CQF - Exam 2

Imports

In [ ]:
```
import pandas as pd
import numpy as np
import math
from scipy.stats import norm
from statistics import mean
from scipy.stats.mstats import gmean
import matplotlib.pyplot as plt
%matplotlib inline
```

Question 1

Parameters

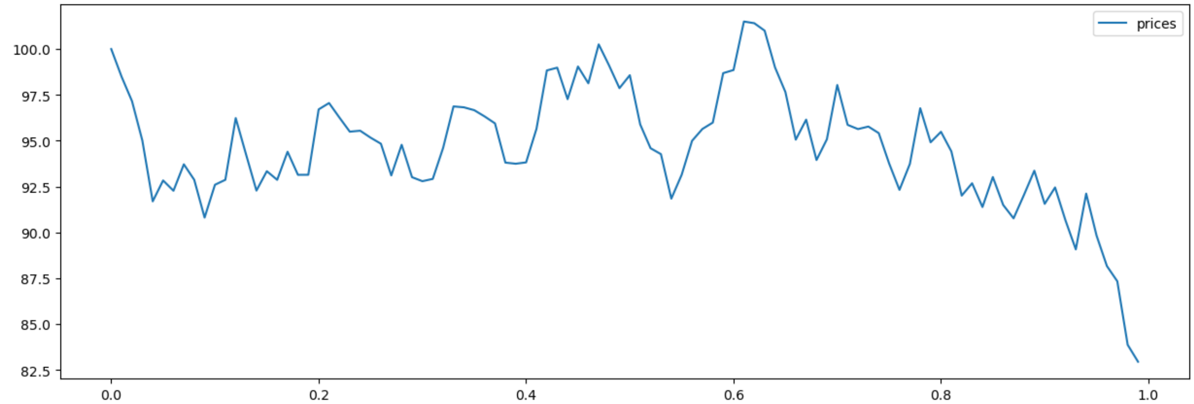
In [ ]:
```
S0 = 100.0
T = 1.0
sigma = 0.2
r = 0.05
N = 100
dt = T / N
ts = np.arange(0, T, dt)
dB = np.concatenate((np.zeros(1), np.random.randn(N-1)*np.sqrt(dt)))
B = np.cumsum(dB)
```

Euler-Maruyama scheme

$$S_{n+1} = S_n + rS_n\Delta t + \sigma S_n \Delta B$$

In [ ]:
```
S = np.zeros(len(ts))
S[0] = S0

for i in range(1, len(ts)):
    S[i] = S[i-1] + r*S[i-1]*dt + sigma*S[i-1]*dB[i]

em_df = pd.DataFrame(index=ts, data=S, columns=['prices'])
em_df.plot(colormap='tab10', figsize=[15,5])
```

Out[ ]: `<AxesSubplot:>`

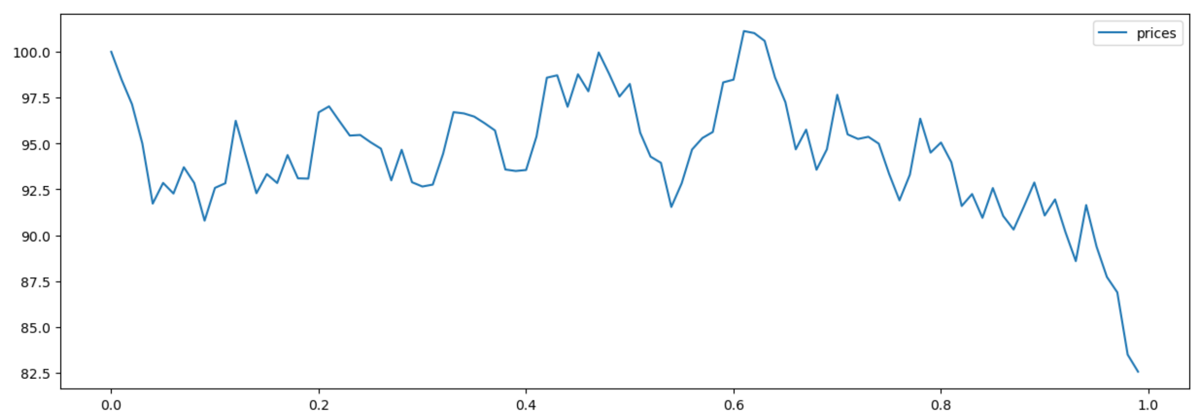


Milstein schema

$$S_{n+1} = S_n + rS_n\Delta t + \sigma S_n\Delta B + \frac{1}{2}\sigma^2 S_n(\Delta B^2 - \Delta t)$$

In [ ]:
```
S = np.zeros(len(ts))
S[0] = S0

for i in range(1, len(ts)):
    S[i] = S[i-1] + r*S[i-1]*dt + sigma*S[i-1]*dB[i] \
        + 0.5*(sigma**2)*S[i-1]*(dB[i]**2-dt)

milstein_df = pd.DataFrame(index=ts, data=S, columns=['prices'])
milstein_df.plot(colormap='tab10', figsize=[15,5])
```

Out[ ]: <AxesSubplot:>

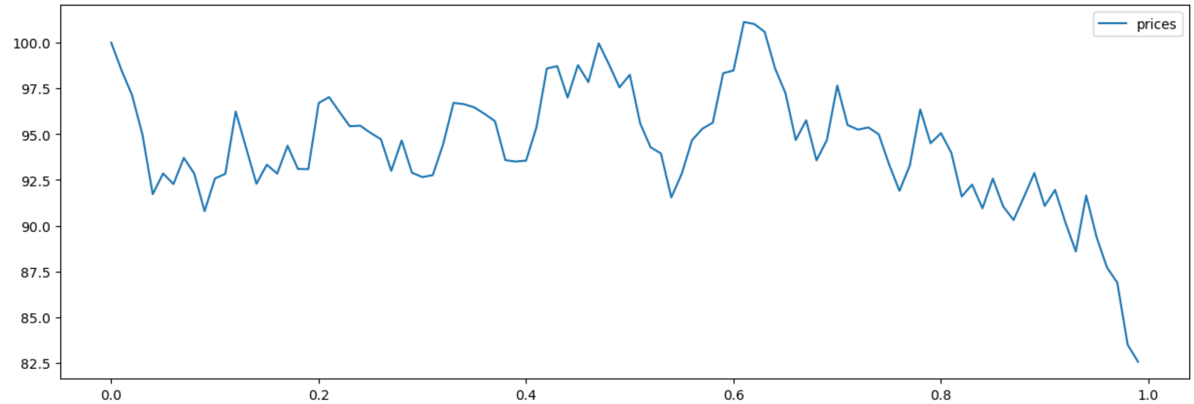


Closed Form Solution

$$S_{n+1} = S_n \exp\left((r - \frac{\sigma^2}{2})t + \sigma B(t)\right)$$

In [ ]:
```
S = S0 * np.exp((r-0.5*sigma**2)*ts + sigma*B)
closed_df = pd.DataFrame(index=ts, data=S, columns=['prices'])
closed_df.plot(colormap='tab10', figsize=[15,5])
```

Out[ ]: <AxesSubplot:>

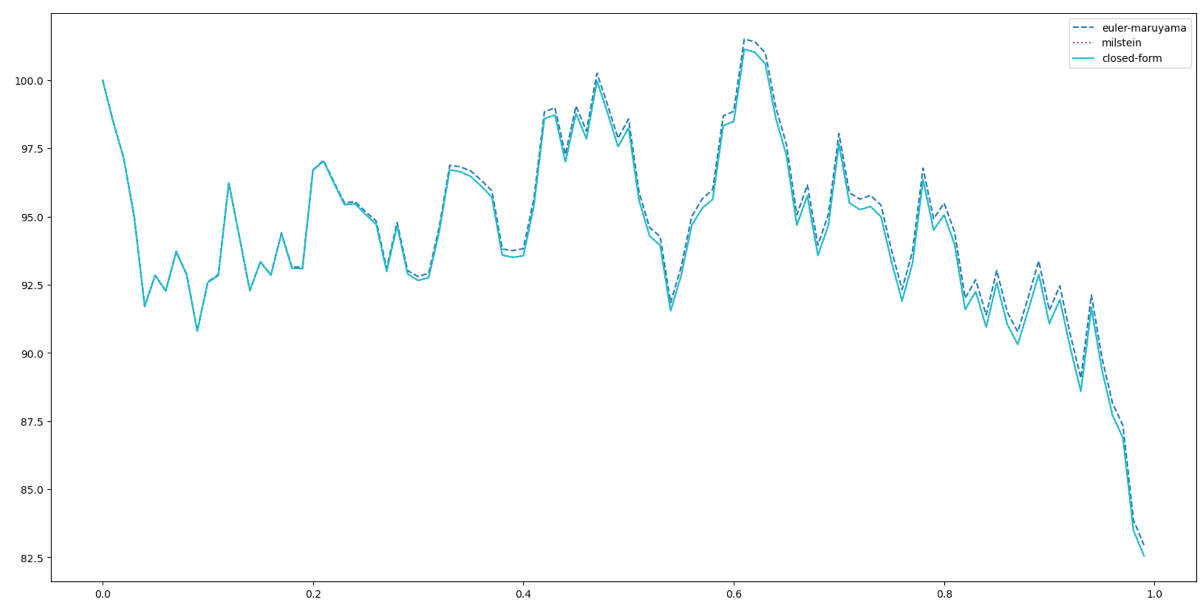


Schemas Comparison

In [ ]:

```
compare_df = em_df.copy()
compare_df.columns = ['euler-maruyama']
compare_df['milstein'] = milstein_df['prices']
compare_df['closed-form'] = closed_df['prices']
compare_df.plot(style=['--', ':', '-'],colormap='tab10', figsize=[20,10])
```

Out[ ]: <AxesSubplot:>



Plot above shows that Milstein method gives us a better approximation to the close form solution. The Milstein method has an error of $O(\delta t^2)$ while Euler-Maruyama has an error of $O(\delta t)$.

Asian Option Payoffs

In [ ]:

```
S0 = 100.0
T = 1.0
sigma = 0.2
r = 0.05
```

```
N = 252*4
dt = T / N
ts = np.arange(0, T, dt)
n_simul = 1000
simulations = [np.concatenate((np.zeros(1), np.random.randn(N-1)*np.sqrt(dt))) for

euler_paths = []
milstein_paths = []
close_paths = []

for s in simulations:
    dB = s
    B = np.cumsum(dB)
    S = np.zeros(len(ts))
    S[0] = S0

    for i in range(1, len(ts)):
        S[i] = S[i-1] + r*S[i-1]*dt + sigma*S[i-1]*s[i]

    euler_paths.append(S)

    S = np.zeros(len(ts))
    S[0] = S0

    for i in range(1, len(ts)):
        S[i] = S[i-1] + r*S[i-1]*dt + sigma*S[i-1]*s[i] \
            + 0.5*(sigma**2)*S[i-1]*(dB[i]**2-dt)

    milstein_paths.append(S)

    S = S0 * np.exp((r-0.5*sigma**2)*ts + sigma*B)
    close_paths.append(S)
```

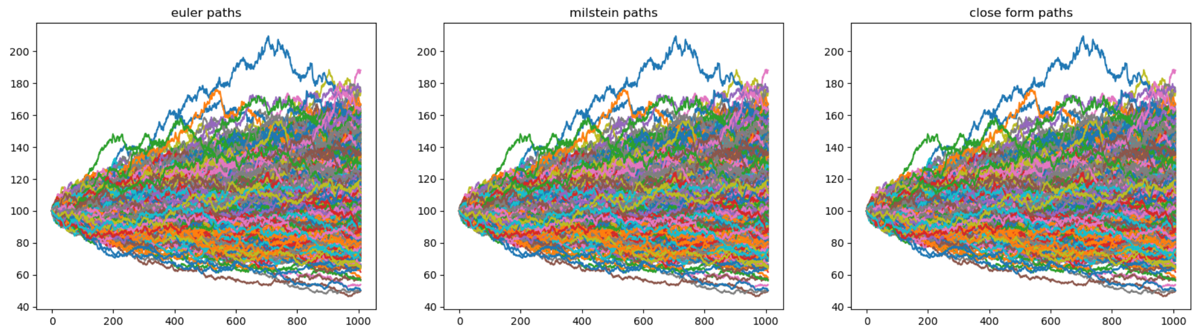
Cheching if paths are been generated accordingly

In [ ]:

```
figure, axis = plt.subplots(1, 3, figsize=[20, 5])

for s in euler_paths:
    axis[0].plot(s)
axis[0].set_title('euler paths')

for s in milstein_paths:
    axis[1].plot(s)
axis[1].set_title('milstein paths')

for s in close_paths:
    axis[2].plot(s)
axis[2].set_title('close form paths')
```

Out[ ]:

```
Text(0.5, 1.0, 'close form paths')
```



In [ ]:
```
S = np.zeros(1000)
print(int(len(S) / 252))
ttt = S[::1]
len(ttt)
```

```
3
```

Out[ ]: 1000

In [ ]:
```
def vanilla_payoff(S, K, CP):
    return max(S[-1]-K, 0) if CP == 'C' else max(K-S[-1], 0)

def asian_payoff(S, K, CP, avg_func, fixed_strike=True, continuous_sampling=True):

    seq = 1 if continuous_sampling else int(len(S) / 252)

    if CP == 'C':
        return max(avg_func(S[::seq])-K, 0) if fixed_strike else max(S[-1]-avg_func
    else:
        return max(K-avg_func(S[::seq]), 0) if fixed_strike else max(avg_func(S[::s

E = 100

df = pd.DataFrame(index=['euler-maruyama','milstein', 'closed'], data=[
    [
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
    ],
    [
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
```

```
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
    ],
    [
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=mean, continuous
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='C', avg_func=gmean, continuou
        np.exp(-r*T)*mean([asian_payoff(S=S, K=E, CP='P', avg_func=gmean, continuou
    ]],
    columns=[
        'call mean fixed strike cont. sampl',
        'put mean fixed strike cont. sampl',
        'call mean float strie cont. sampl',
        'put mean float strike cont. sampl',
        'call gmean fixed strike cont. sampl',
        'put gmean fixed strike cont. sampl',
        'call gmean float strike cont. sampl',
        'put gmean float strike cont. sampl',

        'call mean fixed strike fixed sampl',
        'put mean fixed strike fixed sampl',
        'call mean float strie fixed sampl',
        'put mean float strike fixed sampl',
        'call gmean fixed strike fixed sampl',
        'put gmean fixed strike fixed sampl',
        'call gmean float strike fixed sampl',
        'put gmean float strike fixed sampl'
        ])

df
```

Out[ ]:

	call mean fixed strike cont. sampl	put mean fixed strike cont. sampl	call mean float strie cont. sampl	put mean float strike cont. sampl	call gmean fixed strike cont. sampl	put gmean fixed strike cont. sampl	call gmean float strike cont. sampl	put gmean float strike cont. sampl	ca mea fixe strik fixe samp
euler-maruyama	5.873826	3.396822	6.153336	3.351903	5.650191	3.519078	6.372901	3.225578	5.85865
milstein	5.872158	3.396561	6.151815	3.351107	5.648300	3.518545	6.371413	3.224863	5.85696
closed	5.872420	3.396755	6.152091	3.351296	5.648541	3.518751	6.371709	3.225039	5.85722

Question 2

The model problem:

$$\frac{d^2y}{dx^2} = P(x)\frac{dy}{dx} + Q(x)y = f(x) \tag{1}$$

with boundary conditions:

$$y(a) = \alpha$$
$$y(b) = \beta$$

Let

$$y_i = y(x_i), P_i = P(x_i), Q_i = Q(x_i), f_i = f(x_i)$$

and using a Taylor serie expansion we can approximante the derivative terms as following:

$$\frac{dy}{dx} \approx \frac{y_{i+1} - y_{i-1}}{2\delta x} \tag{2}$$

$$\frac{d^2y}{dx^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\delta x^2} \tag{3}$$

Substituting (2) and (3) in (1), and multiplying both sides by δx^2 we have:

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

and rearranging:

$$\left(1 - \frac{\delta x}{2}P_i\right)y_{i-1} + (-2 + \delta x^2 Q_i)y_i + \left(1 + \frac{\delta x}{2}P_i\right)y_{i+1} = \delta x^2 f_i \tag{4}$$

with boundary conditions:

$$y_0 = \alpha$$
$$y_n = \beta$$

To represent the problem as a matrix inversion problem $Ax = b$, we define:

$$A_i = 1 - \frac{\delta x}{2} P_i$$
$$B_i = -2 + \delta x^2 Q_i$$
$$C_i = 1 + \frac{\delta x}{2} P_i$$

And thus, the matrices A, x and b will have the form:

$$A = \begin{vmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ A_1 & B_1 & C_1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_2 & C_2 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & A_3 & B_3 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & A_{n-2} & B_{n-2} & C_{n-2} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & A_{n-1} & B_{n-1} & C_{n-1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 \end{vmatrix}$$

$$x = \begin{vmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{vmatrix}$$

$$b = \begin{vmatrix} f_0 \delta x^2 = \alpha \delta x^2 \\ f_1 \delta x^2 \\ f_2 \delta x^2 \\ f_3 \delta x^2 \\ \vdots \\ f_{n-2} \delta x^2 \\ f_{n-1} \delta x^2 \\ f_n \delta x^2 = \beta \delta x^2 \end{vmatrix}$$

Implementing the algorithm:

In [ ]:
```
def solve_problem(P, Q, n, x_0, x_n, y_0, y_n, func_problem):
    delta_x = (x_n - x_0)/n
```

```
    xs = np.arange(x_0, x_n+delta_x, delta_x)
    fs = [func_problem(i)* (delta_x**2) for i in xs]
    A = np.identity(n+1)

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

    b = fs
    b[0] = y_0 * (delta_x**2)
    b[n] = y_n * (delta_x**2)
    ys = np.linalg.solve(A,b)
    return xs, ys

P = 3.
Q = 2.
x_0 = 1.
x_n = 2.
y_0 = 1.
y_n = 6.

df = []
for n in [10, 50, 100]:
    xs, ys = solve_problem(P=P, Q=Q, n=n, x_0=x_0, x_n=x_n, y_0=y_0, y_n=y_n, func_
    df.append(pd.DataFrame(data={'x': xs, f'y [n={n}]':ys}))

df = pd.concat(df)
```

Solving the problem for $n = 10$ and show de results for x_i and y_i:

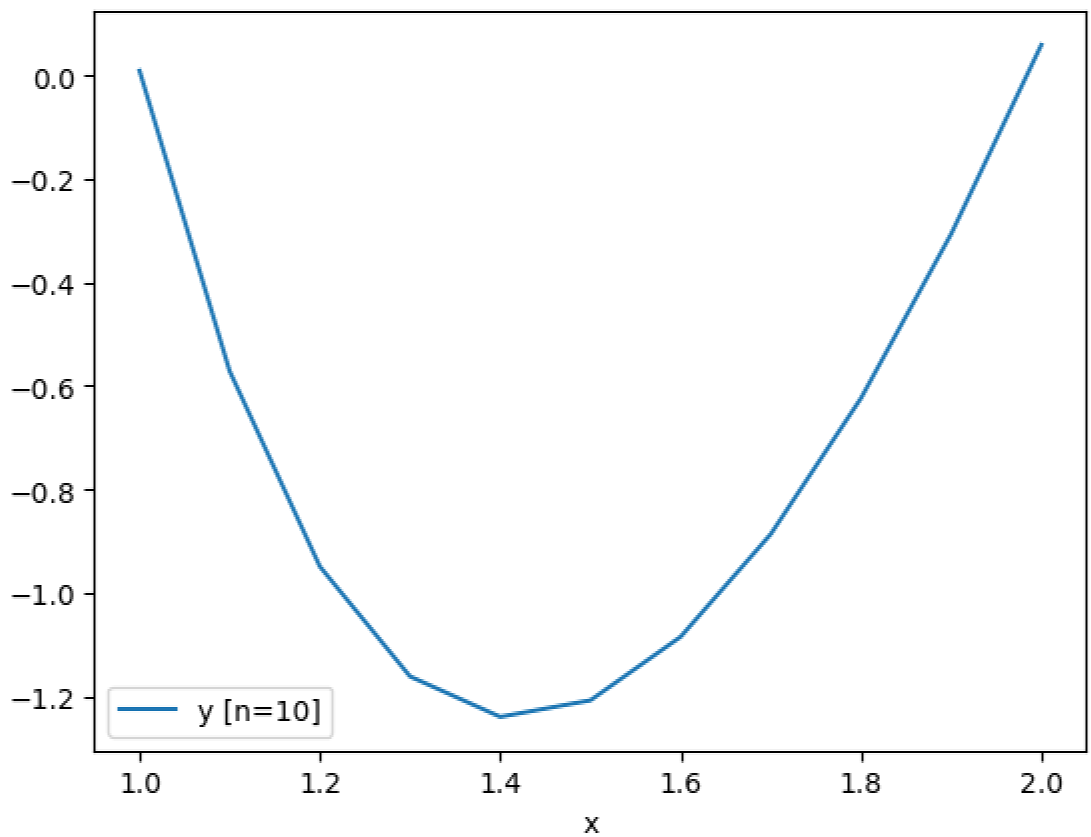
In [ ]:
```
df[['x','y [n=10]']].dropna()
```

Out[ ]:

	x	y [n=10]
0	1.0	0.010000
1	1.1	-0.571491
2	1.2	-0.949263
3	1.3	-1.161890
4	1.4	-1.240060
5	1.5	-1.208097
6	1.6	-1.085201
7	1.7	-0.886449
8	1.8	-0.623606
9	1.9	-0.305791
10	2.0	0.060000

In [ ]:
```
df.plot(x='x', y='y [n=10]')
```

Out[ ]: <AxesSubplot:xlabel='x'>



Solving the problem for $n = 50$ and show de results for x_i and y_i:

In [ ]:
```
df[['x','y [n=50]']].dropna()
```

Out[ ]:

	x	y [n=50]
0	1.00	0.000400
1	1.02	-0.137543
2	1.04	-0.265728
3	1.06	-0.384560
4	1.08	-0.494425
5	1.10	-0.595694
6	1.12	-0.688722
7	1.14	-0.773847
8	1.16	-0.851393
9	1.18	-0.921671
10	1.20	-0.984976
11	1.22	-1.041591
12	1.24	-1.091788
13	1.26	-1.135824
14	1.28	-1.173946
15	1.30	-1.206391
16	1.32	-1.233383
17	1.34	-1.255139
18	1.36	-1.271863
19	1.38	-1.283751
20	1.40	-1.290992
21	1.42	-1.293764
22	1.44	-1.292237
23	1.46	-1.286574
24	1.48	-1.276931
25	1.50	-1.263455
26	1.52	-1.246287
27	1.54	-1.225563
28	1.56	-1.201410
29	1.58	-1.173950
30	1.60	-1.143301
31	1.62	-1.109572
32	1.64	-1.072869

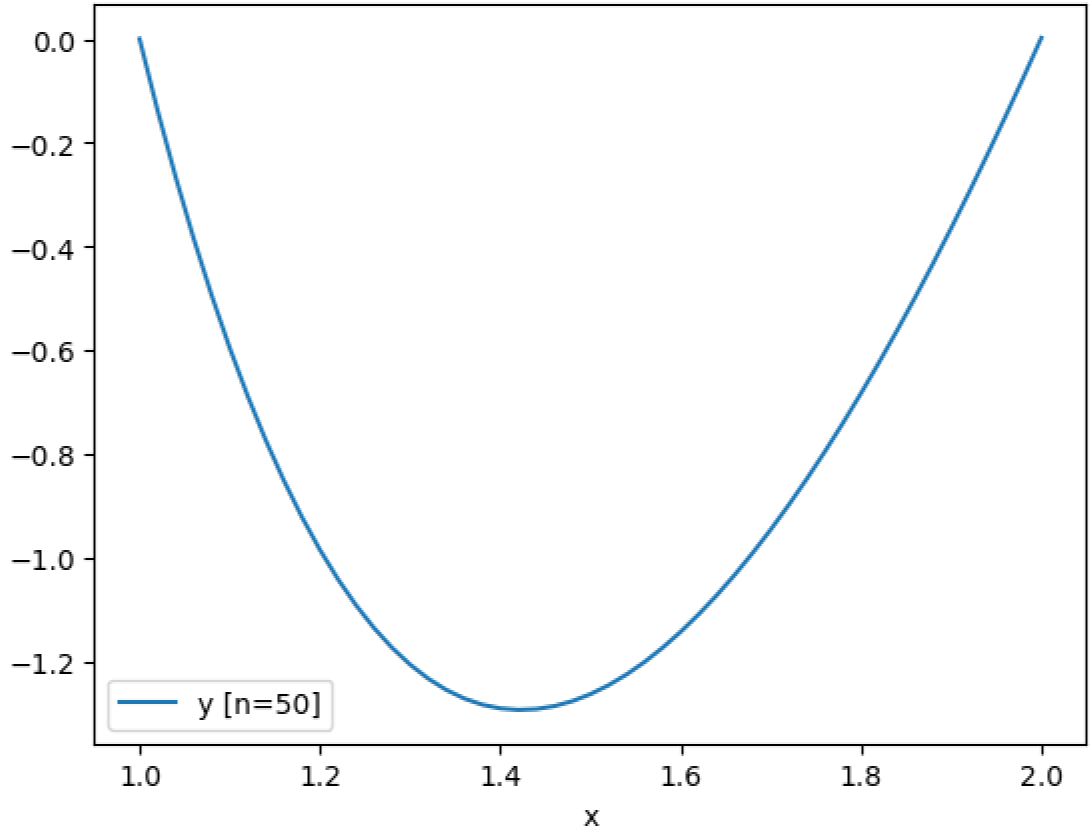
	x	y [n=50]
33	1.66	-1.033293
34	1.68	-0.990939
35	1.70	-0.945899
36	1.72	-0.898258
37	1.74	-0.848099
38	1.76	-0.795500
39	1.78	-0.740536
40	1.80	-0.683276
41	1.82	-0.623789
42	1.84	-0.562136
43	1.86	-0.498379
44	1.88	-0.432575
45	1.90	-0.364778
46	1.92	-0.295039
47	1.94	-0.223407
48	1.96	-0.149928
49	1.98	-0.074645
50	2.00	0.002400

In [ ]:

```
df.plot(x='x', y='y [n=50]')
```

Out[ ]:

```
<AxesSubplot:xlabel='x'>
```



Solving the problem for $n = 100$ and show de results for x_i and y_i:

In [ ]:
```
df[['x','y [n=100]']].dropna()
```

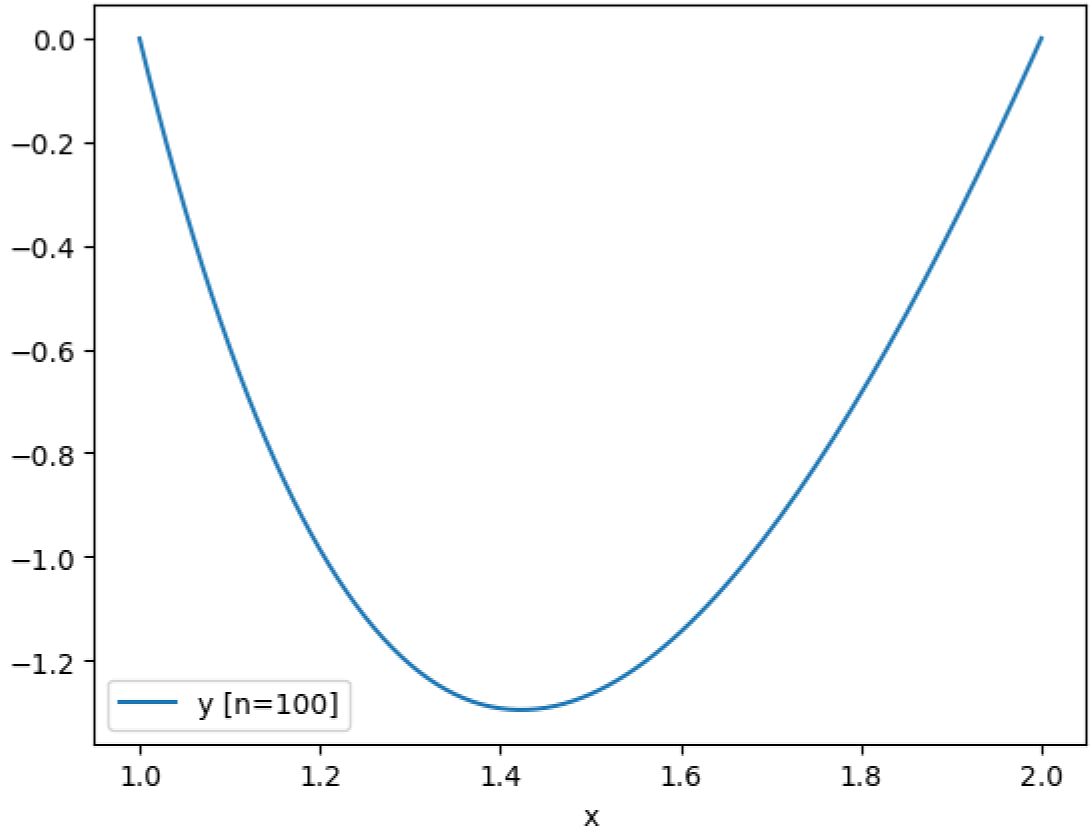
Out[ ]:

	x	y [n=100]
0	1.00	0.000100
1	1.01	-0.070168
2	1.02	-0.137942
3	1.03	-0.203277
4	1.04	-0.266222
...	...	...
96	1.96	-0.151747
97	1.97	-0.114324
98	1.98	-0.076455
99	1.99	-0.038146
100	2.00	0.000600

101 rows × 2 columns

In [ ]:
```
df.plot(x='x', y='y [n=100]')
```

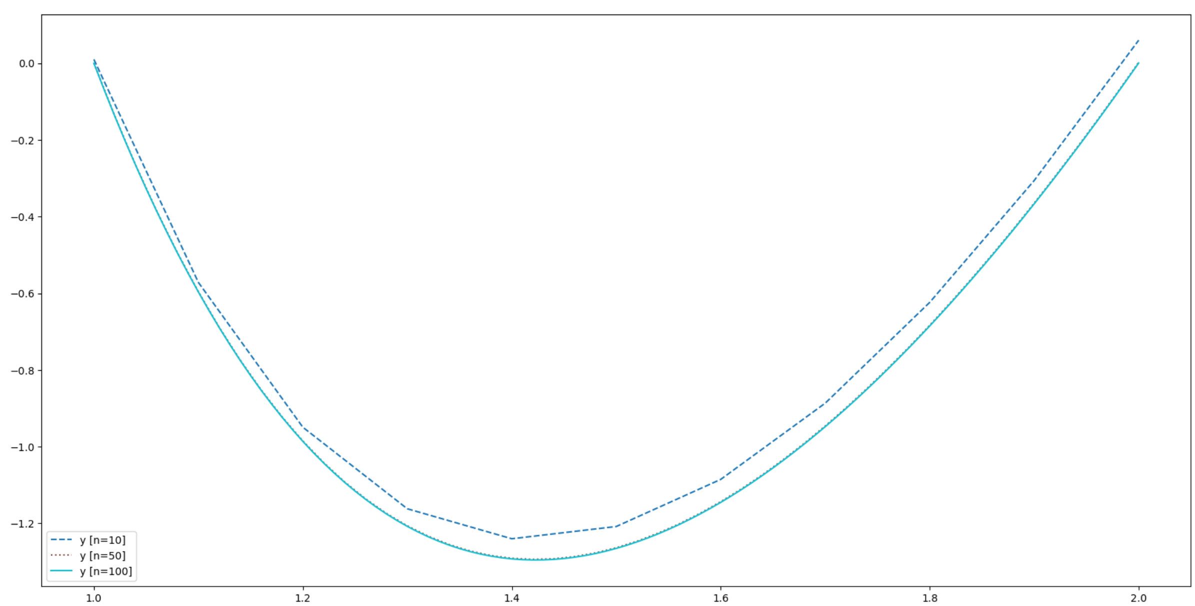
Out[ ]: <AxesSubplot:xlabel='x'>



Ploting the function for $n = 10, 50, 100$, we can see that the more we increase the n the better the approximation and smoother the curve.

In [ ]:
```
df.plot(x='x', style=['--', ':', '-'], colormap='tab10', figsize=[20,10])
```

Out[ ]: <AxesSubplot:xlabel='x'>



Question 3 - Monte Carlo Integration

Define function for Monte Carlo integration

In [ ]:
```
def monte_carlo_integration(func, N, limit_inf, limit_sup):
    xs = np.random.uniform(limit_inf, limit_sup, size=N)
    ys = [func(x) for x in xs]
    return (limit_sup - limit_inf) / N * np.sum(ys)
```

I.

$$\int_1^3 x^2 dx = \frac{1}{3}x^3\Big|_1^3 = \frac{26}{3} = 8.666667$$

In [ ]:
```
def func(x):
    return x**2

exact_value = 8.666667
limit_inf = 1
limit_sup = 3
```

In [ ]:
```
samples = [10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480, 40960, 81
df = pd.DataFrame(data=[exact_value]*len(samples), index=samples, columns=['exact_v
df['mc_value'] = df.apply(lambda x: monte_carlo_integration(func, x.name, limit_inf
df['error'] = df.exact_value - df.mc_value
df
```

Out[ ]:

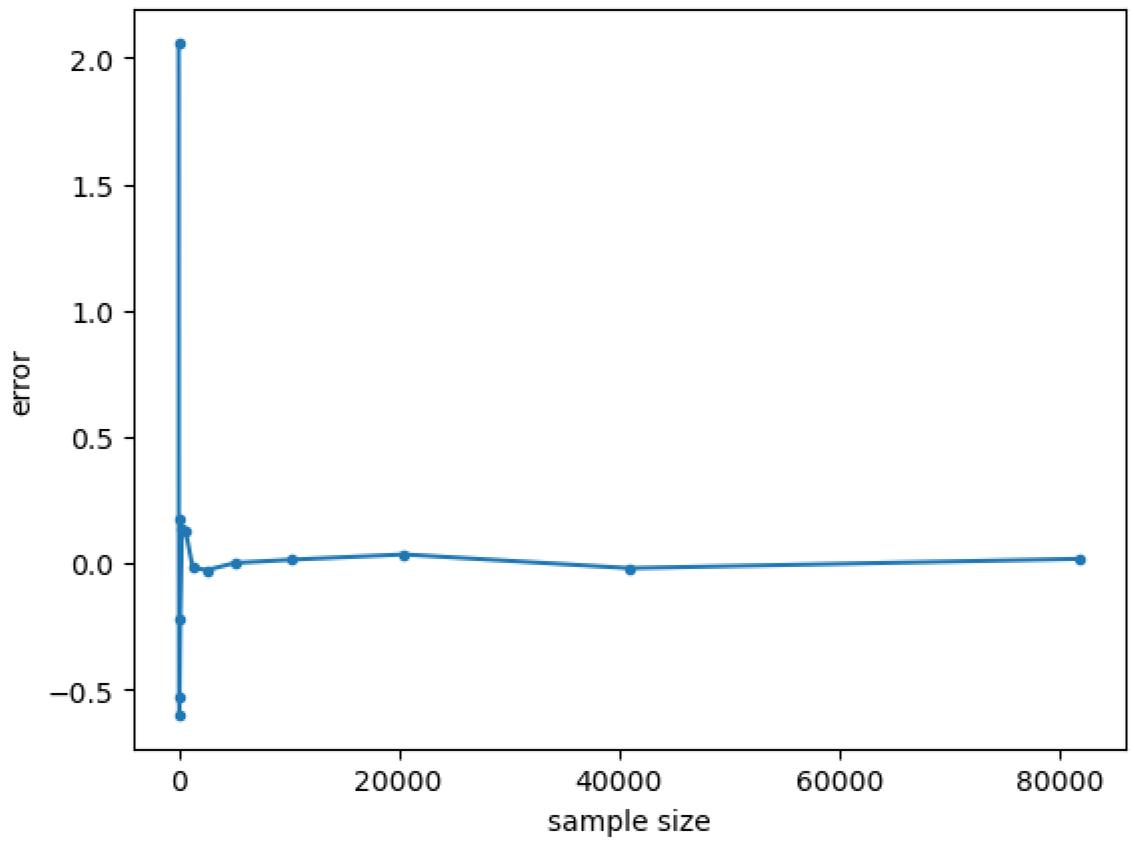
	exact_value	mc_value	error
10	8.666667	6.607261	2.059406
20	8.666667	8.491113	0.175554
40	8.666667	9.196002	-0.529335
80	8.666667	9.266938	-0.600271
160	8.666667	8.887760	-0.221093
320	8.666667	8.527673	0.138994
640	8.666667	8.536157	0.130510
1280	8.666667	8.683538	-0.016871
2560	8.666667	8.694222	-0.027555
5120	8.666667	8.665624	0.001043
10240	8.666667	8.652785	0.013882
20480	8.666667	8.631926	0.034741
40960	8.666667	8.686505	-0.019838
81920	8.666667	8.649351	0.017316

In [ ]:

```
ax = df.error.plot(marker='.')
ax.set_xlabel('sample size')
ax.set_ylabel('error')
```

Out[ ]:

```
Text(0, 0.5, 'error')
```



II.

$$\int_0^\infty e^{-x^2} dx = \frac{1}{2}\sqrt{\pi}\mathrm{erf}(x) \approx 0.886227$$

In [ ]:
```
def func(x):
    return np.exp(-x**2)

exact_value = 0.886227
limit_inf = 0
limit_sup = 500
```

In [ ]:
```
samples = [10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480, 40960, 81
df = pd.DataFrame(data=[exact_value]*len(samples), index=samples, columns=['exact_v
df['mc_value'] = df.apply(lambda x: monte_carlo_integration(func, x.name, limit_inf
df['error'] = df.exact_value - df.mc_value
df
```

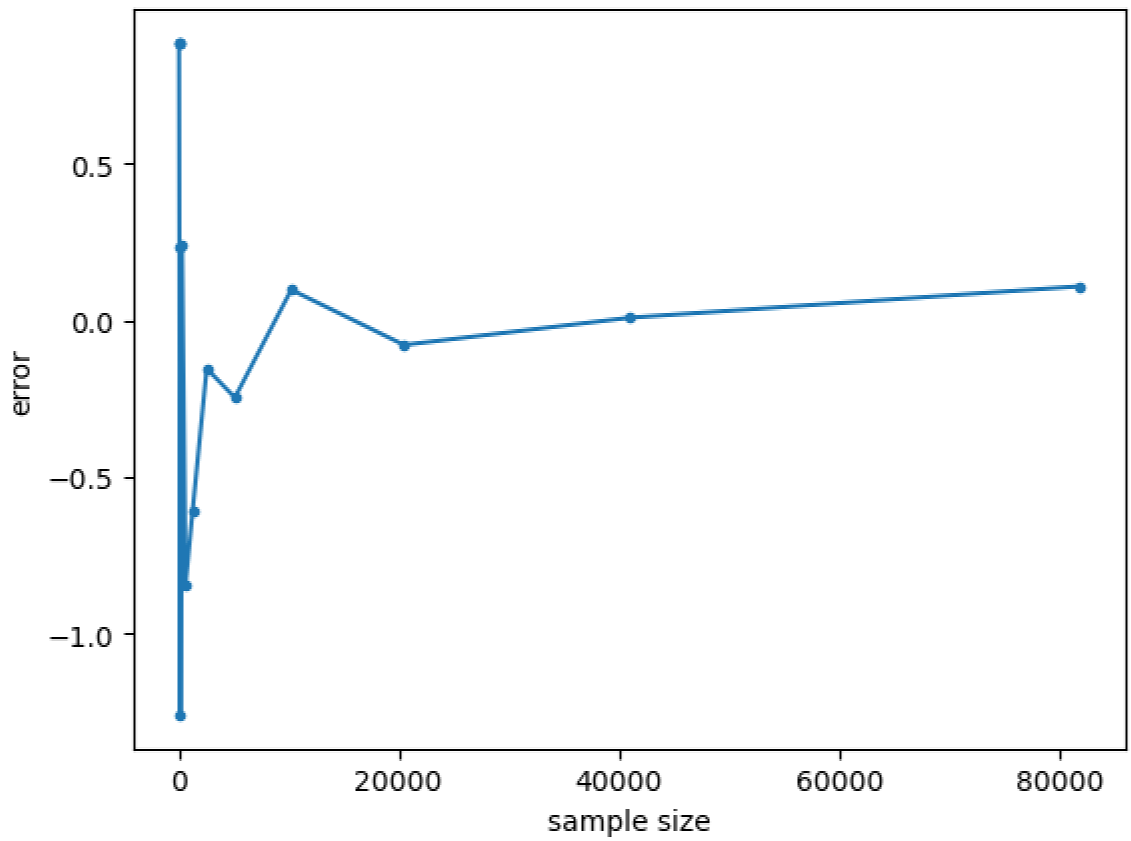
Out[ ]:

	exact_value	mc_value	error
10	0.886227	5.223262e-49	0.886227
20	0.886227	1.721681e-187	0.886227
40	0.886227	7.140643e-05	0.886156
80	0.886227	6.516088e-01	0.234618
160	0.886227	2.145406e+00	-1.259179
320	0.886227	6.482176e-01	0.238009
640	0.886227	1.728493e+00	-0.842266
1280	0.886227	1.493576e+00	-0.607349
2560	0.886227	1.039893e+00	-0.153666
5120	0.886227	1.132399e+00	-0.246172
10240	0.886227	7.882769e-01	0.097950
20480	0.886227	9.644685e-01	-0.078242
40960	0.886227	8.775492e-01	0.008678
81920	0.886227	7.770624e-01	0.109165

In [ ]:

```
ax = df.error.plot(marker='.')
ax.set_xlabel('sample size')
ax.set_ylabel('error')
```

Out[ ]: `Text(0, 0.5, 'error')`



III.

$$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} x^4 e^{-x^2/2} dx = 0.398942\left(3\sqrt{\frac{\pi}{2}}\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right) - e^{-x^2/2}x(x^2+3)\right) = 3$$

In [ ]:
```
def func(x):
    return ( 1.0 / np.sqrt(2*np.pi) ) * x**4 * np.exp(-x**2 / 2)

exact_value = 3.
limit_inf = -500
limit_sup = 500
```

In [ ]:
```
samples = [10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480, 40960, 81
df = pd.DataFrame(data=[exact_value]*len(samples), index=samples, columns=['exact_v
df['mc_value'] = df.apply(lambda x: monte_carlo_integration(func, x.name, limit_inf
df['error'] = df.exact_value - df.mc_value
df
```

Out[ ]:

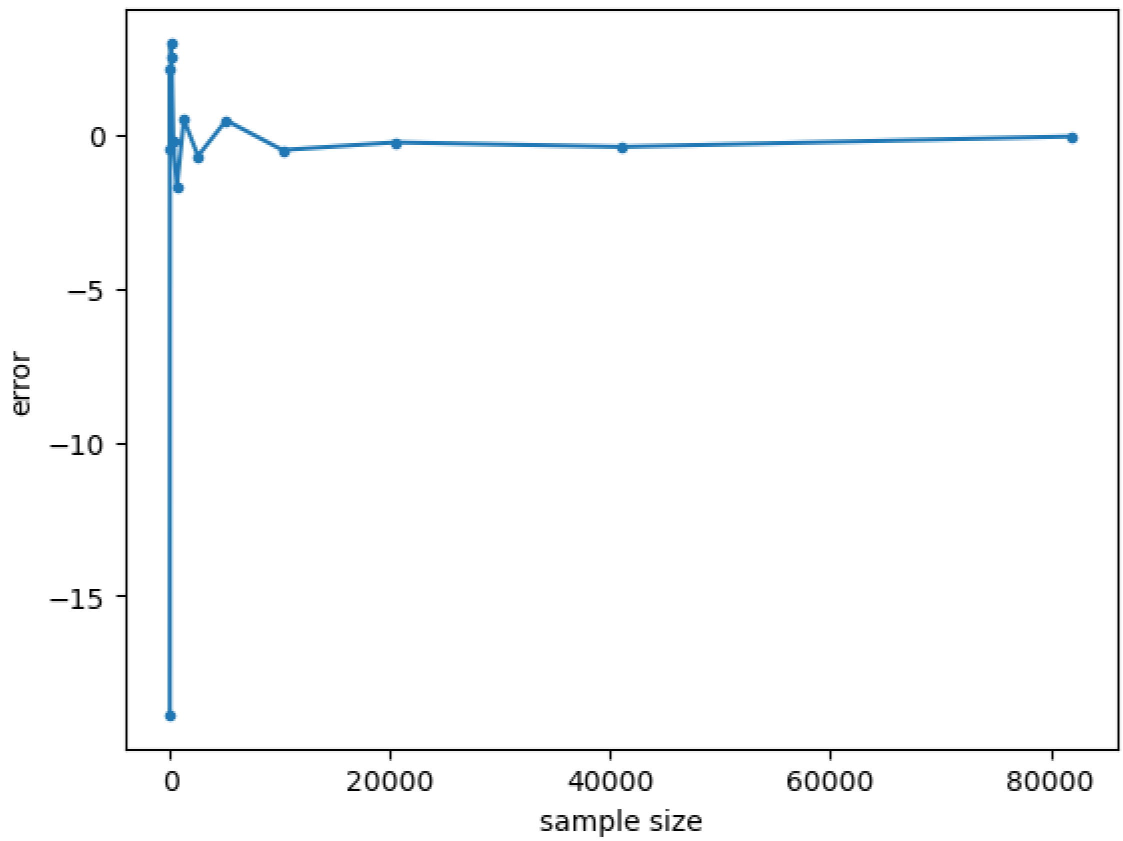
	exact_value	mc_value	error
10	3.0	21.869504	-18.869504
20	3.0	0.821046	2.178954
40	3.0	3.487399	-0.487399
80	3.0	0.000596	2.999404
160	3.0	0.442911	2.557089
320	3.0	3.218282	-0.218282
640	3.0	4.710162	-1.710162
1280	3.0	2.493630	0.506370
2560	3.0	3.684146	-0.684146
5120	3.0	2.523409	0.476591
10240	3.0	3.490200	-0.490200
20480	3.0	3.237281	-0.237281
40960	3.0	3.383781	-0.383781
81920	3.0	3.045230	-0.045230

In [ ]:

```
ax = df.error.plot(marker='.')
ax.set_xlabel('sample size')
ax.set_ylabel('error')
```

Out[ ]:

```
Text(0, 0.5, 'error')
```



Hence, the plots above shows that as we increase the samples size, the error to the exact solution gets closer to 0.

References

- Wilmott, Paul. 2006. Paul Wilmott on Quantitative Finance. 2nd ed. Hoboken, NJ: John Wiley & Sons.
- https://kyleniemeyer.github.io/ME373-book/content/bvps/finite-difference.html