")
134.     plt.xlabel ("Trials")
135.     plt.ylabel ("Value")
136.     plt.legend()
137.
138. #####
139. plt.subplot(2, 2, 4)
140.
141.     n = 0
142.
143.     while n < 1000:
144.         y = MonteCarlo1(n)
145.
146.         plt.plot(n, y, color = 'red', marker = '.')
147.
148.         n += 1
149.
150.     plt.plot(n, y, color = 'red', marker = '.',
151.             label = "samples n = 1000");
152.
153.     plt.title ("Approximation of European Put")
154.     plt.xlabel ("Trials")
155.     plt.ylabel ("Value")
156.
157.     plt.legend()
158.
159.     plt.show()
160.
161. #####
162. # question 6)d)
163.
164. # European asset-or-nothing digital call option
165. # same maturity T = 0.5, same strike K = 110
166.
167. # defining payoff function
168. def payoff2(S, K):
169.     if S > K:
170.         return S
171.     else:
172.         return 0
173.
174. # setting number of simulation runs
175. runs = 10000
176. # initialize array that will have payoffs of option
177. payoffs = np.zeros(0)
```

```
178.  
179.     # looping through  
180.     for i in xrange(runs):  
181.         # generate future stock  
182.         S_T = generateStock(S_0, r, sigma, T)  
183.  
184.         # append to the payoffs list whatever the payoff is  
185.         payoffs = np.append(payoffs,  
186.                             payoff2(S_T, K))  
187.  
188.     Price3 = exp(-r*T) * sum(payoffs)/runs  
189.  
190.     print("European asset-or-  
nothing digital call option price (using Monte Carlo integration) is: ")  
191.     print(round(Price3,2))  
192.  
193.     #####  
194.     # question 6)e)  
195.  
196.     # European cubic put option  
197.     # same maturity T = 0.5, same strike K = 110  
198.  
199.     # defining payoff function  
200.     def payoff3(S, K):  
201.         return (max(0, K - S))**3  
202.  
203.     # setting number of simulation runs  
204.     runs = 10000  
205.     # initialize array that will have payoffs of option  
206.     payoffs = np.zeros(0)  
207.  
208.     # looping through  
209.     for i in xrange(runs):  
210.         # generate future stock  
211.         S_T = generateStock(S_0, r, sigma, T)  
212.  
213.         # append to the payoffs list whatever the payoff is  
214.         payoffs = np.append(payoffs,  
215.                             payoff3(S_T, K))  
216.  
217.     Price4 = exp(-r*T) * sum(payoffs)/runs  
218.  
219.     print("European cubic put option price (using Monte Carlo integration) is: ")  
220.     print(round(Price4,2))  
221.  
222.     #####  
223.     # question 6)f)  
224.  
225.     # European gap call option  
226.     # same maturity T = 0.5, same strike K = 110  
227.  
228.     # L exercise level  
229.     L = 105  
230.  
231.     # defining payoff function  
232.     def payoff4(S, K):  
233.         if S > K:  
234.             return max(0, S - L)  
235.         else:  
236.             return 0
```

```
237.  
238.     # setting number of simulation runs  
239.     runs = 10000  
240.     # initialize array that will have payoffs of option  
241.     payoffs = np.zeros(0)  
242.  
243.     # looping through  
244.     for i in xrange(runs):  
245.         # generate future stock  
246.         S_T = generateStock(S_0, r, sigma, T)  
247.  
248.         # append to the payoffs list whatever the payoff is  
249.         payoffs = np.append(payoffs,  
250.                             payoff4(S_T, K))  
251.  
252.     Price5 = exp(-r*T) * sum(payoffs)/runs  
253.  
254.     print("European gap call option price (using Monte Carlo integration) is: ")  
255.     print(round(Price5,2))  
256.  
257.     #####  
258.     # question 6)g)  
259.  
260.     # European exponential put option  
261.     # same maturity T = 0.5, same strike K = 110  
262.  
263.     # defining payoff function  
264.     def payoff5(S, K):  
265.         return exp(max(0, K - S))  
266.  
267.     # setting number of simulation runs  
268.     runs = 10000  
269.     # initialize array that will have payoffs of option  
270.     payoffs = np.zeros(0)  
271.  
272.     # looping through  
273.     for i in xrange(runs):  
274.         # generate future stock  
275.         S_T = generateStock(S_0, r, sigma, T)  
276.  
277.         # append to the payoffs list whatever the payoff is  
278.         payoffs = np.append(payoffs,  
279.                             payoff5(S_T, K))  
280.  
281.     Price6 = exp(-r*T) * sum(payoffs)/runs  
282.  
283.     print("European exponential put option price (using Monte Carlo integration) is:  
284.     ")  
284.     print(round(Price6,2))
```

Output of code:

European Put option price (using Black Scholes formula) is:

10.97

European Put option price (using Monte Carlo integration) is:

11.03

European asset-or-nothing digital call option price (using Monte Carlo integration) is:

31.27

European cubic put option price (using Monte Carlo integration) is:

5809.78

European gap call option price (using Monte Carlo integration) is:

3.92

European exponential put option price (using Monte Carlo integration) is:

4.1053809922e+19

Figure:

