Introduction to Uncertainty Quantification Coursework

Calum Regan 11013166

November 2025

1 Monte Carlo

We want to derive the forward Euler Approximation of the following system of equations:

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

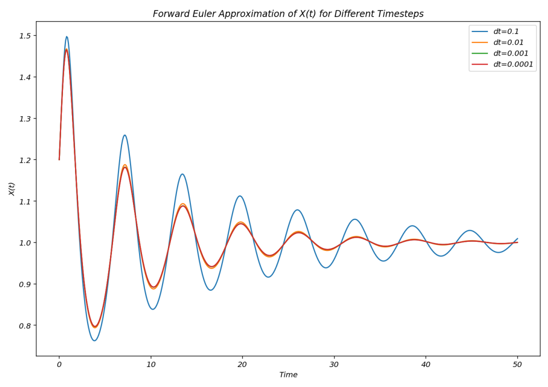
for $A = 1, B = 1.8$ and initial conditions $x(0) = 1.2, y(0) = 2.0$. The forward Euler approximation is given by

$$X_{n+1} = X_n + \Delta t f(X_n, t_n)$$

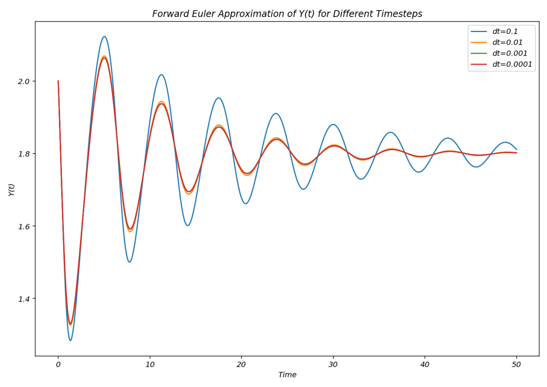
where $t_n = n\Delta t$ for $n = 0, 1, 2, 3, ...$ and $X_n \approx X(n\Delta t)$. In this case, our function f depends on X and Y so our approximation becomes

$$X_{n+1} = X_n + \Delta t(A - (B+1)X_n + X_n^2 Y_n)$$
$$Y_{n+1} = Y_n + \Delta t(BX_n - X_n^2 Y_n)$$

Using Python, we obtained the forward Euler approximation for X and Y at four different time steps $\Delta t = 0.1, 0.01, 0.001, 0.0001$. The graphs are shown in Figure 1 below. Constants are given by $A = 1, B = 1.8$ and $T = 50$.



(a) Approximation for $X(t)$



(b) Approximation for $Y(t)$

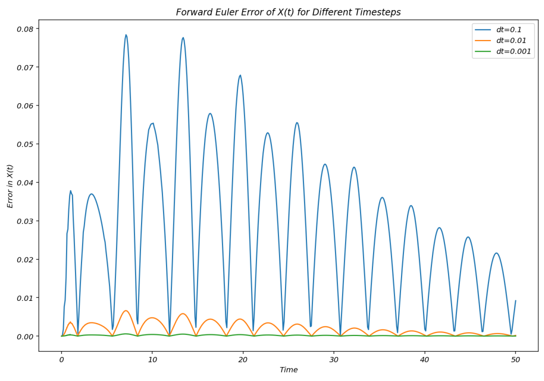
Figure 1: Forward Euler Approximation for $X(t)$ and $Y(t)$ for $\Delta t = 0.1, 0.01, 0.001, 0.0001$

As we can see in Figure 1, as the time step gets smaller, both X and Y begin to converge to a stable solution. Now, we want to consider the error of these approximations.

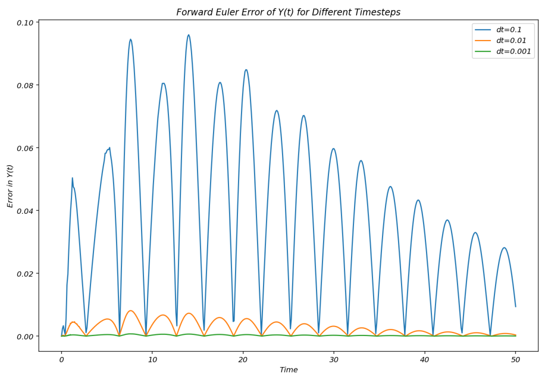
Since the approximation with $\Delta t = 0.0001$ is a very good approximation of the true solution, we will use it as the true solution when we estimate the error of the forward Euler method. The error, e, of the forward Euler method is given by

$$e = |Y_{true} - Y_{estimate}|$$

Once again, we use Python to compute the errors of the approximations for X and Y at three time steps $\Delta t = 0.1, 0.01, 0.001$. The errors are then plotted in Figure 2.



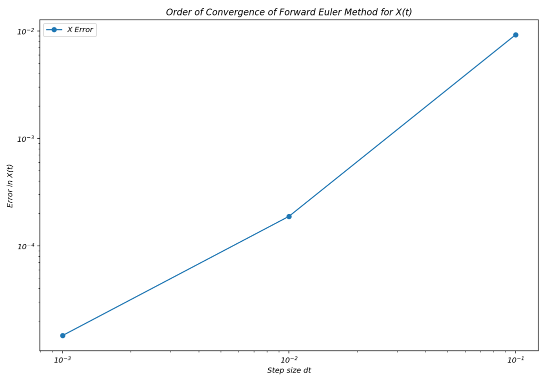
(a) Error for $X(t)$



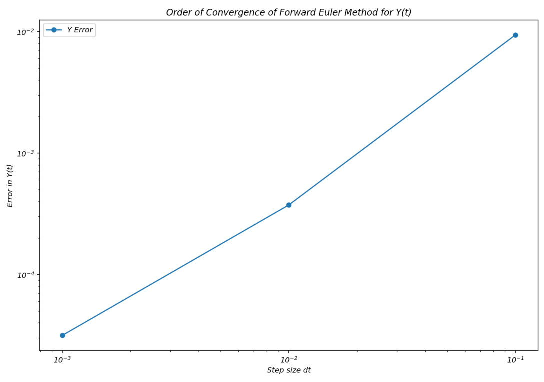
(b) Error for $Y(t)$

Figure 2: Error in the Forward Euler Approximation for $X(t)$ and $Y(t)$ for $\Delta t = 0.1, 0.01, 0.001$,

If we look at the error at $T = 50$ for each time step, we can see the order of convergence for the forward Euler method. We plot time step against error in Figure 3 below.



(a) Order of Convergence for $X(t)$



(b) Order of Convergence for $Y(t)$

Figure 3: Order of Convergence of the Forward Euler Method.

As we can see from Figure 3, the graph is nearly a complete straight line with the form $error = C\Delta t + D$ where C, D are constants, so we can say that the forward Euler method is of order $\mathcal{O}(h)$ where $h = \Delta t$ which is consistent with the theory we developed from the course. The error of X and Y scales as the time step gets larger.

There are some considerations we must think about when choosing Δt. The first is choosing a time step that gives an acceptable error. The forward Euler has error $\mathcal{O}(\Delta t)$ so we should choose a time step that matches our desired error. Secondly, when we have uncertainty in a parameter, we want to choose a time step that makes our approximation stable, otherwise we could have values that "blow-up" or diverge, which makes our approximation less useful.

Now, let us consider when B has uncertainty, modelled by $B \sim Uniform([1.5, 2.5])$. We want to find the time-averaged variance of $x(t)$ in the region $t \in [30, 50]$. Using a Monte Carlo algorithm, we can estimate the expectation of this quantity and plot the density of the samples. We used 5000 samples for the Monte Carlo algorithm, giving a Monte Carlo error of 0.0017.

Using Python, these quantities were computed and density was plotted in a histogram shown in Figure 4.

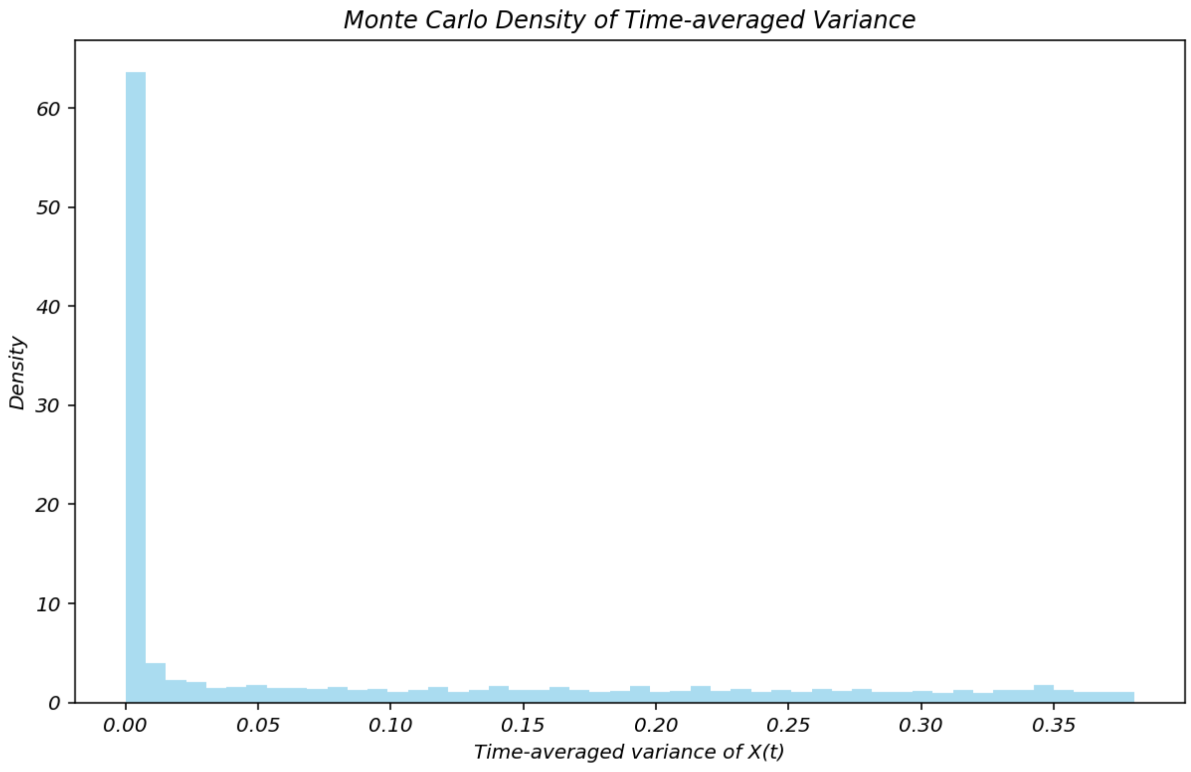


Figure 4: Density of the Time-Averaged Variance of $x(t)$

The expected value of this quantity was 0.0910.

We want to find how many samples M we need to have a standard error (Monte Carlo error) of 10^{-3}. The standard error of a Monte Carlo estimator is given by

$$se(\hat{\theta}) = \sqrt{Var(\hat{\theta}(X))}$$

By rearranging and using the fact $Var(\hat{\theta}(X)) = \sigma^2/M$ where σ^2 is the variance of $x(t)$, we can get the formula

$$M = \frac{\sigma^2}{(se(\hat{\theta}))^2}$$

In this case, $se(\hat{\theta}) = 10^{-3}$ and $\sigma^2 = 0.0142$, we get $M = 14193 \approx 14200$. So we should choose 14200 samples to get a Monte Carlo error of 10^{-3}.

As we can see from Figure 4, the density is concentrated close to 0.0910. If we take a look at Figure 1, the graph starts to stabilises/converge for $t \in [30, 50]$ so the variance will be smaller in this region. This is nearly half of $t \in [0, 50]$, so the lower values of variance will dominated in the density plot, resulting in the high concentration around 1.

2 Euler-Maruyama Monte Carlo

We want to derive the Euler-Maruyama scheme for approximating sample paths of the following stochastic differential equation.

$$dX_t = \left(AV - BX_t - X_t + \frac{X_t(X_t-1)Y_t}{V^2}\right)dt + \sqrt{AV}dW_1 - \sqrt{BX_t}dW_2 - \sqrt{X_t}dW_3 + \sqrt{\frac{X_t(X_t-1)Y_t}{V^2}}dW_4$$

$$dY_t = \left(BX_t - \frac{X_t(X_t-1)Y_t}{V^2}\right)dt + \sqrt{BX_t}dW_2 - \sqrt{\frac{X_t(X_t-1)Y_t}{V^2}}dW_4$$

The Euler-Maruyama scheme is given by

$$X_{n+1} = X_n + \Delta t \mu(t_n, X_n, Y_n) + \sigma(t_n, X_n, Y_n)\sqrt{\Delta t} w_n$$

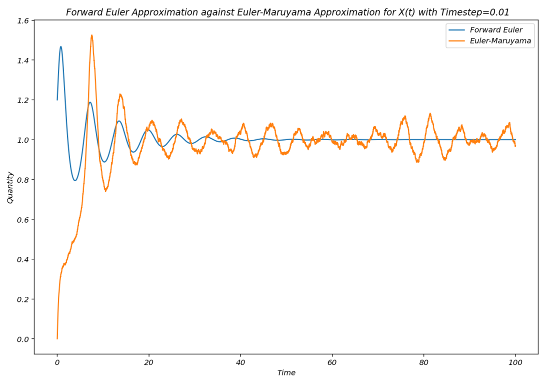
where μ is called the drift term, σ is called the diffusion term, Δt is the time step and w_n is a independent standard Weiner process.

For this problem, the Euler-Maruyama scheme for X and Y is given by

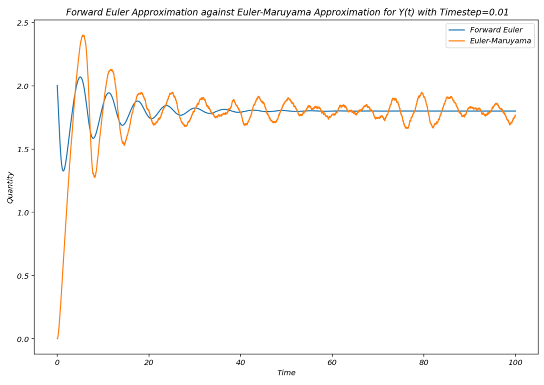
$$X_{n+1} = X_n + \left(AV - BX_t - X_t + \frac{X_t(X_t-1)Y_t}{V^2}\right)\Delta t$$
$$+ \sqrt{AV\Delta t}\, w_n^{(1)} - \sqrt{BX_t\Delta t}\, w_n^{(2)} - \sqrt{X_t\Delta t}\, w_n^{(3)} + \sqrt{\frac{X_t(X_t-1)Y_t}{V^2}\Delta t}\, w_n^{(4)}$$

$$Y_{n+1} = Y_n + \left(BX_t - \frac{X_t(X_t-1)Y_t}{V^2}\right)\Delta t + \sqrt{BX_t\Delta t} w_n^{(2)} - \sqrt{\frac{X_t(X_t-1)Y_t}{V^2}\Delta t} w_n^{(4)}$$

In Python, one realisation of this scheme was plotted for X and Y and then plotted with the forward Euler approximation for the ordinary differential equations in part one. The constants are $A = 1, B = 1.8$ and $\Delta t = 0.01$. The initial conditions for the ODE are $x(0) = 1.2$ and $y(0) = 2.0$. The volume is $V = 10^4$. These plots can be seen in Figure 5.



(a) Euler-Maruyama for $X(t)$



(b) Euler-Maruyama for $Y(t)$

Figure 5: One Realisation of the Euler-Maruyama Approximation compared to the Forward Euler Method for X and Y with time step $\Delta t = 0.001$

In Figure 5, both the deterministic and stochastic solution oscillate up and down as time progresses but the height of the oscillations reduces. Also, the deterministic solution converges for both X and Y shown by the blue line at around $T = 40$. The stochastic solution shown by the orange line begins to stabilise around the deterministic solution for both X and Y. However, we can see as time progresses, the stochastic solution diverges again.

This behaviour from the stochastic equation is surprising because the solution appears to converge, with a small random element and then diverges again. Some random effects are expected because the solution has stochastic components but in $t \in [70, 100]$, these effects become bigger and diverge from the deterministic solution.

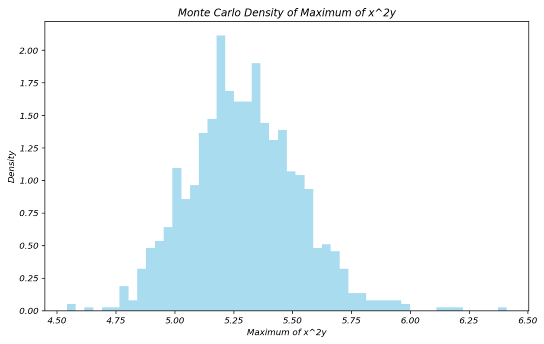
Now we want to estimate the following quantities

$$Q_1 = \max_{t\in[30,50]} x(t)^2 y(t)$$

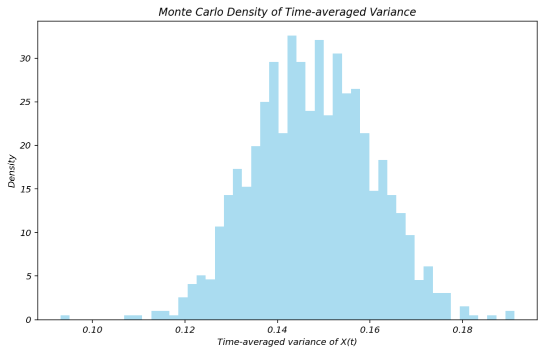
and Q_2 is the time-averaged variance as in (1b). Using the Euler-Maruyama Monte Carlo with 1000 samples, the parameters remain the same apart from we change to $B = 2.2$, we estimated the expected values of Q_1 and Q_2. The expected values were

$$Q_1 = 5.2934$$
$$Q_2 = 0.1477$$

We can plot the densities of these estimators show below in Figure 6.



(a) Density for $\max_{t\in[30,50]}(x(t)^2 y(t))$



(b) Density of Time-averaged variance for $t \in [30, 50]$

Figure 6: Monte Carlo Densities of Quantities Q_1 and Q_2.

We can also plot the joint density of Q_1 and Q_2 in Figure 7.

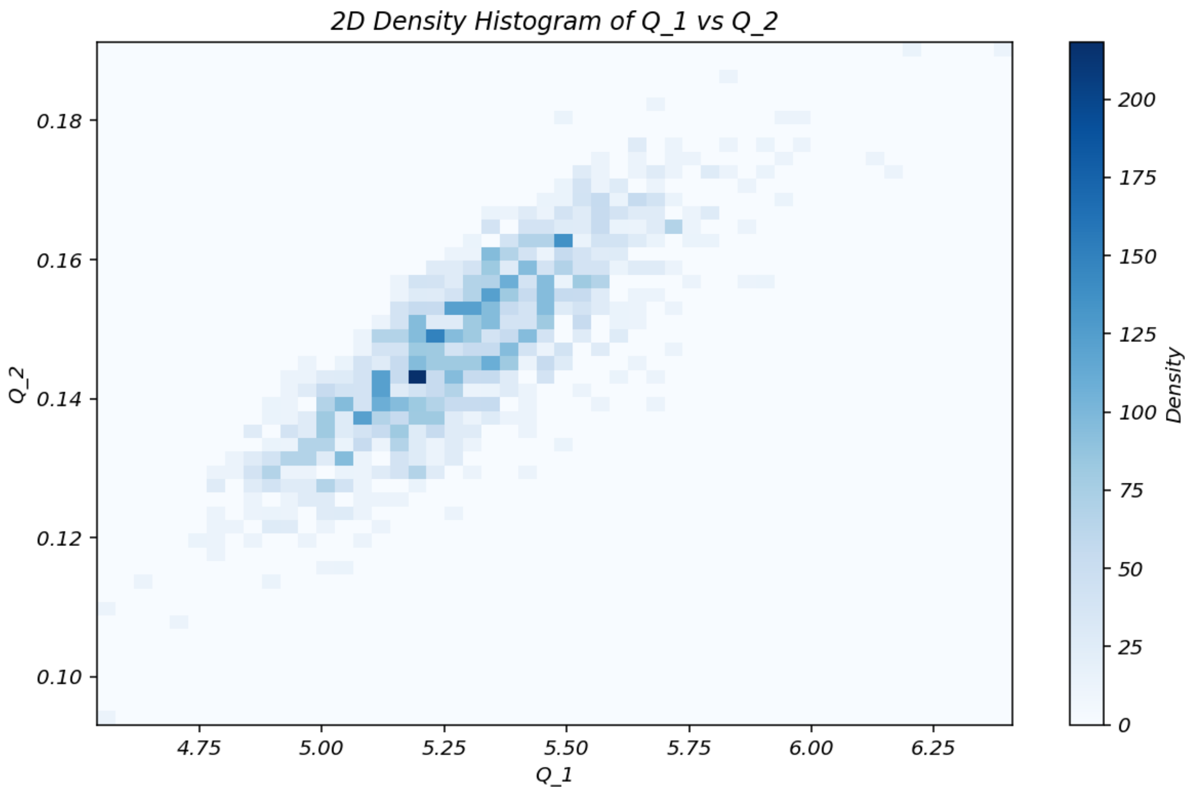


Figure 7: Joint Density of Q_1 and Q_2

Similarly to (1b), We can find the number of samples to get a 10^{-4} standard error using

$$M = \frac{\sigma^2}{(se(\hat{\theta}))^2}$$

Using Python for the calculations, we get

$$M_{Q_1} \approx 5440000 \text{ and } M_{Q_2} \approx 16700$$

There is a large difference between the number of samples needed to obtain the desired error. This is because the standard error of a Monte Carlo estimator scales quadratically, so if the variance is larger, the number of samples needs to be larger as well.

In the case of Q_1 and Q_2, the variance of the time-averaged variance is 0.000168, whereas the variance of the maximum of $x(t)^2y(t)$ is much higher at 0.0544, so the sample size required is far larger.

Figure 7 shows a positive correlation between Q_1 and Q_2. Expanding the Euler-Maruyama scheme shows the term $\frac{X_t^2 Y_t \Delta t}{V^2}$, so when Q_1 is large, the approximation for X_{n+1} will be larger and have a larger variance. This suggests large spikes in Q_1 lead to an increase in the variance. The histogram is also elliptical, suggesting a non-linear relationship. Finally, the density is concentrated at $Q_1 = 5.2934$ and $Q_2 = 0.1477$.

3 Metropolis Hastings

We want to find the posterior density of B, $\pi(B)$ and implement it into Python. We have a proportional relationship between the prior and posterior densities given by

$$\pi(B) \propto \pi_0(B)\pi(Z|B)$$

where $\pi_0(B)$ is the prior distribution and $\pi(Z|B)$ is the likelihood function. We need to find the likelihood function.

Observations subject to Gaussian noise have the form $Z = \mathcal{G}(B) + \eta$ where Z are the observations, $\mathcal{G}(B)$ is the observation operator and η is the Gaussian noise. Then

$$\pi(Z|B) = \pi(\eta = Z - \mathcal{G}(B)) \propto \exp\left(-\frac{1}{2}||\mathcal{G}(B) - Z||_\Sigma^2\right) = \exp\left(-\frac{1}{2\sigma_\eta^2}||\mathcal{G}(B) - Z||^2\right)$$

$$\pi_0(B) \propto \exp\left(-\frac{1}{2\sigma_B^2}(B - \mu_B)^2\right)$$

So the posterior density of B, up to normalisation constant, is

$$\pi(B) \propto \exp\left(-\frac{1}{2\sigma_B^2}(B - \mu_B)^2\right) \exp\left(-\frac{1}{2\sigma_\eta^2}||\mathcal{G}(B) - Z||^2\right)$$

This is the product of two Gaussian densities so Lemma 5.3 from the course states that this product is also a Gaussian density. Using Python, the mean and variance of the posterior can be calculated. We get $\mu = 2.1627$ and $\sigma^2 = 0.0006$

Using Python, we evaluated the posterior density on a uniform grid $B \in [1.4, 2.6]$, with a spacing of 0.001. We can plot the prior and posterior densities on the same graph in Figure 8

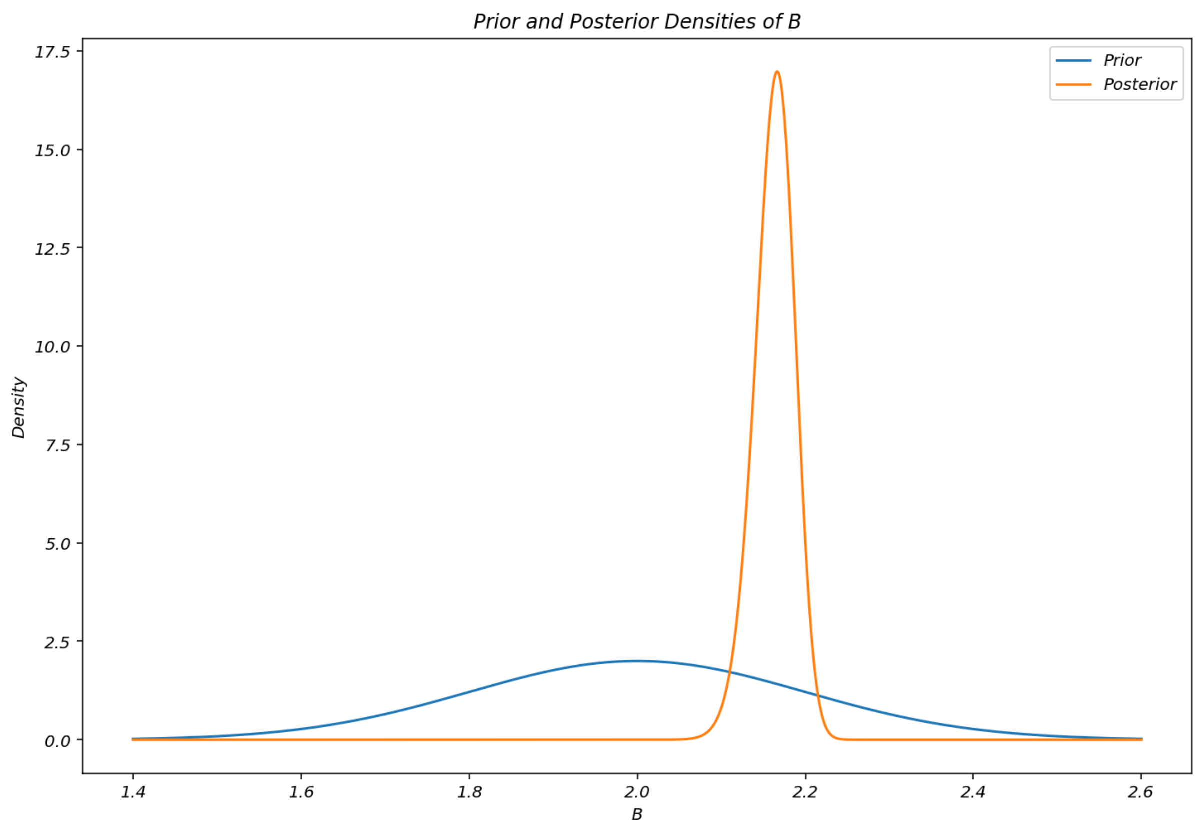


Figure 8: Prior and Posterior Densities for B

In Figure 8, the prior curve is centred on 2.0 and much flatter meaning there is more uncertainty in the parameter B. This is expected as the prior is just an initial belief about the distribution of the parameter. Having considered the data we observed, the posterior is shifted to the right and centred on 2.15, with a sharp peak around this value. This is because there is more certainty about the parameter B based on the observations of the model. This is why there is a difference between prior and posterior curves.

Now we will implement a random walk Metropolis-Hastings (RWMH) algorithm to sample from the density π. We will use $\beta = 0.1$, an initial value of $B = 1.5$ and use $100,000$ samples. The sequence of samples B is plotted against it's iteration number below in Figure 9.

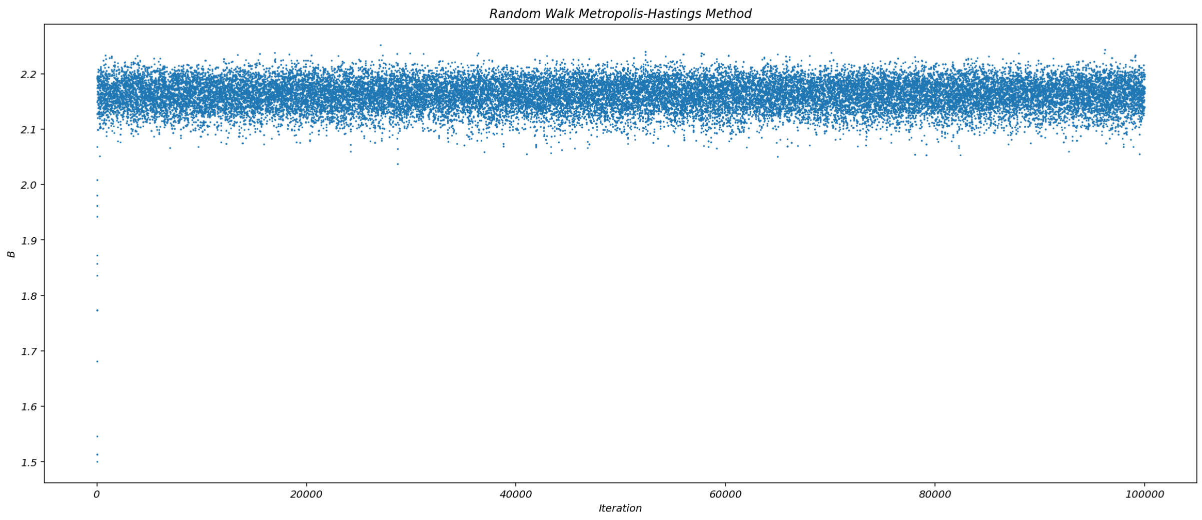


Figure 9: Random Walk Metropolis-Hastings Method for B

In Figure 9, we can see a section of burn-in samples at the beginning of the plot which trace where the samples move from the initial value of $B = 1.5$ to samples from the posterior distribution. This RWMH method moves very quickly from the initial value to samples from the posterior distribution but due to the large number of data points it is hard to tell exactly where the samples being to converge. Hence, we will throw a larger, but still small relative to the number of samples, number of burn-in points to be safe. We will consider 2000 burn-in points, 2% of the total number of samples.

Having thrown away the burn-in samples, we can now plot a histogram of the samples of B with the posterior density $\pi(B)$ we had earlier. This is shown in Figure 10.

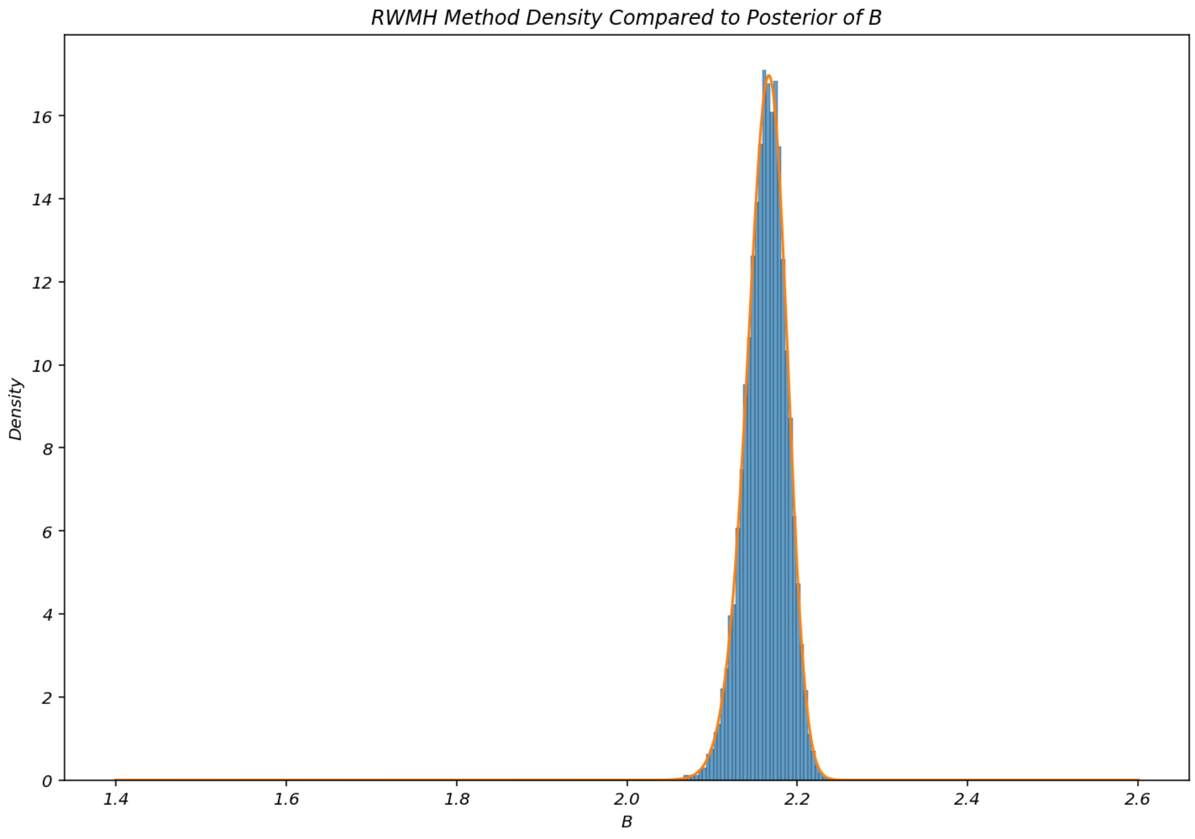


Figure 10: RWMH Histogram Compared to Posterior Density

In Figure 10, there is very little difference between the posterior distribution and the histogram of the samples from the RWMH. Both the histogram and the posterior density curve are concentrated around the same value ($\mu \approx 2.16$), confirming that the RWMH algorithm has sampled from the desired posterior distribution. The slight discrepancies are down to the RWMH method being a discrete approximation.

We can also compute the sample mean and variance in Python which gives the values $\bar{\mu} = 2.1628$ and $\bar{\sigma}^2 = 0.0006$. Using these values and Chebyshev's inequality, we can construct a 95% confidence interval for B. Chebyshev's inequality for some $K > 0$, is given by

$$\mathbb{P}(|B - \bar{\mu}| \geq k\bar{\sigma}) \leq \frac{1}{k^2}$$

Rearranging gives

$$\mathbb{P}(|B - \bar{\mu}| < k\bar{\sigma}) \geq 1 - \frac{1}{k^2}$$

For a 95% confidence interval, we want k such that $1 - 1/k^2 = 0.95$. So $k = \sqrt{20} \approx 4.4721$. Then the confidence interval becomes

$$\bar{\mu} - k\bar{\sigma} < B < \bar{\mu} + k\bar{\sigma}$$

Substituting in these values gives the 95% confidence interval for B; $2.0553 < B < 2.2703$

4 AI Declaration

AI was used in Question 1a to debug my forward Euler function as I had mixed up the power symbol ˆ instead of **.

AI was used in Question 2b to debug the quantities functions as I had not realised max() was part of the numpy package, so needed np.max() instead.

AI was used in Question 3a to debug my prior function as I was using the wrong variance. I was using the variance of the Gaussian noise instead of the variance of the prior distribution.

AI was used to grammar check the whole project. It picked up a few spelling mistakes and repeated words in questions 1 and 3.

5 Appendix

This appendix contains all code relating to this coursework. It was written in Python.

Code for Question 1

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

np.random.seed(0)

#————————QUESTION 1a————————

#function to calculate forward Euler approximation
```

```
#input variables are A,B,X_0,Y_0,dt,T for greater flexibility

def forward_euler(A, B, X_0, Y_0, dt, T):

    X_n = [X_0] #vector of X values, with X_0

    Y_n = [Y_0] #vector of Y values with Y_0

    steps = int(T/dt) #how many steps we want to compute for

    #creates an array of numbers with spacing "steps"
    t = np.linspace(0, T, steps + 1)

    for i in range(0,steps): #for each step do the following

        #calculate X_(n+1) from our forward Euler approximation
        #add the value of X_(n+1) to the vector of X_ns
        X_n.append(X_n[i]+(A-(B+1.0)*X_n[i]+((X_n[i])**2)*Y_n[i])*dt)
        #AI used for ** instead of ^

        #calculate Y_(n+1) from our forward Euler approximation
        #add the value of Y_(n+1) to the vector of Y_ns
        Y_n.append(Y_n[i]+(B*X_n[i]-((X_n[i])**2)*Y_n[i])*dt)

        #AI used here to debug power as I used ^ instead of **

    # Create a DataFrame for X_n, Y_n values for easy malnipulation
    X_Y_df = pd.DataFrame({
    'Time': t,
    'X_n': X_n,
    'Y_n': Y_n
    })

    return X_Y_df #return the dataframe of values of X_n,Y_n

#compute for different time steps 0.1,0.01,0.001,0.0001
approx_1 = forward_euler(1, 1.8, 1.2, 2.0, 0.1, 50) #timestep = 0.1
approx_2 = forward_euler(1, 1.8,1.2, 2.0, 0.01, 50) #timestep = 0.01
approx_3 = forward_euler(1, 1.8,1.2, 2.0, 0.001, 50) #timestep = 0.001
approx_4 = forward_euler(1, 1.8,1.2, 2.0, 0.0001, 50) #timestep = 0.0001

#plot approximations for X on same graph
plt.figure(figsize=(12,8))
plt.plot(approx_1['Time'], approx_1['X_n'], label='dt=0.1')
plt.plot(approx_2['Time'], approx_2['X_n'], label='dt=0.01')
plt.plot(approx_3['Time'], approx_3['X_n'], label='dt=0.001')
plt.plot(approx_4['Time'], approx_4['X_n'], label='dt=0.0001')
plt.xlabel('Time')
plt.ylabel('X(t)')
plt.title('Forward Euler Approximation of X(t) for Different Timesteps')
plt.legend()
plt.show()

#plot approximations for Y on same graph
plt.figure(figsize=(12,8))
plt.plot(approx_1['Time'], approx_1['Y_n'], label='dt=0.1')
plt.plot(approx_2['Time'], approx_2['Y_n'], label='dt=0.01')
plt.plot(approx_3['Time'], approx_3['Y_n'], label='dt=0.001')
plt.plot(approx_4['Time'], approx_4['Y_n'], label='dt=0.0001')
plt.xlabel('Time')
plt.ylabel('Y(t)')
plt.title('Forward Euler Approximation of Y(t) for Different Timesteps')
plt.legend()
plt.show()

#error of the forward Euler method is |"actual solution"-"computed
    solution"|
#dont have actual solution so we will use values at timestep 0.0001
#as it is a good approximation of the true solution.

#function to compute error using timestep 0.0001 as the true value
def approx_error(approximation, dt, T):

    #define fine and coarse, makes sure we compare true and approximate
        values
    #at the right timepoints instead of by the index of the dataframe

    #fine solution (dt = 0.0001)
    fine = approx_4.set_index("Time")

    #coarse solution
    coarse = approximation.set_index("Time")

    #map fine solution onto coarse time grid
    fine_interp = fine.reindex(coarse.index).interpolate()

    #error calculation
    error_X = np.abs(fine_interp['X_n'] - coarse['X_n'])
    error_Y = np.abs(fine_interp['Y_n'] - coarse['Y_n'])

    #add values to a dataframe for easier malnipulation
    error = pd.DataFrame({
        'Time': coarse.index,
        'X_error': error_X,
        'Y_error': error_Y
    })

    return error

#compute the error at each timepoint
error_1 = approx_error(approx_1, 0.1, 50)
error_2 = approx_error(approx_2, 0.01, 50)
error_3 = approx_error(approx_3, 0.001, 50)

#plotting the error of X for the three time steps
plt.figure(figsize=(12,8))
plt.plot(error_1['Time'], error_1['X_error'], label='dt=0.1')
plt.plot(error_2['Time'], error_2['X_error'], label='dt=0.01')
plt.plot(error_3['Time'], error_3['X_error'], label='dt=0.001')
plt.xlabel('Time')
plt.ylabel('Error in X(t)')
plt.title('Forward Euler Error of X(t) for Different Timesteps')
plt.legend()
plt.show()

#plotting the error of X for the three time steps
```

```
plt.figure(figsize=(12,8))
plt.plot(error_1['Time'], error_1['Y_error'], label='dt=0.1')
plt.plot(error_2['Time'], error_2['Y_error'], label='dt=0.01')
plt.plot(error_3['Time'], error_3['Y_error'], label='dt=0.001')
plt.xlabel('Time')
plt.ylabel('Error in Y(t)')
plt.title('Forward Euler Error of Y(t) for Different Timesteps')
plt.legend()
plt.show()

#create one large dataframe of all error values
#we will take values at T=50 and show order of convergence
error_df = pd.DataFrame({
    'Time': error_1['Time'],
    'Error1_X': error_1['X_error'],
    'Error1_Y': error_1['Y_error'],
    'Error2_X': error_2['X_error'],
    'Error2_Y': error_2['Y_error'],
    'Error3_X': error_3['X_error'],
    'Error3_Y': error_3['Y_error'],
})

#select row T=50
error_time50 = error_df[error_df['Time'] == 50]

#time steps as points on x axis
dt_values = [0.1, 0.01, 0.001]

#get scalar errors at T=50 for X
error_X = [
    error_time50['Error1_X'].values[0],
    error_time50['Error2_X'].values[0],
    error_time50['Error3_X'].values[0]
]

#get scalar errors at T=50 for Y
error_Y = [
    error_time50['Error1_Y'].values[0],
    error_time50['Error2_Y'].values[0],
    error_time50['Error3_Y'].values[0]
]

#plot error at three time steps at T=50 for X for order of convergence
plt.figure(figsize=(12,8))
plt.plot(dt_values, error_X, 'o-', label='X Error')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Step size dt')
plt.ylabel('Error in X(t)')
plt.title('Order of Convergence of Forward Euler Method for X(t)')
plt.legend()
plt.show()

#plot error at three time steps at T=50 for Y for order of convergence
plt.figure(figsize=(12,8))
plt.plot(dt_values, error_Y, 'o-', label='Y Error')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Step size dt')
plt.ylabel('Error in Y(t)')
plt.title('Order of Convergence of Forward Euler Method for Y(t)')
plt.legend()
plt.show()


#----------QUESTION 1b----------

#function to compute time-averaged variance for one simulation
#T0, T1 denote start and end times of the period of interest
#other parameters are same as foward_euler
def compute_time_avg_variance(A, B, X_0, Y_0, dt, T, T0, T1):

    #values from forward Euler approximation
    df = forward_euler(A, B, X_0, Y_0, dt, T)

    #look at time period we are interested in
    X_interest = df[(df['Time'] >= T0) & (df['Time'] <= T1)]['X_n'].values

    #compute variance
    var_X = np.var(X_interest, ddof=1)

    return var_X

#function to perform Monte Carlo estimation
#number of samples is set at 5000
#range of possible values of B is [1.5,2.5]
def monte_carlo_variance(A, X_0, Y_0, dt, T, T0, T1, num_MC=5000,
                         B_range=(1.5, 2.5)):

    #stores each estimate of time-averaged variance
    var_list = []

    #for each Monte Carlo simulation the function will
    #1)sample from the uniform distribution B~Uniform([1.5,2.5])
    #2)compute the time average variance for that value of B
    #3)add that value for the variance to the list
    for _ in range(num_MC):
        B = np.random.uniform(*B_range)
        var_X = compute_time_avg_variance(A, B, X_0, Y_0, dt, T, T0, T1)
        var_list.append(var_X)

    #turn into an array for easier handling
    var_array = np.array(var_list)

    #take the expected value of the time-averaged variance
    expected_var = np.mean(var_array)

    return var_array, expected_var

#run Monte Carlo with following parameters
var_array, expected_var = monte_carlo_variance(A=1,
                                               X_0=1.2,
                                               Y_0=2.0,
                                               dt=0.01,
                                               T=50,
                                               T0=30,
                                                                      T1=50,
                                                                      num_MC=5000)

#want to compute number of samples for certain error
#need variance of the quantity we are estimating
variance_var = np.var(var_array)

#desired monte carlo error
monte_carlo_error = 0.001

#formula for number of samples
M = variance_var/((monte_carlo_error)**2)

#displays the expected value of the variance,
#variance of time-averaged variance, number of samples for certain error
print(f"Estimated E[Var[X(t)]] over [30,50]: {expected_var:.4f}")
print(f"Variance of time-averaged variance : {variance_var:.4f}")
print(f"Minimum number of samples to have Monte Carlo error of 0.01:
    {M:.4f}")

#function to plot density
def plot_density(var_array):
    plt.figure(figsize=(10,6))
    plt.hist(var_array, bins=50, density=True, alpha=0.7, color='skyblue')
    plt.xlabel('Time-averaged variance of X(t)')
    plt.ylabel('Density')
    plt.title('Monte Carlo Density of Time-averaged Variance')
    plt.show()

#plot density
plot_density(var_array)
```

Code for Question 2

```
    #----------QUESTION 2a----------

#function for the Euler-Maruyama method.
def euler_mar(X_V, Y_V, A, B, V, dt, T):

    #divide by volume to get X_0,Y_0
    X_0 = X_V / V
    Y_0 = Y_V / V

    #number of steps to find X,Y at
    steps = int(T / dt)

    #create two vectors of length steps and fill with zeros to store data
    X = np.zeros(steps + 1)
    Y = np.zeros(steps + 1)

    #creates an array of numbers with spacing "steps"
    t = np.linspace(0, T, steps + 1)

    #set our vectors of X and Y to contain X_0,Y_0
    X[0] = X_0
    Y[0] = Y_0

    for n in range(steps):
        #for each time step, generate 4 Weier terms with mean 0, sigma 1
        #multiplied by square root of time step
        dW1, dW2, dW3, dW4 = np.sqrt(dt) * np.random.normal(0, 1, 4)

        #drift/deterministic terms for X and Y
        X_drift = A*V - B*X[n] - X[n] + (X[n] * (X[n] - 1) * Y[n]) / V**2
        Y_drift = B*X[n] - (X[n] * (X[n] - 1) * Y[n]) / V**2

        #diffusion/stochastic terms for X,Y
        X_diff = (np.sqrt(A*V) * dW1
              - np.sqrt(B*X[n]) * dW2
              - np.sqrt(X[n]) * dW3
              + np.sqrt((X[n] * (X[n] - 1) * Y[n]) / V**2) * dW4)

        Y_diff = (np.sqrt(B*X[n]) * dW2
              - np.sqrt((X[n] * (X[n] - 1) * Y[n]) / V**2) * dW4)

        #EulerMaruyama method by combining both drift and diffusion
            terms
        X[n+1] = X[n] + X_drift * dt + X_diff
        Y[n+1] = Y[n] + Y_drift * dt + Y_diff

        #X,Y cannot be negative,
        #if either is negative then it is set to 0 for that timepoint
        if X[n+1] < 0:
            X[n+1] = 0.0
        if Y[n+1] < 0:
            Y[n+1] = 0.0

    #create a datframe that contains all information on X,Y
    df = pd.DataFrame({
        "Time": t,
        "X_n": X/V,
        "Y_n": Y/V
    })

    return df

#function to plot Euler-Maruyama approximation
def plot_euler_mar(df,dt):
    plt.figure(figsize=(12,8))
    plt.plot(df['Time'], df['X_n'], label='X')
    plt.plot(df['Time'], df['Y_n'], label='Y')
    plt.xlabel('Time')
    plt.ylabel('Quantity')
    plt.title(f' Euler-Maruyama Approximation of X and Y with Timestep
        {dt}')
    plt.legend()
    plt.show()

df = euler_mar(X_V = 1.2*10000,Y_V = 2*10000,A = 1,B = 1.8,V = 10000,
               dt = 0.01,T = 100.0)

plot_euler_mar(df, 0.01)

#function to plot the ODE next to SDE solution
def plot_ODE_SDE(ode_approx, sde_approx, dt, T):
    plt.figure(figsize=(12,8))
    plt.plot(ode_approx['Time'], ode_approx['X_n'], label='Forward Euler')
    plt.plot(sde_approx['Time'], sde_approx['X_n'], label='Euler-Maruyama')
    plt.xlabel('Time')
    plt.ylabel('Quantity')
    plt.title(f' Forward Euler Approximation against Euler-Maruyama
       Approximation for X(t) with Timestep={dt}')
    plt.legend()
    plt.show()

    plt.figure(figsize=(12,8))
    plt.plot(ode_approx['Time'], ode_approx['Y_n'], label='Forward Euler')
    plt.plot(sde_approx['Time'], sde_approx['Y_n'], label='Euler-Maruyama')
    plt.xlabel('Time')
    plt.ylabel('Quantity')
    plt.title(f' Forward Euler Approximation against Euler-Maruyama
       Approximation for Y(t) with Timestep={dt}')
    plt.legend()
    plt.show()

approx = forward_euler(1, 1.8,1.2, 2.0, 0.01, 100)

plot_ODE_SDE(approx, df, 0.01, 100)

#--------QUESTION 2b--------

#function to compute quantities of interest
#T0,T1 for start and stop time of QoIs
def quantities(A, B, X_V, Y_V, dt, V, T, T0, T1):

    #values from forward Euler approximation
    df = euler_mar(X_V, Y_V, A, B, V, dt, T)

    #look at time period we are interested in
    X_interest = df[(df['Time'] >= T0) & (df['Time'] <= T1)]['X_n'].values
    df_interest = df[(df['Time'] >= T0) & (df['Time'] <= T1)]

    #compute max of x^2y
    Q_1 = np.max((df_interest['X_n']**2)*df_interest['Y_n'])

    #compute variance
    Q_2 = np.var(X_interest, ddof=1)

    return Q_1, Q_2

#function to run monte carlo simulation
def mc_quantities(A, B, X_V, Y_V, dt, V, T, T0, T1, num_MC):

    #stores Q_1,Q_2 values
    Q_1_list = []
    Q_2_list = []

    for _ in range(num_MC):

        #get quantities
        Q_1, Q_2 = quantities(A, B, X_V, Y_V, dt, V, T, T0, T1)

        #add them to corresponding list
        Q_1_list.append(Q_1)
```

```
            Q_2_list.append(Q_2)

        #convert to arrays for easy malnipulation
        Q_1_array = np.array(Q_1_list)
        Q_2_array = np.array(Q_2_list)

        #compute expectation of both quantities
        expected_Q_1 = np.mean(Q_1_array)
        expected_Q_2 = np.mean(Q_2_array)

        return Q_1_array, Q_2_array, expected_Q_1, expected_Q_2

#function to plot density for Q_1, Q_2 and joint density
def plot_sde_denisty(Q_1_array, Q_2_array):

        plt.figure(figsize=(10,6))
        plt.hist(Q_1_array, bins=50, density=True, alpha=0.7, color='skyblue')
        plt.xlabel('Maximum of x^2y')
        plt.ylabel('Density')
        plt.title('Monte Carlo Density of Maximum of x^2y')
        plt.show()

        plt.figure(figsize=(10,6))
        plt.hist(Q_2_array, bins=50, density=True, alpha=0.7, color='skyblue')
        plt.xlabel('Time-averaged variance of X(t)')
        plt.ylabel('Density')
        plt.title('Monte Carlo Density of Time-averaged Variance')
        plt.show()

        plt.figure(figsize=(10,6))
        plt.scatter(Q_1_array, Q_2_array, alpha=0.4, s=10)
        plt.title("Joint Distribution of Q_1, Q_2")
        plt.xlabel("Q_1")
        plt.ylabel("Q_2")
        plt.show()

Q_1_array, Q_2_array, expected_Q_1, expected_Q_2 = mc_quantities(A=1,
                                                                 B=2.2,
                                                                 X_V =
                                                                     1.2*10000,
                                                                 Y_V =
                                                                     2*10000,
                                                                 dt=0.01,
                                                                 V=10000,
                                                                 T=50,
                                                                 T0=30,
                                                                 T1=50,
                                                                 num_MC=1000)

#expectations of Q_1, Q_2
print(f"Estimated E[max(x^2y) over [30,50]: {expected_Q_1:.4f}")
print(f"Estimated E[Var[X(t)]] over [30,50]: {expected_Q_2:.4f}")

plot_sde_denisty(Q_1_array, Q_2_array)

#want to find number of samples to get a ceratin error
#find variances of quantities
Q_1_var = np.var(Q_1_array)
```

```
Q_2_var = np.var(Q_2_array)

#specfified error
EM_monte_carlo_error = 0.0001

#compute number of samples for quantities
M_Q_1 = Q_1_var/((EM_monte_carlo_error)**2)
M_Q_2 = Q_2_var/((EM_monte_carlo_error)**2)

#show number of samples needed to get a certain error
print(f"Samples for Q_1: {M_Q_1:.4f}")
print(f"Samples for Q_2: {M_Q_2:.4f}")
```

Code for Question 3

```
    #----------QUESTION 3a---

#data from Table 1
data = np.array([
    [6.87094*10**(-1), 8.86372*10**(-1), 1.16966*10**(0)],
    [2.57202*10**(0), 2.10889*10**(0), 1.48356*10**(0)]
    ])

#times of interest
T_array = [10,15,20]

#parameters of Gaussian noise
mean_noise = 1.0
sigma_noise = 0.1

#parameters of a Gaussian prior
mean_prior = 2.0
sigma_prior = 0.2

#function to estimate the observation operator G(B)
def observation_operator(B, T_array):

    #store approximations from forward Euler
    x_vals = []
    y_vals = []

    for T in T_array:

        #for each time
        #1)get the values for the approximation
        #2)locate the values at time T
        #3)add the X,Y values to corresponding lists
        df = forward_euler(A=1, B=B, X_0=1.2, Y_0=2.0, dt=0.01, T=T)

        idx = (df['Time'] - T).abs().idxmin()
        row = df.loc[idx]

        x_vals.append(row['X_n'])
        y_vals.append(row['Y_n'])

    #convert into a 2 3  matrix --> same form as in question
    G = np.vstack([x_vals, y_vals])

    return G
```

```
#function to estimate likelihood
def likelihood(B, data, T_array):

    #get observation operator
    G = observation_operator(B, T_array)

    #need the difference for ||G(B)-Y|| part of density
    difference = G-data

    #calculate likelihood based on Gaussian noise density ~N(0,sigma_noise)
    density = np.exp((-1/(2*sigma_noise**2))*np.sum(difference**2))

    return density

#function for density of the prior for B
def prior(B):

    #compute density base on Gaussian density ~N(2.0,0.2^2)
    density =
        (1/(sigma_prior*np.sqrt(2*np.pi)))*np.exp((-1/(2*sigma_prior**2))*(B-mean_prior)

    return density #AI caught i was using sigma noise instead of sigma
        prior

#function to compute posterior density
def posterior(B, data, T_array):

    #get prior
    prior_density = prior(B)

    #get likelihood
    like = likelihood(B, data, T_array)

    #formula for posterior density
    posterior_density = prior_density*like

    return posterior_density

#function for evaluating posterior denisty on a grid for B in [1.4,2.6]
#start,stop,spacing denote interavl and space between values on uniform
    grid
def evaluate_and_plot(start, stop, spacing, data, T_array):

    #define our grid of B values
    B_grid = np.linspace(start, stop, int((stop-start)/spacing) + 1)

    #compute prior, posterior values for each B
    prior_values = np.array([prior(B) for B in B_grid])
    posterior_values = np.array([posterior(B, data, T_array) for B in
        B_grid])

    #normalise prior, posterior values with trapeziod method
    #this involves numerically intergrating of the grid
    #np.trapezoid is a function in numpy that follows this method
    normalised_prior_values = prior_values/np.trapezoid(prior_values,
        B_grid)
    normalised_posterior_values =
        posterior_values/np.trapezoid(posterior_values, B_grid)

    #plot the density of the prior and posterior
    plt.figure(figsize=(12,8))
    plt.plot(B_grid, normalised_prior_values, label='Prior')
    plt.plot(B_grid, normalised_posterior_values, label='Posterior')
    plt.xlabel('B')
    plt.ylabel('Density')
    plt.title('Prior and Posterior Densities of B')
    plt.legend()
    plt.show()

    return normalised_posterior_values, B_grid

normalised_posterior_values, B_grid = evaluate_and_plot(1.4, 2.6, 0.001,
                                                         data=data, T_array=T_array)

#calculate posterior mean and density
posterior_mean = np.trapezoid(B_grid * normalised_posterior_values, B_grid)
posterior_variance = np.trapezoid((B_grid - posterior_mean)**2 *
    normalised_posterior_values, B_grid)

print(f'Posterior mean: {posterior_mean:.4f}')
print(f'Posterior variance: {posterior_variance:.4f}')

#-----------QUESTION 3b-----------

#function for random walk Metropolis Hastings
def RWMH(beta, B_0, RWMH_samples, data, T_array):

    #stores values of B starting with initial value
    B = [B_0]

    #dictionary to cache posterior evaluations
    posterior_cache = {}

    #precompute posterior for initial state
    posterior_cache[B_0] = posterior(B_0, data=data, T_array=T_array)

    for i in range(RWMH_samples):

        #for each sample number
        #1)sample Y from N(B_(i-1),beta^2)
        #2)calculate the acceptance probability
        #with Metropolis-Hastings formula
        #3)accept or reject Y
        Y = np.random.normal(loc=B[-1], scale=beta)

        #posterior cache speeds up the calculation of
        #posteriors to speed up overall runtime
        #by checking if the posterior for that value of
        #B has already been calculated
        #if not it is calculated anyway and added to the cache
        if Y in posterior_cache:
            post_Y = posterior_cache[Y]
        else:
            post_Y = posterior(Y, data=data, T_array=T_array)
            posterior_cache[Y] = post_Y
        post_current = posterior_cache[B[-1]]

        #acceptance probability from Metropolis-Hastings method
        prob_ratio = post_Y / post_current
        acceptance_prob = min(1, prob_ratio)

        U = np.random.uniform()

        #acceptance => B_i=Y_i
        if U < acceptance_prob:
            B.append(Y)
        #rejection => B_i=B_(i-1)
        else:
            B.append(B[-1])

        #I added this as a sanity check because it took so long to run
        #so I did not know if it was working correctly
        #Sorry if it is a bit annoying when you run my code :)
        if (i+1) % 10000 == 0:  # print every 1000 iterations
            print(f"Iter ({(i+1)/RWMH_samples*100:.1f}%)")

    #create dataframe for easier malnipulation
    df = pd.DataFrame({
        "iteration": np.arange(0, RWMH_samples + 1),
        "B": B
    })

    return df

B_df = RWMH(beta = 0.1, B_0 = 1.5, RWMH_samples = 100000,
            data=data, T_array=T_array)

#function to plot RWMH
def plot_RWMH(B_df):

    plt.figure(figsize=(20,8))
    plt.scatter(B_df['iteration'], B_df['B'], s=0.5)
    plt.xlabel('Iteration')
    plt.ylabel('B')
    plt.title('Random Walk Metropolis-Hastings Method')
    plt.show()

plot_RWMH(B_df)

#function to remove certain number of burn-in samples
def remove_burn_in(B_df, N_remove):

    B_df_new = B_df.iloc[N_remove:].reset_index(drop=True)

    return B_df_new

#function to plot histogram against posterior
def hist_vs_posterior(B_df, B_grid, posterior):

    plt.figure(figsize=(12,8))
    plt.hist(B_df['B'], bins=50, density=True, alpha=0.7,
             edgecolor='black', linewidth=0.2, label ='RWMH Method')
    plt.plot(B_grid, posterior, label = 'Posterior ')
    plt.xlabel('B')
    plt.ylabel('Density')
    plt.title('RWMH Method Density Compared to Posterior of B')
    plt.show()

B_df_new = remove_burn_in(B_df, N_remove = 2000)
hist_vs_posterior(B_df=B_df_new, B_grid= B_grid,
   posterior=normalised_posterior_values)

#calculate mean, variance, standard deviation and confidence interval
sample_mean = np.mean(B_df_new['B'])
sample_variance = np.var(B_df_new['B'])
sample_std = np.std(B_df_new['B'])
lower_CI = sample_mean-(np.sqrt(20)*sample_std)
upper_CI = sample_mean+(np.sqrt(20)*sample_std)

print(f'Sample mean of samples of B: {sample_mean:.4f}')
print(f'Sample variance of samples of B: {sample_variance:.4f}')
print(f'Sample standard deviation of samples of B: {sample_std:.4f}')
print(f'95% confidence interval for B: [{lower_CI:.4f},{upper_CI:.4f}]')
```