Project 4

Group members: Yilin Liu, Pufeng Sun, Meng Zhou

1.

Let W_t be a standard Wiener process, that is the drift parameter is zero and the Variance parameter $\sigma^2 = 1$. Suppose that we divide the interval $[0,2]$ $into\ L\ subintervals\ [t_i, t_{i+1}], with\ t_i = i\delta t\ and\ \delta t = 2/L$. Let $W_i = W(t_i)\ \ and\ \ \delta W_i = W_{i+1} - W_i$. Verify numerically that

a) $\sum_{i=0}^{L-1} |\delta W_i|$ is unbounded as δt goes to zero

b) $\sum_{i=0}^{L-1} \delta {W_i}^2$ converges to 2 in probability as δt goes to zero

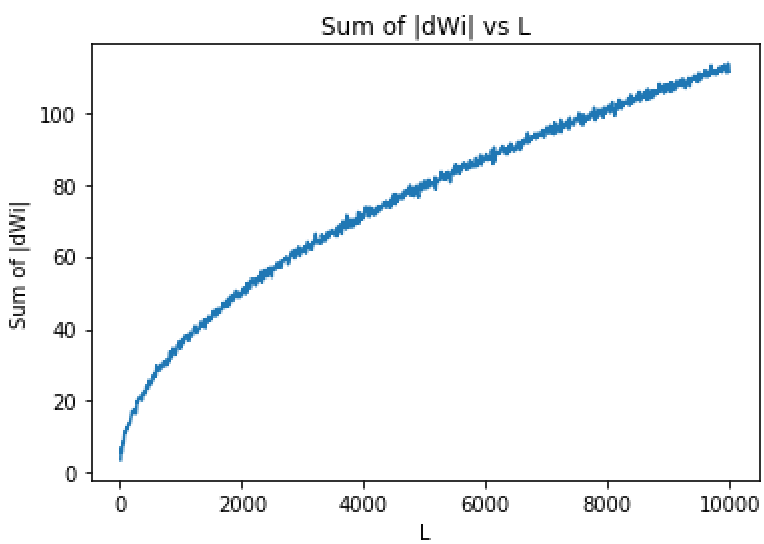
Since Wt is a standard Wiener process, we can simulate dWt using standard normal distribution with size L in Python.

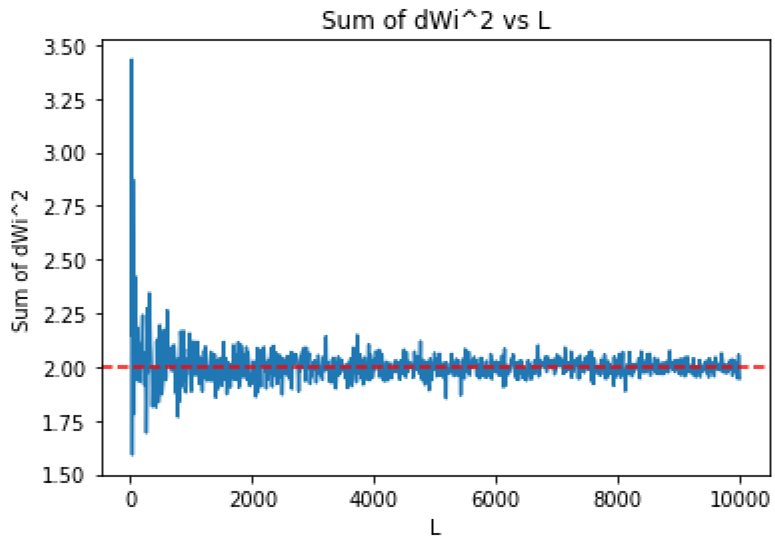
```
dWi = np.sqrt(dt) * np.random.normal(0, 1, L)
```

And then we can simulate sum of |dWi| and sum of (dWi)^2 respectively.

```
abs_dWi_sum = np.sum(np.abs(dWi))
dWi_squared_sum = np.sum(dWi ** 2)
```

To verify a) and b) numerically, we simulate them with different L and plot them with L varied from 10 to 10000. There is also a table below showing the result.





L	dt	∑|dW_t|	∑ d（W_t ^2）
20	0.1	5.509385036	2.108317365
100	0.02	11.77176284	2.112601318
200	0.01	15.72708098	1.870845178
1000	0.002	35.79839572	1.962149222
2000	0.001	113.2322233	2.019519564
10000	0.0002	159.1568192	2.001725873
100000	0.00002	358.0450032	2.007631438

As we can see from both plots and table, the sum of |Wi| keeps increasing when dt gets closer to 0. Also, $\sum d(W_t{}^2)$ converges around 2 with dt gets to 0.
Here is the complete code for Q1

```
import numpy as np
import matplotlib.pyplot as plt
def simulate_wiener_process(dt, L):
    dWi = np.sqrt(dt) * np.random.normal(0, 1, L)
    abs_dWi_sum = np.sum(np.abs(dWi))
    dWi_squared_sum = np.sum(dWi ** 2)
    return abs_dWi_sum, dWi_squared_sum
def run_simulations(L_values):
    abs_dWi_sums = []
    dWi_squared_sums = []

    for L in L_values:
        dt = 2 / L
        abs_dWi_sum, dWi_squared_sum = simulate_wiener_process(dt, L)
        abs_dWi_sums.append(abs_dWi_sum)
        dWi_squared_sums.append(dWi_squared_sum)

    return abs_dWi_sums, dWi_squared_sums
L_values = np.arange(10, 10001, 10)
abs_dWi_sums, dWi_squared_sums = run_simulations(L_values)

plt.figure()
plt.plot(L_values, abs_dWi_sums)
plt.xlabel("L")
plt.ylabel("Sum of |dWi|")
plt.title("Sum of |dWi| vs L")
plt.show()

plt.figure()
plt.plot(L_values, dWi_squared_sums)
plt.axhline(y=2, color='r', linestyle='--')
plt.xlabel("L")
plt.ylabel("Sum of dWi^2")
plt.title("Sum of dWi^2 vs L")

plt.show()
```

2.Evaluate numerically the stochastic integrals

In this problem, we use Euler-Maruyama to simulate each process.

a) Itô $\int_0^2 W(t)dW(t)$ ↵

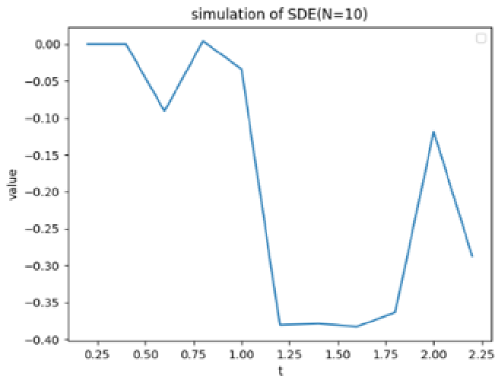
First, we divided [0,2] into N small distance equally.Then, we let W[i+1] = W[i] + *N(0,1),

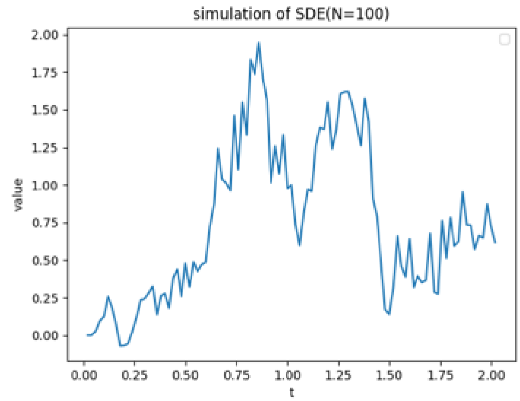
So the integral result is

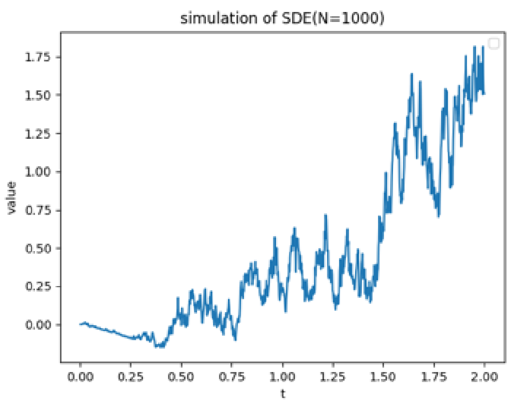
$$\sum_{i=1}^{N} w(i-1) * (w(i) - w(i-1)).$$

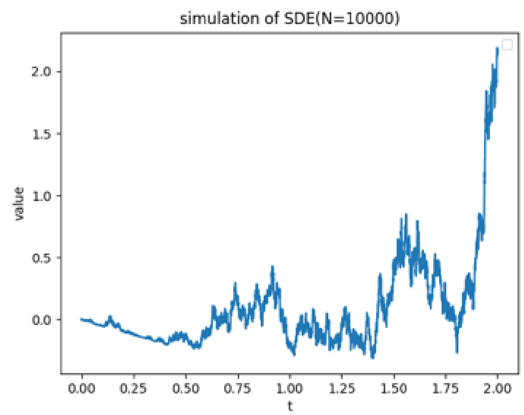
This is the result of numerical simulation calculations. Also by ito formula, we obtain the analytical solution containing W (t), 0.5*w(t)^2-0.5*2. Then we get the error of the simulation. After that, we change N and simulated more times to get the following results.

N	simulation value	Analytical value	error
10	-0.81256	-0.58275	0.256045
100	-0.84758	-0.95886	0.111279
1000	-0.92139	-0.94253	0.021146
10000	-0.85187	-0.87564	0.023772









From the results, we clearly see that the keeps getting smaller as N gets larger, and the error between the numerical solution obtained from our simulation and the true value is also decreasing.

b) Stratonovich $\int_0^2 W(t) \circ dW(t)$

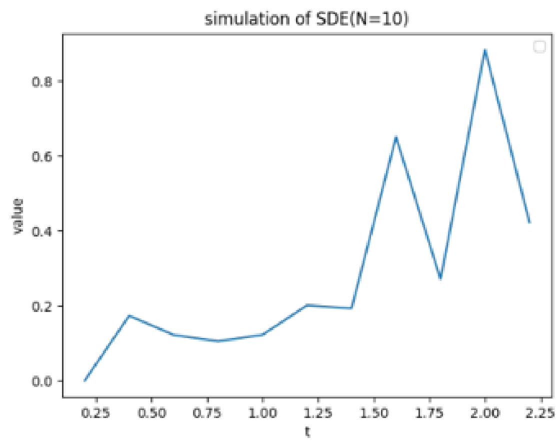
First, Similarly divided [0,2] into N small distance equally. let W[i+1] = W[i] + *N(0,1).

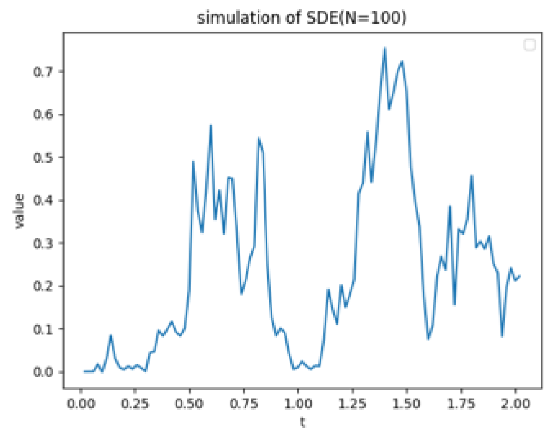
For the Stratonovich integral, integral result is

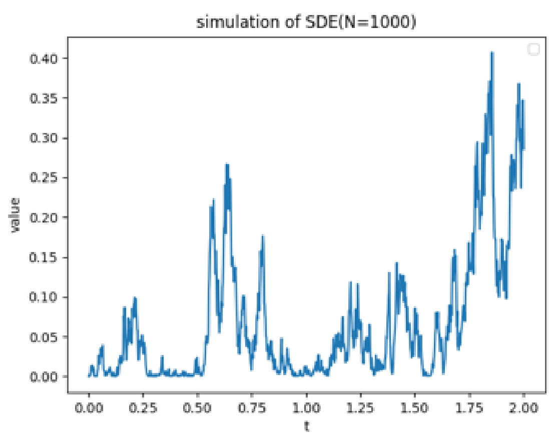
$$\sum_{i=1}^{N}(0.5 * (w[i-1] + w[i]) + \sqrt{\Delta T/4} * N(0,1)) \; * (w(i) - w(i-1)).$$

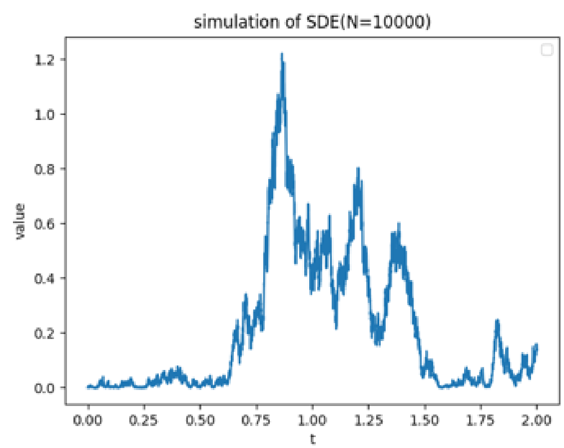
Also by Ito formula, we obtain the analytical solution 0.5*w(t)^2. Then we get the error of the simulation.

After that, we change N and simulated more times to get the following results.









N	simulation value	Analytical value	error
10	0.180045	0.102133	0.077912

100	0.378844	0.393502	-0.01466
1000	0.003387	0.005278	-0.00189
10000	0.033783	0.033797	-1.35E-05

From the results, we clearly see that the keeps getting smaller as N gets larger, the error between the numerical solution obtained from our simulation and the true value is also decreasing.

c) $E\ [\int_0^2 W(t)dW(t)\ ]$

Following problem a, after obtaining the integral value, we will repeat the simulation n times(In this question, we use n=1000) and take the average value. In this way we obtain the expectation of the stochastic calculus.

And we know that w(t)~N(0,t), so the real value of $E\ [\int W(t)dW(t)\ ]$ equals to 0.

Compare the simulation with the real expectation, we get the following answer.

N	expectation	absolute error
10	0.058631	0.058631
100	-0.04589	0.045886
1000	-0.0118	0.011798

10000	-0.0004	0.0004

From the results, we clearly see that the keeps getting smaller as N gets larger, the error between the numerical solution obtained from our simulation and the real value is also decreasing.

d) $E\ (\int_0^2 W(t)dW(t))^2)$

Also, following problem a, after obtaining the integral value, we will square it repeat the simulation n times(In this question, we use n=1000) and take the average value. In this way we obtain the expectation of the stochastic calculus.

By Ito, we know that $(\int_0^2 W(t)dW(t)\ )^2 = \int_0^2 W(t)^2 dt$, so the $E\ (\ \int_0^2 W(t)dW(t)\ )^2 = \int_0^2 E[W(t)^2]dt$.

Therefore, $\int_0^2 E[W(t)^2]dt = \int_0^2 t dt = 2$

Compare the simulation with the real expectation, we get the following answer.

N	expectation	absolute error
10	1.880237	0.119763
100	1.941309	0.058691
1000	1.985103	0.014897

10000	1.931447	0.068553

From the results, we clearly see that the keeps getting smaller as N gets larger, the error between the numerical solution obtained from our simulation and the real value is also decreasing.

e) $E\ [\int_0^2 \mathrm{W(t)^\wedge 2dt}]$

First, simulate this process. We let W[i+1] = W[i] + *N(0,1),

So the integral result is .

$$\sum_{i=1}^{N} w(i-1)^\wedge 2 * \Delta \mathrm{T}.$$

Then we repeat the simulation n times(In this question, we use n=1000) and take the average value. In this way we obtain the expectation of the stochastic calculus.

Like the question d, we know the real value of

$$E\ [\int_0^2 \mathrm{W(t)^\wedge 2dt}] = \int_0^2 \mathrm{E[W(t)^2]dt} = \int_0^2 \mathrm{tdt} = 2$$

Compare the simulation with the real expectation, we get the following answer.

N	expectation	absolute error
10	1.770458	0.229542

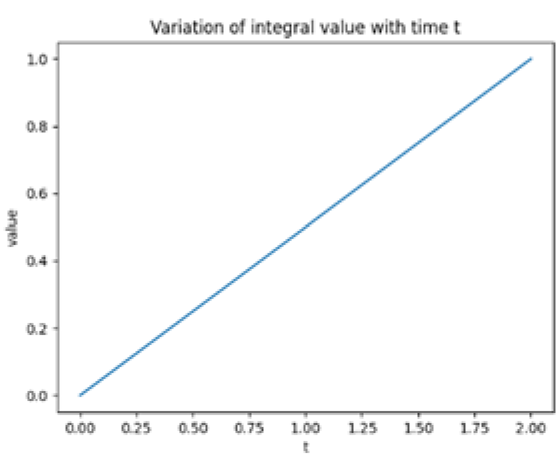
100	1.86531	0.13469
1000	1.916359	0.083641
10000	2.028355	0.028355

From the results, we clearly see that the keeps getting smaller as N gets larger, the error between the numerical solution obtained from our simulation and the real value is also decreasing.

f) For $t \in [0,2]$ evaluate $\int_0^t W(t)dW(t)$, $\int_0^t W(t) \circ dW(t)$ and 1/2*$\int_0^t dt$ what do you observe.

First we use the same simulation as in problem a and b, then adding a for loop to it, letting t take values on the interval [02]. In this way we obtain a plot of the variation of the stochastic integral g with respect to the upper limit of integration t.

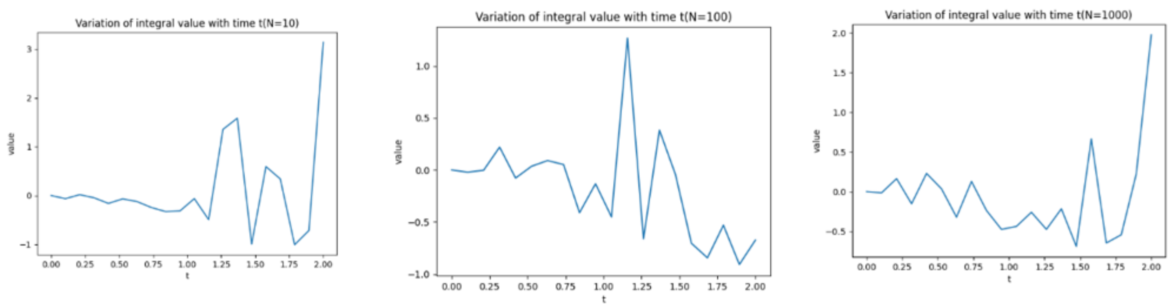
For 1/2*, because this integral is a deterministic integral, the change in its integral value shows a straight line as the upper limit of integration t increases, as shown in the following figure



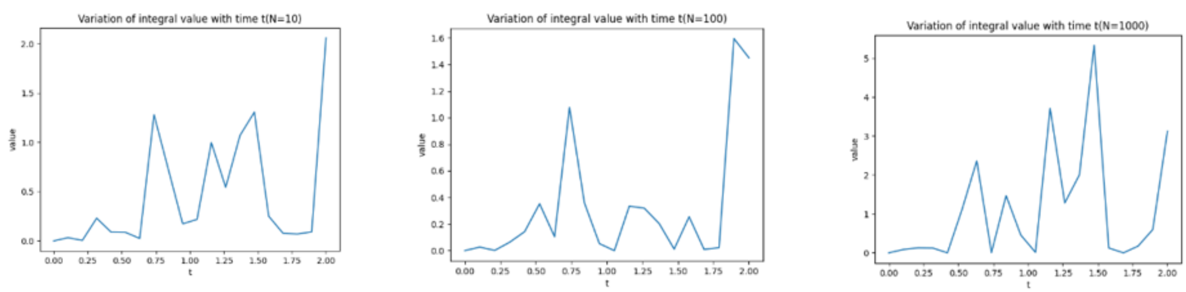
For, $\int_0^t W(t)dW(t)$, $\int_0^t W(t) \circ dW(t)$

we first divided [0,2] into 20 small distance equally. Then we ran the program and got the following results

$$\int_0^t W(t)dW(t)$$

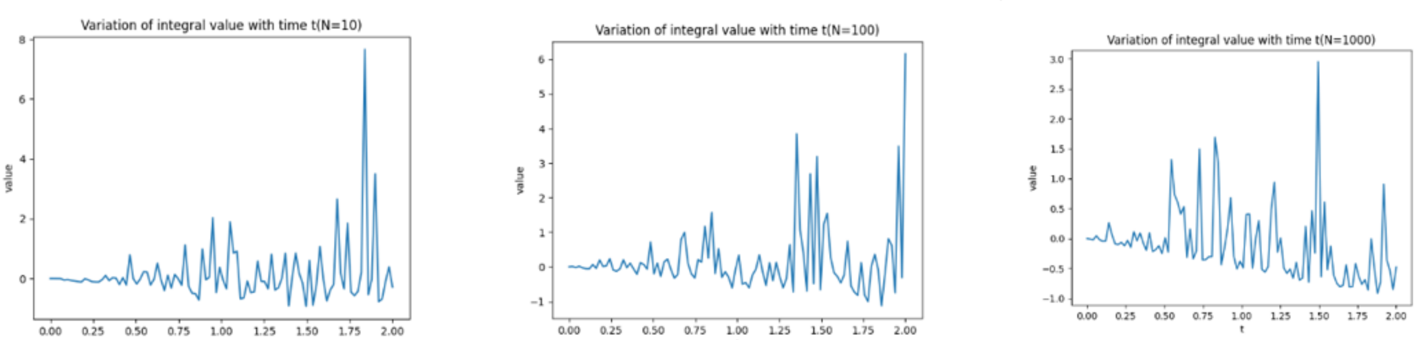


$$\int_0^t W(t) \circ dW(t)$$

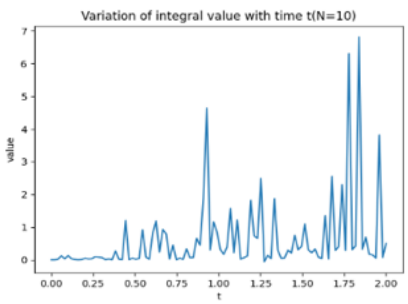


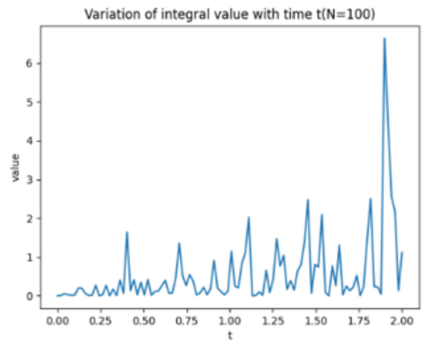
Then we first divided [0,2] into 100 small distances equally

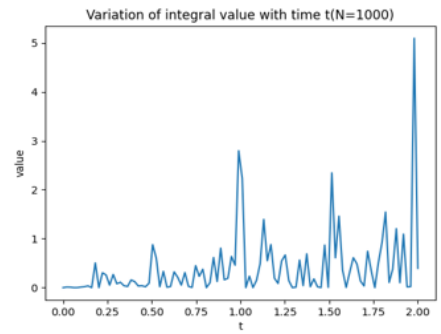
$$\int_0^t W(t)dW(t)$$



$$\int_0^t W(t) \circ dW(t)$$







From the above results, we observe that as the upper limit of integration t increases in the interval [0,2], the probability that the value of our simulated stochastic integrals becomes larger increases as well.

Here is the code for questions a through f in order

```
import matplotlib.pyplot as plt
import numpy as np
my_list = [10,100,1000,10000]
for N in my_list: # number of time steps
    delta_t = 2/N # time step
    W = np.zeros(N+1) # initialize Wiener process
    for i in range(1, N+1):
        W[i] = W[i-1] + np.sqrt(delta_t)*np.random.normal()
    sum =0
    path =np.zeros(N+1)
    for i in range(1, N+1):
        sum +=W[i-1]*(W[i]-W[i-1])
        path[i]=sum
    approximation = sum
    print('the num sol is{}'.format(approximation))
    print('the ana sol is{}'.format(0.5 * W[-1] ** 2-0.5*2))
    print('the error is{}'.format(approximation-0.5 * W[-1] ** 2+0.5*2))
    x=np.arange(1,N+2)
    plt.plot(x*2/N,path)
    plt.xlabel("t")
    plt.ylabel("value")
    plt.title("simulation of SDE(N={})".format(N))
    plt.legend()
    plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
my_list = [10,100,1000,10000]
for N in my_list: # number of time steps
    delta_t = 2/N # time step
    W = np.zeros(N+1) # initialize Wiener process
    for i in range(1, N+1):
        W[i] = W[i-1] + np.sqrt(delta_t)*np.random.normal()
    sum = 0
    path = np.zeros(N + 1)
    for i in range(1, N+1):
        sum += (0.5*(W[i-1]+W[i])+np.random.normal(0,1/4*delta_t))*(W[i]-W[i-1])
        path[i] = sum
    approximation = sum
    print('the num sol is{}'.format(approximation))
    print('the ana sol is{}'.format(0.5 * W[-1] ** 2))
    print('the error is{}'.format(approximation-0.5 * W[-1] ** 2))
    x = np.arange(1, N + 2)
    plt.plot(x*2/N, path)
    plt.xlabel("t")
    plt.ylabel("value")
    plt.title("simulation of SDE(N={})".format(N))
    plt.legend()
    plt.show()
```

```
import numpy as np
# Expected value of \int_0^2 W(t) dW(t)
n=1000
N = 10
delta_t = 2/N
W = np.zeros(N+1)
integral=0
ans=[]
for i in range(n):
    for i in range(1, N + 1):
        W[i] = W[i - 1] + np.sqrt(delta_t) * np.random.normal()
    sum = 0
    for i in range(1, N + 1):
        sum += W[i - 1] * (W[i] - W[i - 1])
    ans.append(sum)
expectation = np.mean(ans)
print(expectation)
```

```python
import numpy as np
n=1000
N = 10000
delta_t = 2/N
W = np.zeros(N+1)
integral=0
ans=[]
for i in range(n):
    for i in range(1, N + 1):
        W[i] = W[i - 1] + np.sqrt(delta_t) * np.random.normal()
    sum = 0
    sum1=0
    for i in range(1, N + 1):
        sum += W[i - 1] * (W[i] - W[i - 1])
    approximation = sum**2
    ans.append(approximation)
expectation = np.mean(ans)
print(expectation)
```

```python
import numpy as np
n=1000
N = 100
delta_t = 2/N
W = np.zeros(N+1)
integral=0
ans=[]
for i in range(n):
    for i in range(1, N + 1):
        W[i] = W[i - 1] + np.sqrt(delta_t) * np.random.normal()
    sum = 0
    for i in range(1, N + 1):
        sum += W[i - 1]**2 * delta_t
    ans.append(sum)
expectation = np.mean(ans)
print(expectation)
```

```python
import matplotlib.pyplot as plt
import numpy as np
approximation = np.zeros(100)
my_list = [10,100,1000,10000]
t= np.linspace(0,2,100)
for N in my_list: # number of time steps
    for j in range(1,101):
        delta_t = t[j-1]/N # time step
        W = np.zeros(N+1) # initialize Wiener process
        for i in range(1, N+1):
            W[i] = W[i-1] + np.sqrt(delta_t)*np.random.normal()
        sum =0
        path =np.zeros(N+1)
        for i in range(1, N+1):
            sum +=W[i-1]*(W[i]-W[i-1])
            path[i]=sum
        approximation[j-1] = sum
    print(approximation)
    x = np.arange(1, 101)
    plt.xlabel("t")
    plt.ylabel("value")
    plt.title("Variation of integral value with time t(N={})".format(N))
    plt.plot(t, approximation)
    plt.show()
```

```python
import matplotlib.pyplot as plt
import numpy as np
approximation = np.zeros(100)
my_list = [10,100,1000,10000]
t= np.linspace(0,2,100)
for N in my_list: # number of time steps
    for j in range(1,101):
        delta_t = t[j-1]/N # time step
        W = np.zeros(N+1) # initialize Wiener process
        for i in range(1, N+1):
            W[i] = W[i-1] + np.sqrt(delta_t)*np.random.normal()
        sum =0
        path =np.zeros(N+1)
        for i in range(1, N+1):
            sum += (0.5*(W[i-1]+W[i])+np.random.normal(0,1/4*delta_t))*(W[i]-W[i-1])
            path[i]=sum
        approximation[j-1] = sum
    print(approximation)
    plt.xlabel("t")
    plt.ylabel("value")
    plt.title("Variation of integral value with time t(N={})".format(N))
    x = np.arange(1, 101)
    plt.plot(t, approximation)
    plt.show()
```

3).Consider the following SDE: $dX(t)=\mu X(t)dt+\sigma X(t)dW(t)$, $X(0)=3$, $\mu=2$, $\sigma=0.10$
Where $t \in [0,1]$.

a) Show that the Euler Maruyama method has weak order of convergence equal to one. That is $|E[X_1] - E[X(1)]| = C\Delta t$. Here $X(1)$ is the exact solution at time 1 and X_1 is the computed solution at time 1.

In this problem The Euler-Maruyama method for this SDE is given by:
X_{n+1} = X_n + μ X_n Δt + σ X_n ΔW_n
where ΔW_n = W_{(n+1)Δt} - W_{nΔt} is the increment of a Wiener process over the time step Δt. Let X(t) be the exact solution to the SDE. Then we have:
X(Δt) = X(0) + μ X(0) Δt + σ X(0) ΔW_0 + O(Δt^2)
Then we plug it in to the python, and calculate the weak error.
Here is the code

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(100)
_lambda = 2
mu = 0.1
Xzero = 3
T = 1
N = 2 ** 9
dt = T / N
M = 1000

Xweak_err = np.zeros(5)

# Compute the true expected value
Xtrue_exp = Xzero * np.exp(_lambda * T)
```

Firstly, we set up the initial condition, here lambda is mu, mu is sigma.

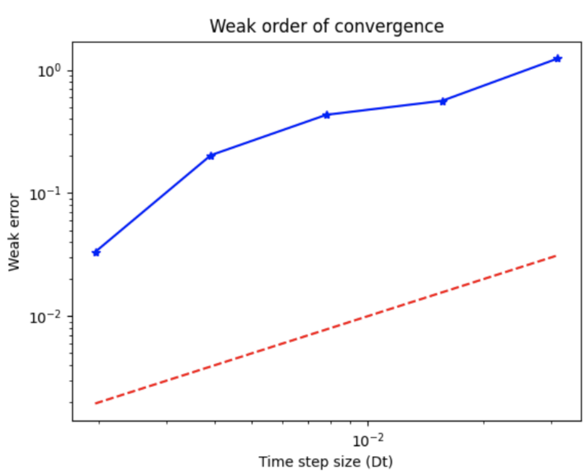
```python
# Compute the true expected value
Xtrue_exp = Xzero * np.exp(_lambda * T)

for p in range(5):
    R = 2 ** p
    Dt = R * dt
    L = N // R
    Xnumerical = np.zeros(M)

    for s in range(M):
        dw = np.sqrt(dt) * np.random.randn(1, N)
        W = np.cumsum(dw)
        Xtemp = Xzero

        for j in range(L):
            Winc = np.sum(dw[0, R * j:R * (j + 1)])
            Xtemp = Xtemp + Dt * _lambda * Xtemp + mu * Xtemp * Winc

        Xnumerical[s] = Xtemp

    # Compute the weak error as the absolute difference between the expected values
    Xweak_err[p] = abs(np.mean(Xnumerical) - Xtrue_exp)
```

Then we compute the the true exponential expected value for the x, which we can use ito sde that the dw part of the function is elimilated. Which is equal to Xzero * exp(mu * T)
Then we rearrange the dt into 5 discrate points. And use the Euler-Maruyama method to calculate the X. After that we take the Expected value. Compute the weak error as the absolute difference between the expected values

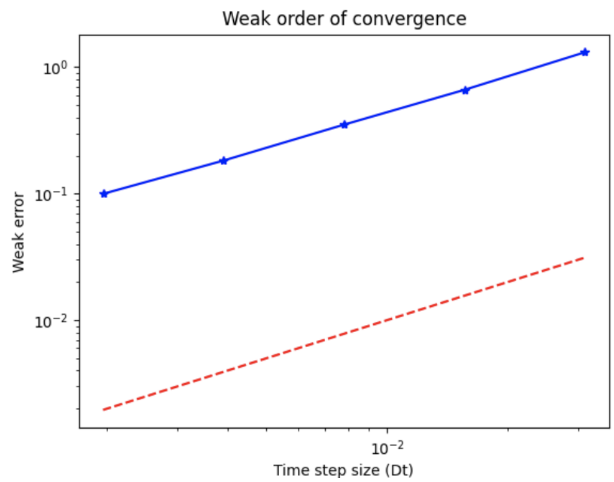
```python
Xweak_err[p] = abs(np.mean(Xnumerical) - Xtrue_exp)
```

```
Dtvals = dt * np.array([2 ** i for i in range(5)])
plt.loglog(Dtvals, Xweak_err, 'b*-')
plt.loglog(Dtvals, Dtvals, 'r--')
plt.xlabel('Time step size (Dt)')
plt.ylabel('Weak error')
plt.title('Weak order of convergence')
plt.show()

A = np.vstack((np.ones(5), np.log(Dtvals))).T
rhs = np.log(Xweak_err).reshape(-1, 1)
sol = np.linalg.lstsq(A, rhs, rcond=None)[0]
q_weak = sol[1][0]
resid = np.linalg.norm(A @ sol - rhs)
print(q_weak)
print(resid)
```

In the end, we plug it into a diagram, we compare it with the loglog function of convergence of 1, which is indicated the slope of the function.

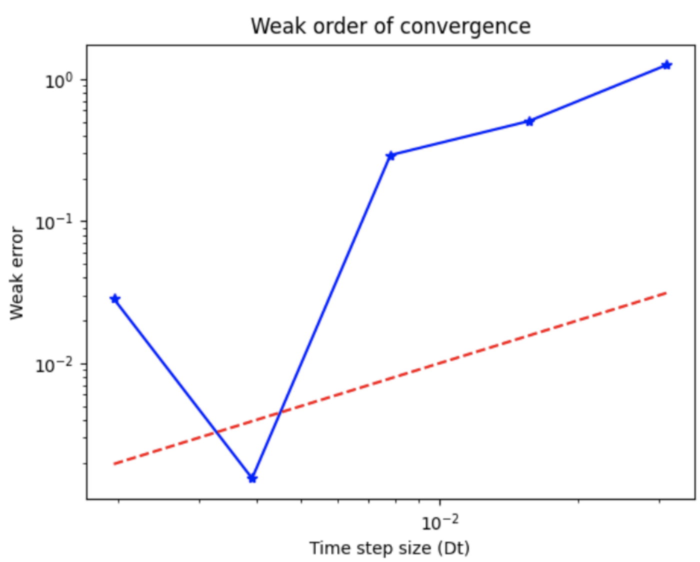


In this case, we get q_weak=1.190814851813077
resid= 0.8437258168797771 for m=1000
If m = 10000
q_weak=0.9296334317340579
resid=0.040153380941455714



If m =100

q_weak=1.9282709369144726
resid=3.3539035579572283 which is too big to use this data



And we change the dt, as N goes bigger, the subtraction goes to 0

N	Error=mean(Xnumerical) - Xtrue_exp
2^4	13.1854603
2^5	10.12549307
2^6	7.015924019
2^7	4.064014323
2^8	2.495104883
2^9	1.25538007
2^10	0.581979297
2^11	0.512599949
2^12	0.056532044

As result, we can see that as more time of simulation of the fcunton, the mean of the function is closer to the analitical mean. And the Euler Maruyama method has weak order of convergence equal to one.

b) Show that the Euler Maruyama method has strong order of convergence equal to one half. That is
$E|X_1 - X(1)| = C\Delta t^{0.5}$. Here $X(1)$ is the exact solution at time 1 and X_1 is the computed solution at time 1.

To show that the Euler-Maruyama method has strong order of convergence equal to 0.5, we need to show that the expected error between the numerical solution and the exact solution is proportional to the square root of the time step size Δt.

Using the same notation as in part a), the exact solution at t = 1 is given by:X(1) = X(0) exp[(μ - σ^2/2)t + σ W_t] where W_t is a Wiener process with mean zero and variance t. The numerical solution at t = 1 using the Euler-Maruyama method is given by: X_1 = X_0 exp[(μ - σ^2/2)Δt + σ ΔW_0] where ΔW_0 is the increment of the Wiener process over the time interval [0, Δt].
In the end, we just need to use $E|X_1 - X(1)| = C\Delta t^{0.5}$

```
np.random.seed(42)
_lambda = 2
mu = 0.1
Xzero = 3
T = 1
N = 2 ** 9
dt = T / N
M = 10000

Xerr = np.zeros((M, 5))
```

Firstly, we set up the initial condition, here lambda is mu, mu is sigma.

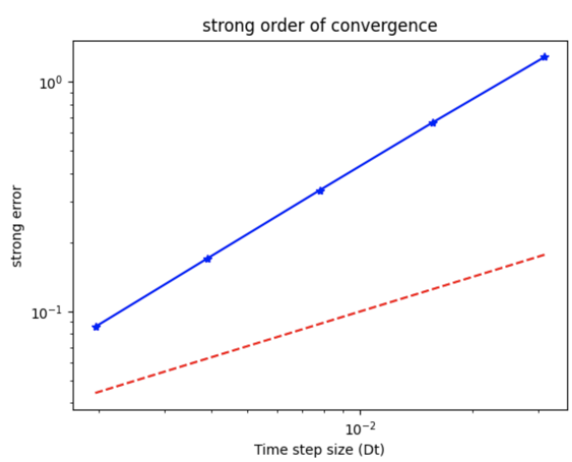
```
for s in range(M):
    dw = np.sqrt(dt) * np.random.randn(1, N)
    W = np.cumsum(dw)

    for p in range(5):
        R = 2 ** p
        Dt = R * dt
        L = N // R
        Xtemp = Xzero

        Xtrue = Xzero * np.exp((_lambda - 0.5 * mu ** 2) * (L * Dt) + mu * W[R * L - 1])

        for j in range(L):
            Winc = np.sum(dw[0, R * j:R * (j + 1)])
            Xtemp = Xtemp + Dt * _lambda * Xtemp + mu * Xtemp * Winc

        Xerr[s, p] = abs(Xtemp - Xtrue)
```

Then we compute the the true exponential expected value for the x, which we can use ito sde Which is equal to `Xzero * np.exp((mu - 0.5 * sigma ** 2) * (L * Dt) + sigma * W[R * L - 1])`
Then we rearrange the dt into 5 discrate points. And use the Euler-Maruyama method to calculate the X. then we get the difference between the two methods. After that we take the Expected value.
Compute the weak error as the absolute difference between the expected values

```python
Dtvals = dt * np.array([2 ** i for i in range(5)])
plt.loglog(Dtvals, np.mean(Xerr, axis=0), 'b*-')
plt.loglog(Dtvals, Dtvals ** 0.5, 'r--')
plt.xlabel('Time step size (Dt)')
plt.ylabel('strong error')
plt.title('strong order of convergence')
plt.show()

A = np.vstack((np.ones(5), np.log(Dtvals))).T
rhs = np.log(np.mean(Xerr, axis=0)).reshape(-1, 1)
sol = np.linalg.lstsq(A, rhs, rcond=None)[0]
q = sol[1][0]
resid = np.linalg.norm(A @ sol - rhs)

print(f"Strong order of convergence (q): {q}")
print(f"Residual: {resid}")
```

In the end, we plug it into a diagram, we compare it with the loglog function of convergence of 0.5, which is indicated the slope of the function. But here something wired happened.

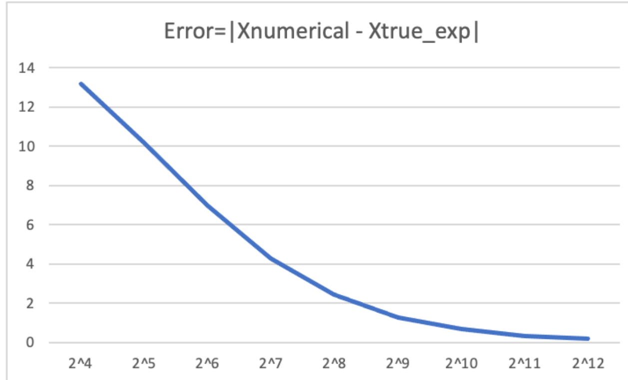
Here is the graph



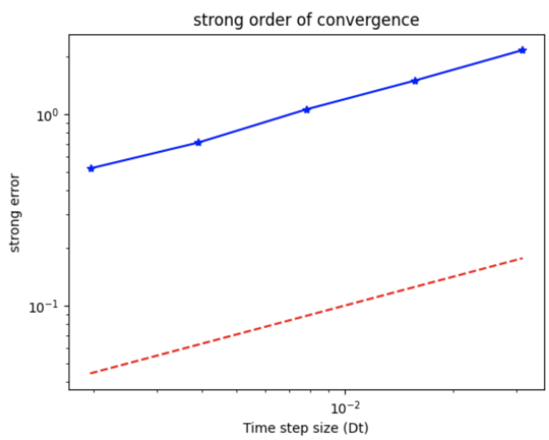
And here is the data
Strong order of convergence (q): 0.9767603160492175
Residual: 0.017732618142960038
Which indicate that the Strong order of convergence of the function is 1 rather than 0.5.

This is the examination:

N	Error=|Xnumerical - Xtrue_exp|
2^4	13.16907254
2^5	10.16056076
2^6	6.99345518
2^7	4.296305306
2^8	2.417748418
2^9	1.283572632
2^10	0.663647916
2^11	0.337498417
2^12	0.170737718

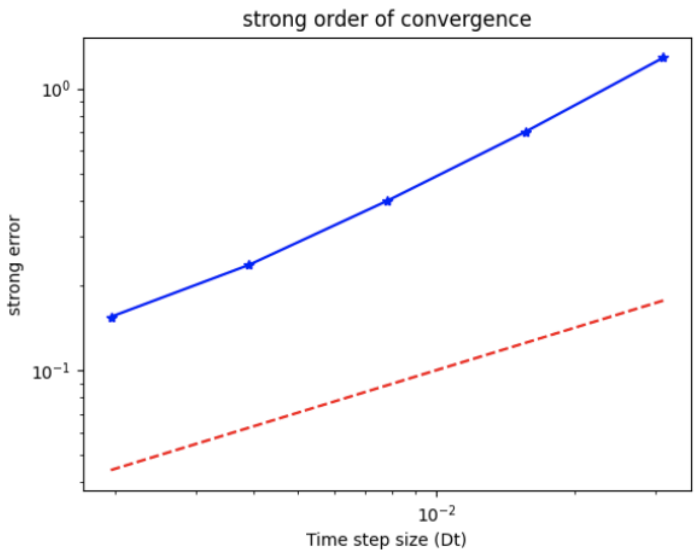


To figuring out why this is not 0.5 convergence, we do several trials with different parameter. if we change the sigma to 1 rather than 0.1:



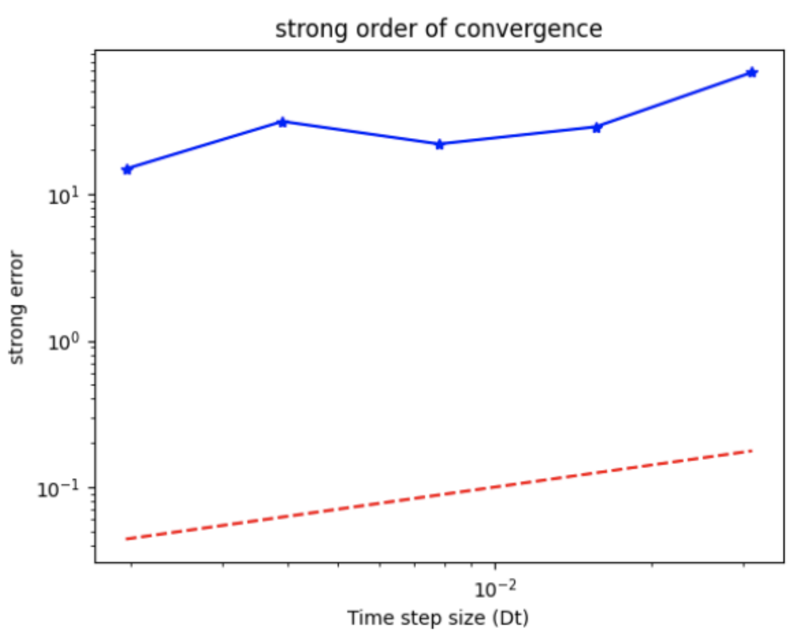
Strong order of convergence (q): 0.5182110094916422
Residual: 0.03703552713324291
Now, the Strong order of convergence of the function is 0.5.

If the sigma is 0.5



Strong order of convergence (q): 0.7683998515733361
Residual: 0.10992691302697705

If we change sigma=4



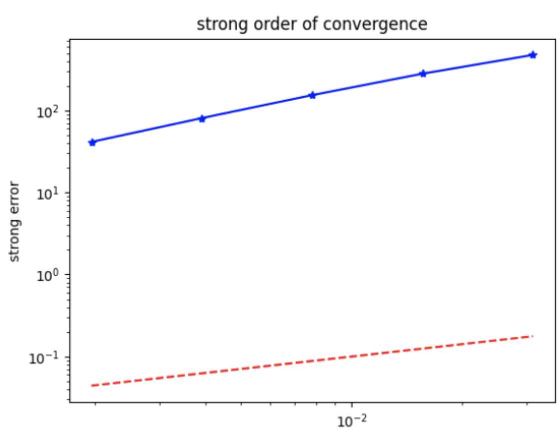
Strong order of convergence (q): 0.4253321766311653
Residual: 0.6168491143113144

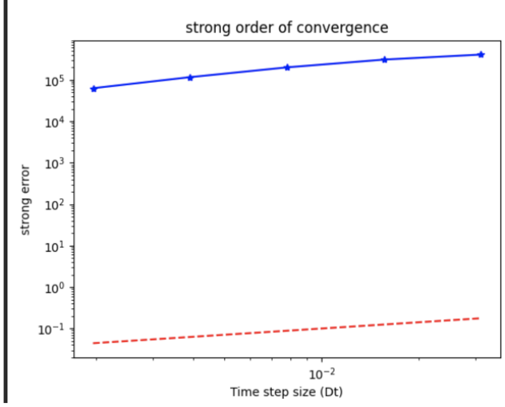
sigma	Strong order of convergence	Residual
0.1	0.977	0.018
0.5	0.768	0.110
1.0	0.518	0.037
4.0	0.425	0.616

From above observation, as sigma increases, Strong order of convergence decrease.

If we fix sigma, increase the mu,
mu=6
Strong order of convergence (q): 0.8843782873321784
Residual: 0.08871003340662138

mu=12
Strong order of convergence (q): 0.6793623977412382
Residual: 0.21031375499224111





We may conclude that as mu increases, Strong order of convergence decreases.

4.

Consider the following SDE:

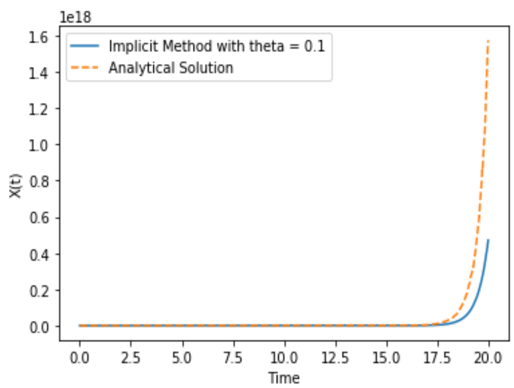
$$dX(t) = \mu X(t)dt + \sigma X(t)dW(t)\,, \;\; X(0) = 3, \qquad \mu = 2\,, \qquad \sigma = 0.10$$

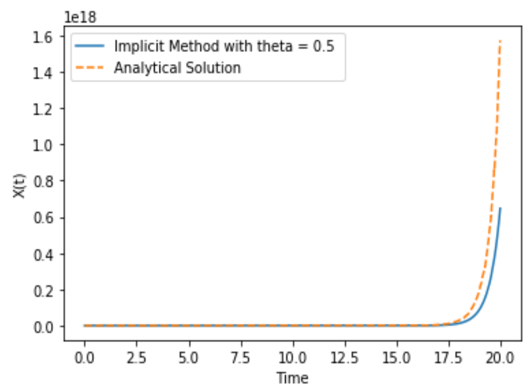
a) Simulate (over the interval [0,20]) this stochastic process using an implicit method of the form

$$X_{n+1} = X_n + (1-\theta)\Delta t f(X_n) + \theta \Delta t f(X_{n+1}) + \sqrt{\Delta t}\alpha_n g(X_n)$$

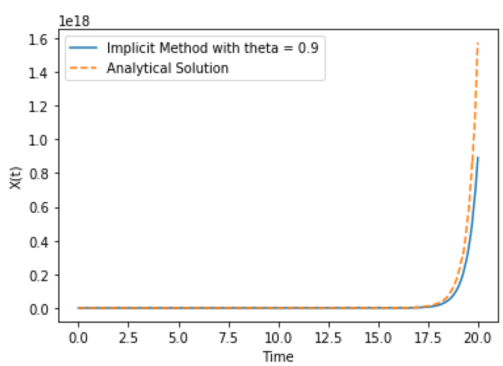
And b) We choose different values for theta (=0.1, 0.5, 0.9) to simulate using the implicit method. Here are three graphs compared with the analytical solution.

OBJ





OBJ



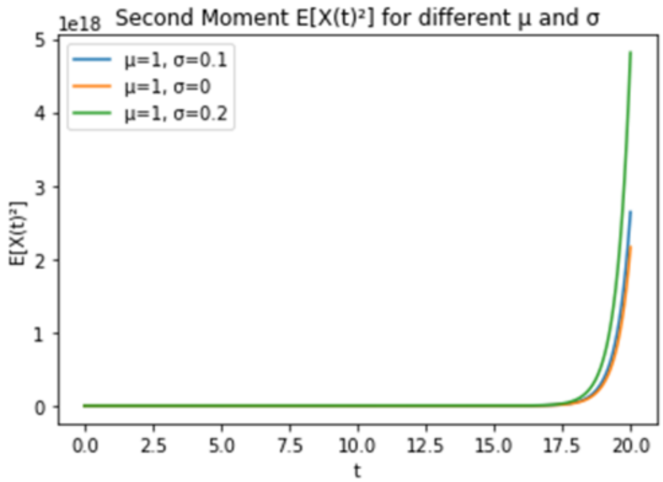
OBJ OBJ

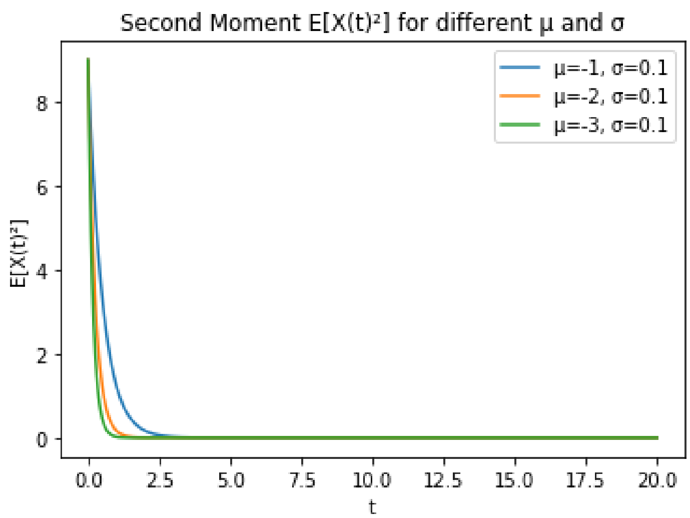
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

X0 = 3
mu = 2
sigma = 0.10
T = 20
dt = 0.01
theta = 0.1
n_steps = int(T / dt)

np.random.seed(42)
dW = np.random.normal(0, np.sqrt(dt), n_steps)
W = np.concatenate(([0], np.cumsum(dW)))

def implicit_equation(X_n_plus_1, X_n, dt, dW_n, theta, mu, sigma):
    return X_n_plus_1 - X_n - (1 - theta) * dt * mu * X_n - theta * dt * mu * X_n_plus_1 + (dt)**0.5 * dW_n * sig

X = np.zeros(n_steps + 1)
X[0] = X0
for i in range(n_steps):
    X[i + 1] = fsolve(implicit_equation, X[i], args=(X[i], dt, dW[i], theta, mu, sigma))

# Analytical solution
X_analytical = X0 * np.exp((mu - 0.5 * sigma**2) * np.arange(0, T + dt, dt)[:n_steps + 1] + sigma * W)

# Plot the results
time = np.arange(0, T + dt, dt)[:n_steps + 1]
plt.plot(time, X, label='Implicit Method with theta = 0.1')
plt.plot(time, X_analytical, label='Analytical Solution', linestyle='--')
plt.xlabel('Time')
plt.ylabel('X(t)')
plt.legend()
plt.show()
```

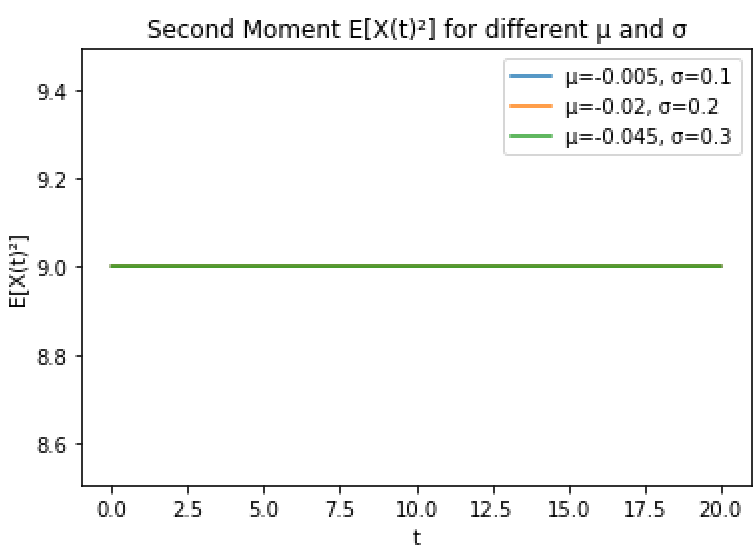
c).We use the following code to calculate the second moment and simulate with different mu and delta.H

```python
import numpy as np
import matplotlib.pyplot as plt

X0 = 3
T = 20
dt = 0.01
t = np.arange(0, T + dt, dt)

# Define a function to compute the second moment E[X(t)²]
def second_moment(t, mu, sigma, X0):
    mean = X0 * np.exp(mu * t)
    variance = X0**2 * np.exp(2 * mu * t) * (np.exp(sigma**2 * t) - 1)
    second_moment = mean**2 + variance
    return second_moment
```

Here are three different cases, and we can find that mu should always be negative to make sure the SDE is mean-square stable.


Second Moment E[X(t)²] for different μ and σ
1e18
μ=1, σ=0.1
μ=1, σ=0
μ=1, σ=0.2
E[X(t)²]
t


Second Moment E[X(t)²] for different μ and σ
μ=-1, σ=0.1
μ=-2, σ=0.1
μ=-3, σ=0.1
E[X(t)²]
t

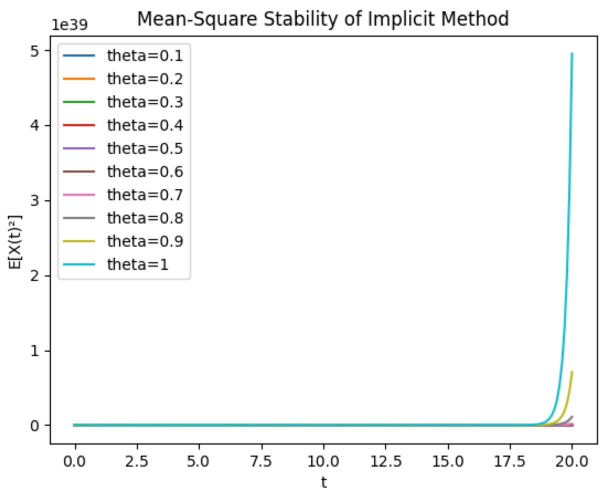

Second Moment E[X(t)²] for different μ and σ
μ=-0.005, σ=0.1
μ=-0.02, σ=0.2
μ=-0.045, σ=0.3
E[X(t)²]
t

d).

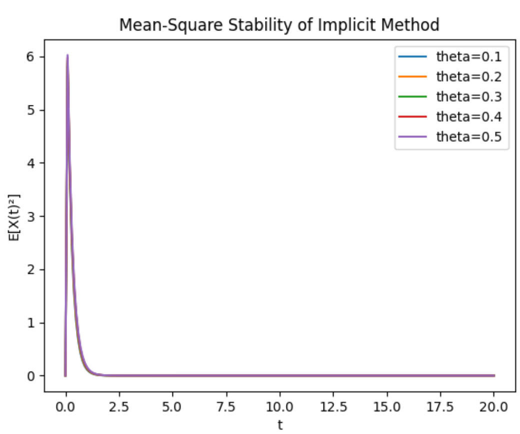
$$X_{n+1} = X_n + (1-\theta)\Delta t f(X_n) + \theta \Delta t f(X_{n+1}) + \sqrt{\Delta t}\alpha_n g(X_n)$$

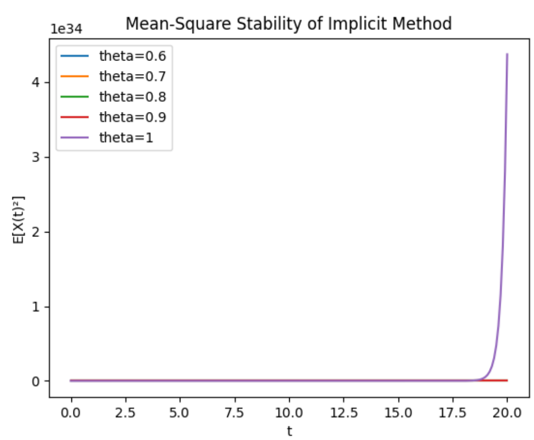
Using this implicit method to simulate and then calculate the E[X(t)²].
First, we let u>0(actually we let u+½*sigma^2>0) and change the theta.

u=2, sigma = 0.01

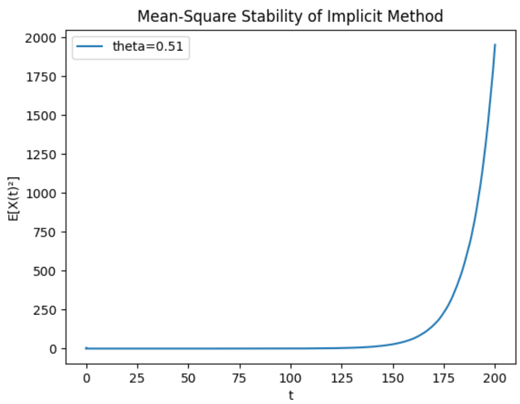


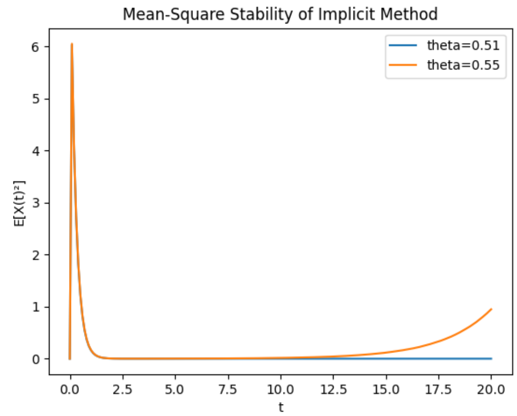
We find that it is not mean-square stable.
Then we let u<0(actually we let u+½*sigma^2<0) and change the theta.





We find that it is mean-square stable when theta is in [0 0.5]. When theta is bigger than 0.5, It is still not mean-square stable.





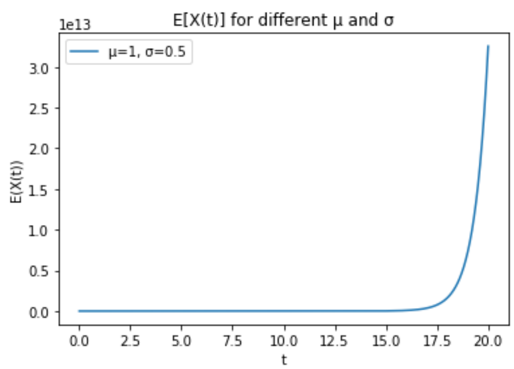
Code:

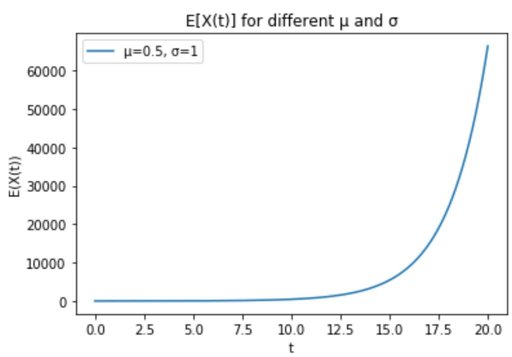
```
import numpy as np
import matplotlib.pyplot as plt
# Define the SDE parameters and initial condition
mu = -2
sigma = 0.01
X0 = 3# Define the time interval and time step size
T = 20
dt = 0.1
t = np.arange(0, T + dt, dt)# Define the number of time steps
N = int(T / dt)# Define the different values of theta to simulate
thetas = [0.1,0.2,0.3,0.4,0.5]
# Initialize the array to store the second moment for each value of theta
second_moments = np.zeros(len(thetas))
S = np.zeros(shape=(1000,N+1))
M=np.zeros(N+1)
V=np.zeros(N+1)
ans=np.zeros(N+1)
# Loop over each value of theta
for i, theta in enumerate(thetas):
    # Initialize the array to store the solution of the SDE
    X = np.zeros(N+1)
    X[0] = X0
    for j in range(1000):
        # Initialize the array to store the Gaussian random variables
        Z = np.random.normal(size=N)
        # Loop over each time step
            for n in range(N):
                # Calculate the next value of X using the implicit method
                alpha_n = mu * X[n]
                X[n+1] = (X[n] + (1 - theta) * dt * alpha_n
                          + theta * dt * mu * X[n+1]
                          + np.sqrt(dt) * sigma * X[n] * Z[n])
                S[j,n+1] = X[n+1]
        for k in range(N+1):
            M[k]=np.mean(S[:,k])
            V[k]=np.var(S[:,k])
            ans[k]=M[k]*M[k]+V[k]
        # Calculate the second moment of X and store it in the array
        #     second_moments[i] = np.mean(X**2)
    plt.plot(t,ans,label=f"theta={theta}")
# Plot the second moment as a function of theta
plt.xlabel('t')
plt.ylabel('E[X(t)²]')
plt.legend()
plt.title('Mean-Square Stability of Implicit Method')
plt.show()
```

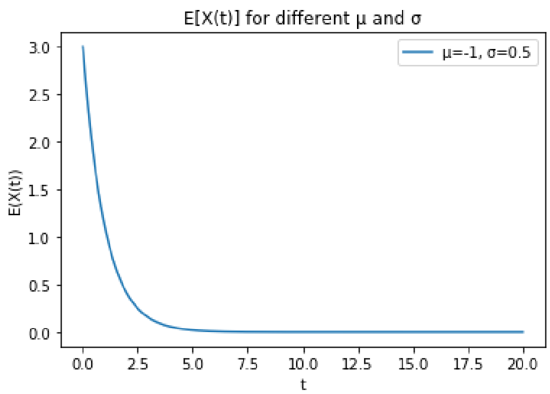
e)To determine the values of μ and σ for which the SDE is asymptotically stable numerically, we can use Monte Carlo simulations. The basic idea is to generate many sample paths of the SDE and observe their behavior as time goes to infinity.

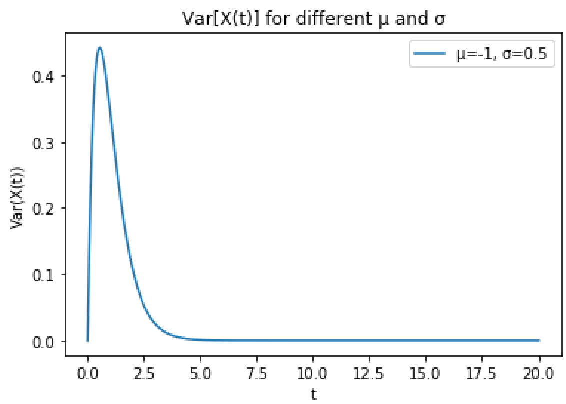
```python
import numpy as np

# Define SDE parameters
mu = -2
sigma = 0.1

# Define simulation parameters
dt = 0.01
T = 10.0
N = int(T/dt)
M = 1000

# Initialize arrays to store final values
X = np.zeros(M)

# Generate sample paths
for i in range(M):
    x = 3.0
    for j in range(N):
        x += mu*x*dt + sigma*x*np.sqrt(dt)*np.random.normal()
    X[i] = x

# Compute sample mean and sample variance
mean_X = np.mean(X)
var_X = np.var(X)


print("Sample mean of X:", mean_X)
print("Sample variance of X:", var_X)
```

Then we put different values of μ and σ to verify if μ − 0.5 δ2 < 0 is the condition.









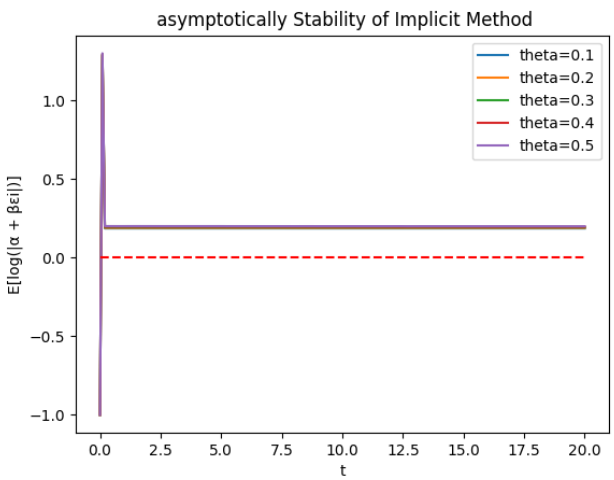
When $\mu = -1$, $\delta = 0.5$,E[X(t)] and Var[X(t)] are both going to 0 when t goes to zero, which means SDE is asymptotically stable.

```
import numpy as np
import matplotlib.pyplot as plt

X0 = 3
T = 20
dt = 0.01
t = np.arange(0, T + dt, dt)

# Define a function to compute the second moment E[X(t)²]
def mean(t, mu, sigma, X0):
    mean = X0 * np.exp(mu * t)
    return mean
def var(t, mu, sigma, X0):
    variance = X0**2 * np.exp(2 * mu * t) * (np.exp(sigma**2 * t) - 1)
    return variance
# Different combinations of mu and sigma

plt.plot(t, mean(t, 1.5, 0.5, X0), label=f"μ={1}, σ={0.5}")


plt.xlabel("t")
plt.ylabel("E(X(t))")
plt.legend()
plt.title("E[X(t)] for different μ and σ")
plt.show()

plt.plot(t, mean(t, -1, 0.5, X0), label=f"μ={-1}, σ={0.5}")


plt.xlabel("t")
plt.ylabel("E(X(t))")
plt.legend()
plt.title("E[X(t)] for different μ and σ")
plt.show()
```

f). For what values of θ is the Implicit method asymptotically stable.

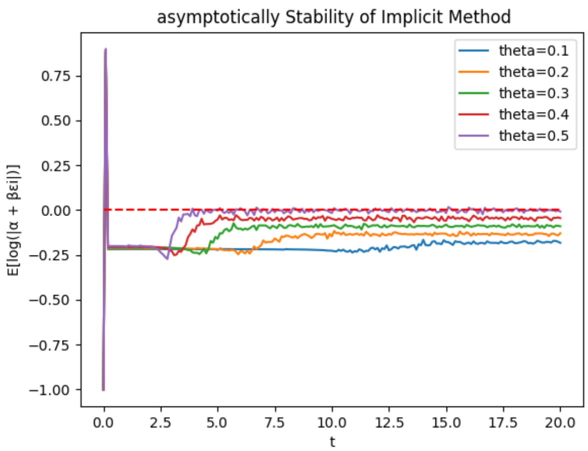
Using implicit method to simulate and then calculate theE[ln(|α + βεi|)]
First, we let u>0(actually we let u-½*sigma^2>0) and change the theta.

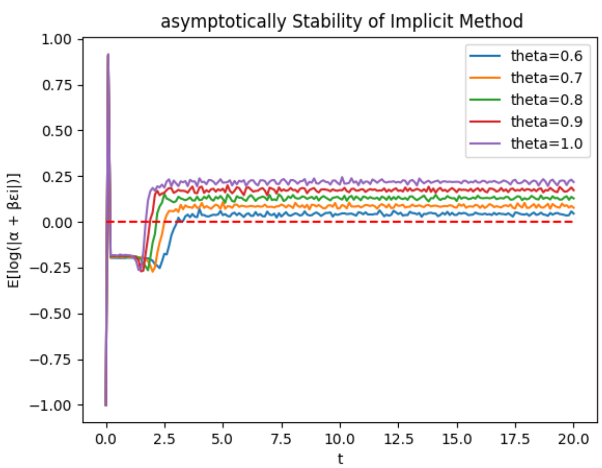
u=2, sigma = 0.01



We find that it is not asymptotically stable.
Then we let u<0(actually we let u+½*sigma^2<0) and change the theta.

u=-2, sigma = 0.01





We find that it is asymptotically stable when theta is in [0 0.5]. When theta is bigger than 0.5, It is still not asymptotically stable.

Code：

```python
import numpy as np
import matplotlib.pyplot as plt
# Define the SDE parameters and initial condition
mu = -2
sigma = 0.01
X0 = 3# Define the time interval and time step size
T = 20
dt = 0.1
t = np.arange(0, T + dt, dt)# Define the number of time steps
N = int(T / dt)# Define the different values of theta to simulate
thetas = [0.6,0.7,0.8,0.9,1.0]
# Initialize the array to store the second moment for each value of theta
second_moments = np.zeros(len(thetas))
S = np.zeros(shape=(1000,N+1))
x_n=np.zeros(shape=(1000,N+1))
x_n_1=np.zeros(shape=(1000,N+1))
x_n_1[:,0]=1
ans=np.zeros(N+1)
spf=np.zeros(N+1)
# Loop over each value of theta
for i, theta in enumerate(thetas):
    # Initialize the array to store the solution of the SDE
    X = np.zeros(N+1)
    X[0] = X0
```


```python
import numpy as np
import matplotlib.pyplot as plt
# Define the SDE parameters and initial condition
mu = -2
sigma = 0.01
X0 = 3# Define the time interval and time step size
T = 20
dt = 0.1
t = np.arange(0, T + dt, dt)# Define the number of time steps
N = int(T / dt)# Define the different values of theta to simulate
thetas = [0.6,0.7,0.8,0.9,1.0]
# Initialize the array to store the second moment for each value of theta
second_moments = np.zeros(len(thetas))
S = np.zeros(shape=(1000,N+1))
x_n=np.zeros(shape=(1000,N+1))
x_n_1=np.zeros(shape=(1000,N+1))
x_n_1[:,0]=1
ans=np.zeros(N+1)
spf=np.zeros(N+1)
# Loop over each value of theta
for i, theta in enumerate(thetas):
    # Initialize the array to store the solution of the SDE
    X = np.zeros(N+1)
    X[0] = X0
```


5).

Consider the following SDE: $dX(t)=\mu X(t)dt+\sigma X(t)dW(t)$, $X(0)=2$,
Let $a=0.5$ *and* $b=3$.
Compute the mean exit time function $v(x)$ *for* $x \in [0.5, 3]$

For this question, firstly, we need to consider The script first calculates the mean exit time function v(x) using the finite difference method, then uses Monte Carlo simulation to estimate the mean exit time for an initial condition x0 = 2.
Here we need to figure out the exit time X += mu * X * dt + sigma * X * dW(dt)for n loop. We are consider the riemann sum method to add the movement of x in each dt. Do several trials and get the expected value of them.
And compare them with the analytical solution as below.
ea=(1 / (0.5 * sigma**2 - mu)) * (log(X0 / a) - np.log(b / a) * (1 - (X0 / a)**(1 - 2 * mu / sigma**2)) / (1 - (b / a)**(1 - 2 * mu / sigma**2)))

Here is the code

We set up the initial condition for the question:

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)  # For reproducibility

mu = 0.2
sigma = 0.15
X0 = 2
a = 0.5
b = 3
dt = 0.001  # Time step
num_simulations = 1000  # Number of simulations

q52
num_simulations: int = 1000
    return np.random.normal(0, dt)

def simulate_SDE(X0, mu, sigma, dt, a, b):
    X = X0
    exit_time = 0
    while a < X < b:
        X += mu * X * dt + sigma * X * dW(dt)
        exit_time += dt
    return exit_time
```

Then, we need to get the simulation of the SDE, by having the $dX(t)=\mu X(t)dt+\sigma X(t)dW(t)$
Dwt is just the cumulative sum of n N(0,1) times dt^0.5.

After setting up the function, we can plug them into the function we drive above

```python
exit_times = [simulate_SDE(X0, mu, sigma, dt, a, b) for _ in range(num_simulations)]
ea=(1 / (0.5 * sigma**2 - mu)) * (np.log(X0 / a) - np.log(b / a) * (1 - (X0 / a)**(1 - 2 * mu / sigma**2)) / (1 - (b / a)**(1 - 2 * mu / sigma**2)))
```

$$dx = f(x,t)\,dt + g(x,t)\,dW.$$

$$f(x,t) = \mu x \;;\quad g(x,t) = \sigma x.$$

$$a < x_0 < b.$$

we can show that in this case

$u(x)$ = mean exit time, x is the initial point.

$$\begin{cases} \frac{1}{2} g^2 \frac{d^2u}{dx^2} + f(x)\frac{du}{dx} = -1 & (i). \\ u(a) = u(b) = 0. & a < x < b. \end{cases}$$

in this case.

$$\frac{1}{2}\sigma^2 x^2 \frac{d^2u}{dx^2} + \mu x \frac{du}{dx} = -1.$$

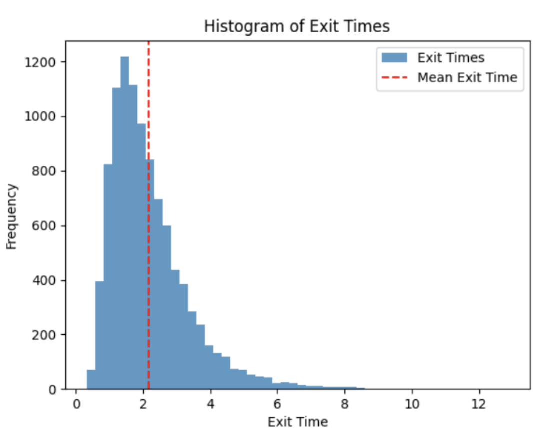
and we know the sol of this

$$u(x) = \frac{1}{\frac{1}{2}\sigma^2 - \mu}\left[\log\left(\frac{x}{a}\right) - \frac{1-\left(\frac{x}{a}\right)^{1-2\mu/\sigma^2}}{1-\left(\frac{b}{a}\right)^{1-2\mu/\sigma^2}}\log\left(\frac{b}{a}\right)\right]$$

This is how to get the analytical solution of the expected exit time.

```
# Perform simulations and store exit times
exit_times = [simulate_SDE(X0, mu, sigma, dt, a, b) for _ in range(num_simulat
ea=(1 / (0.5 * sigma**2 - mu)) * (np.log(X0 / a) - np.log(b / a) * (1 - (X0 / 
# Calculate mean exit time
mean_exit_time = np.mean(exit_times)
error=abs(ea-mean_exit_time)
print("Mean Exit Time (Simulation):", mean_exit_time)
print("Mean Exit Time (ana):", ea)
print("Mean Exit Time (error):", error)
# Plot histogram of exit times
plt.hist(exit_times, bins=50, alpha=0.75, label="Exit Times")
plt.axvline(mean_exit_time, color="r", linestyle="--", label="Mean Exit Time")
plt.xlabel("Exit Time")
plt.ylabel("Frequency")
plt.title("Histogram of Exit Times")
plt.legend()
plt.show()
```

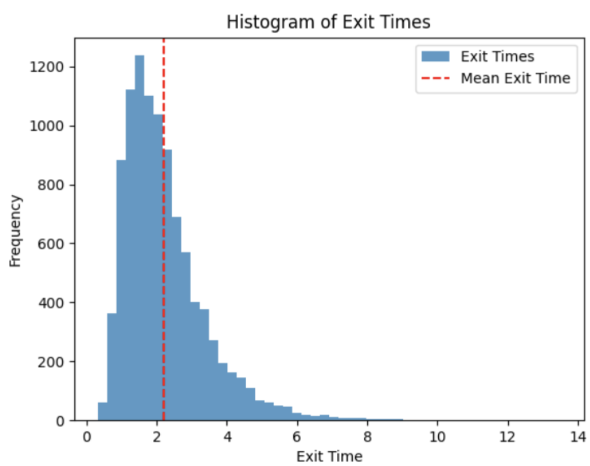
In the end, we get the error between two methods and plot it.



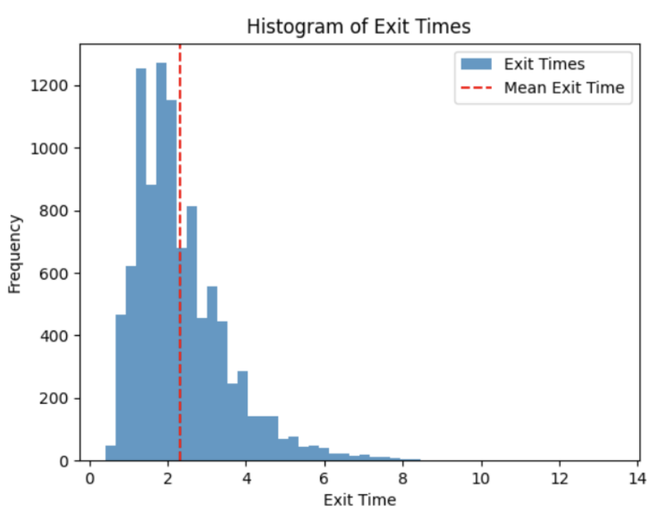
dt=0.001
Mean Exit Time (Simulation): 2.1625030999999053
Mean Exit Time (ana): 2.148159512404766
Mean Exit Time (error): 0.014343587595139429

If dt= 0.01

Mean Exit Time (Simulation): 2.2118439999999895
Mean Exit Time (analytical): 2.148159512404766
Mean Exit Time (error): 0.0636844875952236



If dt=0.1
Mean Exit Time (Simulation): 2.1259000000000006
Mean Exit Time (ana): 2.148159512404766
Mean Exit Time (error): 0.022259512404765314



dt	Mean Exit Time (Simulation)	Mean Exit Time (analytical)	Mean Exit Time (error)
0.1	2.1259	2.148159512	0.022259512
0.01	2.164844	2.148159512	0.016684488
0.001	2.1625031	2.148159512	0.014343588

Thus, we can indicate that as dt=>0, the Mean Exit Time (Simulation) will be equal to Mean Exit Time (ana), and for this question, Mean Exit Time should be close to 2.148159512404766

Then what we want to do is to Calculate mean exit times for different starting points (simulation)

Similar to what we have done above, we have to compare the simulation to the theoretical solution: (1 / (0.5 * sigma**2 - mu)) * (log(x / a) - log(b / a) * (1 - (x / a)**(1 - 2 * mu / sigma**2)) / (1 - (b / a)**(1 - 2 * mu / sigma**2))).

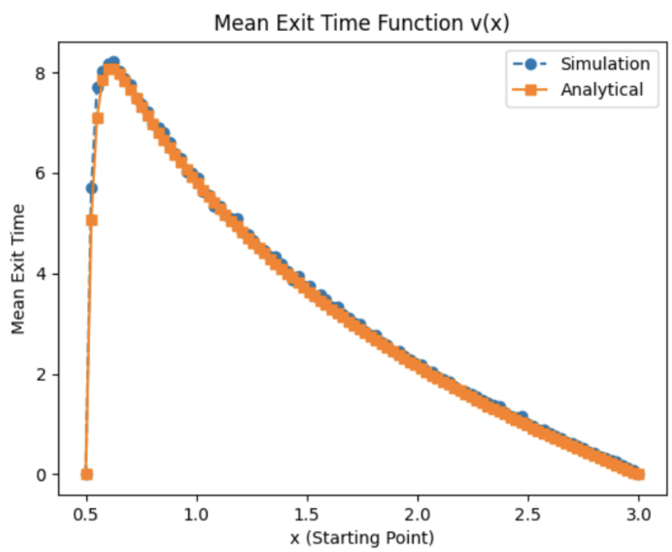
Here is the code:

The initial part is same as above.

```
x_values = np.linspace(0.5, 3, 100)

# Calculate mean exit times for different starting points (simulation)
simulated_mean_exit_times = []
for x in x_values:
   exit_times = [simulate_SDE(x, mu, sigma, dt, a, b) for _ in
range(num_simulations)]
   mean_exit_time = np.mean(exit_times)
   simulated_mean_exit_times.append(mean_exit_time)

# Calculate mean exit times for different starting points (analytical solution)
analytical_mean_exit_times = [analytical_v(x, mu, sigma, a, b) for x in
x_values]
error=abs(simulated_mean_exit_times-analytical_mean_exit_times)

# Plot mean exit time function v(x) for both simulation and analytical solution
plt.plot(x_values, simulated_mean_exit_times, label='Simulation',
linestyle='--', marker='o')
plt.plot(x_values, analytical_mean_exit_times, label='Analytical',
linestyle='-', marker='s')
plt.xlabel("x (Starting Point)")
plt.ylabel("Mean Exit Time")
plt.title("Mean Exit Time Function v(x)")
plt.legend()
```

```
plt.show()


plt.plot(x_values,error,label='error')
plt.xlabel("x (Starting Point)")
plt.ylabel("Mean Exit Time")
plt.title("compare v(x)")
plt.legend()
plt.show()
```

In the end, I compare the error between the two method with the same starting point X.



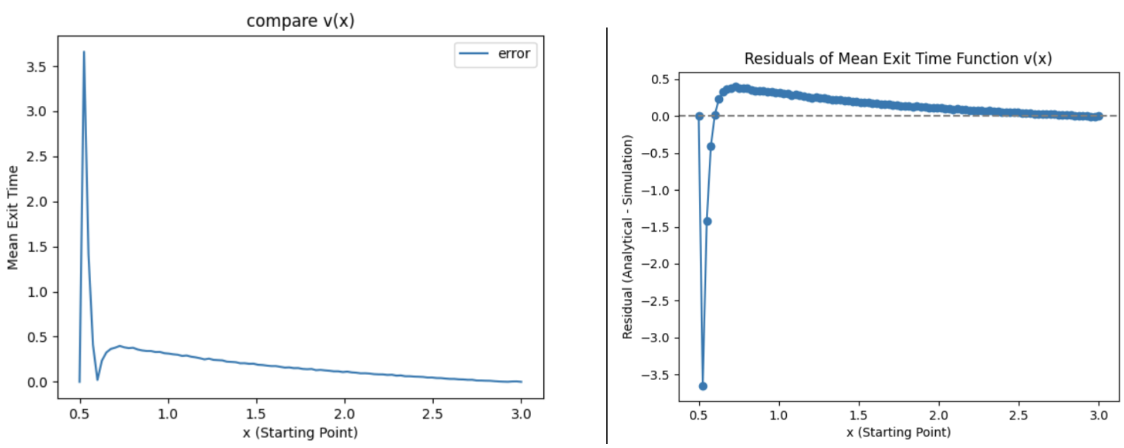
Here is the graph, we can see that it's like a right skewed normal distribution, and the mean exit time is below 9.

To check how good is the simulation. We the calculated the error between them and residuals .

```
analytical_mean_exit_times = [analytical_v(x, mu, sigma, a, b) for x in
x_values]

error=[]
for i in range(len(simulated_mean_exit_times)):
   error.append(abs(simulated_mean_exit_times[i] -
analytical_mean_exit_times[i]))
plt.plot(x_values,error,label='error')
plt.xlabel("x (Starting Point)")
plt.ylabel("Mean Exit Time")
plt.title("compare v(x)")
plt.legend()
plt.show()
```

```python
residuals = np.array(analytical_mean_exit_times) -
np.array(simulated_mean_exit_times)

# Plot residuals
plt.plot(x_values, residuals, linestyle='-', marker='o')
plt.xlabel("x (Starting Point)")
plt.ylabel("Residual (Analytical - Simulation)")
plt.title("Residuals of Mean Exit Time Function v(x)")
plt.axhline(0, color='gray', linestyle='--')
plt.show()
```

Here is the graph



we can tell that as x increase, the analytical and numerical solution are perfectly agreed.

	0	1	2	3	4
0	-0.00000	-3.66012	-1.41482	-0.41168	0.02060

.
.
.

95	96	97	98	99
0.00196	-0.00063	-0.00404	-0.00469	-0.00000

Here is the residual numbers that we get as x approach to 3. We can see taht as it close to the end. It converge to 0.