MFE 405: Project 1

Aliaksei Kanstantsinau

Problem 1

Using the LGM method, generate Uniformly distributed random numbers on [0,1] to do the following:

(a) Generate 1,000 random numbers with Binomial distribution with $n = 44$ and $p = 0.64$. Compute the probability that the random variable X, that has Binomial (44, 0.64) distribution, is at least 40: $P(X \geq 40)$. Use any statistics textbook or online resources for the exact number for the above probability and compare it with your finding and comment. Hint: A random variable with Binomial distribution (n, p) is a sum of n Bernoulli (p) distributed random variables, so you will need to generate 44,000 Uniformly distributed random numbers, to start with.

In [147…

```
def uniform_variable(seed):
    '''This function generates non-normalized X_i'''
    U_i = (13 * seed + 29) % 128
    return U_i

def uniform_variable_normalized(seed):
    '''Copy of uniform variable function generating normalized value'''
    U_i = (13 * seed + 29) % 128
    return U_i/128

def probability(list, value):
    '''This function calculates probability'''
    empty_list = []
    for number in list:
        if number >= value:
            empty_list.append(number)
    return len(empty_list)/len(list)
```

In [100…

```
def bernoulli_variable(seed, trials, samples, p, X_prob):
    '''This function generates a randomly generated benoulli variable'''
    # create a list to store generated variables
    generated_numbers = []
    # create a list to store probability values
    prob_list = []
    # iterate through trials setting new seed each time
    for sample in range(seed, samples + seed):
        U_n = seed + sample
        # create a list to store samples
        sample_list = []
        # iterate through the formula sample times
        for number in range(trials):
            U_nplusone = uniform_variable(U_n)
            if uniform_variable_normalized(U_n) < p:
```

```
                sample_list.append(1)
            # set x_n as x_n+1
            U_n = U_nplusone

        # append sum of the sample list to generated numbers list
        generated_numbers.append(sum(sample_list))

    # calculate the probability
    print(f'Probability that X>={X_prob} =', probability(generated_numbers,
    # return the list
    return generated_numbers

# test values
first_try = bernoulli_variable(6, 44, 1000, .64, 35)
print(first_try[:15])
```

```
Probability that X>=35 = 0.0
[27, 29, 30, 22, 31, 30, 27, 26, 26, 27, 28, 28, 29, 28, 27]
```

(b) Generate 10,000 Exponentially distributed random numbers with parameter $\lambda = 1.5$. Estimate $P(X \geq 1)$; $P(X \geq 4)$; and compute the empirical mean and the standard deviation of the sequence of 10,000 numbers. Draw the histogram by using the 10,000 numbers you have generated. Note: Random variable X that is exponentially distributed with parameter λ has the following cdf: $F(t) = P(X \leq t) = 1 - e$^-$t/\lambda$ for $t \geq 0$ (and $E(X) = \lambda$)

In [149…

```
# import numpy for log
import numpy as np
# import matplotlib for plotting
import matplotlib.pyplot as plt
```

In [913…

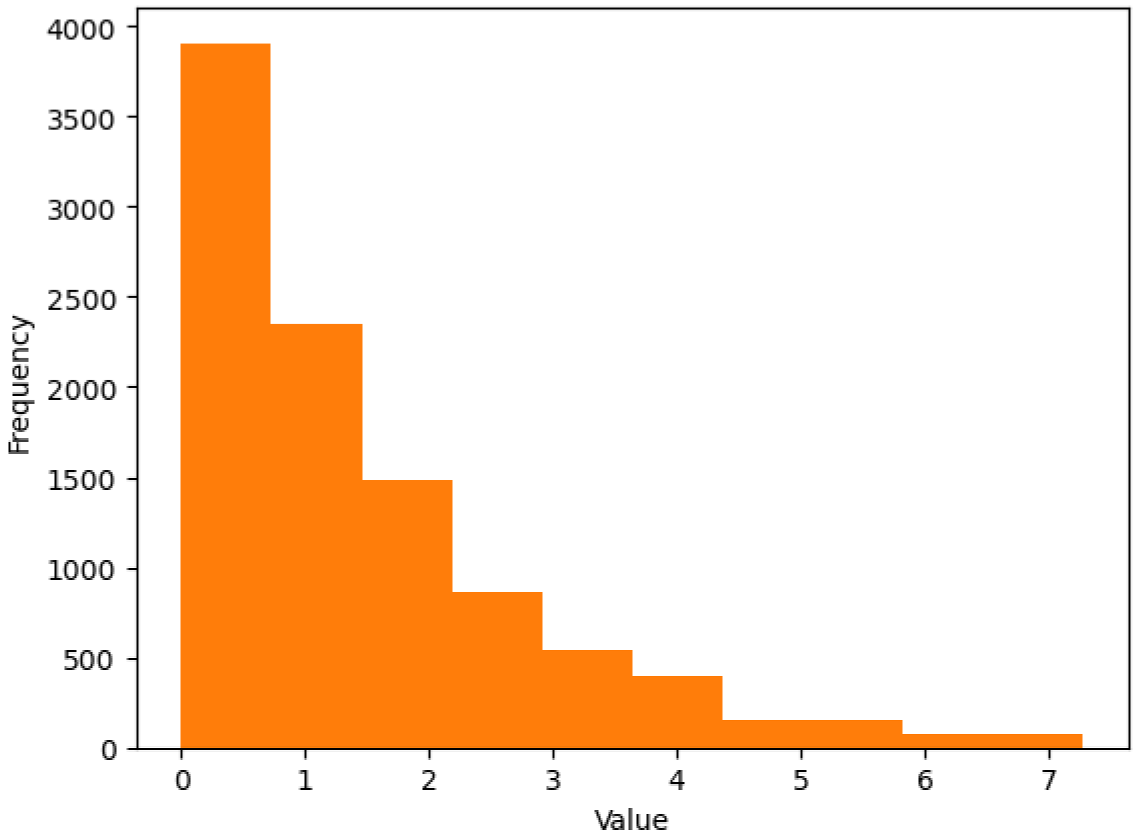
```
def exponentially_distributed(seed, samples, lambd, X_prob):
    '''This function generates exponentially distributed random numbers'''
    # create a list to store generated numbers
    generated_numbers = []
    prob_list = []
    # iterate through each sample with a new starting point
    for sample in range(samples):
        U_n = seed+sample
        U_nplusone = uniform_variable(U_n)
        # append X_i
        X_i = -lambd*np.log(1 - uniform_variable_normalized(U_n))
        generated_numbers.append(X_i)

    # plot the result
    plt.hist(generated_numbers)
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    # get the probability
    print(f'Probability that X>={X_prob} =', probability(generated_numbers,
    #return the result
    return generated_numbers

second_try = exponentially_distributed(3, 10000, 1.5, 1)
third_try = exponentially_distributed(1, 10000, 1.5, 4)
```

```
print(second_try[:15])
print(third_try[:15])
```

```
Probability that X>=1 = 0.508
Probability that X>=4 = 0.0625
[1.1365285525462747, 1.5028239933143381, 1.9885046089551837, 2.7112617392942
91, 4.1588830833596715, 0.05976886282079951, 0.2273248471908014, 0.415978928
12435155, 0.6318201976144553, 0.8840255803174526, 1.1873808800597967, 1.5680
516612239463, 2.0794415416798357, 2.861386927129765, 4.590406192037344]
[0.5965244514991641, 0.8423562341568391, 1.1365285525462747, 1.5028239933143
381, 1.9885046089551837, 2.711261739294291, 4.1588830833596715, 0.0597688628
2079951, 0.2273248471908014, 0.41597892812435155, 0.6318201976144553, 0.8840
255803174526, 1.1873808800597967, 1.5680516612239463, 2.0794415416798357]
```



(c) Generate 5,000 Normally distributed random numbers with mean 0 and variance 1, by using the Box- Muller Method.

In [914…

```
def box_mueller(seed, sample_size):
    '''This function generates normally distributed random variables using t
    generated_values = []
    # set sample size
    true_sample = int(sample_size/2)
    # iterate and generate 2 numbers
    for sample in range(true_sample):
        # ensure seeds are different
        uniform_1 = uniform_variable_normalized(seed+sample)
        uniform_2 = uniform_variable_normalized(seed+sample+145917)
        # z1 and z2 from slides
        z_1 = np.sqrt(-2*np.log(uniform_1)) * np.cos(2*np.pi*uniform_2)
        z_2 = np.sqrt(-2*np.log(uniform_1)) * np.sin(2*np.pi*uniform_2)
```

```
        generated_values.append(z_1)
        generated_values.append(z_2)

    return generated_values

test_1 = box_mueller(1, 5000)
print(test_1[:15])
```

```
[1.4767319648460795, 0.21905249433488735, 0.9190740333004445, 0.919074033300
4443, 0.16503396323575265, 1.1125686084308335, -0.4509574988370361, 0.843682
1396337612, -0.710348645237785, 0.33596988779959164, -0.5871643401386492, -
0.11679424893789352, -0.24127295715551628, -0.2662036859635395, -0.249610702
49575365]
```

```
/var/folders/vv/3nnd1g4506z6vdqnf44fkr2c0000gn/T/ipykernel_89548/1278283725.
py:12: RuntimeWarning: divide by zero encountered in log
  z_1 = np.sqrt(-2*np.log(uniform_1)) * np.cos(2*np.pi*uniform_2)
/var/folders/vv/3nnd1g4506z6vdqnf44fkr2c0000gn/T/ipykernel_89548/1278283725.
py:13: RuntimeWarning: divide by zero encountered in log
  z_2 = np.sqrt(-2*np.log(uniform_1)) * np.sin(2*np.pi*uniform_2)
```

In [915…

```
def polar_marsaglia(seed, sample_size):
    '''This function generates normally distributed random variables using t
    generated_values = []
    true_sample = int(sample_size/2)
    for sample in range(true_sample):
        uniform_1 = uniform_variable_normalized(seed+sample)
        uniform_2 = uniform_variable_normalized(seed+sample+145917)
        v_1 = 2*uniform_1 - 1
        v_2 = 2*uniform_2 - 1
        w = v_1**2 + v_2**2
        if w <= 1:
            z_1 = v_1 * np.sqrt((-2*np.log(w))/w)
            z_2 = v_2 * np.sqrt((-2*np.log(w))/w)
        generated_values.append(z_1)
        generated_values.append(z_2)

    return generated_values


test_2 = box_mueller(1, 5000)
print(test_2[:15])
```

```
[1.4767319648460795, 0.21905249433488735, 0.9190740333004445, 0.919074033300
4443, 0.16503396323575265, 1.1125686084308335, -0.4509574988370361, 0.843682
1396337612, -0.710348645237785, 0.33596988779959164, -0.5871643401386492, -
0.11679424893789352, -0.24127295715551628, -0.2662036859635395, -0.249610702
49575365]
```

```
/var/folders/vv/3nnd1g4506z6vdqnf44fkr2c0000gn/T/ipykernel_89548/1278283725.
py:12: RuntimeWarning: divide by zero encountered in log
  z_1 = np.sqrt(-2*np.log(uniform_1)) * np.cos(2*np.pi*uniform_2)
/var/folders/vv/3nnd1g4506z6vdqnf44fkr2c0000gn/T/ipykernel_89548/1278283725.
py:13: RuntimeWarning: divide by zero encountered in log
  z_2 = np.sqrt(-2*np.log(uniform_1)) * np.sin(2*np.pi*uniform_2)
```

Problem 2

(a) Estimate the following expected values by simulation: $A(t) = E\left(Wt\ 2 + sin(Wt\ )\right)$ and $B(t) = E\left(et\ 2cos(Wt\ )\right)$ for t = 1, 3, 5. Here, Wt is a Standard Wiener Process.

In [999…
```
# simulate wiener process
def wiener_process(t):
    '''This function simulates wiener process starting at 0'''
    dW = np.random.normal(0, 1, 1000) * np.sqrt(t)
    W = dW.mean()
    return W
```

In [100…
```
# simulate a
def simulate_a(t):
    '''This function simulates A(t)'''
    a_t = wiener_process(t)**2 + np.sin(wiener_process(t))
    return a_t

print(f'Simulate A with t=1:', simulate_a(1))
print(f'Simulate A with t=3:', simulate_a(3))
print(f'Simulate A with t=5:', simulate_a(5))
```

```
Simulate A with t=1: -0.00025970264584806996
Simulate A with t=3: -0.04829706081642063
Simulate A with t=5: -0.018755438960304408
```

In [100…
```
# simulate b
def simulate_b(t):
    '''This function simulates B(t)'''
    b_t = np.exp(t/2)*np.cos(wiener_process(t))
    return b_t

print(f'Simulate B with t=1:', simulate_b(1))
print(f'Simulate B with t=3:', simulate_b(3))
print(f'Simulate B with t=5:', simulate_b(5))
```

```
Simulate B with t=1: 1.6486226789168696
Simulate B with t=3: 4.456429723542384
Simulate B with t=5: 12.149800968652029
```

(b) How are the values of $B(t)$ (for the cases t = 1, 3, 5) related?

- the values of B(t) are basically being multiplied by exponent when there is a large number of draws, if we do one draw they will be completely unrelated to each other.

(c) Now use a variance reduction technique (whichever you want) to compute the expected value $B(5)$. Do you see any improvements? Comment on your findings.

- I find that value stays more consistent thoughout simulations

In [100…
```
antithetic_variates = (simulate_b(5) + simulate_b(5))/2
print(f'Antithetic variates values: ', antithetic_variates)
```

```
Antithetic variates values:  12.137111969941653
```

Problem 3

Let St be a Geometric Brownian Motion process: $St = S0e(\sigma Wt+(r-\sigma 2\ )t)$, where $r = 0.055$, $\sigma = 0.2$, $S0 = \$100$; Wt is a Standard Brownian Motion process (Standard Wiener process). (a) Estimate the price c of a European Call option on the stock with $T = 5$, $X = \$100$ by using Monte Carlo simulation.

In [868…
```
def monte_carlo(S0, r, sigma, t, X):
    '''Monte Carlo Function'''
    # generate Z
    W = np.random.normal(0, 1, 10000) * np.sqrt(t)
    # compute the payoff function
    St = S0*np.exp((r - .5 * sigma**2)*t + sigma * W)
    payoff = np.maximum(St - X, 0)
    av_payoff = payoff.mean()
    # calculate the value
    c = np.exp(-r*t)*(av_payoff)
    return c

monte_carlo(100, .055, .2, 5, 100)
```

Out[868…
```
30.34687067583183
```

(b) Compute the exact value of the option c using the Black-Scholes formula. $$C = S_0N(d_1) - Ke^{-rT}N(d_2) = 30.373$$

(c) Use variance reduction techniques (whichever one(s) you want) to estimate the price in part (a) again using the same number of simulations. Did the accuracy improve? Compare your findings and comment.

- Yes, the option price is more accurate

In [812…
```
antithetic_variates_call = (monte_carlo(100, .055, .2, 5, 100) + monte_carlo
print(f'Using antithetic variates: ', antithetic_variates_call)
```

```
Using antithetic variates:  30.69677410226282
```

Problem 4

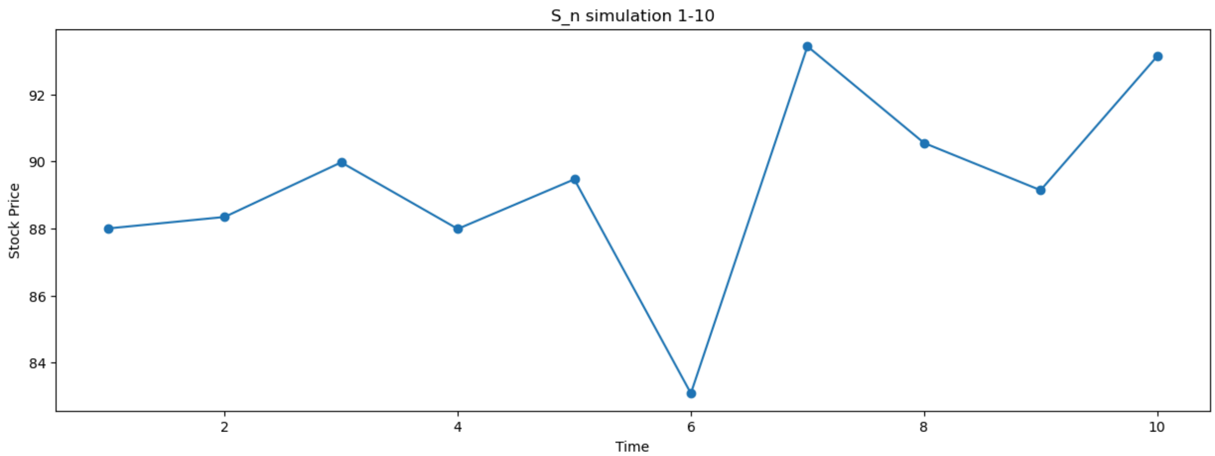
(a) For each integer number n from 1 to 10, use 1,000 simulations of Sn to estimate $E(Sn)$, where St is a Geometric Brownian Motion process: $St = S0e(\sigma Wt+(r-\sigma 2\ 2\ )t)$, where $r = 0.055$, $\sigma = 0.20$, $S0 = \$88$. Plot all of the above $E(Sn\ )$, for n ranging from 1 to 10, in one graph.

In [883…
```
def simulations(range_start, range_end, S0, r, sigma):
    '''GBM simulation function'''
    S = []
    for year in range(range_start-1, range_end):
        W = np.random.normal(0, 1, range_end) * np.sqrt(year)
        St = S0 * np.exp(sigma*W + (r * .5 * sigma**2)*year)
        S.append(St.mean())

    return np.array(S)
```

```
s_values = simulations(1, 10, 88, .2, .055)

plt.figure(figsize=(15, 5))
plt.plot(range(1, 11), s_values, marker='o')
plt.title('S_n simulation 1-10')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.show()
```



(b) Now simulate 3 paths of St for $0 \le t \le 10$ (defined in part (a)) by dividing up the interval [0, 10] into 1,000 equal parts.

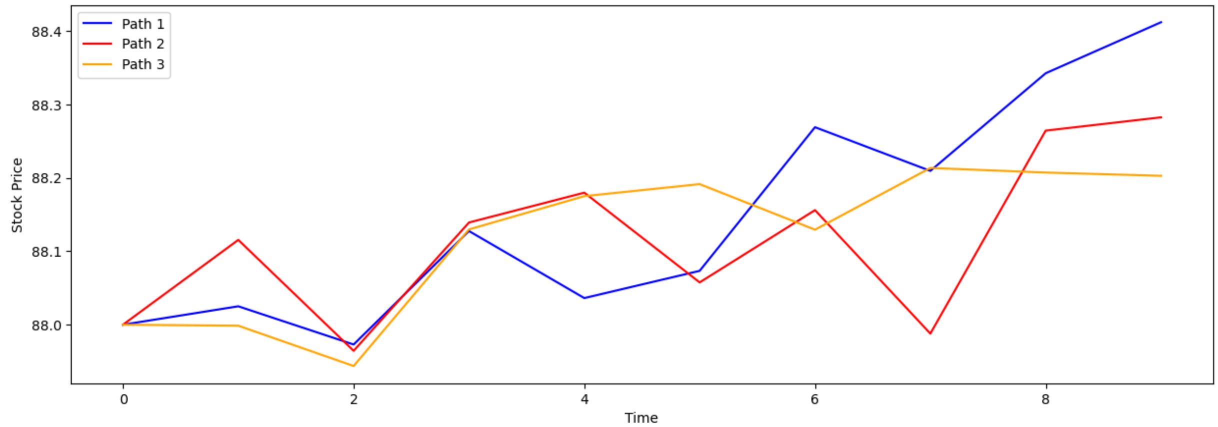
In [887…

```
def simulations_steps(range_start, range_end, S0, r, sigma, steps):
    '''GBM simulation function with steps'''
    S = []
    for year in range(range_start-1, range_end):
        W = np.random.normal(0, 1, range_end) * np.sqrt(year/steps)
        St = S0 * np.exp(sigma*W + (r * .5 * sigma**2)*year)
        S.append(St.mean())

    return np.array(S)

path1 = simulations_steps(1, 10, 88, .2, .055, 1000)
path2 = simulations_steps(1, 10, 88, .2, .055, 1000)
path3 = simulations_steps(1, 10, 88, .2, .055, 1000)

plt.figure(figsize=(15, 5))
plt.plot(path1, label='Path 1', color='blue')
plt.plot(path2, label='Path 2', color='red')
plt.plot(path3, label='Path 3', color='orange')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

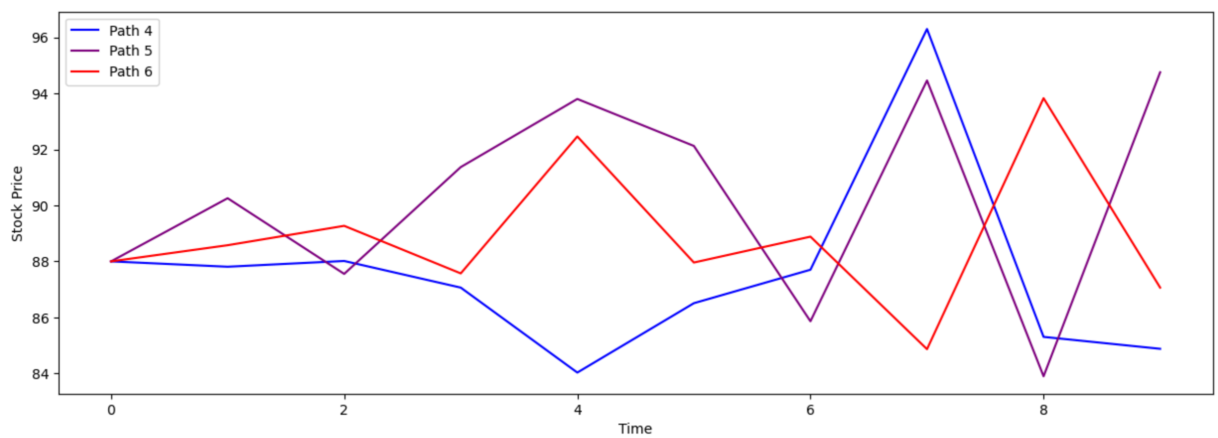


(d) What would happen to the $E(Sn)$ graph if you increased σ from 20% to 30%? What would happen to the 3 plots of St for $0 \leq t \leq 10$, if you increased σ from 20% to 30%?

In [889…

```
path4 = simulations(1, 10, 88, .3, .055)
path5 = simulations(1, 10, 88, .3, .055)
path6 = simulations(1, 10, 88, .3, .055)

plt.figure(figsize=(15, 5))
#plt.plot(s_values, label='Original', color='green')
plt.plot(path4, label='Path 4', color='blue')
plt.plot(path5, label='Path 5', color='purple')
plt.plot(path6, label='Path 6', color='red')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```



Problem 5

(a) Write a code to compute prices of European Call options via Monte Carlo simulation of paths of the stock price process. Use Euler's discretization scheme to discretize the SDE for the stock price process. The code should be generic: for any input of the 5 model parameters - $S0$, T, X, r, σ – the output is the corresponding price of the European call option and the standard error of the estimate.

In [890…
```
def euler_discretization_call_option(S0, T, X, r, sigma):
    '''This function prices European Call Option using Eulers Discretizatior
    # lets use 10000 simulations and 100 steps
    # increments t/100
    dt = T/100
    # create a matrix of zeros, where each row stores simulated path values
    S = np.zeros((10000, 101))
    # set initial value in each row to s0
    S[:, 0] = S0
    # iterate through each time step and simulate a value
    for t in range(1, 101):
        W = np.random.normal(0, 1, 10000) * np.sqrt(dt)
        # Xtk+1 = Xtk + a(Xtk)dt + b(Xtk)W
        S[:, t] = S[:, t-1] + r*S[:, t-1]*dt + sigma*S[:, t-1]*W
    # get the final payoff
    payoff = np.maximum(S[:, -1] - X, 0)
    # discount the payoff
    disc_payoff = np.exp(-r*T)*payoff
    # get mean and standard error
    price = disc_payoff.mean()
    std = disc_payoff.std()/10
    # print price and standard error
    print(f'Estimated Option Price: ', price)
    print(f'Standard Error: ', std)

euler_discretization_call_option(90, 6, 100, .05, .16)
```

```
Estimated Option Price:  22.16638916294021
Standard Error:  3.082548545678942
```

b) Write a code to compute prices of European Call options via Monte Carlo simulation of paths of the stock price process. Use Milshtein's discretization scheme to discretize the SDE for the stock price process. The code should be generic: for any input of the 5 model parameters - $S0$, T, X, r, σ – the output is the corresponding price of the European call option and the standard error of the estimate.

In [891…
```
def millsteins_discretization_call_option(S0, T, X, r, sigma):
    '''This function prices European Call Option using Eulers Discretizatior
    # lets use 10000 simulations and 100 steps
    # increments t/100
    dt = T/100
    # create a matrix of zeros, where each row stores simulated path values
    S = np.zeros((10000, 101))
    # set initial value in each row to s0
    S[:, 0] = S0
    # iterate through each time step and simulate a value
    for t in range(1, 101):
        W = np.random.normal(0, 1, 10000) * np.sqrt(dt)
        # Xtk+1 = Xtk + a(Xtk)dt + b(Xtk)W + 1/2*b(Xtk)*db(Xtk)(W^2 - dt)
        S[:, t] = S[:, t-1] + r*S[:, t-1]*dt + sigma*S[:, t-1]*W + .5*(sigma
    # get the final payoff
    payoff = np.maximum(S[:, -1] - X, 0)
    # discount the payoff
    disc_payoff = np.exp(-r*T)*payoff
```

```
    # get mean and standard error
    price = disc_payoff.mean()
    std = disc_payoff.std()/10
    # print price and standard error
    print(f'Estimated Option Price: ', price)
    print(f'Standard Error: ', std)

millsteins_discretization_call_option(90, 6, 100, .05, .16)
```

```
Estimated Option Price:  22.073173735312448
Standard Error:  3.0965467607222648
```

(c) Write code to compute the prices of European Call options by using the Black - Scholes formula. Use the approximation of $N(\cdot)$ described in Chapter 3. The code should be generic: for any input values of the 5 parameters - $S0$, T, X, r, σ - the output is the corresponding price of the European call option.

In [893…
```
from scipy.stats import norm
def black_scholes_call(S0, X, T, r, sigma):
    # calculate d1 and d2
    d1 = (np.log(S0/X)+(.5*sigma**2)*T)/(sigma*np.sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)
    # calculate N(d1) and N(d2)
    n_d1 = norm.cdf(d1)
    n_d2 = norm.cdf(d2)
    # calculate the price
    bs = ((S0*n_d1)-(X*np.exp(-r*T)*n_d2))
    # append values to the list
    print (f'BS Price: ', bs)

black_scholes_call(90, 6, 100, .05, .16)
```

```
BS Price:  89.39633149624808
```

(d) Use the results of (a) to (c) to compare the two schemes of parts (a) and (b)

(e) Estimate the European call option's greeks – delta (Δ), gamma (Γ), theta (θ), and vega (ν) - and graph them as functions of the initial stock price $S0$. Use X = 100, σ = 0.25, r = 0.055 and T = 0.5 in your estimations. Use the range [95, 105] for $S0$, with a step size of 1. You will have 4 different graphs for each of the 4 greeks. In all cases, dt (time-step) should be user-defined. Use dt = 0.05 as a default value.

- To see more curvature in the graph we need larger range

In [340…
```
import pandas as pd
```

In [911…
```
def greeks(start, stop, X, T, r, sigma):
       '''This function calculates and plots greeks'''
       greeks = []
       for price in range(start, stop+1):
              dt = .05
              # calculate d1 and d2
              d1 = (np.log(price/X)+(r+.5*(sigma**2))*T)/(sigma*np.sqrt(T))
```

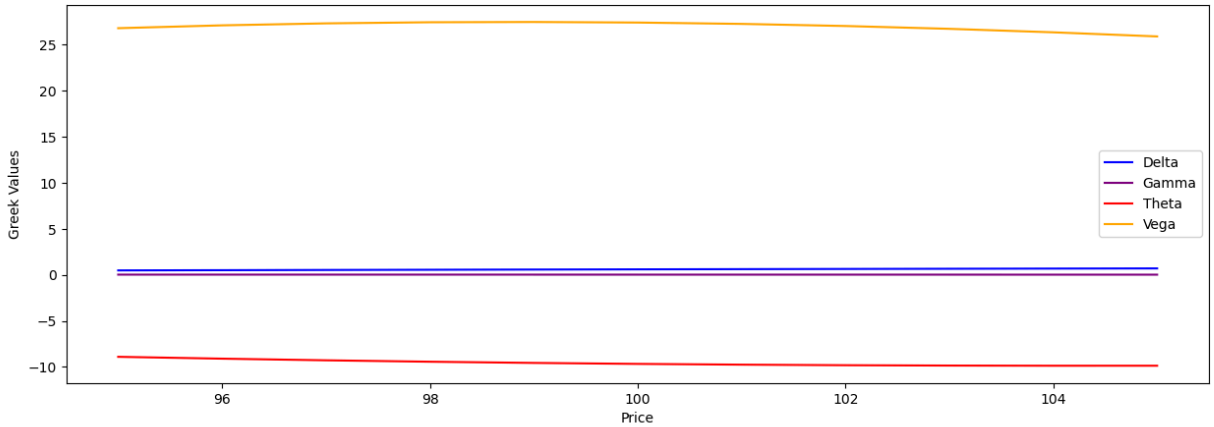
```
            d2 = d1 - sigma*np.sqrt(T)
            # calculate N(d1) and N(d2)
            N_d1 = norm.cdf(d1)
            N_d2 = norm.cdf(d2)
            n_d1 = norm.pdf(d1)
            # calculate delta
            delta = N_d1
            # calculate gamma
            gamma = n_d1/(price*sigma*np.sqrt(T))
            # calculate theta
            theta = (-price*sigma*n_d1)/(2*np.sqrt(T)) - r*X*np.exp(-r*T)*
            # calculate vega
            vega = price*np.sqrt(T)*n_d1
            greeks.append({'Price': price, 'Delta': delta, 'Gamma': gamma,
    df = pd.DataFrame(greeks)
    df.set_index('Price', inplace=True)
    plt.figure(figsize=(15, 5))
    plt.plot(df.index, df['Delta'], label='Delta', color='blue')
    plt.plot(df.index, df['Gamma'], label='Gamma', color='purple')
    plt.plot(df.index, df['Theta'], label='Theta', color='red')
    plt.plot(df.index, df['Vega'], label='Vega', color='orange')
    plt.xlabel('Price')
    plt.ylabel('Greek Values')
    plt.legend()

    return df

greeks(95, 105, 100, .5, .055, .25)
```

Out[911…

Price	Delta	Gamma	Theta	Vega
95	0.481573	0.023730	-8.895928	26.770412
96	0.505197	0.023506	-9.097113	27.078802
97	0.528559	0.023206	-9.274667	27.293050
98	0.551585	0.022835	-9.428345	27.413852
99	0.574208	0.022400	-9.558145	27.442826
100	0.596366	0.021906	-9.664303	27.382437
101	0.618003	0.021359	-9.747268	27.235916
102	0.639069	0.020767	-9.807684	27.007173
103	0.659523	0.020134	-9.846371	26.700703
104	0.679327	0.019469	-9.864299	26.321490
105	0.698451	0.018775	-9.862567	25.874912



Problem 6

Consider the following 2-factor model for stock prices with stochastic volatility:
$dSt = rSt\ dt + \sqrt{Vt}\ St\ dWt1$
$dVt = \alpha(\beta - Vt\ )dt + \sigma\sqrt{Vt}\ dWt2$
where the Brownian Motion processes above are correlated: $dWt1dWt2 = \rho dt$, where the correlation ρ is a constant in $[-1,1]$. Estimate the price of a European Call option (via Monte Carlo simulation) that has a strike price of X and matures in T years. Use the following default parameters of the model: $\rho = -0.6$, $r = 0.055$, $S0$ = \$100, X = \$100, $V0$ = 0.05, $\sigma = 0.42$, $\alpha = 5.8$, $\beta = 0.0625$, $dt = 0.05$, N = 10,000. Use the Full Truncation, Partial Truncation, and Reflection methods, and provide 3 price estimates by using the tree methods

In [527…

```
def full_truncation(rho, r, S0, X, V0, sigma, alpha, beta):
    '''Full Truncation Function'''
    # lets use 10000 simulations and 100 steps
    #lets set t = .5
    T = 5
    dt = .05
    # create a matrix of zeros, where each row stores simulated path values
    S = np.zeros((10000, 101))
    V = np.zeros((10000, 101))
    # set initial value in each row to s0
    S[:, 0] = S0
    V[:, 0] = V0
    # iterate through each time step and simulate a value
    for t in range(1, 101):
        z = np.random.normal(0, 1, 10000)
        z_2 = np.random.normal(0, 1, 10000)
        z_2 = rho * z + np.sqrt(1 - rho**2) * z_2
        V[:, t] = V[:, t-1] + alpha*(beta - np.maximum(V[:, t-1], 0))*dt + s
        S[:, t] = S[:, t-1] + r*S[:, t-1]*dt + np.sqrt(V[:, t-1])*S[:, t-1]*
    # get the final payoff
    payoff = np.maximum(S[:, -1] - X, 0)
    # discount the payoff
    disc_payoff = np.exp(-r*T)*payoff
    # get mean and standard error
```

```
    price = disc_payoff.mean()
    return price
```

In [528…

```
def partial_truncation(rho, r, S0, X, V0, sigma, alpha, beta):
    '''Partial Truncation Function'''
    # lets use 10000 simulations and 100 steps
    #lets set t = .5
    T = 5
    dt = .05
    # increments t/100
    # create a matrix of zeros, where each row stores simulated path values
    S = np.zeros((10000, 101))
    V = np.zeros((10000, 101))
    # set initial value in each row to s0
    S[:, 0] = S0
    V[:, 0] = V0
    # iterate through each time step and simulate a value
    for t in range(1, 101):
        z = np.random.normal(0, 1, 10000)
        z_2 = np.random.normal(0, 1, 10000)
        z_2 = rho * z + np.sqrt(1 - rho**2) * z_2
        V[:, t] = V[:, t-1] + alpha*(beta - V[:, t-1])*dt + sigma*np.sqrt(np
        S[:, t] = S[:, t-1] + r*S[:, t-1]*dt + np.sqrt(V[:, t-1])*S[:, t-1]*
    # get the final payoff
    payoff = np.maximum(S[:, -1] - X, 0)
    # discount the payoff
    disc_payoff = np.exp(-r*T)*payoff
    # get mean and standard error
    price = disc_payoff.mean()
    return price
```

In [529…

```
def reflection_methods(rho, r, S0, X, V0, sigma, alpha, beta):
    '''Reflection Methods Function'''
    # lets use 10000 simulations and 100 steps
    #lets set t = .5
    T = 5
    dt = .05
    # increments t/100
    # create a matrix of zeros, where each row stores simulated path values
    S = np.zeros((10000, 101))
    V = np.zeros((10000, 101))
    # set initial value in each row to s0
    S[:, 0] = S0
    V[:, 0] = V0
    # iterate through each time step and simulate a value
    for t in range(1, 101):
        z = np.random.normal(0, 1, 10000)
        z_2 = np.random.normal(0, 1, 10000)
        z_2 = rho * z + np.sqrt(1 - rho**2) * z_2
        V[:, t] = np.abs(V[:, t-1]) + alpha*(beta - V[:, t-1])*dt + sigma*np
        S[:, t] = S[:, t-1] + r*S[:, t-1]*dt + np.sqrt(V[:, t-1])*S[:, t-1]*
    # get the final payoff
    payoff = np.maximum(S[:, -1] - X, 0)
    # discount the payoff
    disc_payoff = np.exp(-r*T)*payoff
```

```
    # get mean and standard error
    price = disc_payoff.mean()
    return price
```

In [533…

```
def heston_model(rho, r, S0, X, V0, sigma, alpha, beta):
    '''Heston Model Function'''
    # full truncation
    full = full_truncation(rho, r, S0, X, V0, sigma, alpha, beta)
    # partial truncation
    partial = partial_truncation(rho, r, S0, X, V0, sigma, alpha, beta)
    # reflection methods
    reflection = reflection_methods(rho, r, S0, X, V0, sigma, alpha, beta)

    print(f'Full Truncation Value: ', full)
    print(f'Partial Truncation Value: ', partial)
    print(f'Reflection Method Value: ', reflection)

heston_model(.6, .055, 100, 100, .05, .42, 5.8, .0625)
```

```
Full Truncation Value:  80.82246916758398
Partial Truncation Value:  68.81240996744513
Reflection Method Value:  70.21333396719066
```

Problem 7.

The objective of this exercise is to compare a sample of Pseudo-Random numbers with a sample of Quasi-Monte Carlo numbers of $Uniform[0,1]x[0,1]$: Use 2-dimensional Halton sequences to estimate the integral: Default parameter values: N=10,000; (2,3) for bases.

In [550…

```
# define halton sequence function. Code from lecture materials, MATLAB conve
def GetHalton(howmany, base):
    seq = np.zeros(howmany)
    numbits = 1 + int(np.ceil(np.log(howmany)/np.log(base)))
    vetbase = 1/(base ** np.arange(1, numbits+1))
    workvet = np.zeros(numbits)

    for i in range(howmany):
        j = 0
        ok = False
        while not ok:
            workvet[j] += 1
            if workvet[j] < base:
                ok = True
            else:
                workvet[j] = 0
                j += 1
        seq[i] = np.dot(workvet, vetbase)
    return seq

def integral(x, y):
    '''Define Integral'''
    return np.exp(-x * y) * (np.sin(6 * np.pi * x) + np.cbrt(np.cos(2 * np.p

def halton_integral(b1, b2, N):
    '''This function calculates integral value using haltons numbers'''
```

```
    x = GetHalton(N, b1)
    y = GetHalton(N, b2)
    integral_value = np.mean(integral(x, y))
    return integral_value

halton_integral(2, 3, 10000)
```

Out[550… 0.026261973509314397