Question 1.

Closed Form Solution:

$$S_T = S_0 \exp\left\{\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\phi\sqrt{T}\right\}$$

Euler Maruyama Scheme:

$$S_{t+\delta t} \sim S_0\left(1 + r\delta t + \sigma\phi\sqrt{T}\right)$$

Milstein Scheme (E-M Scheme with correction term):

$$S_{t+\delta t} \sim S_0\left(1 + r\delta t + \sigma\phi\sqrt{T} + \frac{1}{2}\sigma^2(\sigma^2 - 1)\delta t\right)$$

[Below is the Question 1 complete Jupyter Notebook, saved as PDF. Explanations and comments are embedded as markdown].

This document is my exam report submitted for Exam 2: Equities and Currencies.
Graded 89.5/100
All workings and codes are independently written from scratch.

OVERVIEW

The first part of the question asks the candidate to compare simulated values of stock price at maturity, using the Closed Form solution, Euler Maruyama Scheme, and Milstein Scheme.

The second part of the question asks the candidate to build an Asian option pricer for calls and puts, taking into account the variation of input parameters.

The third part of the question asks the candidate to compare the value of the Asian option whilst varying the input parameters.

Importing Libraries

In [1]:
```
# Importing libraries
import pandas as pd
from numpy import *
import matplotlib.pyplot as plt

# Set max row to 300
pd.set_option('display.max_rows', 300)
```

PART 1: Comparing Closed Form Solution, EM Scheme, MS Scheme.

First we defining Functions for the Closed Form solution, E-M Scheme and Milstein Scheme

(1) Closed Form Solution

In [2]:
```
# define simulation function
def simulate_path_CF(s0, mu, sigma, horizon, n_sims):

    # set random seed for reproducibility
    random.seed(1000)

    # read parameters
    S0 = s0                 # initial spot price
    r = mu                  # mu = rf in risk neutral framework
    sigma = sigma           # volatility
    T = horizon             # time horizon
    n = n_sims              # number of simulation

    # simulate 'n' asset price path with '2' time points - 0 and T
    S = zeros((2, n))
    S[0] = S0

    w = random.standard_normal(n)
    S[1] = S0*exp((r-sigma**2/2)*T+sigma*(sqrt(T)*w))

    return S
```

(2) Euler Maruyama Scheme

In [3]:
```
# define simulation function
def simulate_path_EM(s0, mu, sigma, horizon, timesteps, n_sims):

    # set random seed for reproducibility
    random.seed(1000)

    # read parameters
    S0 = s0                 # initial spot price
    r = mu                  # mu = rf in risk neutral framework
    vol = sigma           # volatility
    T = horizon             # time horizon
    t = timesteps           # number of time steps
    n = n_sims              # number of simulation

    # define dt
    dt = T/t                # length of time interval

    # simulate 'n' asset price path with 't' timesteps
    S = zeros((t+1,n))
    S[0] = S0

    for i in range(0, t):
```

```
        w = random.standard_normal(n)
        S[i+1] = S[i] * (1 + r*dt + vol*sqrt(dt)*w)

    return S
```

(3) Milstein Scheme

In [4]:
```
# define simulation function
def simulate_path_MS(s0, mu, sigma, horizon, timesteps, n_sims):

    # set random seed for reproducibility
    random.seed(1000)

    # read parameters
    S0 = s0                 # initial spot price
    r = mu                  # mu = rf in risk neutral framework
    vol = sigma           # volatility
    T = horizon             # time horizon
    t = timesteps           # number of time steps
    n = n_sims              # number of simulation

    # define dt
    dt = T/t                # length of time interval

    # simulate 'n' asset price path with 't' timesteps
    S = zeros((t+1,n))
    S[0] = S0

    for i in range(0, t):
        w = random.standard_normal(n)
        S[i+1] = S[i] * (1 + r*dt + vol*sqrt(dt)*w + vol**2*(w**2-1)*dt/2)

    return S
```

In [5]:
```
## Assign simulated price path to check function is working as expected
price_path_CF = pd.DataFrame(simulate_path_CF(100,0.05,0.2,1,1000))
price_path_EM = pd.DataFrame(simulate_path_EM(100,0.05,0.2,1,20,1000))
price_path_MS = pd.DataFrame(simulate_path_MS(100,0.05,0.2,1,20,1000))
```

In [6]:
```
price_path_CF.head()
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	...	
0	100.000000	100.000000	100.00000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	...	100.0
1	87.731282	109.876444	102.52161	117.217942	97.029092	111.393068	100.854887	93.613403	116.068123	93.900591	...	93.9

2 rows × 1000 columns

In [7]:
```
price_path_EM.head()
```

Out[7]:

	0	1	2	3	4	5	6	7	8	9	...	
0	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	...	100
1	96.652353	101.685250	100.136037	103.131504	98.904796	101.991783	99.769526	98.103450	102.911080	98.171944	...	9
2	96.735080	96.937133	106.083085	101.704968	105.510414	103.904861	100.532006	90.221314	108.549984	101.142612	...	95
3	99.109860	96.271501	96.986034	106.501537	96.621627	99.305448	111.759193	90.806013	109.941224	102.017823	...	97
4	106.312144	98.657092	103.648377	107.572122	100.293693	97.639413	104.817255	88.631427	111.060974	108.898408	...	96

5 rows × 1000 columns

In [8]:
```
price_path_MS.head()
```

Out[8]:

	0	1	2	3	4	5	6	7	8	9	...	
0	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	...	100
1	96.617068	101.595549	100.036102	103.073019	98.813844	101.906952	99.670680	98.026489	102.846486	98.093535	...	98
2	96.603278	96.872960	106.039059	101.557966	105.518768	103.729998	100.334052	90.388903	108.519630	101.001533	...	95
3	98.901702	96.115146	97.252701	106.347331	96.920761	99.148026	112.036777	90.885015	109.807737	101.776435	...	97
4	106.233444	98.424602	104.049172	107.313052	100.568363	97.403906	105.199444	88.649438	110.819562	108.753600	...	96

5 rows × 1000 columns

Next, we compare resultsbetween the CF Solution, EM Scheme and MS Scheme

To do this, we define a function that takes in the asset data, simulation parameters, the variable to vary (timesteps number of simulations), and the list of values for comparison.

The function outputs a dataframe of the asset value at maturity and volatility at various values of the comparison variable (timesteps number of simulations).

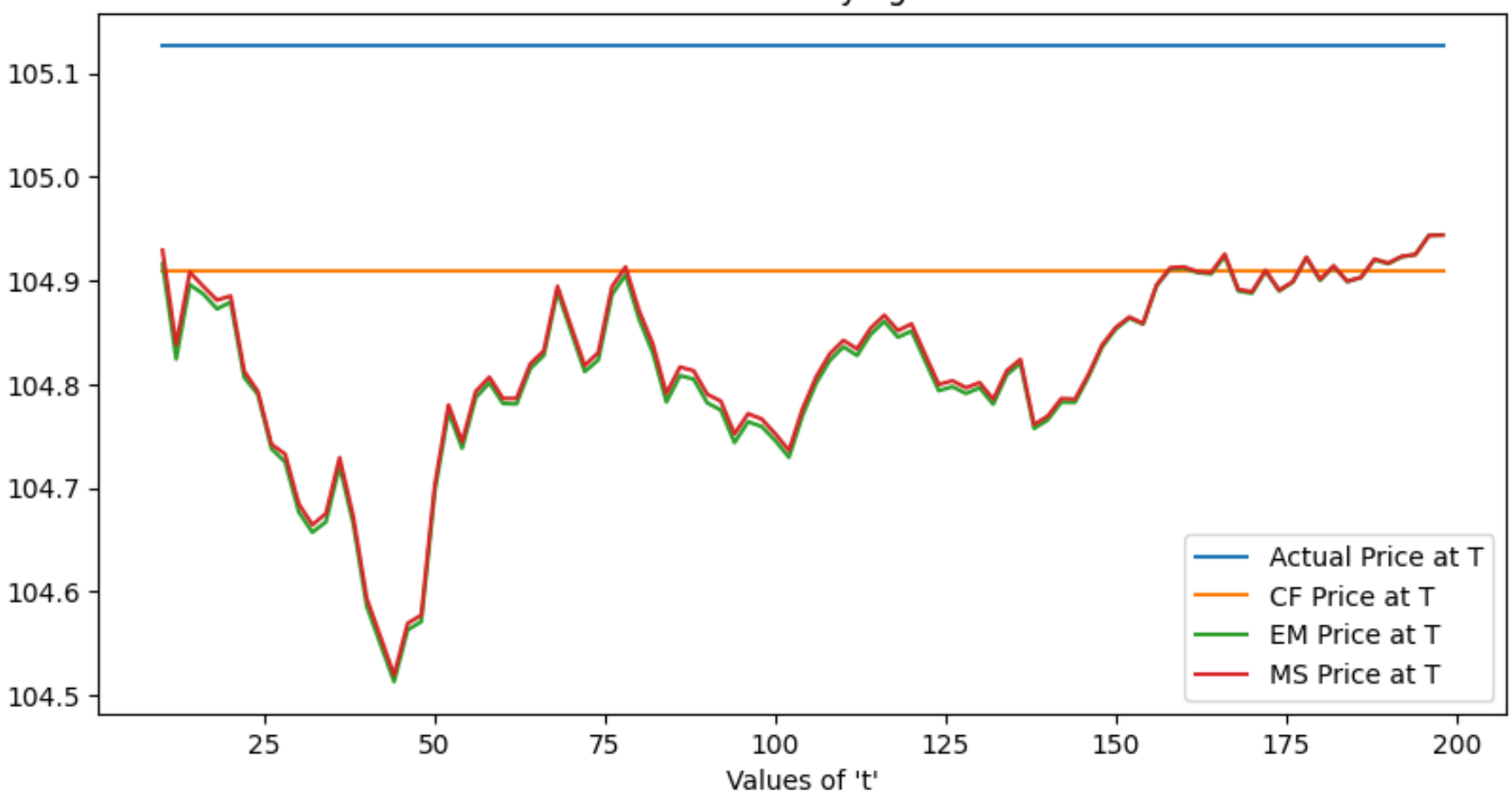
In [9]:
```
def compare_schemes(s0, mu, sigma, horizon, compare, val_list, timesteps=20, n_sims=1000):


    '''
    Compares the price and volatility at maturity between
    (1) The Closed Form Solution,
    (2) Euler Maruyama Scheme,
    (3) MilStein Scheme.

    Parameters:
    s0:        Initial spot price,
    mu:        mu = rf in risk neutral framework,
    sigma:     volatility of the underlying asset,
    horizon:   Time horizon,

    compare:   The variable to compare, takes in a string value -
               "n" for number of simulations,
                "t" for number of time steps

    val_list:  The list of values to compare.

    timesteps: Number of time steps - default value = 20,
    n_sims:    Number of simulation - default value = 1000,

    Output:
    Outputs a dataframe that stores in each column:

    (1) The list of values assigned to the comparison variable,
        i.e., values in the input val_list
    (2) The price at time T simulated using the Closed Form solution
    (3) The volatility at time T simulated using the Closed Form solution
    (4) The price at time T simulated using the Euler Maruyama Scheme
    (5) The volatility at time T simulated using the Euler Maruyama Scheme
    (6) The price at time T simulated using the Milstein Scheme
    (7) The volatility at time T simulated using the Milstein Scheme

    The function also plots two graphs:
    (1) Comparing the calculated price at time T using
        the three schemes vs the actual (theoretical) price at time T
    (2) Comparing the calculated volatility using
        the three schemes vs the actual (theoretical) price at time T

    '''

    # set random seed for reproducibility
    random.seed(10000)

    # read parameters
    S0 = s0                 # initial spot price
    r = mu                  # mu = rf in risk neutral framework
    vol = sigma             # volatility
    T = horizon             # time horizon

    if compare.lower() != "t" and compare.lower() != "n":
        return "Invalid input for the 'compare' parameter."

    # Comparing results whilst varying timesteps "t"
    if compare.lower() == "t":

        # Calculating actual forward price at maturity & actual volatility
        act_St = [S0*exp(r*T) for timestep in val_list]
        act_vol = [sigma for timestep in val_list]

        # Calculating forward price at maturity & actual volatility using
        # closed form solution
        CF_St = pd.DataFrame(simulate_path_CF(S0,r,vol,T,n_sims)).iloc[-1].mean()
        CF_vol = log(simulate_path_CF(S0,r,vol,T,n_sims)[-1]/S0).std()

        # Calculating forward price at maturity & actual volatility using
        # Euler Maruyama Scheme
```

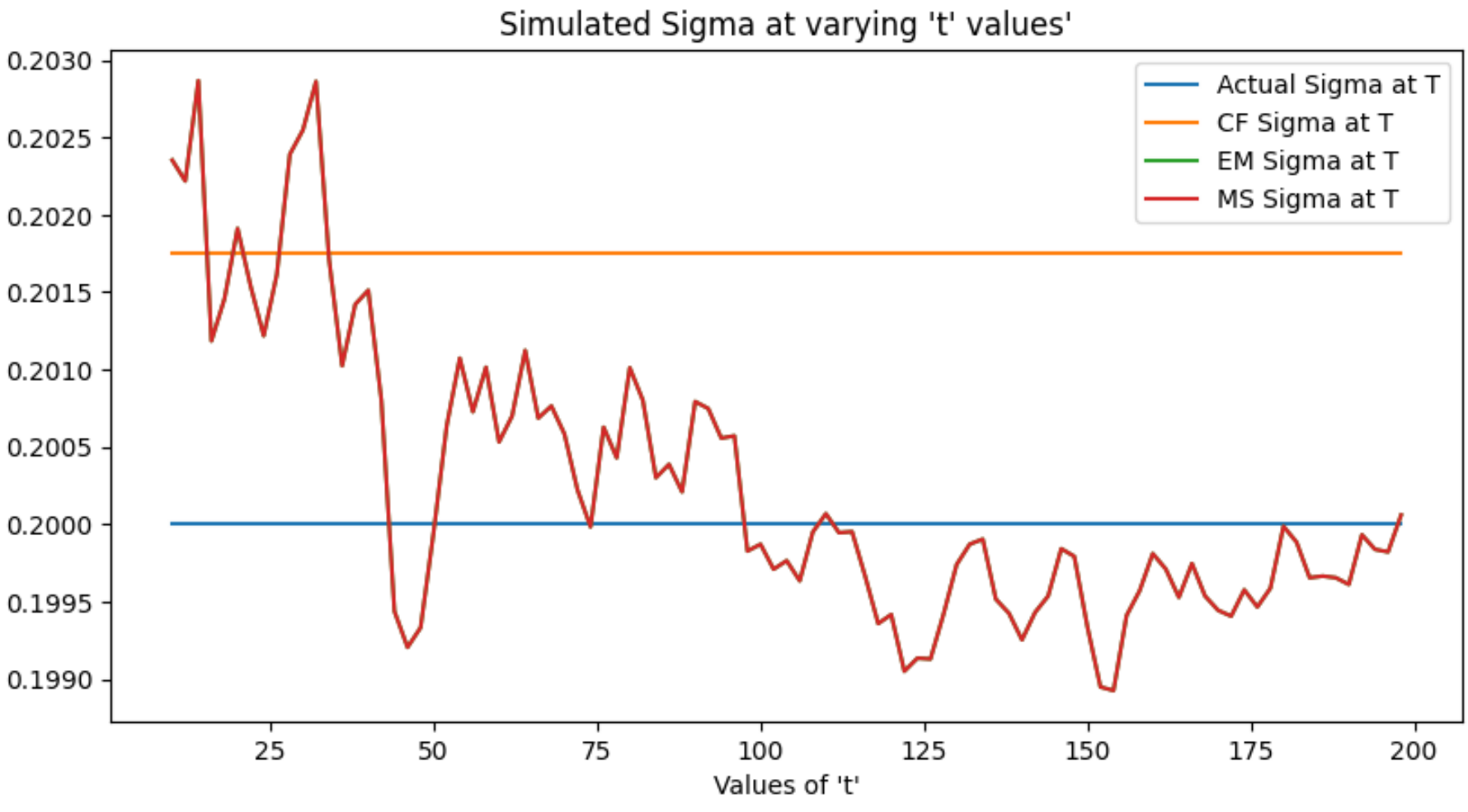
```
        EM_St = [pd.DataFrame(simulate_path_EM(S0,r,vol,T,timestep,n_sims)).iloc[-1].mean()
                        for timestep in val_list]
        EM_vol = [log(simulate_path_EM(S0,r,vol,T,timestep,n_sims)[-1]/S0).std()
                        for timestep in val_list]

        # Calculating forward price at maturity & actual volatility using
        # Milstein Scheme
        MS_St = [pd.DataFrame(simulate_path_MS(S0,r,vol,T,timestep,n_sims)).iloc[-1].mean()
                        for timestep in val_list]
        MS_vol = [log(simulate_path_EM(S0,r,vol,T,timestep,n_sims)[-1]/S0).std()
                        for timestep in val_list]

    # Comparing results whilst varying number of simulations "n"
    if compare.lower() == "n":

        # Calculating actual forward price at maturity & actual volatility
        act_St = [S0*exp(r*T) for n in val_list]
        act_vol = [sigma for n in val_list]

        # Calculating forward price at maturity & actual volatility using
        # closed form solution
        CF_St = [pd.DataFrame(simulate_path_CF(S0,r,vol,T,n)).iloc[-1].mean()
                            for n in val_list]
        CF_vol = [log(simulate_path_CF(S0,r,vol,T,n)[-1]/S0).std()
                            for n in val_list]

        # Calculating forward price at maturity & actual volatility using
        # Euler Maruyama Scheme
        EM_St = [pd.DataFrame(simulate_path_EM(S0,r,vol,T,timesteps,n)).iloc[-1].mean()
                            for n in val_list]
        EM_vol = [log(simulate_path_EM(S0,r,vol,T,timesteps,n)[-1]/S0).std()
                            for n in val_list]

        # Calculating forward price at maturity & actual volatility using Milstein Scheme
        MS_St = [pd.DataFrame(simulate_path_MS(S0,r,vol,T,timesteps,n)).iloc[-1].mean()
                            for n in val_list]
        MS_vol = [log(simulate_path_MS(S0,r,vol,T,timesteps,n)[-1]/S0).std()
                            for n in val_list]

    results = pd.DataFrame.from_dict({
        f"Values of '{compare.lower()}'": val_list,
        "Actual Price at T": act_St,
        "Actual Sigma at T": act_vol,
        "CF Price at T": CF_St,
        "CF Sigma at T": CF_vol,
        "EM Price at T": EM_St,
        "EM Sigma at T": EM_vol,
        "MS Price at T": MS_St,
        "MS Sigma at T": MS_vol,
    })

    # Actual price and sigma at time T
    ax1 = results.plot(x=f"Values of '{compare.lower()}'", y='Actual Price at T')
    results.plot(ax=ax1, x=f"Values of '{compare.lower()}'", y='CF Price at T')
    results.plot(ax=ax1, x=f"Values of '{compare.lower()}'", y='EM Price at T')
    results.plot(ax=ax1, x=f"Values of '{compare.lower()}'", y='MS Price at T')
    ax1.set_title(f"Simulated St at varying '{compare.lower()}' values'")
    box1 = ax1.get_position()
    ax1.set_position([box1.x0, box1.y0, box1.width * 1.5, box1.height]);

    ax2 = results.plot(x=f"Values of '{compare.lower()}'", y='Actual Sigma at T')
    results.plot(ax=ax2, x=f"Values of '{compare.lower()}'", y='CF Sigma at T')
    results.plot(ax=ax2, x=f"Values of '{compare.lower()}'", y='EM Sigma at T')
    results.plot(ax=ax2, x=f"Values of '{compare.lower()}'", y='MS Sigma at T')
    ax2.set_title(f"Simulated Sigma at varying '{compare.lower()}' values'")
    box2 = ax2.get_position()
    ax2.set_position([box2.x0, box2.y0, box2.width * 1.5, box2.height]);

    return results
```

Comparing the three schemes by varying MC Simulation parameters

(1) Varying timesteps

In [10]:
```
# Vary the timesteps to see the effect on simulated forward price

compare_timestep = compare_schemes(100, 0.05, 0.2, 1, "t",
                list(range(10, 200, 2)),
                timesteps=20, n_sims=10000)
```





Comment:

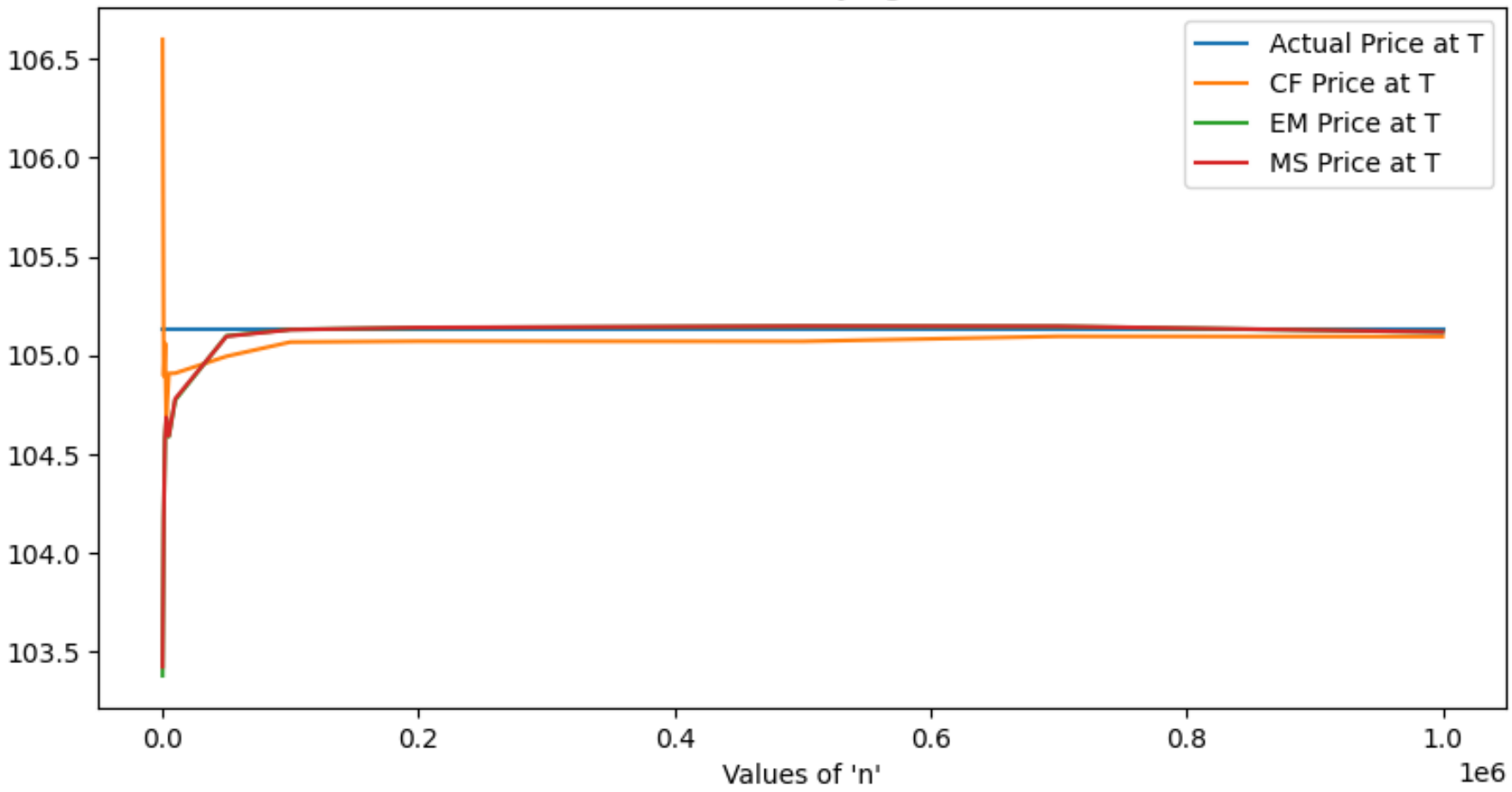
We can see that generally the forward stock price and sigma from all three simulations are reasonably close to the actual values. As 't' becomes larger, the forward price tends toward the actual result and the volatility fluctuates around the actual result. In both cases, the error is satisfactorily small.

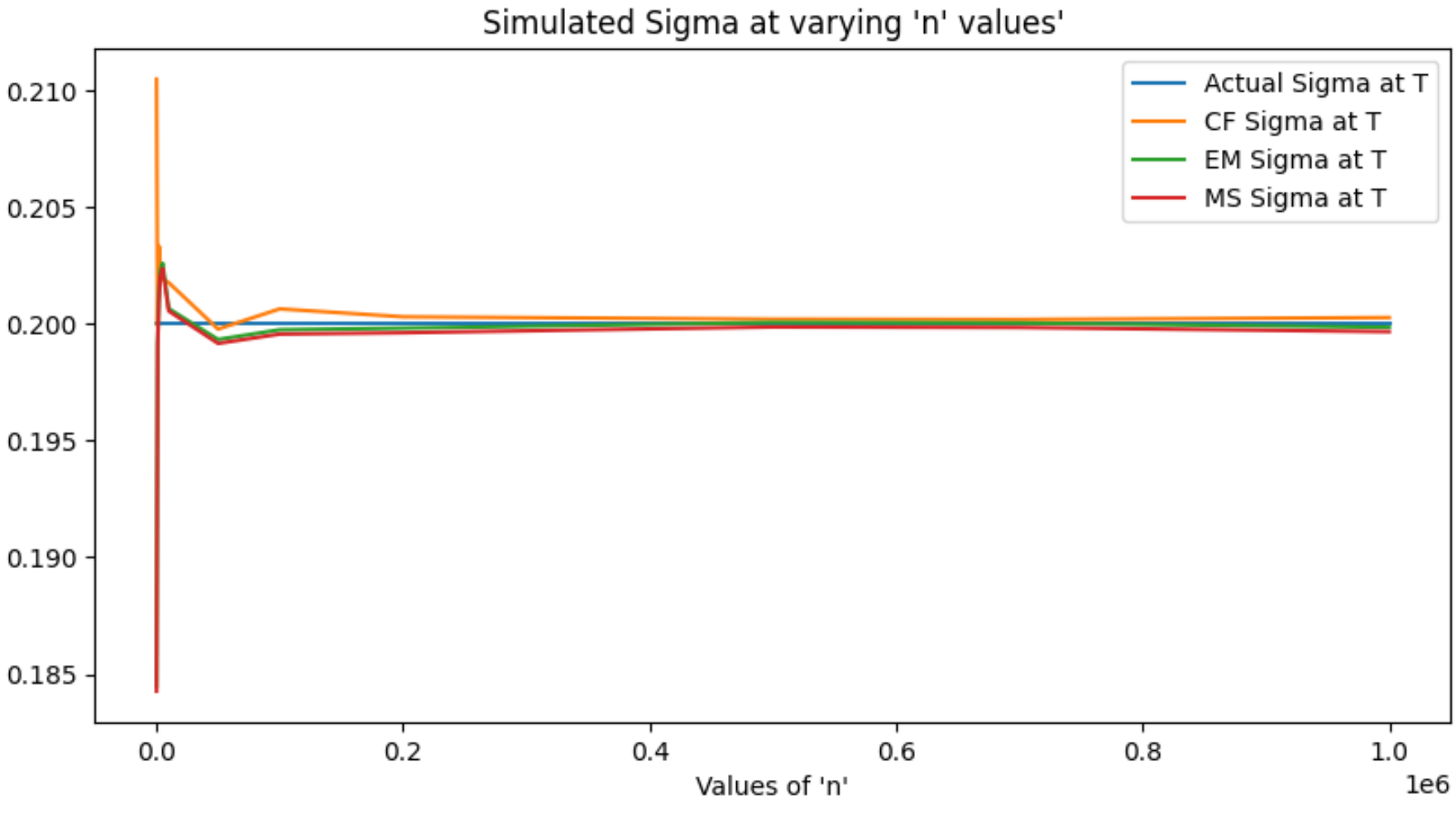
The Euler Maruyama Scheme and the Milstein Scheme have similar values. For the forward price, we see that they quite clearly converge as timesteps increase, for the volatility, the values seem to be very close throughout.

We can informally conclude that as the time step increases, the subsequent results from the simulated paths become more accurate.

(2) Varying the number of simulations

In [11]:
```
# Vary the number of simulations to see the effect on simulated forward price

comepare_n = compare_schemes(100, 0.05, 0.2, 1, "n",
                [100, 1000, 2000, 3000, 5000, 10000, 50000,
                 100000, 200000, 500000, 700000, 1000000],
                timesteps=52, n_sims=10000)
```





Comments:

Similar to above, generally the forward stock price and sigma from all three simulations converge to the actual value as the number of simulations increase.

The results from the EM Scheme and the Milstein Scheme are very close and follow the same trends.

We can conclude that as the number of simulations increase, the subsequent result becomes more accurate.

PART 2: Asian Option Pricer

Asian Option Pricer Function

First we create an Asian Option pricer function.

In [12]:
```
def price_asian(s0=100, mu=0.05, sigma=0.2, horizon=1,
                timesteps=252, n_sims=10000, strike=100,
                OpType="C", StrikeType="fixed",
                SampleType="C", AvgType="A", AvgStep=-1):

    '''
```

```
    Prices an Asian option with the specifications given in parameter in puts.

    Parameters:
    s0:           Initial spot price - default value = 100
    mu:           mu = rf in risk neutral framework - default value = 0.05
    sigma:        volatility of the underlying asset - default value = 0.2
    horizon:      Time horizon - default value = 9
    timesteps:    Number of time steps - default value = 20,
    n_sims:       Number of simulation - default value = 1000,
    strike:       Strike of the option (at fixed strike) - default value = 100,

    OpType:       Type of the option, takes one of two values, case insensitive:
                  (1) "C" for call,
                  (2) "P" for put.

    StrikeType:   Type of the strike, takes one of two values, case insensitive:
                  (1) "Fixed" for fixed strike,
                  (2) "Floating" for floating strike.

    SampleType:   Type of sampling method, takes one of two values, case insensitive:
                  (1) "D" for discrete sampling,
                  (2) "C" for continuous sampling.

    AvgType:      Type of averaging method used, takes one of two values, case insensitive:
                  (1) "A" for arithmetic average,
                  (2) "G" for geometric average.

    AvgStep:      If SampleType = "D" (discrete sampling), specifies the step \
                  difference between each sampled point.
                  e.g. AvgStep = 3, then the sample is S(t), S(t-3), S(t-6)...

    Output:       Output the price of the option specified.



    '''

    # implementing checks

    if not OpType.lower() in ['c','p']:
        return "Invalid option type - please choose between 'C' or 'P'."

    if not StrikeType.lower() in ['fixed','floating']:
        return "Invalid strike type - please choose between 'fixed' or 'floating'."

    if not SampleType.lower() in ['d','c']:
        return "Invalid sampling method - please choose between 'D' or 'C'."

    if not AvgType.lower() in ['a','g']:
        return "Invalid option type - please choose between 'A' or 'G'."

    # set random seed for reproducibility
    random.seed(1000)

    # read parameters
    S0 = s0                 # initial spot price
    r = mu                  # mu = rf in risk neutral framework
    vol = sigma             # volatility
    T = horizon             # time horizon
    t = timesteps           # number of time steps
    n = n_sims              # number of simulation
    K = strike              # the Strike price


    # simulate asset path using the Milstein Scheme path generation function defined
    SPath = simulate_path_MS(S0,r,vol,T,t,n)

    # calculating averages in accordance with SampleType & AvgType & AvgStep
    if SampleType.lower() == "c":

        if AvgType.lower() == "a":
            avg = mean(SPath, axis=0)
        elif AvgType.lower() == "g":
            avg = exp(mean(log(SPath), axis=0))

    elif SampleType.lower() == "d":

        if AvgType.lower() == "a":
            avg = mean(SPath[list(range(t,-1,AvgStep))], axis=0)
        elif AvgType.lower() == "g":
```

```
            avg = exp(mean(log(SPath[list(range(t,-1,AvgStep))]), axis=0))

    # replacing S or K with the calculated average, in accordance with StrikeType
    if StrikeType.lower() == 'fixed':
        S = avg

    elif StrikeType.lower() == 'floating':
        S = SPath[-1]
        K = avg

    # calculating discounted value of the expected payoff

    C0 = exp(-r*T) * mean(maximum(0, S-K))
    P0 = exp(-r*T) * mean(maximum(0, K-S))

    # output price depending on the OpType

    if OpType.lower() == "c":
        return C0

    if OpType.lower() == "p":
        return P0
```

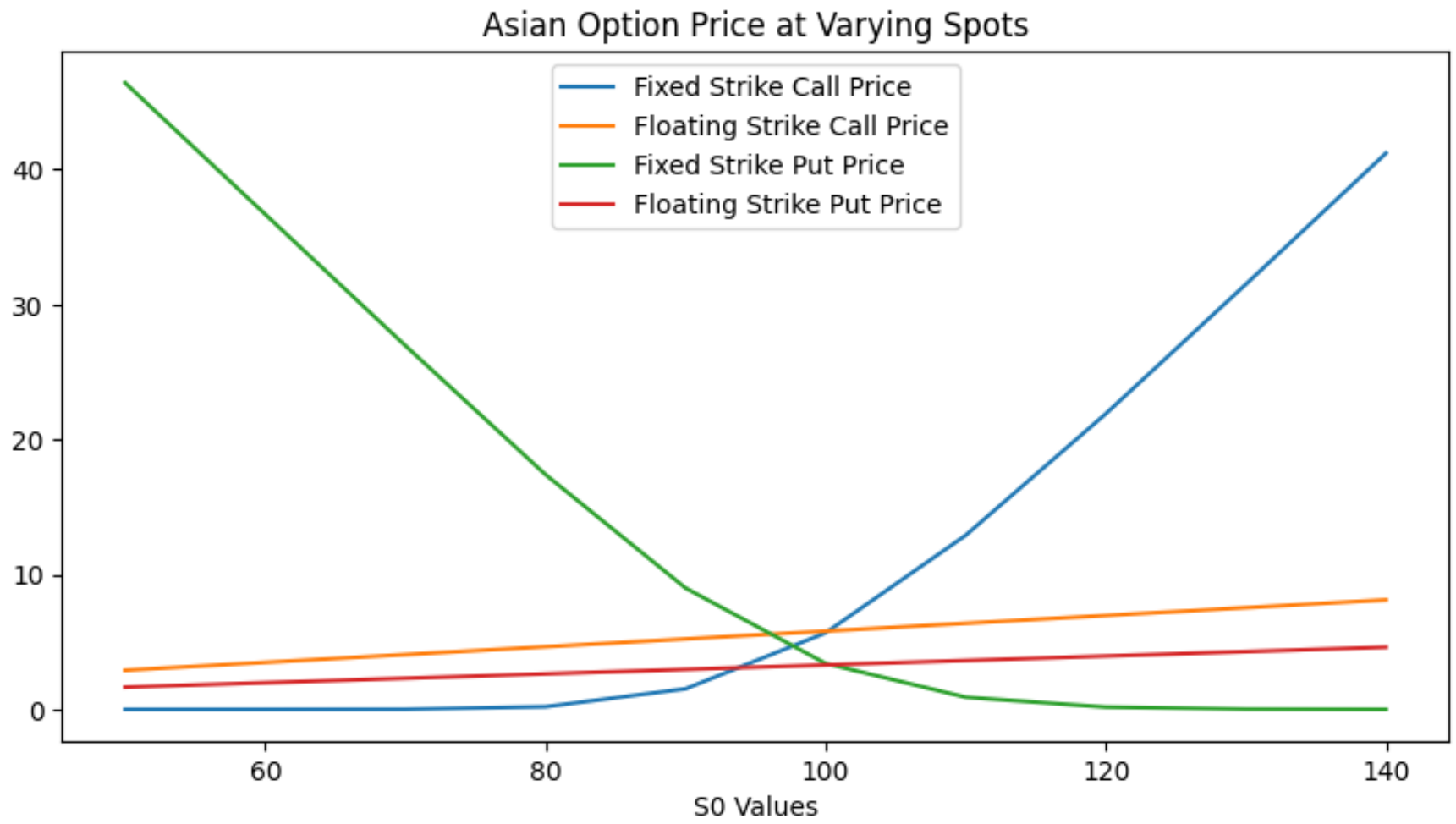
PART 3: Asian Option Price at Varying Parameters

(1) Varying the Spot S0

Holding fixed: sigma=0.2, mu=0.05, horizon=1, timesteps=252, strike=100, SampleType='C', AvgType='A'

In [13]:

```
# Values of S0
S0_vals = list(range(50, 150, 10))

# Fixed Strike Call
S0_FX_C = [price_asian(s0=val, OpType="C", StrikeType="fixed") for val in S0_vals]

# Floating Strike Call
S0_FL_C = [price_asian(s0=val, OpType="C", StrikeType="floating") for val in S0_vals]

# Fixed Strike Put
S0_FX_P = [price_asian(s0=val, OpType="P", StrikeType="fixed") for val in S0_vals]

# Floating Strike Put
S0_FL_P = [price_asian(s0=val, OpType="P", StrikeType="floating") for val in S0_vals]

# Create dataframe for varying S0 values and price calculated at each S0
S0_df = pd.DataFrame.from_dict({
    'S0 Values': S0_vals,
    'Fixed Strike Call Price': S0_FX_C,
    'Floating Strike Call Price': S0_FL_C,
    'Fixed Strike Put Price': S0_FX_P,
    'Floating Strike Put Price': S0_FL_P,
})


# Plot results
ax = S0_df.plot(x='S0 Values', y='Fixed Strike Call Price');
S0_df.plot(ax=ax, x='S0 Values', y='Floating Strike Call Price');
S0_df.plot(ax=ax, x='S0 Values', y='Fixed Strike Put Price');
S0_df.plot(ax=ax, x='S0 Values', y='Floating Strike Put Price');
ax.set_title("Asian Option Price at Varying Spots");
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height]);
```



Interpretation

Fixed Strike Call

As the Spot at time 0 increases, the option price increases. Because a larger starting pushes up the end value at maturity, hence increasing the value of S - K. Moreover, larger values of St gives a larger diffusion of the geometric Brownian Motion, which also contributes to a larger expected value of ST, further increasing the call payoff.

Fixed Strike Put

As the Spot at time 0 increases, the option price decreases. Because a larger starting pushes up the end value at maturity, hence decreasing the value of K - S. Moreover, larger values of St gives a larger diffusion of the geometric Brownian Motion, which also contributes to a larger expected value of ST, further decreasing the put payoff.

Floating Strike Call

As the Spot at time 0 increases, the option price slowly increases. The effects of the previous two products is heavily diminished as now S and K move together. However, we can still see a slower increase in the option price as a larger S0 still gives a larger diffusion to the geometric Brownian Motion, which contributes to a larger average of ST. we can still see a slower increase in the option price as a larger S0 still gives a larger diffusion to the geometric Brownian Motion, meaning that at maturity the distribution of ST becomes more widely dispersed, producing a larger expected payoffs S-K.

Floating Strike Put

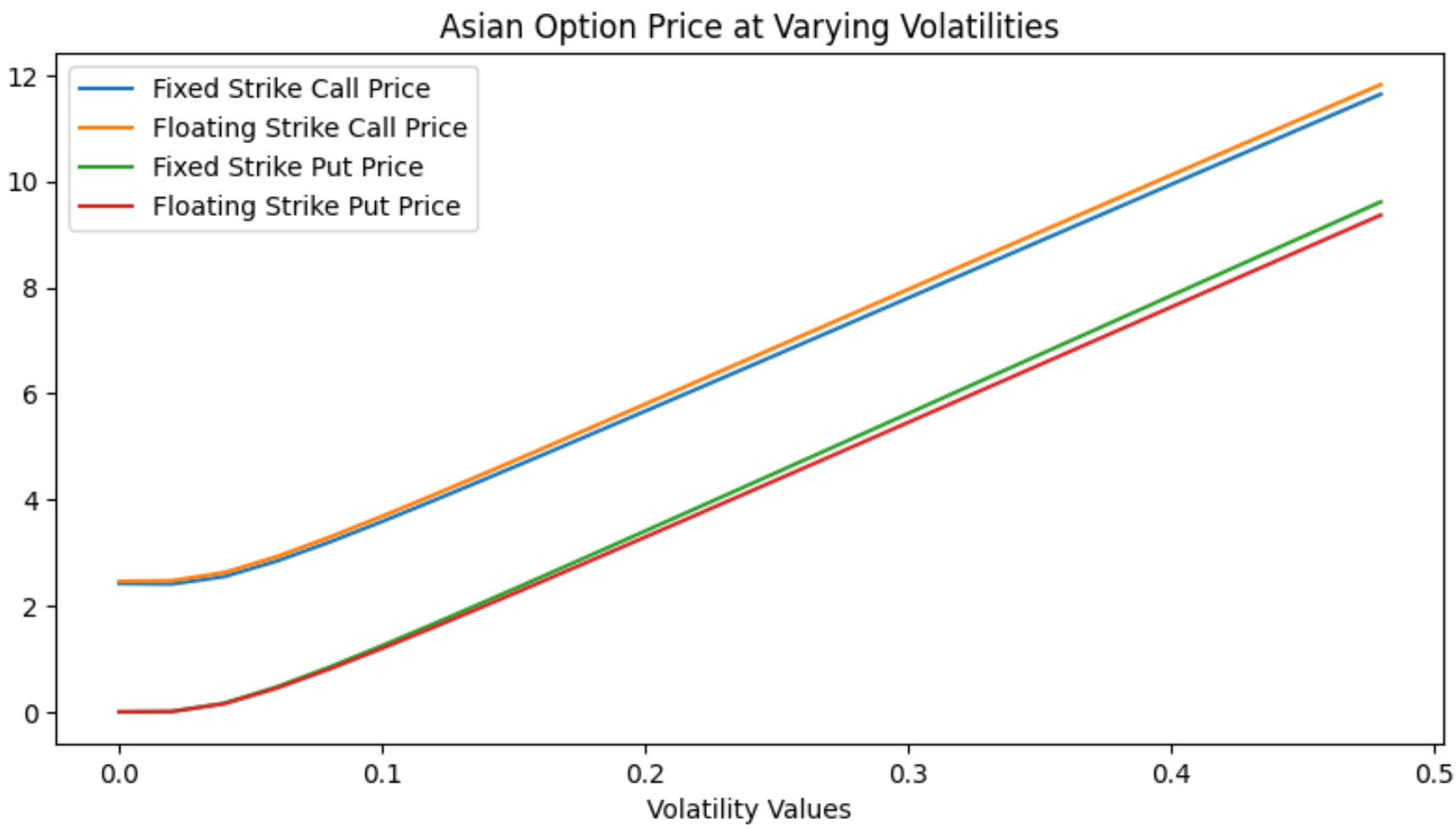
As the Spot at time 0 increases, the option price slowly increases. The effects of the previous two products is heavily diminished as now S and K move together. However, we can still see a slower increase in the option price as a larger S0 still gives a larger diffusion to the geometric Brownian Motion, which contributes to a larger average of ST. we can still see a slower increase in the option price as a larger S0 still gives a larger diffusion to the geometric Brownian Motion, meaning that at maturity the distribution of ST becomes more widely dispersed, producing a larger expected payoffs K-S.

(2) Varying the Volatility

Holding fixed: S0=100, mu=0.05, horizon=1, timesteps=252, strike=100, SampleType='C', AvgType='A'

In [14]:
```
# Values of Volatility (sigma)
vol_vals = list(arange(0, 0.5, 0.02))

# Fixed Strike Call
vol_FX_C = [price_asian(sigma=val, OpType="C", StrikeType="fixed") for val in vol_vals]

# Floating Strike Call
vol_FL_C = [price_asian(sigma=val, OpType="C", StrikeType="floating") for val in vol_vals]

# Fixed Strike Put
vol_FX_P = [price_asian(sigma=val, OpType="P", StrikeType="fixed") for val in vol_vals]
```

```
# Floating Strike Put
vol_FL_P = [price_asian(sigma=val, OpType="P", StrikeType="floating") for val in vol_vals]

# Create dataframe for varying volatility values and price calculated at each S0
vol_df = pd.DataFrame.from_dict({
    'Volatility Values': vol_vals,
    'Fixed Strike Call Price': vol_FX_C,
    'Floating Strike Call Price': vol_FL_C,
    'Fixed Strike Put Price': vol_FX_P,
    'Floating Strike Put Price': vol_FL_P,
})

# Plot results
ax = vol_df.plot(x='Volatility Values', y='Fixed Strike Call Price');
vol_df.plot(ax=ax, x='Volatility Values', y='Floating Strike Call Price');
vol_df.plot(ax=ax, x='Volatility Values', y='Fixed Strike Put Price');
vol_df.plot(ax=ax, x='Volatility Values', y='Floating Strike Put Price');
ax.set_title("Asian Option Price at Varying Volatilities");
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height]);
```



Interpretation

For all four product categories:

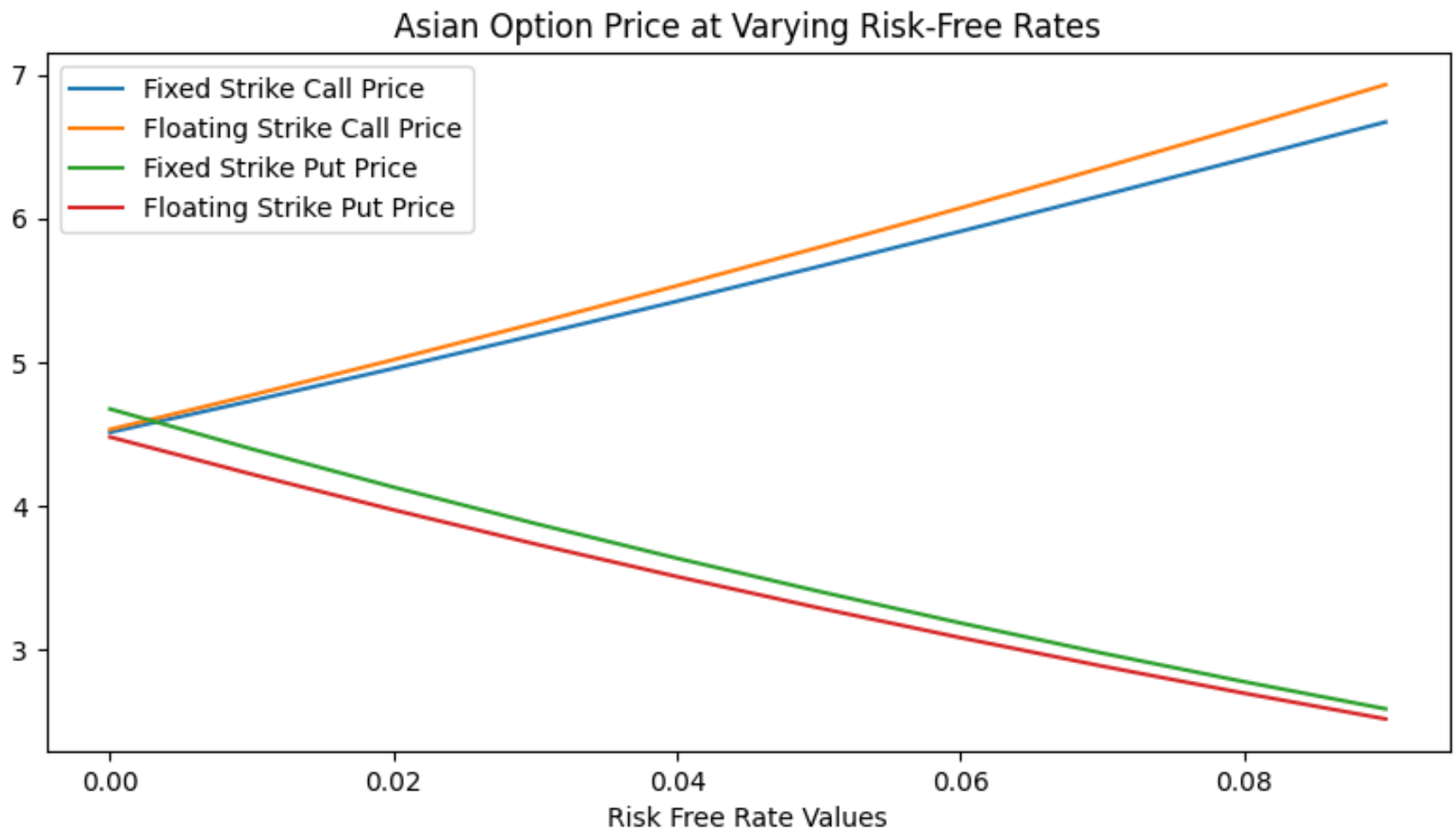
As the volatility increases, the option price increases. Options are generally long volatility products since the payoff of the option depends on how much the asset moves. i.e., the wider the diffusion, the more likely that the final S-K (for calls) and K-S (for puts) can have larger values.

(3) Varying the Risk Free rate

Holding fixed: S0=100, sigma=0.2, horizon=1, timesteps=252, strike=100, SampleType='C', AvgType='A'

In [15]:
```
# Values of Risk Free Interest Rate (mu)
r_vals = list(arange(0, 0.1, 0.01))

# Fixed Strike Call
r_FX_C = [price_asian(mu=val, OpType="C", StrikeType="fixed") for val in r_vals]

# Floating Strike Call
r_FL_C = [price_asian(mu=val, OpType="C", StrikeType="floating") for val in r_vals]

# Fixed Strike Put
r_FX_P = [price_asian(mu=val, OpType="P", StrikeType="fixed") for val in r_vals]

# Floating Strike Put
r_FL_P = [price_asian(mu=val, OpType="P", StrikeType="floating") for val in r_vals]

# Create dataframe for varying risk free rate values and price calculated at each S0
```

```
r_df = pd.DataFrame.from_dict({
    'Risk Free Rate Values': r_vals,
    'Fixed Strike Call Price': r_FX_C,
    'Floating Strike Call Price': r_FL_C,
    'Fixed Strike Put Price': r_FX_P,
    'Floating Strike Put Price': r_FL_P,
})

# Plot results
ax = r_df.plot(x='Risk Free Rate Values', y='Fixed Strike Call Price');
r_df.plot(ax=ax, x='Risk Free Rate Values', y='Floating Strike Call Price');
r_df.plot(ax=ax, x='Risk Free Rate Values', y='Fixed Strike Put Price');
r_df.plot(ax=ax, x='Risk Free Rate Values', y='Floating Strike Put Price');
ax.set_title("Asian Option Price at Varying Risk-Free Rates");
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height]);
```



Interpretation

Fixed Strike Call

As the Risk Free Rate increases, the option price increases. Because a higher interest rate pushes up the forward price at time T. Thereby decreasing the payoff of the call.

Fixed Strike Put

As the Risk Free Rate increases, the option price decreases. Because a higher interest rate pushes up the forward value at time T. Thereby decreasing the payoff of the put.

Floating Strike Call

As the Risk Free Rate increases, the option price increases. A higher interest rate will on average push values of the underlying asset throughout the time horison, therefore the difference between a sampled average and the value at maturity (S - K) will become larger. This can be easily thought of using a simple example of the sampled asset values being S(t-4) = 10, S(t-2) = 20, S(t) = 30: initially the payoff is 30 - 20, where 20 is the arithmetic mean of the three samples. If interest increases and now each of these sampled point values become 11, 22, 33, then the arithmetic mean now becomes 22, which makes the payoff 33 - 22 = 11 > 10.

Floating Strike Put

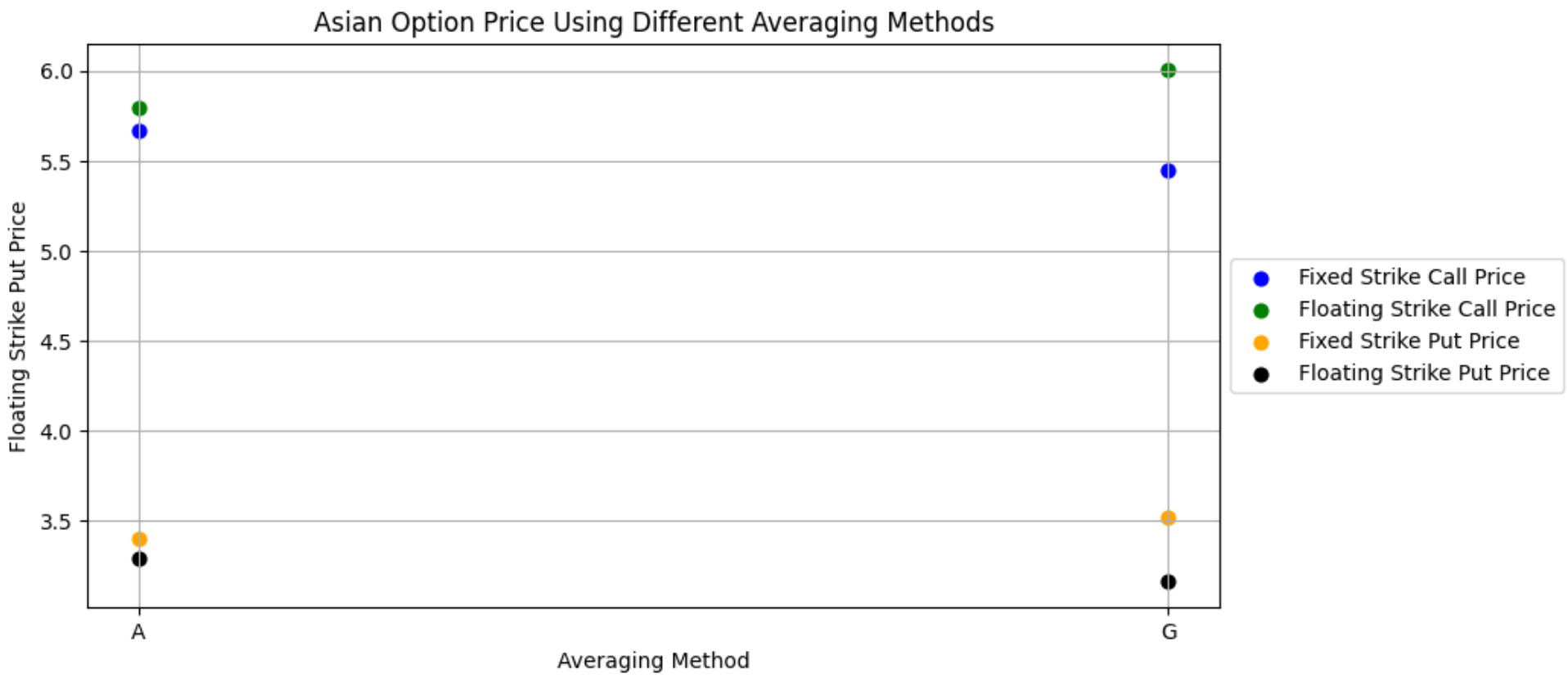
As the Risk Free Rate increases, the option price decreases. This can be inferred from the same logic as above for Floating Strike Calls.

Note: For all cases above the effect of discounting is less significant than the increase of forward values resulting from rising interest rates.

(4) Varying the Averaging method: Arithmetic vs Geometric

Holding fixed: S0=100, sigma=0.2, mu = 0.05, horizon=1, timesteps=252, strike=100, SampleType='C'

In [16]:
```
# Values denoting different averaging methods
avg_vals = ['A', 'G']

# Fixed Strike Call
avg_FX_C = [price_asian(AvgType=val, OpType="C", StrikeType="fixed") for val in avg_vals]

# Floating Strike Call
avg_FL_C = [price_asian(AvgType=val, OpType="C", StrikeType="floating") for val in avg_vals]

# Fixed Strike Put
avg_FX_P = [price_asian(AvgType=val, OpType="P", StrikeType="fixed") for val in avg_vals]

# Floating Strike Put
avg_FL_P = [price_asian(AvgType=val, OpType="P", StrikeType="floating") for val in avg_vals]

# Create dataframe for different averaging methods and price calculated at each S0
avg_df = pd.DataFrame.from_dict({
    'Averaging Method': avg_vals,
    'Fixed Strike Call Price': avg_FX_C,
    'Floating Strike Call Price': avg_FL_C,
    'Fixed Strike Put Price': avg_FX_P,
    'Floating Strike Put Price': avg_FL_P,
})

# Plot results
ax = avg_df.plot.scatter(x='Averaging Method', y='Fixed Strike Call Price',
                         s=40, c='blue', grid=True, legend=True);
avg_df.plot.scatter(ax=ax, x='Averaging Method', y='Floating Strike Call Price',
                    s=40, c='green', grid=True, legend=True);
avg_df.plot.scatter(ax=ax, x='Averaging Method', y='Fixed Strike Put Price',
                    s=40, c='orange', grid=True, legend=True);
avg_df.plot.scatter(ax=ax, x='Averaging Method', y='Floating Strike Put Price',
                    s=40, c='black', grid=True, legend=True);
ax.set_title("Asian Option Price Using Different Averaging Methods");
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height]);
ax.legend(['Fixed Strike Call Price','Floating Strike Call Price'
           ,'Fixed Strike Put Price','Floating Strike Put Price']
          ,loc='center left', bbox_to_anchor=(1, 0.5));
```



Interpretation

Fixed Strike Call

The price is lower when taking geometric mean. The geometric mean is always smaller than the arithmetic mean. Therefore the forward value is smaller when taking geometric mean, reducing the option price.

Fixed Strike Put

The price is higher when taking the geometric mean. The geometric mean is always smaller than the arithmetic mean. Therefore the forward value is smaller when taking geometric mean, increasing the option price..

Floating Strike Call

The price is higher when taking geometric mean. The geometric mean is always smaller than the arithmetic mean. Therefore the strike is smaller when taking geometric mean, increasing the option price.

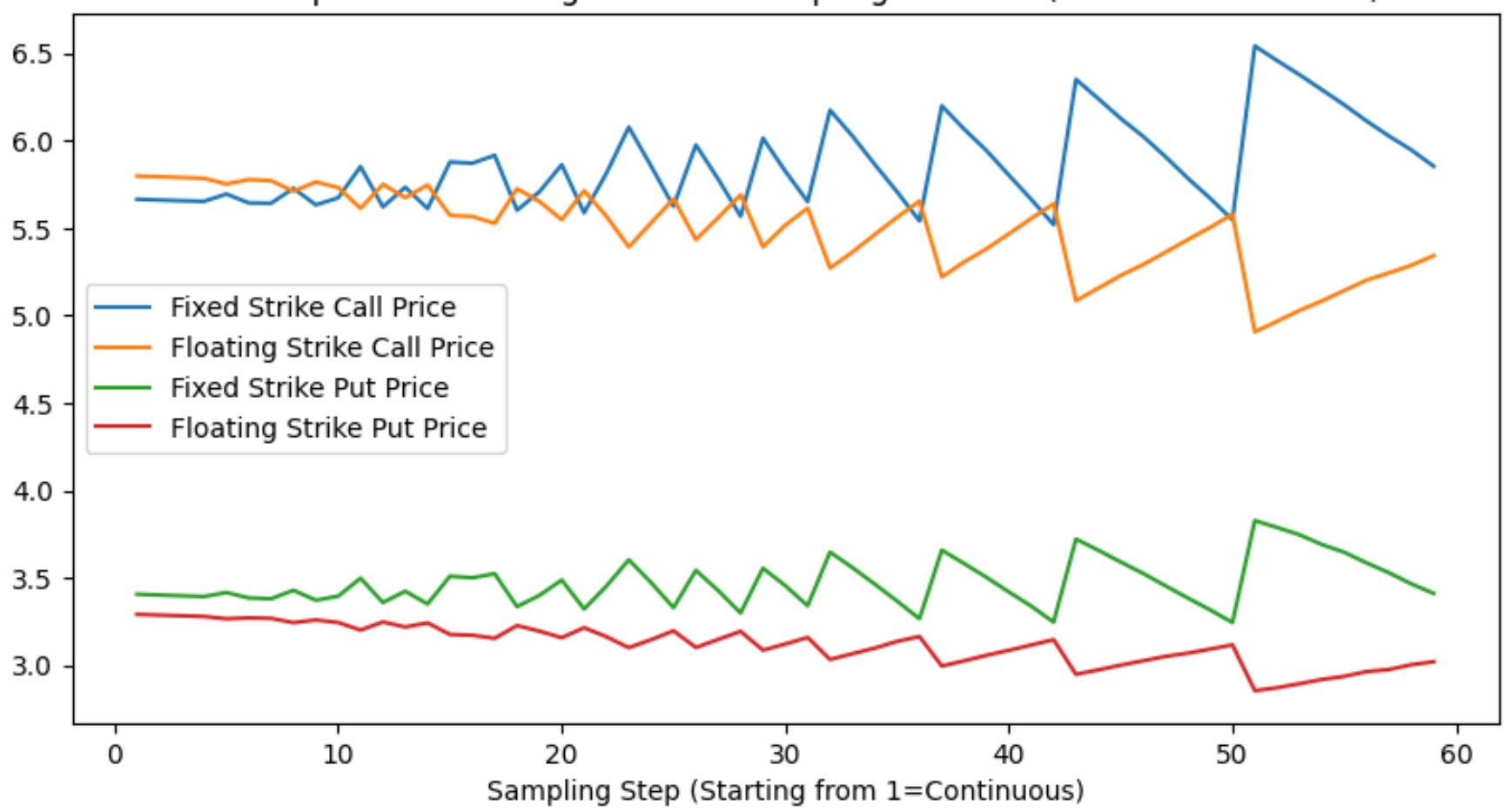
Floating Strike Put

The price is lower when taking geometric mean. The geometric mean is always smaller than the arithmetic mean. Therefore the strike is smaller when taking geometric mean, decreasing the option price.

(5) Varying the Sampling method: Continuous vs Discrete (at various distance between each sample)

Holding fixed: S0=100, sigma=0.2, mu = 0.05, horizon=1, timesteps=252, strike=100, AvgType='A'

In [17]:
```
# Values denoting different sampling methods
### Note since continuous sampling is equivalent to discrete sampling but with a step of -1,
### We letavg_vals = 'D', and input a list of step lengths
### Values starting from -1, which denotes continuous sampling
step_vals = list(range(-1, -60, -1))

# Fixed Strike Call
step_FX_C = [price_asian(SampleType="D", AvgStep=val, OpType="C", StrikeType="fixed")
             for val in step_vals]

# Floating Strike Call
step_FL_C = [price_asian(SampleType="D", AvgStep=val, OpType="C", StrikeType="floating")
             for val in step_vals]

# Fixed Strike Put
step_FX_P = [price_asian(SampleType="D", AvgStep=val, OpType="P", StrikeType="fixed")
             for val in step_vals]

# Floating Strike Put
step_FL_P = [price_asian(SampleType="D", AvgStep=val, OpType="P", StrikeType="floating")
             for val in step_vals]

# Create dataframe for different averaging methods and price calculated at each S0
step_df = pd.DataFrame.from_dict({
    'Sampling Step (Starting from 1=Continuous)': [val*(-1) for val in step_vals],
    'Fixed Strike Call Price': step_FX_C,
    'Floating Strike Call Price': step_FL_C,
    'Fixed Strike Put Price': step_FX_P,
    'Floating Strike Put Price': step_FL_P,
})

# Plot results
ax = step_df.plot(x='Sampling Step (Starting from 1=Continuous)',
                  y='Fixed Strike Call Price');
step_df.plot(ax=ax, x='Sampling Step (Starting from 1=Continuous)',
             y='Floating Strike Call Price');
step_df.plot(ax=ax, x='Sampling Step (Starting from 1=Continuous)',
             y='Fixed Strike Put Price');
step_df.plot(ax=ax, x='Sampling Step (Starting from 1=Continuous)',
             y='Floating Strike Put Price');
ax.set_title("Asian Option Price Using Different Sampling Methods (from 1=Continuous)");
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height]);
```



Interpretation

At first glance, when we increase the sampling step from 0 to 60, the option price for all four products show a sawtooth shape. This is because as the discrete sampling step increases, every so often a "jump" occurs when the step crosses a point that decreases the sample size by 1. For example, suppose we are sampling backward from 100 to 0 out of 100 values, 100, 99, 98, ..., 0. When sample step = 21, 22, 23, 24, 25, their sample sizes all equal to 5. However, when the sampling step increases to 26, the sample size becomes 3 (100, 74, 48, 18). Therefore compared to when step=25, the sample "shifts" abruptly towards 100, thereby lifting the average and bumping up the price.

Another observation comes from comparing the many 'Peaks' on the saw tooth. We can see that the peak values slowly increase as well. Similar to the explanation of the jumps, at every new peak the sample size decreases by one and the sample shifts towards the point at maturity (larger values), and therefore the sample mean goes up with every new peak.

Aside from these "Jumps" and the resulting "Peaks" identified, we can also see that between each two jumps, the sample mean gradually decreases. Taking the same example as above, when step=21, sampled values = [100, 79, 58, 37, 16]; when step=22, sampled values = [100, 78, 56, 34, 12]; when step=22, sampled values = [100, 78, 56, 34, 12]; when step=23, sampled values = [100, 77, 54, 31, 8]. We can easily calculate that the sample mean at step=23 is smaller than at 22, which is smaller than at 21.

Because in all four products, the average replaces one of the forward (at maturity) or the strike values whilst the other one stays constant in our analysis, the payoff thereby driven by the sawtooth shaped change in the average. This can also explain why the price of the floating strike call and the fixed strike call (as with the floating strike put and fixed strike put) move in opposite directions, depending on whether the average replaces S or K.

CONCLUSION

The follwing conclusions are drawn from the discussion above

The Closed Form Solution, Euler Maruyama Scheme and Milstein Scheme are all valid schemes for simulating a stock's forward value. As the number of simulations increase, the values of from the EM Scheme converges very quickly to the Milstein Scheme (which we know adds a correction term to the EM Scheme) - which we have explored using both the sigma and the actual forward stock price.

We then varied parameters of the Asian Option (built on the simulated stock) and examined the impact on price for the fixed strike call, fixed strike put, floating strike call, floating strike call. From this we can see that the choices of parameters can have sizeable effects on the option price, mostly through impacting the sampled mean or the variance of the geometric Brownian Motion.

Limitations

An important limitation of the Asian Option pricer is the discrete sampling method. Here the method adopted is that we first define a sample step *n*. After that we the value at maturity as the first sample, and then take the values at t-n, t-2n, t-3n, ... until we go beyond t0. In real life, the contract of the option can take on various forms, and a different discrete sampling method may well be

used. For example, we could instead choose a time period (such as t-100 to t, then specify either a sample size *m* (from which we take *m* evenly spaced time points from the range (t-100, t), or, define a time step the same way as the pricer does, and keep sampling from t until we go beyond t0.

Question 2.

Overview:

The model problem is a linear second-order homogenous differential equation, which we know how to solve by hand using the method covered in the math primer (more on this in Question 2 Part 3). However, the question guides us through the steps of solving it using numerical methods, namely Finite Difference Method using tridiagonal matrices. The FDM produces results that are approximations of the exact solution, which in this case we can compare with the hand-solved exact solution to evaluate its accuracy. Aligned with the question guideline, the answer below is structured into three parts:

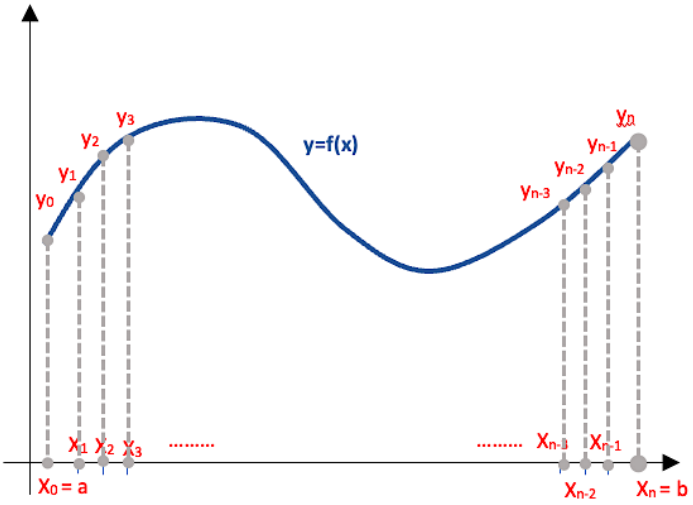
- Part 1 applies numerical approximations for each derivative term.
- Part 2 expresses the results of Part 1 into a matrix inversion problem.
- Part 3 computes the problem in Part 2 using Python
- Part 4 solves the model problem DE by hand to get the precise solution and compares it with the numerically solved solutions in Part 3.

Part 1:

Applying the definition of the interior mesh points of interval [a, b], we can conceptualise the function $y(x)$, which is represented by the dark blue line in the graph below. The domain of $f(x)$ is represented by the interval [a, b] is partitioned into n intervals, where each interval $\delta x = \frac{b-a}{n}$. Hence, we have:

$$\begin{cases} x_0 = a \\ x_1 = a + \delta x \\ x_2 = a + 2\delta x \\ \dots \\ x_{n-1} = a + \delta x \\ x_n = b \end{cases}$$

$y(x)$ at each point of x_i (i=1, 2, 3, ..., n) can be expressed as $y_i = y(x_i)$, in other words, instead of seeing y(x) as a continuous line, we have effectively broken it down as a series of points which approximates the continuous line: (x_0, y_0) (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), by applying the function $y_i = y(x_i)$, to each interior mesh point x_i on the interval [a, b].



Similarly, since P, Q and f are also funtions of x, we have:

$$\begin{cases} Q_i = Q(x_i), \\ P_i = P(x_i), \\ f_i = f(x_i) \end{cases}$$

Next we derive numerical approximations for each derivative term, i.e., $\frac{dy}{dx}$, $\frac{d^2y}{dx^2}$. Here we use the centered approximation which is more accurate than forward and backward approximation.

From a Taylor Series expansion of y about the point $x - \delta x,\ x - \delta x$, we have:

$$y(x_i + \delta x) = y(x_i) + \delta x \frac{dy}{dx} + \frac{1}{2!}\delta x^2 \frac{d^2y}{dx^2} + \frac{1}{3!}\delta x^3 \frac{d^3y}{dx^3} + O(\partial x^4) \quad (1)$$

$$y(x_i - \delta x) = y(x_i) - \delta x \frac{dy}{dx} + \frac{1}{2}\delta x^2 \frac{d^2y}{dx^2} - \frac{1}{3!}\delta x^3 \frac{d^3y}{dx^3} + O(\partial x^4) \quad (2)$$

From (1) – (2):

$$y(x_i + \delta x) - y(x_i - \delta x) = 2\delta x \frac{dy}{dx} + O(\partial x^3)$$

Upon rearranging is:

$$\frac{dy}{dx} = \frac{y(x_i + \delta x) - y(x_i - \delta x)}{2\delta x} + O(\partial x^3)$$

Which can be written as:

$$\frac{dy}{dx} = \frac{y(x_{i+1}) - y(x_{i-1})}{2\delta x} + O(\partial x^3)$$

i.e.,

$$\frac{dy}{dx} \approx \frac{y_{i+1} - y_{i-1}}{2\delta x} \quad (3)$$

From (1) + (2):

$$y(x_i + \delta x) + y(x_i - \delta x) = +\delta x^2 \frac{d^2y}{dx^2} + O(\partial x^4)$$

Upon rearranging is:

$$\frac{d^2y}{dx^2} = \frac{y(x_i + \delta x) - 2y(x_i) + y(x_i - \delta x)}{\delta x^2} + O(\partial x^2)$$

Which can be written as:

$$\frac{d^2y}{dx^2} = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{\delta x^2} + O(\partial x^2)$$

i.e.,

$$\frac{d^2y}{dx^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\delta x^2} \quad (4)$$

Equation (3) are numerical approximations for the derivative terms $\frac{dy}{dx}$, $\frac{d^2y}{dx^2}$.

Going back to the model problem: For any point x_i, (i =1, 2, 3, ..., n-1),

$$\frac{d^2y}{dx^2}|_{x_i} + P(x_i)\ \frac{dy}{dx}|_{x_i} + Q(x_i)y = f(x_i)$$

Substituting in our numerical approximations for the derivative terms:

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{\delta x^2} + P_i \frac{y_{i+1} - y_{i-1}}{2\delta x} + Q_i y_i = f_i$$

Simpify:

(Multiply both sides by δx^2)

$$y_{i+1} - 2y_i + y_{i-1} + \frac{\delta x}{2} P_i \frac{y_{i+1} - y_{i-1}}{2} + \delta x^2 Q_i y_i = \delta x^2 f_i$$

(Group together the y_{i+1}, y_i, y_{i-1} terms)

$$(1 + \tfrac{\delta x}{2} P_i) y_{i+1} + (-2y_i + \delta x^2 Q_i) y_{i-1} + (1 - \tfrac{\delta x}{2} P_i)\, y_i = \delta x^2 f_i;\ i = 1, 2, 3, \dots, n-1 \qquad (5)$$

Part 2:

Until now we have:

$$(1 + \tfrac{\delta x}{2} P_i) y_{i+1} + (-2 + \delta x^2 Q_i) y_{i-1} + (1 - \tfrac{\delta x}{2} P_i)\, y_i = \delta x^2 f_i;\ i = 1, 2, 3, \dots, n-1$$

And two boundary conditions:

$$y_0 = y(x_0) = y(a) = \alpha$$
$$y_n = y(x_n) = y(b) = \beta$$

Written together as n+1 equations:

$$\begin{cases} y_0 = \alpha \\ (1 + \frac{\delta x}{2} P_1) y_2 + (-2 + \delta x^2 Q_1) y_0 + (1 - \frac{\delta x}{2} P_1)\, y_1 = \delta x^2 f_1 \\ (1 + \frac{\delta x}{2} P_2) y_3 + (-2 + \delta x^2 Q_2) y_1 + (1 - \frac{\delta x}{2} P_2)\, y_2 = \delta x^2 f_2 \\ (1 + \frac{\delta x}{2} P_3) y_4 + (-2 + \delta x^2 Q_3) y_2 + (1 - \frac{\delta x}{2} P_3)\, y_3 = \delta x^2 f_3 \\ \dots\ \dots \\ \dots\ \dots \\ (1 + \frac{\delta x}{2} P_n) y_{n+1} + (-2 + \delta x^2 Q_n) y_n + (1 - \frac{\delta x}{2} P_n)\, y_3 = \delta x^2 f_n \\ y_n = \beta \end{cases}$$

This set of equations can be written in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 1 - \frac{\delta x}{2} P_1 & -2 + \delta x^2 Q_1 & 1 + \frac{\delta x}{2} P_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 - \frac{\delta x}{2} P_2 & -2 + \delta x^2 Q_2 & 1 + \frac{\delta x}{2} P_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 - \frac{\delta x}{2} P_3 & -2 + \delta x^2 Q_3 & 1 + \frac{\delta x}{2} P_3 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 - \frac{\delta x}{2} P_{n-1} & -2 + \delta x^2 Q_{n-1} & 1 + \frac{\delta x}{2} P_{n-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 - \frac{\delta x}{2} P_n & -2 + \delta x^2 Q_n & 1 + \frac{\delta x}{2} P_n \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = \begin{bmatrix} \alpha \\ \delta x^2 f_1 \\ \delta x^2 f_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \delta x^2 f_n \\ \beta \end{bmatrix}$$

Matrix A: (n+1) × (n+1) **Matrix x: (n+1) × 1** **Matrix b: (n+1) × 1**

Hence expressed as a matrix inversion problem $Ax = b$; the form of A, x and b as shown above.

Part 3:

Applying the method above to the differential equation:

$$\frac{d^2y}{dx^2} + 3\frac{dy}{dx} + 2y = 4x^2,\ \mathrm{y}(1) = 1, y(2) = 6$$

We have

$$\begin{cases} P = 3 \\ Q = 2 \\ f = 4x^2 \end{cases}$$

And

$$\begin{cases} a = 1 \\ b = 2 \\ \alpha = 1 \\ \beta = 6 \end{cases}$$

The problem is coded in Python in Jupyter Notebook, as below. The full Jupyter Notebook can be found in the Zip folder named as "Question 2 Script".

First we import packages as needed.

```
import numpy as np
import pandas as pd

import math as m

import cufflinks as cf
cf.set_config_file(offline=True, dimensions=((1000,600)))

from matplotlib import pyplot as plt

import plotly.express as px
px.defaults.width, px.defaults.height = 1000, 600
```

Next we write three functions to generate:

- The $(n+1)\times(n+1)$ matrix A as expressed in Part 2,
- The vector of $x_i (i = 1, 2, 3, \dots, n)$, denoted in the code as small x,
- The $(n+1)\times 1$ matrix b as expressed in Part 2.

Function `gen_A` takes the boundaries a, b and number of interior mesh points n as arguments and returns a generated matrix A.

```
def gen_A(a, b, n):

    """
    Takes three inputs
    n: Number of intervals on X-axis created by interior mesh points -> Hence n-1 mesh points.
    a: Lower boundary for x.
    b: Upper boundary for x.

    Returns a tridiagonal matrix A of shape (n+1)*(n+1).

    """
    # Compute P, Q

    P = np.full((n+1), 3)
    Q = np.full((n+1), 2)

    # Initialise a (n+1)*(n+1) matrix of zeros

    A = np.zeros((n+1,n+1))

    # Derive delta_x
    delta_x = (b-a)/n

    # Compute matrix values

    for i in range(n+1):
        if i==0:
            A[i,i] = 1
        elif i==n:
            A[i,i] = 1
        else:
            A[i,i-1] = 1-delta_x/2*P[i]
            A[i,i] = -2+delta_x**2*Q[i]
            A[i,i+1] = 1+delta_x/2*P[i]

    return A
```

Function gen_x_i takes the boundaries a, b and number of interior mesh points n as arguments and returns a generated vector x_i.

```
# Create vector x_i

def gen_x_i(a, b, n):

    # Initialise
    x_i = np.zeros(n+1)

    # Derive delta_x
    delta_x = (b-a)/n

    # Update values
    for i in range(n+1):
        x_i[i] = a+i*delta_x
    return x_i
```

Function gen_b takes the domain boundaries (a, b), boundary values of y (α, β), number of interior mesh points n, and vector of x_i values as arguments and returns a generated matrix b. Note b is denoted _B to distinguish from the domain upper boundary b.

```
# Create Matrix b - knowing f(x) = 4*x**2

def gen_b(a, b, alpha, beta, n, x_i):

    # Initialise
    _B = np.zeros(n+1)

    # Derive delta_x
    delta_x = (b-a)/n

    # Update values
    for i in range(n+1):
        if i == 0:
            _B[i] = alpha
        elif i == n:
            _B[i] = beta
        else:
            _B[i] = delta_x**2*4*x_i[i]**2
    return _B
```

Next, using the three functions defined above, we create a master function $calc\_y$ that takes the domain boundaries (a, b), boundary values of y (α, β) and number of interior mesh points n as arguments, generates matrices A, x_i, b. Then taking the dot product of inverse A and b we compute the "x" (which is a $(n+1)\times 1$ matrix of y_i values for each value of x_i $(i = 1, 2, 3, \dots, n)$) in the $Ax = b$ matrix inversion problem. The function returns two outputs: the matrix x which stores y_i values, and the x_i vector which stores all values of x_i.

We call the function three times with the corresponding parameters, and let $n = 10, 50, 100$.

The resulting outputs are then stored in three dataframes results_10, results_50, results_100, each dataframe has column 'x' storing values of, and 'y' storing values of y_i $(i = 1, 2, 3, \dots, n)$.

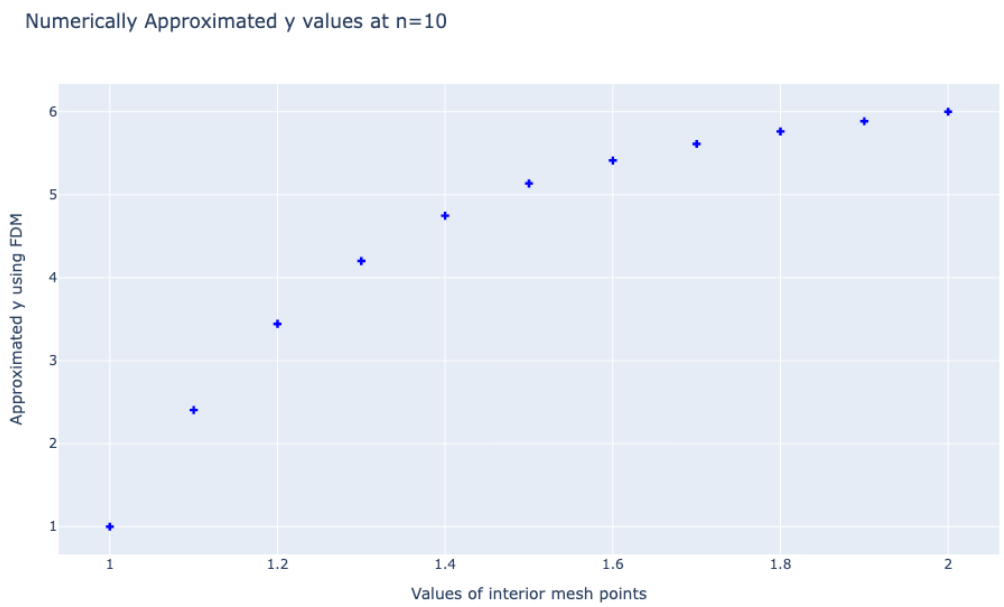
To visualise the results, we plot each dataframe into scatter plots using Plotly package. The code and generated plots are listed below.

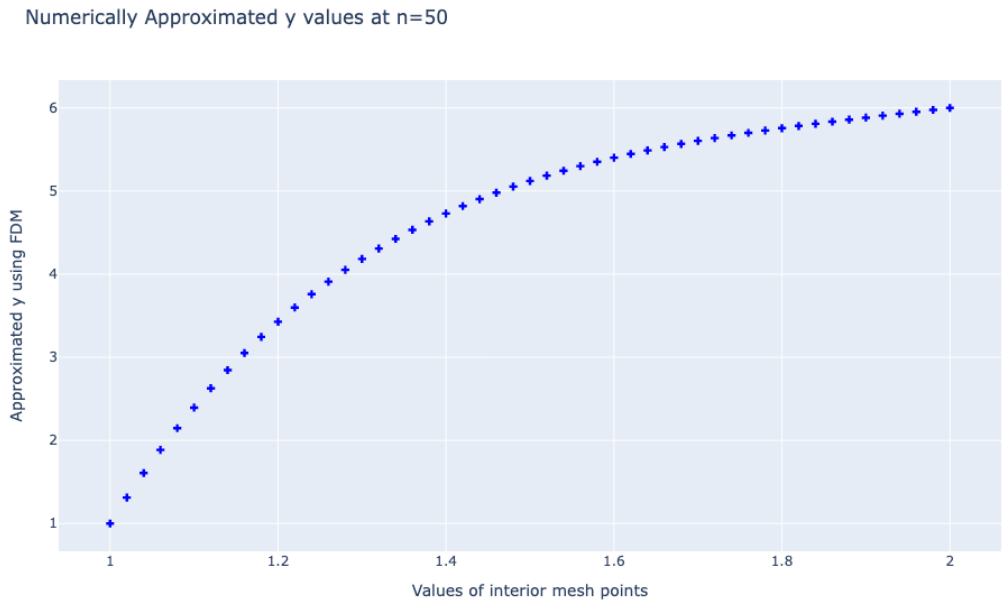
```
# Plot x's and y's when n=10

fig = px.scatter(
    results_10, x='x', y='y',
    labels={'x': 'Values of interior mesh points', 'y': 'Approximated y using FDM'},
    title="Numerically Approximated y values at different n"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```

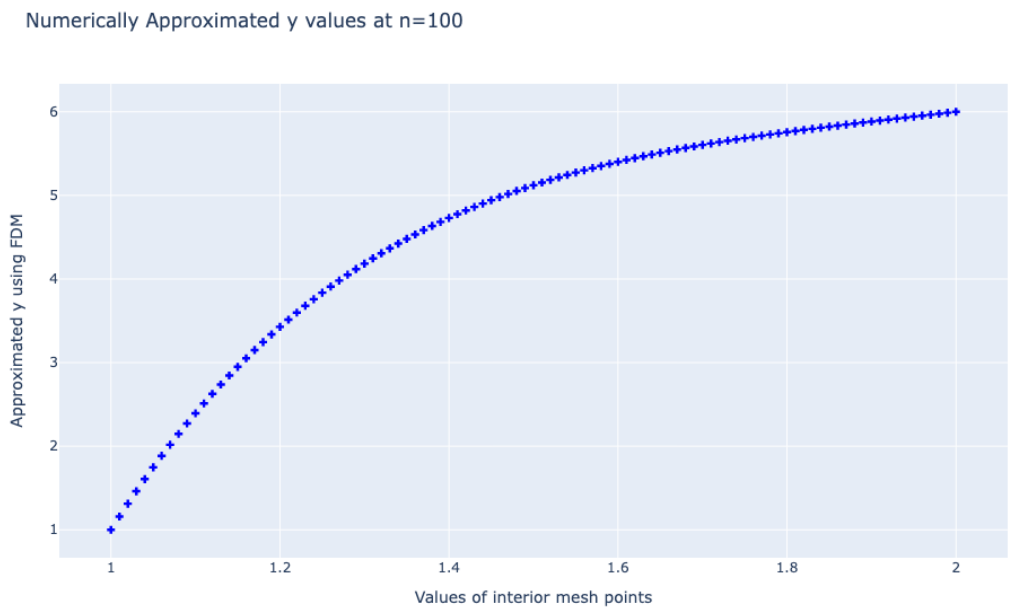
```
# Plot x's and y's when n=50

fig = px.scatter(
    results_50, x='x', y='y',
    labels={'x': 'Values of interior mesh points', 'y': 'Approximated y using FDM'},
    title="Numerically Approximated y values at n=50"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```

```
# Plot x's and y's when n=100

fig = px.scatter(
    results_100, x='x', y='y',
    labels={'x': 'Values of interior mesh points', 'y': 'Approximated y using FDM'},
    title="Numerically Approximated y values at n=100"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```

The three plots are shown below. The x axis denotes the value of the x_i $(i = 1, 2, 3, \dots, n)$ and the y axis denotes value of numerical approximations of y at each x_i.







Comparing the three plots, we see that the dots seem follow a very similar trend, approximating the same curve.

Moreover, in line with the definition of numerical methods, which approximates a continuous process with discrete points on a grid, we further make the guess that the larger the n the more accurate the approximation (to be tested in Part 4).

Part 4:

To better interpret the approximated results produced by FDM, we solve the differential equation in Part 3 by hand to derive the precise solution.

We know that the GS of a non-homogenous second-order linear differential equation takes the form:

$$y = C.F + P.I = y_c + y_p$$

We can derive the Auxilliary Equation:

$$\lambda^2 + 3\lambda + 2 = 0$$

Which has a two distinct real roots:

$$\lambda = -1, -2$$

Hence,

$$y_c = ae^{-x} + be^{-2x}$$

Now we know the non-homogenous part is:

$$f(x) = 4x^2$$

Therefore for y_p we try:

$$y_p = p_0 + p_1 x + p_2 x^2$$

$$\rightarrow\ y_p' = p_1 + 2p_2 x$$

$$\rightarrow\ y_p'' = 2p_2$$

Substitute the derivative terms into the original differential equation:

$$2p_2 + 3(p_1 + 2p_2x) + 2(p_0 + p_1x + p_2x^2) = 4x^2$$
$$2p_2 + 3p_1 + 6p_2x + 2p_0 + 2p_1x + 2p_2x^2 = 4x^2$$

$$2p_2 + 3p_1 + 2p_0 + (6p_2 + 2p_1)x + 2p_2x^2 = 4x^2$$

And equate the coefficients of x^n:

O(x^2): $2p_2 = 4 \rightarrow p_2 = 2$

O(x): $6p_2 + 2p_1 = 0 \rightarrow p_1 = -6$

O(x^0): $2p_2 + 3p_1 + 2p_0 = 0 \rightarrow p_0 = 7$

$$\rightarrow y_p = 2x^2 - 6x + 7$$

Therefore,

$$y = y_c + y_p = ae^{-x} + be^{-2x} + 2x^2 - 6x + 7$$

Substitute in the boundary conditions $y(1) = 1, y(2) = 6$

$$\begin{cases} ae^{-1} + be^{-2} + 2 - 6 + 7 = 1 \\ ae^{-2} + be^{-4} + 8 - 12 + 7 = 6 \end{cases}$$

$$\begin{cases} ae^{-1} + be^{-2} = -2 & (6) \\ ae^{-2} + be^{-4} = 3 & (7) \end{cases}$$

(6) LHS and RHS both multiply by e^{-1}, then subtract by (7):

$$b = \frac{2e^3 + 3e^4}{1 - e}$$

Substitute into (6) or (7):

$$a = \frac{-3e^3 - 2e}{1 - e}$$

Substitute into the expression of y above:

$$y = \left(\frac{-3e^3 - 2e}{1 - e}\right)e^{-x} + (\frac{2e^3 + 3e^4}{1 - e})e^{-2x} + 2x^2 - 6x + 7$$

$$= \frac{-3e^{3-x} - 2e^{1-x} + 2e^{3-2x} + 3e^{4-2x}}{1 - e} + 2x^2 - 6x + 7$$

Now we know the precise solution of y, we want to visualise its graph to compare with our approximated solutions using FDM in Part 3.

First we define a function precise_y that takes in a value of x and computes the corresponding value of y using the expression above.

```
# Defining Function y

def precise_y(x):
    return (-3*m.exp(3-x)-2*m.exp(1-x)+2*m.exp(3-2*x)+3*m.exp(4-2*x))/(1-m.exp(1))+2*x**2-6*x+7
```

Next we calculate a series of precise values of y for a range of x on the [1, 2] domain. To do this, we generate 500 equally spaced x values in the range [1, 2] and apply the precise_y function to each x value to derive the precise y value. The series of $(x, y)'s$ are gathered into a dataframe results_actual with two columns respectively storing x and y.

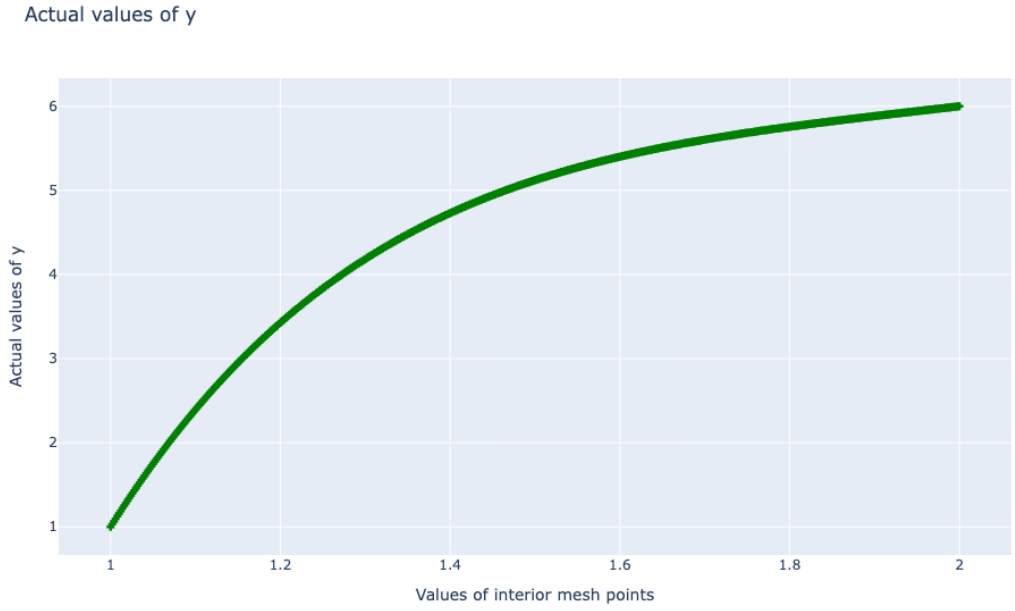
```
# Generate 500 x values in domain [1,2]

x = list(np.linspace(1, 2, num=500))
y = [precise_y(x) for x in x]

# Store values of x and y into a dataframe

results_actual = pd.DataFrame.from_dict({'x': x,'y': y})
results_actual
```

	x	y
0	1.000000	1.000000
1	1.002004	1.032101
2	1.004008	1.064018
3	1.006012	1.095751
4	1.008016	1.127301
...	...	...
495	1.991984	5.990602
496	1.993988	5.992949
497	1.995992	5.995298
498	1.997996	5.997648
499	2.000000	6.000000

500 rows × 2 columns

We then plot values of x and y to visualise the solution function.

```
# Plot x's and actual y's

fig = px.scatter(
    results_actual, x='x', y='y',
    labels={'x': 'Values of interior mesh points', 'y': 'Actual values of y'},
    title="Actual values of y"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='green'))
fig.show()
```



Comparing the graph for the precise solution function with the numerically approximated plots in Part 3, we can spot (from eyeballing) that the approximated plots seem to follow the same trend as the precise solution.

To test whether a larger n contributes to better accuracy of the numerically approximated solution, we calculate (in Python) the errors of the numerical solution compared to the actual solution.

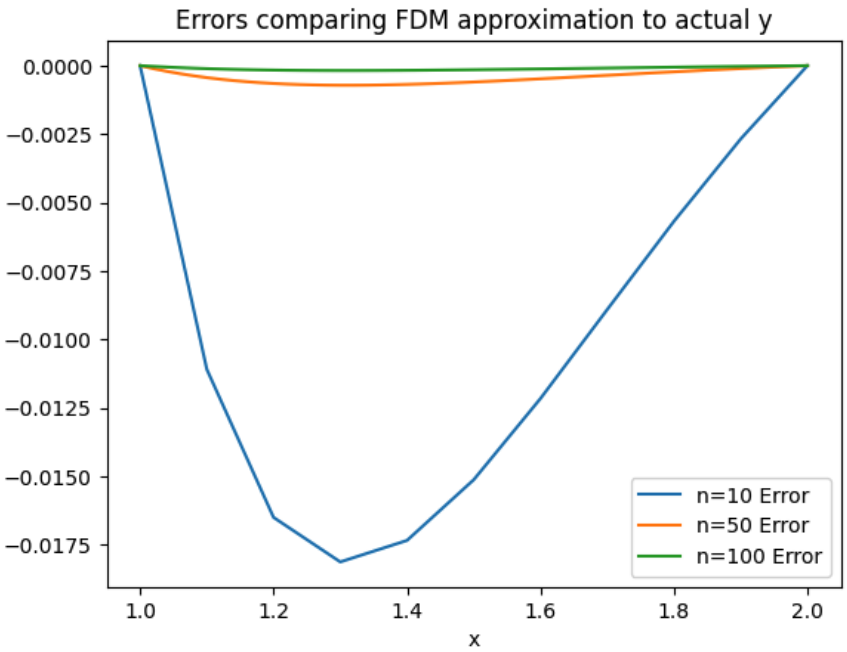
First, using the precise_y function defined above, we calculate the actual values of y for each value of x_i $(i = 1, 2, 3, \dots, n)$, for $n = 10, 50, 100$. The actual y values are then appended to the results_10, results_50, results_100 dataframes as a new column "Actual y".

Then for each dataframe, we compute an error column that is the difference between the actual y calues and the numerically approximated y values.

```
# Calculating errors of approximated solutions

# Calculating actual values of y
actual_y_10 = [precise_y(x) for x in list(_X_10[1])]
actual_y_50 = [precise_y(x) for x in list(_X_50[1])]
actual_y_100 = [precise_y(x) for x in list(_X_100[1])]

# Appending actual values of y to each result dataset for n=10, 50, 100
results_10['Actual y'] = actual_y_10
results_50['Actual y'] = actual_y_50
results_100['Actual y'] = actual_y_100

# Calculating an error column that stores the error of the numerically approximated y to the actual y
results_10['n=10 Error'] = results_10['Actual y'] - results_10['y']
results_50['n=50 Error'] = results_50['Actual y'] - results_50['y']
results_100['n=100 Error'] = results_100['Actual y'] - results_100['y']
```

Next we plot the errors approximated with $n = 10, 50, 100$ onto one plot.

```
# Plotting errors onto the same graph

ax = results_10.plot(x='x',y='n=10 Error')
results_50.plot(ax=ax, x='x', y='n=50 Error')
results_100.plot(ax=ax, x='x', y='n=100 Error')
ax.set_title("Errors comparing FDM approximation to actual y")
```



Here we can see that the error decreases as n becomes larger - the error is the smallest when $n = 100$. Therefore we visually confirm that the accuracy of the numerical solution increases as n becomes larger.

To solidify this conclusion, we further increase the n and repeat the process above.

A function get_results is written so that it takes an input n, calculates the numerical solution and outputs a tuple where the first element is a dataframe storing value of x_i $(i = 1, 2, 3, \dots, n)$, the corresponding numerical solutions of y_i, the precise solution of y for each of x_i, and the errors between of y_i and the actual solution of y, and the second element is the input value of n.

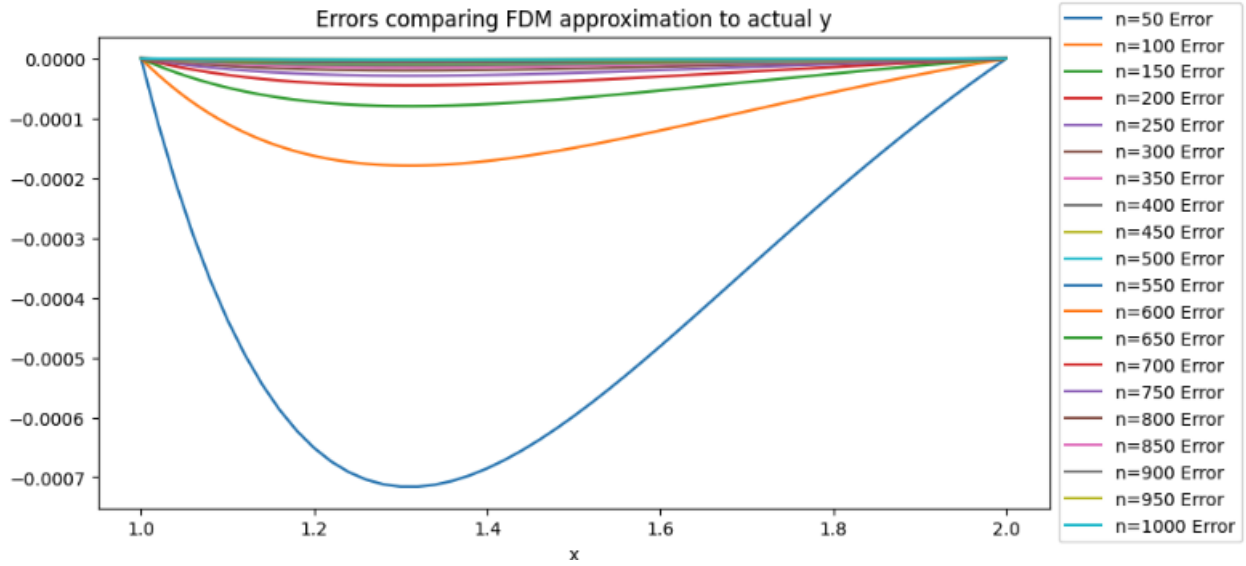
```
def get_results(n):
    _X = calc_y(1,2,1,6,n)
    actual_y = [precise_y(x) for x in list(_X[1])]

    results = pd.DataFrame.from_dict({
        'y': _X[0].flatten(),
        'x': _X[1].flatten(),
        'Actual y': actual_y
    })

    results[f'n={n} Error'] = results['Actual y'] - results['y']

    return results, n
```

We then create a list for 20 increasing values for n, equally spaced between 50 and 1000. Each value is input into the get_results function. Results are stored in a list nameed results_list.

```
# Define a list of n values

N = [int(n) for n in list(np.linspace(50, 1000, 20))]

# Calculating a list of results ()

results_list = [get_results(n) for n in N]
```

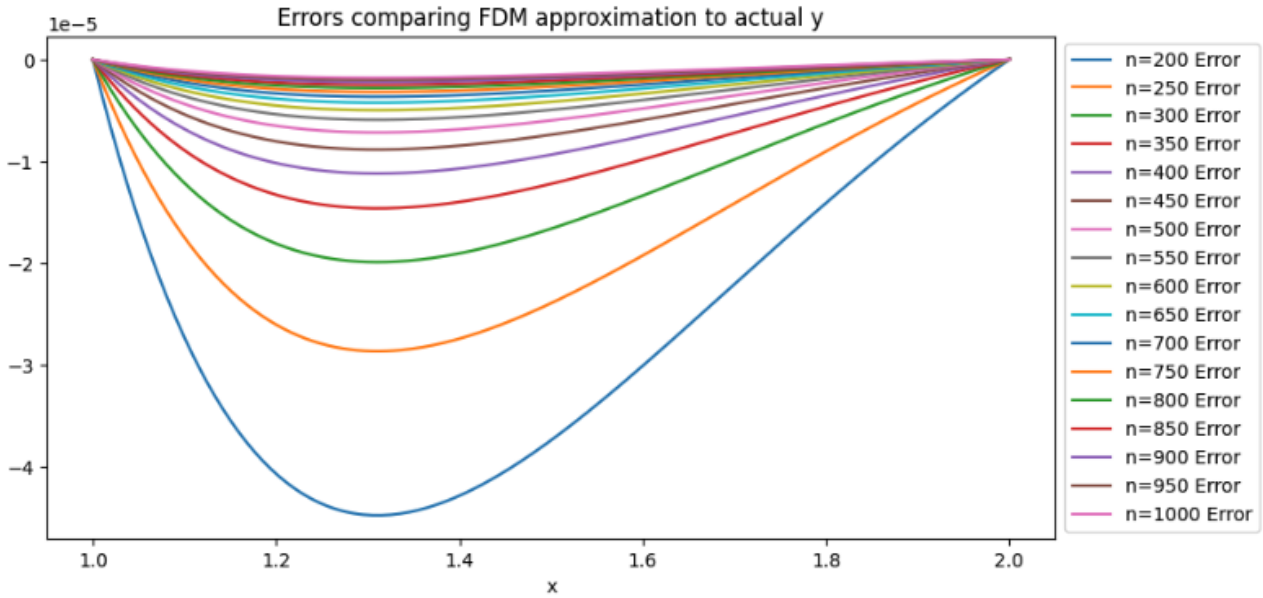
Next we plot the errors at each value of n on the same graph.

```
# Plot errors at various n's in one graph

# Create axis with results at n=50 (first value of n)
ax = results_list[0][0].plot(x='x',y=f'n={results_list[0][1]} Error')

# Loop through results for all n's
for i in range(1, len(results_list)):
    results_list[i][0].plot(ax=ax, x='x', y=f'n={results_list[i][1]} Error')
    ax.set_title("Errors comparing FDM approximation to actual y")

# Graph specifications
ax.set_title("Errors comparing FDM approximation to actual y")
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
```



As we can see, due to the errors rapidly decreasing as n increases, the lines become harder to distinguish. Hence we remove the first three n values (50, 100, 150) and plot the errors for larger values of n.

```
# Zooming-in to look closer at the error lines with larger n values
results_trimmed = results_list[3:]

# Create axis with results at n=50
ax = results_trimmed[0][0].plot(x='x',y=f'n={results_trimmed[0][1]} Error')

# Loop through results for all n's
for i in range(1, len(results_trimmed)):
    results_trimmed[i][0].plot(ax=ax, x='x', y=f'n={results_trimmed[i][1]} Error')
    ax.set_title("Errors comparing FDM approximation to actual y")

# Graph specifications
ax.set_title("Errors comparing FDM approximation to actual y")
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 1.5, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
```

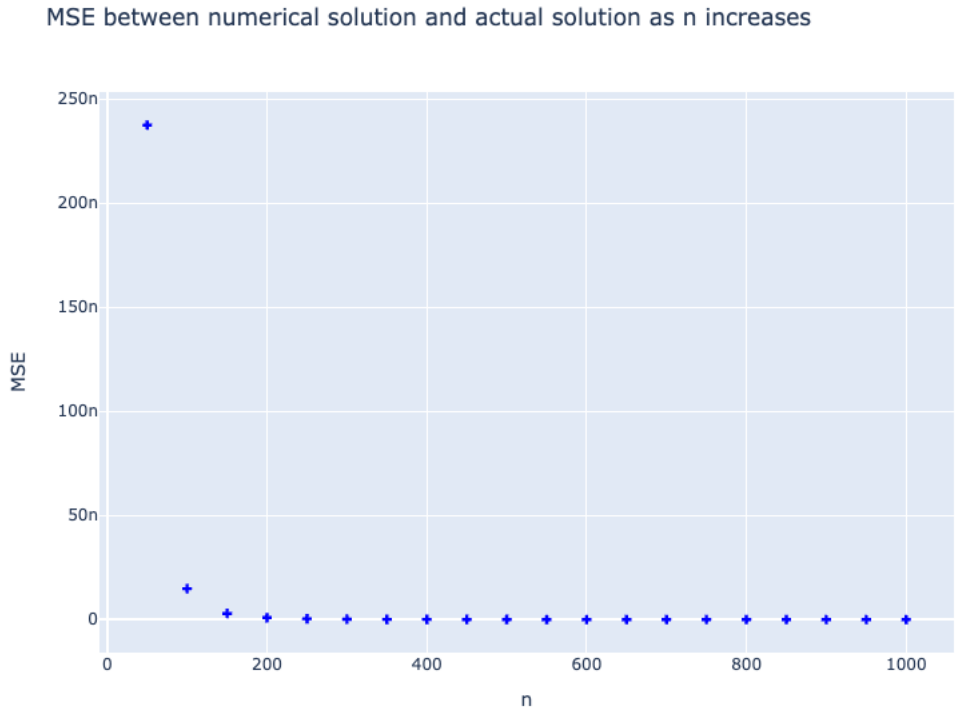


Now we can quite clearly see that as n increases, the errors become smaller, i.e. numerical solution becomes more accurate.

To mathematically demonstrate this, we compute the Mean Squared Error between the numerical solution and actual solution for each value of n. For each n, we simply square the values in the error column, take the sum of all squares, and then calculate the average. The computed MSE's are then stored in a dataframe `MSE_df` for plotting.

```
# Calculating mean squared error
MSE = []
for i in range(len(results_list)):
    df = results_list[i][0]
    error_list = list(df[df.columns[len(df.columns)-1]])
    MSE.append(sum([error**2 for error in error_list])/results_list[i][1])

# Storing MSE and their respective n into a dataframe for visualisation
MSE_df = pd.DataFrame.from_dict({
        'n': N,
        'MSE': MSE
    })

# Plotting MSE as n increases
fig = px.scatter(
    MSE_df, x='n', y='MSE',
    width=800, height=600,
    title="MSE between numerical solution and actual solution as n increases"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```

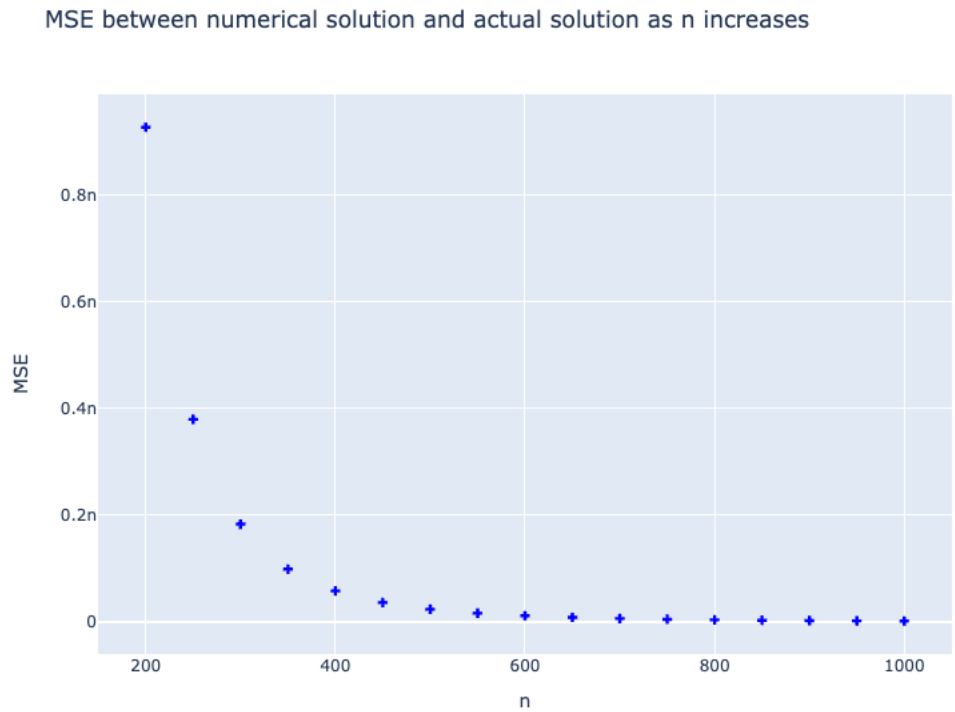
The graph below is scatter plot of the MSE values for the numerical approximation at various values of n.



To better compare the MSEs as n increases beyond 200, we remove the first three MSE's at $n = 50, 100, 150$ and plot again.

```
# Removing n=50 to zoom in
fig = px.scatter(
    MSE_df.loc[MSE_df['n']>150], x='n', y='MSE',
    width=800, height=600,
    title="MSE between numerical solution and actual solution as n increases"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```

From the new graph we can clearly see that the MSE reduces (converges to zero) as n becomes larger.



Examining the actual values of the MSEs, we can also see that they become neglibly small.

	n	MSE
0	50	2.374570e-07
1	100	1.482977e-08
2	150	2.928923e-09
3	200	9.266842e-10
4	250	3.795611e-10
5	300	1.830420e-10
6	350	9.880087e-11
7	400	5.791482e-11
8	450	3.615613e-11
9	500	2.372173e-11
10	550	1.620231e-11
11	600	1.143994e-11
12	650	8.305606e-12
13	700	6.174990e-12
14	750	4.685740e-12
15	800	3.619886e-12
16	850	2.840134e-12
17	900	2.259752e-12
18	950	1.820497e-12
19	1000	1.482797e-12

On a last (and not so important) note, we also notice that all errors are negative, meaning that all numerical approximations are smaller than the value of the actual solution. This can be intuitively explained by numerical approximation (which approximates values of discrete points connected by straight lines) not capturing the full curvature of the downward concaving curve, hence becoming a "flattened" version of the precise solution.

Conclusion:

In this question, we solved a second-order linear differential equation numerically using Finite Difference Methods. Through approximating the derivative terms using Taylor Series expansion, we obtained n +1 equations which were then converted to a matrix inversion problem. Meanwhile, we were also able to derive the precise solution using auxiliary equation and the general solution form for non-homogenous second-order linear differential equations. Our computations show that as n becomes larger, our numerically approximated solution becomes more accurate (i.e., closer to the hand-solved actual solution). This is further demonstrated by taking larger n values and comparing the MSE.

Question 3.

Overview:

The question asks for calculating three integrations numerically using the Monte Carlo simulations and compare them to exact values. To do this, we need to first "map" the limits of each integral onto the range (0, 1), then compute the integral as an expectation. The answer below consists of four parts:

- Part 1 calculates each integral by hand to derive exact values
- Part 2 transforms each integral into the Monte Carlo integration form (convert domain to [0, 1]
- Part 3 performs Monte Carlo integration in Python to numerically calculate each integral
- Part 4 examines how the error behaves as we increase the random number

Part 1:

We first solve each integral by hand to get the exact value.

Integral (i)

$$\begin{aligned} I &= \int_1^3 x^2 dx \\ &= \left[\frac{1}{3}x^3\right]_1^3 \\ &= \frac{1}{3}3^3 - \frac{1}{3}1^3 \\ &= \frac{26}{3} \end{aligned}$$

Integral (ii)

$$I = \int_0^\infty \exp(-x^2) dx$$

We know that the Gaussian Integral is the same function integrated from -∞ to ∞.

i.e.,

$$\int_{-\infty}^{\infty} \exp(-x^2) dx = \sqrt{\pi}$$

The function $f(x) = \exp(-x^2)$ is symmetric about the y axis ($f(x) = f(-x)$). Therefore its integral from 0 to ∞ should be half the value of the Gaussian Integral.

$$\int_0^\infty \exp(-x^2)\, dx = \frac{\sqrt{\pi}}{2}$$

Integral (iii)

$$I = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^4 \exp(-x^2/2) dx$$

Using integration by parts, we have:

$$\begin{aligned} I &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^4 \exp\left(-\frac{x^2}{2}\right) dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^3 \exp\left(-\frac{x^2}{2}\right) x dx \end{aligned}$$

$$= \frac{1}{\sqrt{2\pi}}\left(\left[-x^3 \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} -3x^2 \exp\left(-\frac{x^2}{2}\right) dx\right)$$

$$= \frac{1}{\sqrt{2\pi}}\left\{\left[-x^3 \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} + \int_{-\infty}^{\infty} x \cdot 3x \exp\left(-\frac{x^2}{2}\right) dx\right\}$$

$$= \frac{1}{\sqrt{2\pi}}\left\{\left[-x^3 \exp\left(-\frac{x^2}{2}\right) + (-1)\, 3x \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} 3\,(-1)\exp\left(-\frac{x^2}{2}\right) dx\right\}$$

$$= \frac{1}{\sqrt{2\pi}}\left\{\left[-x^3 \exp\left(-\frac{x^2}{2}\right) - 3x \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} + 3\int_{-\infty}^{\infty} \exp\left(-\frac{x^2}{2}\right) dx\right\}$$

$$= \frac{1}{\sqrt{2\pi}}\left[-x^3 \exp\left(-\frac{x^2}{2}\right) - 3x \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} + 3\int_{-\infty}^{\infty} \underbrace{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)}_{\text{PDF of N(0, 1)}} dx$$

$$= \frac{1}{\sqrt{2\pi}}\left[-x^3 \exp\left(-\frac{x^2}{2}\right) - 3x \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} + 3$$

$$= \frac{1}{\sqrt{2\pi}}\left[-(x^3 + 3x) \cdot \exp\left(-\frac{x^2}{2}\right)\right]_{-\infty}^{\infty} + 3$$

$$= \frac{1}{\sqrt{2\pi}}\left[\frac{x^3 + 3x}{e^{\frac{x^2}{2}}}\right]_{-\infty}^{\infty} + 3$$

We stop at this point as the numerator and denominator longer both tend to infinity.

Applying L'Hospital's rule:

$$\lim_{x\to\pm\infty} \frac{x^3 + 3x}{e^{\frac{x^2}{2}}} = \lim_{x\to\pm\infty} \frac{3x^2 + 3}{xe^{\frac{x^2}{2}}} = \lim_{x\to\pm\infty} \frac{6x}{e^{\frac{x^2}{2}} \cdot (x^2 + 1)} = \lim_{x\to\pm\infty} \frac{6}{e^{\frac{x^2}{2}} \cdot (x^2 + 3x)}$$

Because

$$\lim_{x\to\infty} e^{\frac{x^2}{2}} \cdot (x^2 + 3x) = \infty$$

$$\lim_{x\to-\infty} e^{\frac{x^2}{2}} \cdot (x^2 + 3x) = -\infty$$

Therefore, in both cases

$$\lim_{x\to\pm\infty} \frac{6}{e^{\frac{x^2}{2}} \cdot (x^2 + 3x)} = 0$$

→

$$\frac{1}{\sqrt{2\pi}}\left[\frac{x^3 + 3x}{e^{\frac{x^2}{2}}}\right]_{-\infty}^{\infty} = \frac{1}{\sqrt{2\pi}}(0 - 0) = 0$$

Therefore,

$$I = 3$$

Part 2:

Monte-Carlo scheme uses simulation methods to approximate integrals in the form of expectations.

For a function $f(x)$ such that $f: [0, 1]$ → $\mathbb{R}$, and we want to evaluate the integral

$$I = \int_0^1 f(x)\, dx$$

We can rewrite the function $f(x)$ as $f(U)$ where $U \sim U(0,1)$.
To express the "area under the curve" for $f(U)$ on the domain 0 and 1, we have

$$\int_0^1 f(U)p(u)\, du$$

Where $p(u)$ is the PDF of the random variable $U(0,1)$.
We also know that integrals can be interchangeable with expectations, hence linking the above integral with its expectation form,

$$\mathbb{E}[f(U)] = \int_0^1 f(U)p(u)\, du$$

Meanwhile the density of $U(0,1)$ is 1, i.e., $p(u) = 1$. Therefore

$$\mathbb{E}[f(U)] = \int_0^1 f(U)\, du = I$$

Therefore, the problem of estimating I becomes equivalent of estimating $\mathbb{E}[f(U)]$ where $U \sim U(0,1)$.

However, this method is only suited for functions on the domain [0, 1], which often does not apply to the problem we want to solve. This means that for function with arbitrary domains will first need to be transformed so that the arbitrary domain [a, b] is converted to [0, 1].

From the above, we summarise the steps for a Monte Carlo integration:

(1) If the function has a domain other than [0, 1], transform the function so that its domain is converted to [0, 1]
(2) Generate a sequence $U_1, U_2, \dots, U_n \sim U(0,1)$ where U_i are i.i.d (independent and identically distributed)
(3) Compute $Y_i = f(U_i)$ $(i = 1, 2, \dots, n)$
(4) Estimate θ by

$$\hat{\theta} \equiv \frac{1}{n}\sum_{i=1}^{n} Y_i$$

i.e., calculating the sample mean of the Y_i terms.

Integral (i)

$$I = \int_1^3 x^2 dx$$

First, we transform the problem so that domain [1, 3] is converted to [0, 1].

Let

$$y = \frac{x-1}{3-1}$$

$$\rightarrow x = 2y + 1$$

So that as $x \to 1, y \to 0$ and x$\to 3, y \to 1$.

This gives:

$$dy = \frac{dx}{3-1} = \frac{dx}{2}$$

i.e.,

$$dx = 2\, dy$$

Substituting into I

$$I = 2\int_0^1 (2y+1)^2 dy$$
$$= 2\int_0^1 (2U+1)^2 dU$$
$$= \mathbb{E}[f(U)]$$

Where,

$$f(U) = 2(2U+1)^2$$

Integral (ii)

$$I = \int_0^\infty \exp(-x^2) dx$$

First, we transform the problem so that domain [0, ∞] is converted to [0, 1].

Let

$$y = \frac{1}{1+x}$$

$$\rightarrow x = \frac{1}{y} - 1$$

So that as $x \to 0, y \to 1$ and x→ $\infty, y \to 0$.

This gives:

$$dy = -\frac{1}{(1+x)^2} dx = -y^2 dx$$

i.e.,

$$dx = -\frac{1}{y^2} dy$$

Substituting into *I*

$$I = \int_0^1 -\frac{1}{y^2} \exp(-(\frac{1}{y} - 1)^2) dy$$
$$= \int_0^1 -\frac{1}{U^2} exp\left(1 - \frac{1}{U}\right)^2 dU$$
$$= \mathbb{E}[f(U)]$$

Where,

$$f(U) = -\frac{1}{U^2} exp\left(1 - \frac{1}{U}\right)^2$$

Integral (iii)

$$I = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^\infty x^4 \exp(-x^2/2) dx$$

First, we transform the problem so that domain [−∞, ∞] is converted to [0, 1].

Using the logistic transformation, let

$$y = \frac{1}{1+e^{-x}}$$

$$\rightarrow x = \log(\frac{y}{1-y})$$

So that as $x \to -\infty, y \to 1$ and x→ $\infty, y \to 0$.

This gives:

$$\begin{aligned}\frac{dy}{dx} &= \frac{d}{dx}\frac{1}{1+e^{-x}} \\ &= -(1+e^{-x})^{-2} \times \frac{d}{dx}(1+e^{-x}) \\ &= -(1+e^{-x})^{-2} \times \frac{d}{dx}e^{-x} \\ &= -(1+e^{-x})^{-2} \times e^{-x}\frac{d}{dx}(-x) \\ &= (1+e^{-x})^{-2} \times e^{-x} \\ &= \frac{e^{-x}}{(1+e^{-x})^2}\end{aligned}$$

i.e.,

$$dx = \frac{(1+e^{-x})^2}{e^{-x}}dy$$

Further simplify:

$$\begin{aligned}dx &= (1+e^{-x})^2 e^{x} dy \\ &= \left(1+e^{-\log\left(\frac{y}{1-y}\right)}\right)^2 e^{\log\left(\frac{y}{1-y}\right)} dy \\ &= \left(1+e^{\log\left(\frac{1-y}{y}\right)}\right)^2 \frac{y}{1-y} dy \\ &= \left(1+\frac{1-y}{y}\right)^2 \frac{y}{1-y} dy \\ &= \left(1+\frac{1-y}{y}\right)^2 \frac{y}{1-y} dy \\ &= \left(\frac{1}{y}\right)^2 \frac{y}{1-y} dy \\ &= \frac{1}{y(1-y)} dy\end{aligned}$$

Substituting into *I*

$$\begin{aligned}I &= \frac{1}{\sqrt{2\pi}}\int_0^1 \frac{1}{y(1-y)}\left(\log\left(\frac{y}{1-y}\right)\right)^4 \exp\left[-\frac{1}{2}\left(\log\left(\frac{y}{1-y}\right)\right)^2\right]dy \\ &= \frac{1}{\sqrt{2\pi}}\int_0^1 \frac{1}{U(1-U)}\left(\log\left(\frac{U}{1-U}\right)\right)^4 \exp\left[-\frac{1}{2}\left(\log\left(\frac{U}{1-U}\right)\right)^2\right]dU \\ &= \mathbb{E}[f(U)]\end{aligned}$$

Where,

$$f(U) = \frac{1}{\sqrt{2\pi}}\frac{1}{U(1-U)} \times \left(\log\left(\frac{U}{1-U}\right)\right)^4 \times \exp\left[-\frac{1}{2}\left(\log\left(\frac{U}{1-U}\right)\right)^2\right]$$

Hence, we have successfully transformed all integrals onto a domain of [0, 1].

Part 3:

From here, we perform the simulations in Python. The full Jupyter Notebook can be found in the Zip folder named as "Question 3 Script".

Packages are imported as needed.

```
import numpy as np
import pandas as pd

import cufflinks as cf
cf.set_config_file(offline=True, dimensions=((1000,600)))

from matplotlib import pyplot as plt

import plotly.express as px
px.defaults.width, px.defaults.height = 1000, 600
```

First, for each of the three transformed integrals, we define a function (respectively named int_i, int_ii, int_iii) that takes in n as an input, denoting the number of random simulations. The function simulates a sequence $U_1, U_2, \dots, U_n \sim U(0,1)$ and calculates the integral for each U_i, which are stored respectively in a list named y_i, y_ii, y_iii for integrals i, ii, iii. Then we calculate $\frac{1}{n}\sum_{i=1}^{n} f(U_i)$ to arrive at the MC integration result. The function outputs the MC integration result, calculated as the expectation of all simulated $f(U)$ values.

Integral (i)

```
# MC Integration function for problem [i]

def int_i(n):

    # Generate random numbers on U(0,1)
    np.random.seed(1000)
    U = np.random.uniform(0,1,n)

    # Defining parameters (initial x limits) a and b
    a = 1
    b = 3

    # Calculating integrals for all simulated U
    y_i = [(b-a)*(u*(b-a)+a)**2 for u in list(U)]

    # MC Integration result
    I_i = sum(y_i)/len(y_i)

    return I_i
```

Integral (ii)

```
# MC Integration function for problem [ii]

def int_ii(n):

    # Generate random numbers on U(0,1)
    np.random.seed(1000)
    U = np.random.uniform(0,1,n)

    # Calculating integrals for all simulated U
    y_ii = [np.exp(-(-1+1/u)**2)/u**2 for u in list(U)]

    # MC Integration result
    I_ii = sum(y_ii)/len(y_ii)

    return I_ii
```

Integral (iii)

```
# MC Integration function for problem [ii]

def int_iii(n):

    # Generate random numbers on U(0,1)
    np.random.seed(1000)
    U = np.random.uniform(0,1,n)

    # Calculating integrals for all simulated x
    y_iii = [(1/np.sqrt(2*np.pi))*(u*(1-u))**(-1)*(np.log(u/(1-u)))**4*np.exp(-(np.log(u/(1-u)))**2/2)
             for u in list(U)]

    # MC Integration result
    I_iii = sum(y_iii)/len(y_iii)

    return I_iii
```

Next, we create a list of increasing values for number of simulations n. We start from 1,000 and end at 10,000,000 simulations, creating 20 evenly spaced values. For each n, we compute the MC integration value for all three integrals. The results are stored in three listsI_i, I_ii, I_iii (each list is a list of 20 numerically calculated integral values).

```
# Create 20 increasing values of n from 1,000 to 10,000,000
N = list(np.linspace(1000, 10000000, 20))
N = [int(n) for n in N]

# Calculating the Monte Carlo integration for each integral at each n value
I_i = [int_i(n) for n in N]
I_ii = [int_ii(n) for n in N]
I_iii = [int_iii(n) for n in N]
```

We then create three variables exact_i, exact_ii, exact_iii storing the exact solutions calculated in Part I.

```
# Create variables storing the exact solutions

exact_i = 26/3
exact_ii = np.sqrt(np.pi)/2
exact_iii = 3

print(exact_i, exact_ii, exact_iii)
```

```
8.666666666666666 0.8862269254527579 3
```

We then get the error between the MC integration result vs. the actual result at each n, stored in three lists errors_i, errors_ii, errors_iii. The errors are merged into one data frame for plotting.

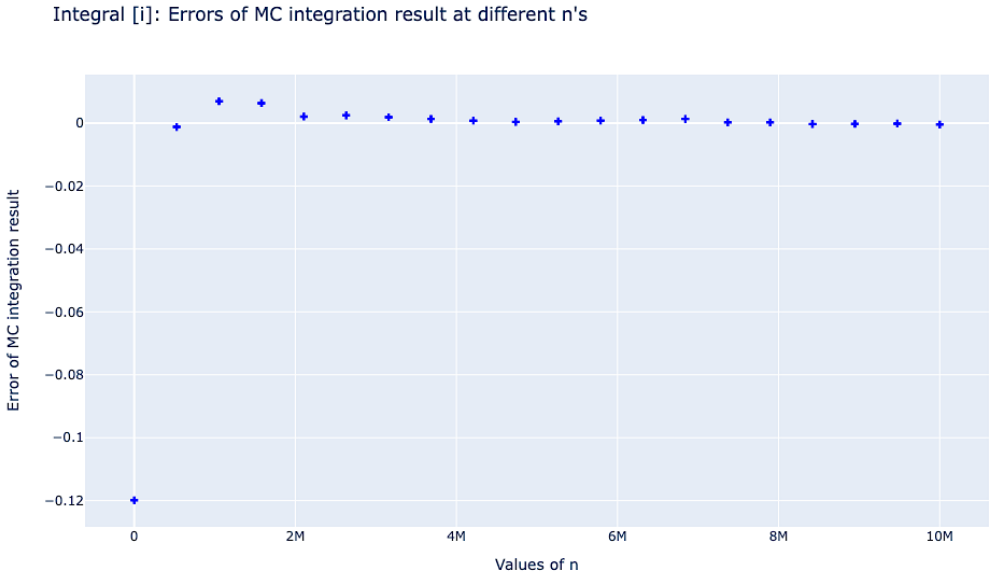
```
# Create three lists storing the errors from exact solution at each n

errors_i = [I_i-exact_i for I_i in I_i]
errors_ii = [I_ii-exact_ii for I_ii in I_ii]
errors_iii = [I_iii-exact_iii for I_iii in I_iii]

# Put the final errors into a dataframe for plotting
errors = pd.DataFrame.from_dict({
    'n': N,
    'Error i': errors_i,
    'Error ii': errors_ii,
    'Error iii': errors_iii,
})
```

The errors for each integral are then plotted. For each plot, the x axis denotes each value of n, the y axis denotes the error values.

Integral (i)

$$I = \int_1^3 x^2 dx$$

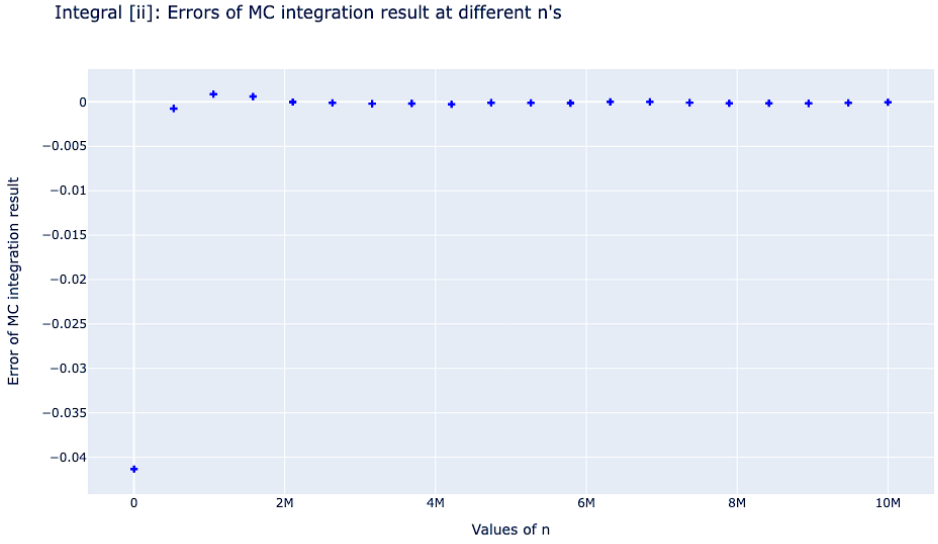
```
# Plot the final errors at different n's - Integral i
fig = px.scatter(
    errors, x='n', y='Error i',
    labels={'n': 'Values of n', 'Error i': 'Error of MC integration result'},
    title="Integral [i]: Errors of MC integration result at different n's"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```



Integral (ii)

$$I = \int_0^\infty \exp(-x^2) dx$$

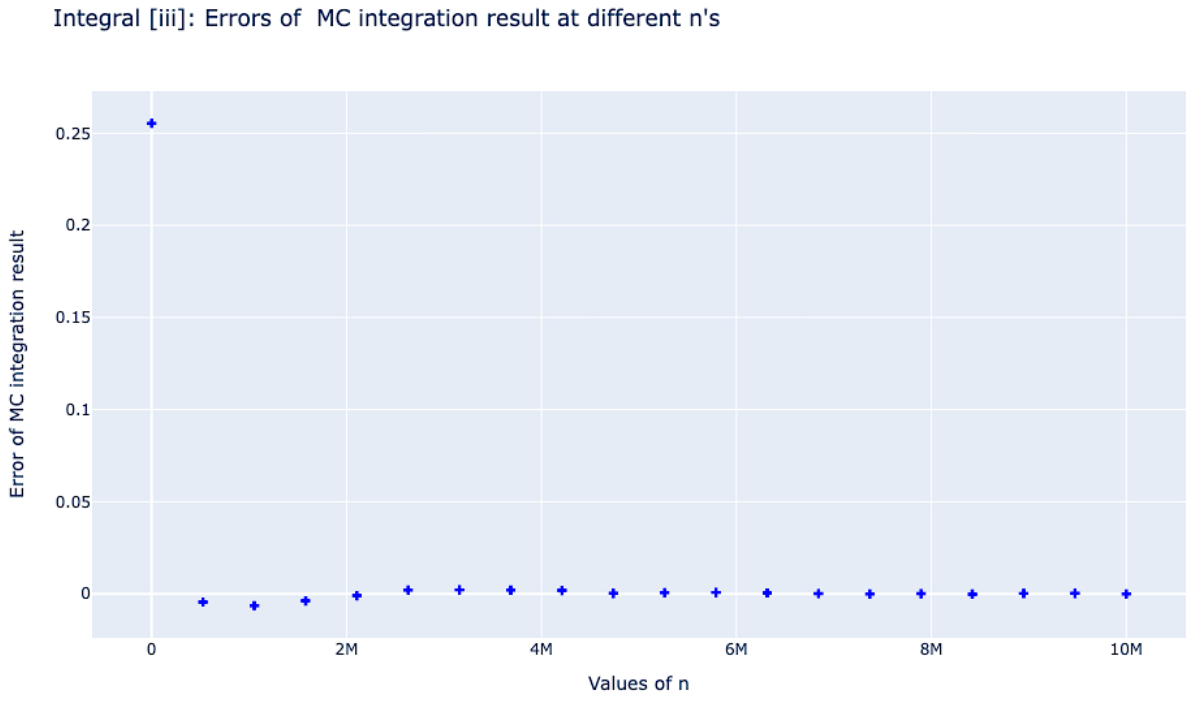
```
# Plot the final errors at different n's - Integral ii
fig = px.scatter(
    errors, x='n', y='Error ii',
    labels={'n': 'Values of n', 'Error ii': 'Error of MC integration result'},
    title="Integral [ii]: Errors of MC integration result at different n's"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```



Integral (iii)

$$I = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^4 \exp\left(-x^2/2\right) dx$$

```
# Plot the final errors at different n's - Integral iii
fig = px.scatter(
    errors, x='n', y='Error iii',
    labels={'n': 'Values of n', 'Error iii': 'Error of MC integration result'},
    title="Integral [iii]: Errors of  MC integration result at different n's"
     ).update_traces(mode='markers', marker=dict(symbol='cross', color='blue'))
fig.show()
```



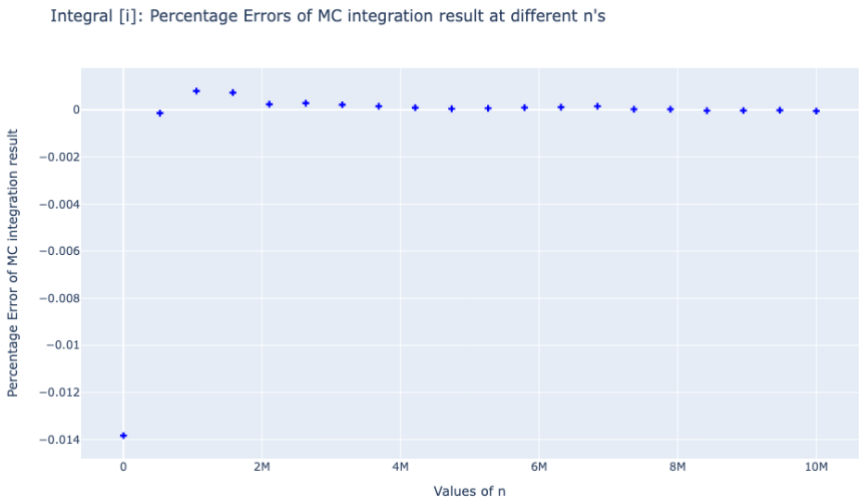
From all three plots, we observe see that as n increases, the error converges to zero. In other words, the integral calculated numerically using MC simulation becomes more accurate as the random numbers increase.

However, we should note that the absolute value of the errors may be misleading as it is the percentage errors that says more about the accuracy. Therefore, we add three columns to the data frame storing the percentage errors from the actual result at every value of n.

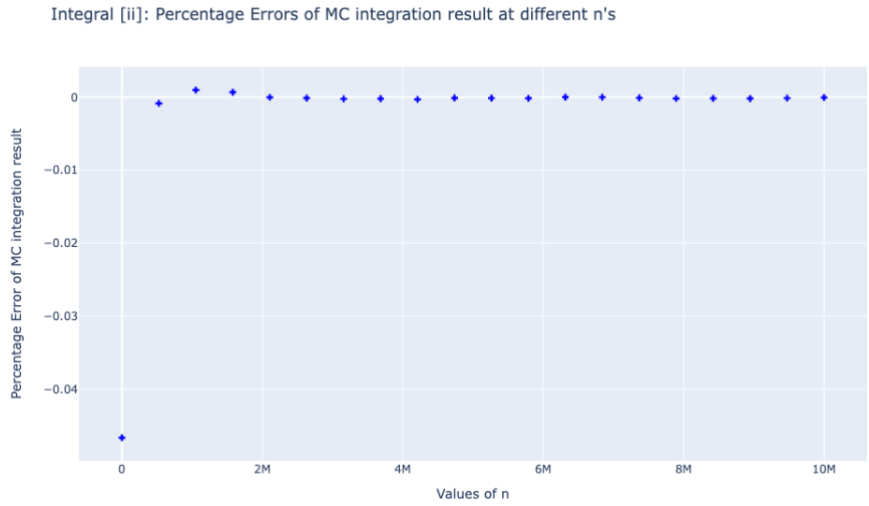
```
# Calculating percentage errors
errors['Error i Percentage'] = errors['Error i']/exact_i
errors['Error ii Percentage'] = errors['Error ii']/exact_ii
errors['Error iii Percentage'] = errors['Error iii']/exact_iii
```

Then we plot the percentage errors the same way as before:

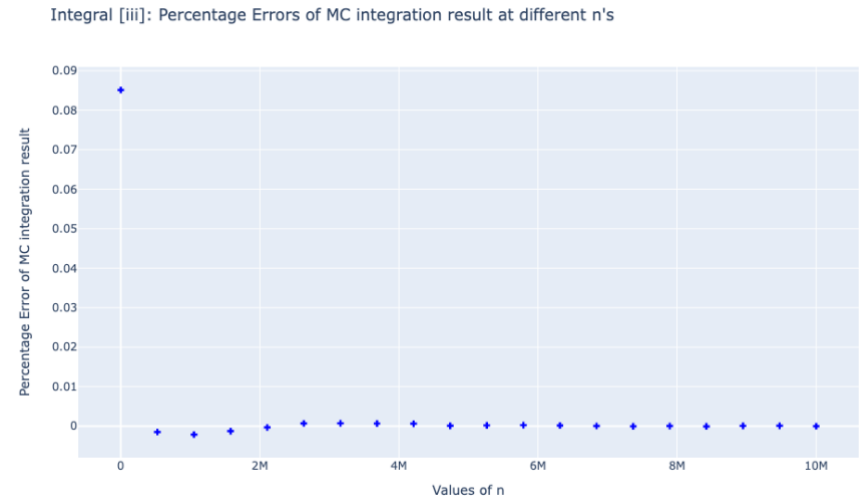
Integral (i)



Integral (ii)

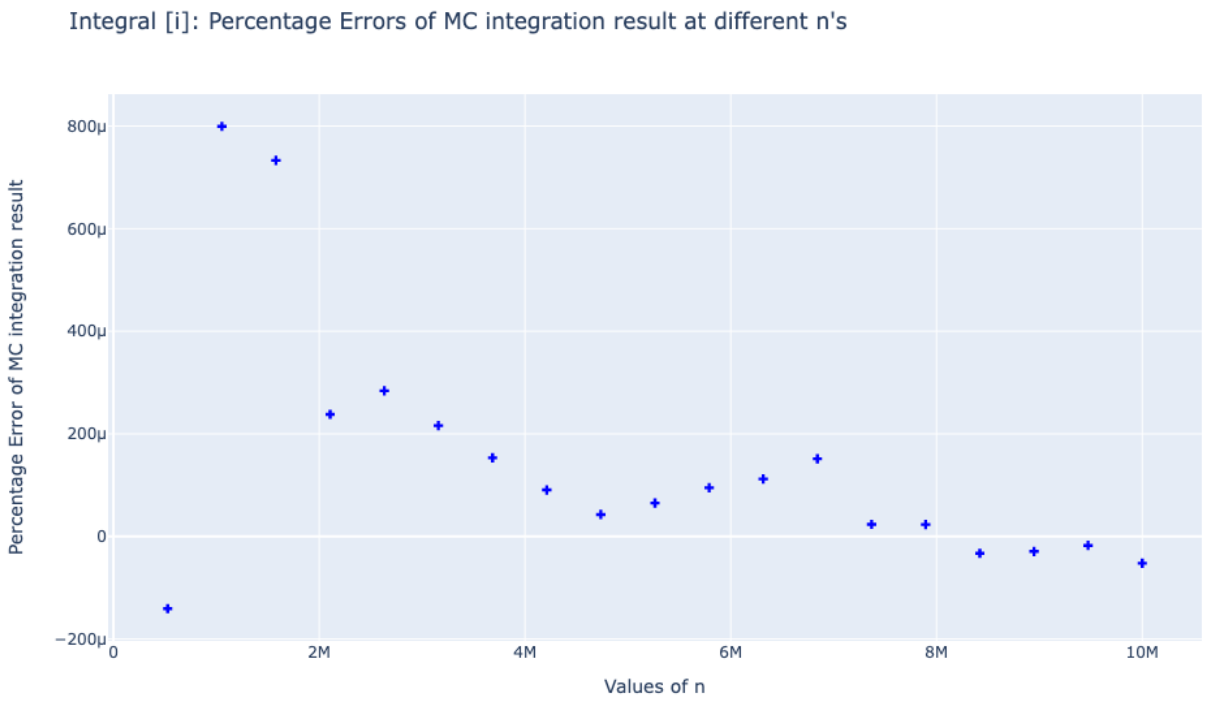


Integral (iii)

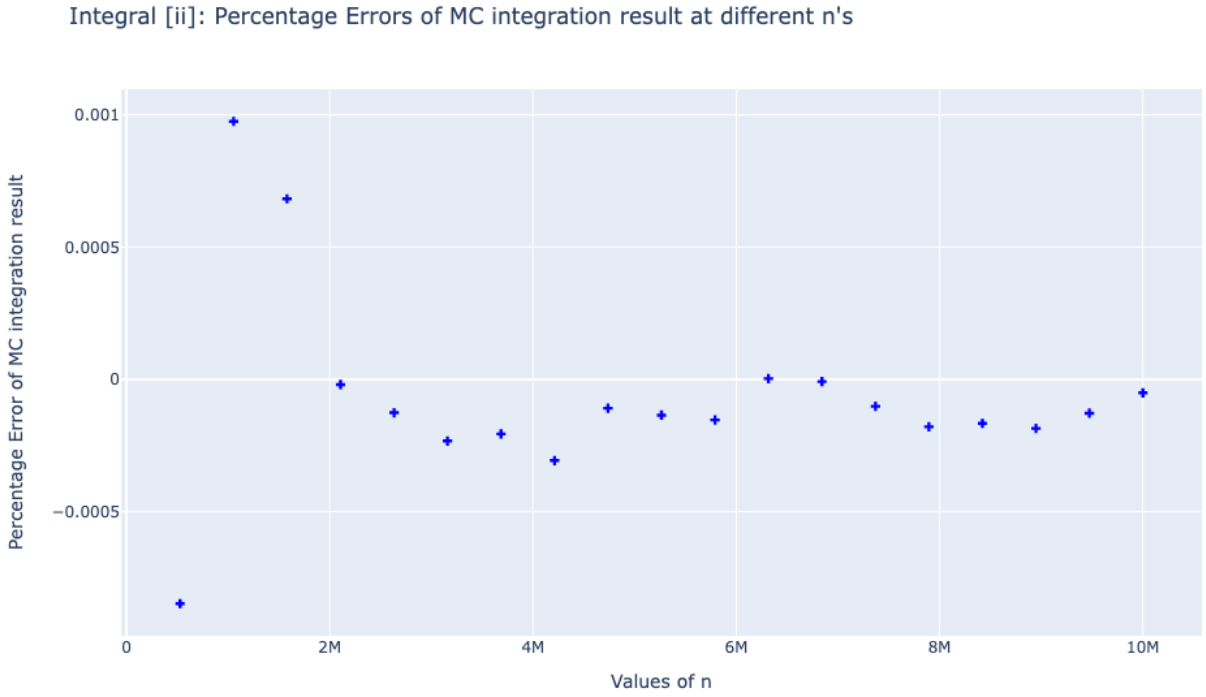


For the purpose of adjusting the scaling (since the value at the first observation is disproportionally large, making it hard to see how the errors change from the second observation onwards), we remove the first value from our plot.

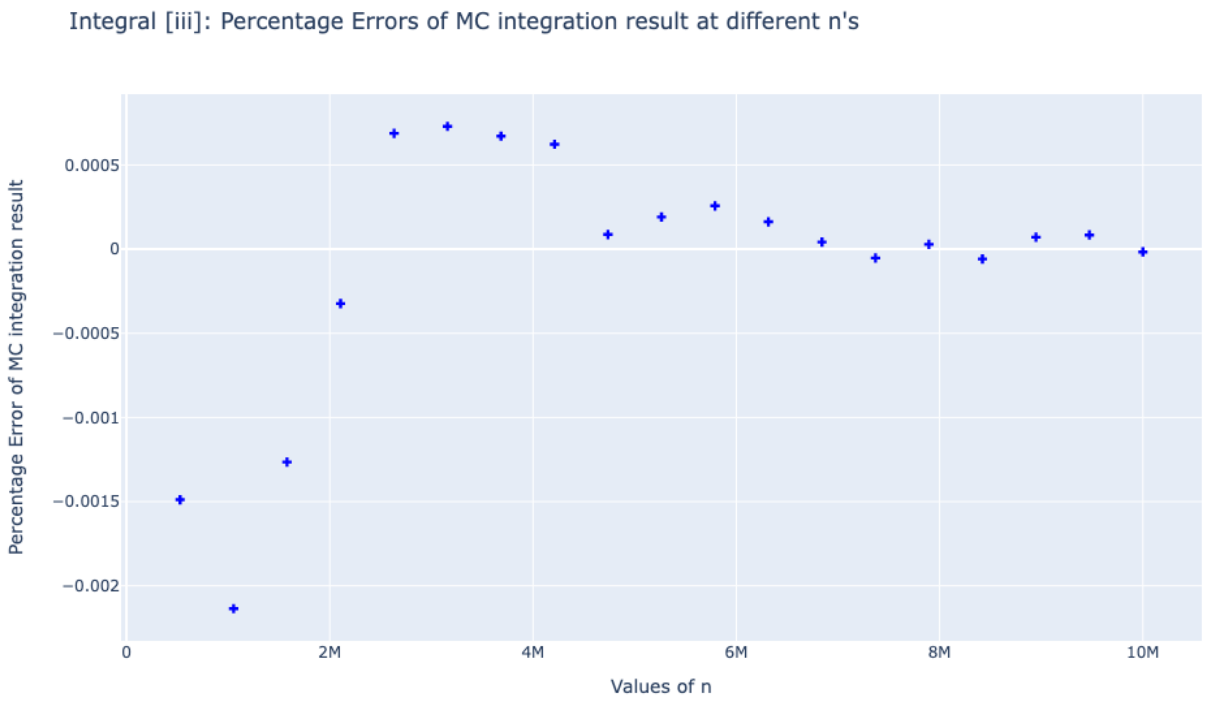
Integral (i)



Integral (ii)



Integral (iii)



From all three plots, we observe see that as n increases, the error converges to zero. In other words, the integral calculated numerically using MC simulation becomes more accurate as the random numbers increase.

This implies that when approaching similar problems in future using Monte Carlo simulations, we should be able to roughly choose the number of simulations to perform, depending on the accuracy requirements of the question / context.

Conclusion:

In this question, we solved a three integrals numerically using Monte Carlo simulation and compared it to the exact values computed by hand. Using the fact that integrals are also expectations, we transformed the integrals so that the arbitrary domains are converted to [0, 1] and expressed the integrals as an expectation of a function of the uniform distribution $U(0,1)$, i.e., $\mathbb{E}[f(U)]$. We performed simulations in Python, generating random numbers from $U(0,1)$ and then computing the expectations. We find that as the number of simulations increase, the percentage error to the actual values converges to zero, implying that numerically calculated integrals using MC simulation becomes more accurate as the number of simulations become larger.

END OF REPORT