Scientific Computing Mini-Project 1

Hugo Bergendahl Hansson, Md Masudul Islam, Mhd Rateb Almajni

February 23, 2024

1 Introduction

This mini-project explores the differences between deterministic and stochastic methods in simulating the circadian clock, a natural internal process that regulates the daily cycles of living organisms.

In simpler terms, deterministic models give the same results every time with the same starting conditions. They help us understand patterns in how biological systems behave, like the daily rhythms in our bodies. For the circadian clock, we focus on two key players: activator protein A, which helps kick-start certain daily processes, and repressor protein R, which slows them down. Together, they create a daily cycle.

However, life is often unpredictable, and this is where stochastic models come in. These models use chance and probability to capture the unexpected and varied outcomes we see in nature. They are especially useful for the tiny, random events that happen inside cells, like how proteins interact.

This project's goal is to recreate findings from a research article and show how both deterministic and stochastic models can help us understand the daily move of these proteins in the circadian clock.

2 Method

To investigate the genetic oscillator, our approach incorporates numerical methods for both deterministic and stochastic simulations. The deterministic analysis uses the `solve_ivp` function from `scipy.integrate`, solving the system's ODEs to observe the time evolution of activator protein A and repressor protein R. For the stochastic simulation, we apply the Gillespie algorithm, generating event times with an exponential distribution and selecting reactions based on their propensity functions.

We utilize the `append` and `copy` methods for efficient data handling during simulations. These methods are designed to manage dynamic array resizing, optimizing the process by reducing the frequency of memory reallocations, thereby enhancing computational efficiency. This allows us to maintain an accurate tracking of the system states over time without incurring significant performance costs.

The combination of `numpy` for numerical operations and `matplotlib.pyplot` for visualization provides a robust framework for analyzing and displaying the model's behavior over a 400-hour period. Through this computational lens, we examine the contrasts between deterministic predictability and the inherent randomness captured by stochastic models, pivotal for understanding biological systems at the molecular level.

We utilized the following discrete Markov process for simulating and visualizing the reactions of different reactants.

$$A + R \xrightarrow{\gamma_C} C$$
$$A \xrightarrow{\delta_A} \emptyset$$
$$C \xrightarrow{\delta_A} R$$

$$R \xrightarrow{\delta_R} \emptyset$$
$$D_A + A \xrightarrow{\gamma_A} D'_A$$
$$D_R + A \xrightarrow{\gamma_R} D'_R$$

$$D'_A \xrightarrow{\theta_A} A + D_A$$
$$D_A \xrightarrow{\alpha_A} D_A + M_A$$
$$D'_A \xrightarrow{\alpha'_A} D'_A + M_A$$
$$M_A \xrightarrow{\delta_{MA}} \emptyset$$
$$M_A \xrightarrow{\beta_A} A + M_A$$
$$D'_R \xrightarrow{\theta_R} A + D_R$$

$$D_R \xrightarrow{\alpha_R} D_R + M_R$$
$$D'_R \xrightarrow{\alpha'_R} D'_R + M_R$$
$$M_R \xrightarrow{\delta_{MR}} \emptyset$$
$$M_R \xrightarrow{\beta_R} M_R + R$$

Furthermore, the rates of the reactants, as presented in the research paper, were as follows.

$$\begin{aligned}
\alpha_A &= 50\,\mathrm{h}^{-1}, & \alpha_{A'} &= 500\,\mathrm{h}^{-1}, & \alpha_R &= 0.01\,\mathrm{h}^{-1},\\
\alpha_{R'} &= 50\,\mathrm{h}^{-1}, & \beta_A &= 50\,\mathrm{h}^{-1}, & \beta_R &= 5\,\mathrm{h}^{-1},\\
\delta_{MA} &= 10\,\mathrm{h}^{-1}, & \delta_{MR} &= 0.5\,\mathrm{h}^{-1}, & \delta_A &= 1\,\mathrm{h}^{-1},\\
\delta_R &= 0.2\,\mathrm{h}^{-1}, & \gamma_A &= 1\,\mathrm{mol}^{-1}\mathrm{hr}^{-1}, & \gamma_R &= 1\,\mathrm{mol}^{-1}\mathrm{hr}^{-1},\\
\gamma_C &= 2\,\mathrm{mol}^{-1}\mathrm{hr}^{-1}, & \theta_A &= 50\,\mathrm{h}^{-1}, & \theta_R &= 100\,\mathrm{h}^{-1}.
\end{aligned}$$

Additionally, the system was subjected to both deterministic and stochastic simulations based on the following implementations.

```
# Deterministic (ODE) solutions
teval = np.linspace(0, final_time, 400)
sol = solve_ivp(ode_function, [0, final_time], initial, t_eval=teval)
```

Figure 1: Implementation of the deterministic simulation

```
# SSA Simulation
times, states = ssa(initial, state_change_matrix, final_time)
```

Figure 2: Implementation of the stochastic simulation

3 Result

In figures 4, 6, and 7, you can see the results derived from the method described in the methods section. These plotted results align perfectly with figures 2a, 2b, 2c, 2d, and 7, which are found in the research paper mentioned in the problem description. Furthermore, figures 5 and 2 showcase the code that was used to generate deterministic and stochastic simulations.

```
# Plotting deterministic solution
fig, axs = plt.subplots(2, 1, figsize=(8, 6))
axs[0].plot(sol.t, sol.y[5], 'b-', label='A')
axs[0].set_ylabel('A (molecules)')
axs[0].set_xlabel('Time (hr)')  # X-axis label for A
axs[0].set_xlim([0, 400])  # Setting x-axis range for A
axs[0].set_title('Deterministic Solution of A and R')
axs[0].legend(loc='upper right')
axs[0].grid(True)

axs[1].plot(sol.t, sol.y[7], 'r-', label='R')
axs[1].set_ylabel('R (molecules)')
axs[1].set_xlabel('Time (hr)')  # X-axis label for R
axs[1].set_xlim([0, 400])  # Setting x-axis range for R
axs[1].legend(loc='upper right')
axs[1].grid(True)

plt.tight_layout()
plt.show()
```

Figure 3: Code for replicating figures 2a and 2b in the research paper

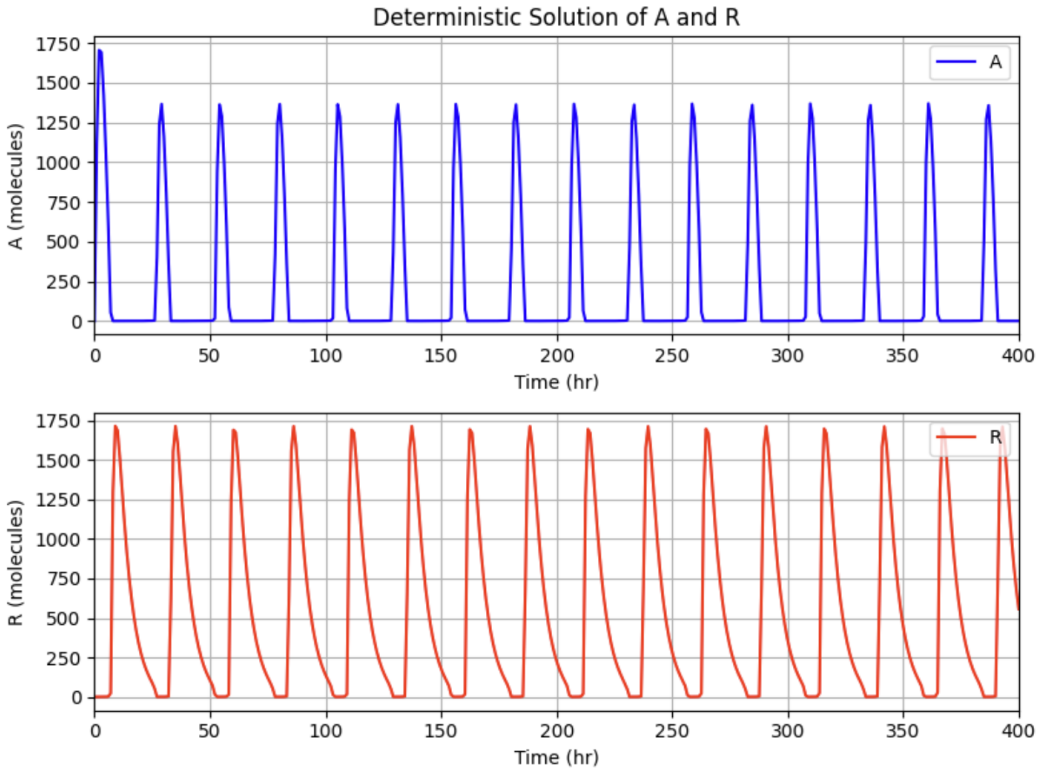

Figure 4: Simulation showing amounts of reactants A and R using a deterministic method

```python
# Plotting SSA solution
fig, axs = plt.subplots(2, 1, figsize=(8, 6))
axs[0].plot(times, states[5, :], 'b-', label='A')
axs[0].set_ylabel('A (molecules)')
axs[0].set_xlabel('Time (hr)')  # X-axis label for A
axs[0].set_xlim([0, 400])  # Setting x-axis range for A
axs[0].set_title('Stochastic Simulation of A and R')
axs[0].legend(loc='upper right')
axs[0].grid(True)

axs[1].plot(times, states[7, :], 'r-', label='R')
axs[1].set_ylabel('R (molecules)')
axs[1].set_xlabel('Time (hr)')  # X-axis label for R
axs[1].set_xlim([0, 400])  # Setting x-axis range for R
axs[1].legend(loc='upper right')
axs[1].grid(True)

plt.tight_layout()
plt.show()
```

Figure 5: Code for replicating figures 2c and 2d in the research paper

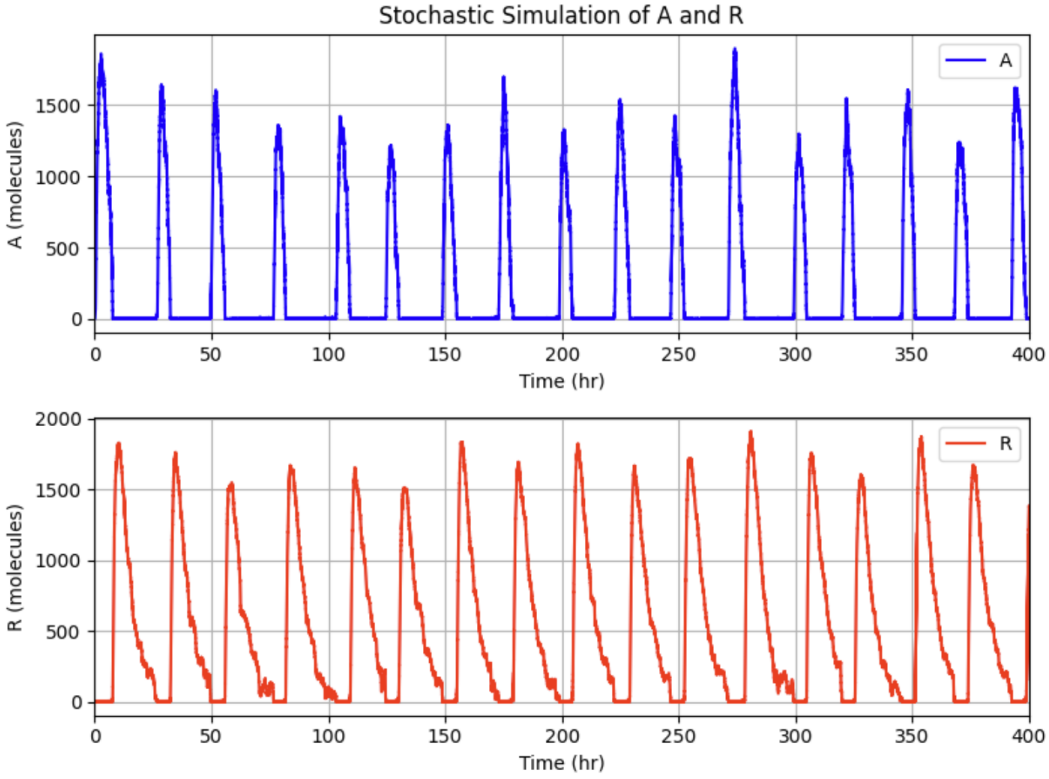

Figure 6: Simulation showing amounts of reactants A and R using a stochastic method

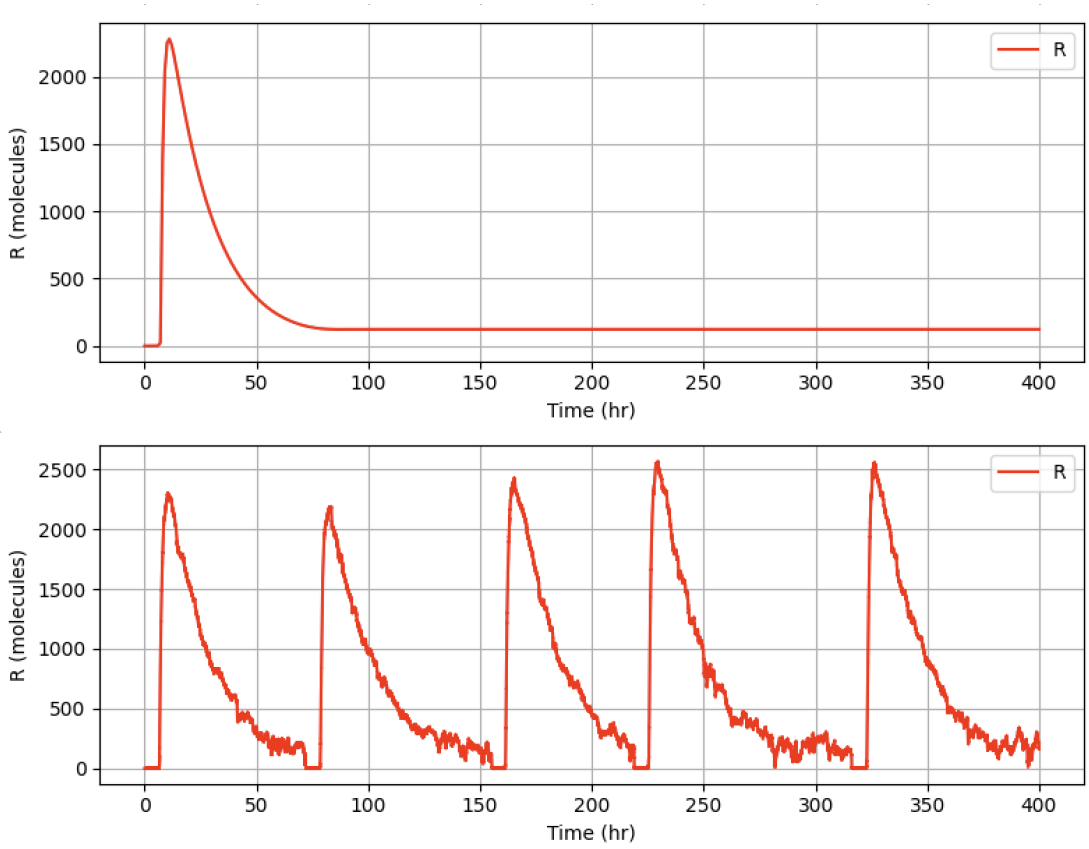

Figure 7: Deterministic and stochastic simulation for reactant R with δR = 0.05h-1

4 Discussion

From the results, both graphs using a deterministic and a stochastic method, resulted in a periodic behavior of the reactants A and R. However, this behavior is expected for this reaction, as seen in the article (1). Comparing the two methods, the deterministic solution (figure 4) has acquired a more precise profile than the stochastic graph (figure 6). This is due to the stochastic method always depending on probabilities. By simulating the stochastic method multiple times, its solution will approach the deterministic solution, known as the Monte-Carlo method. Although a deterministic solution appears more precise, it may not always represent reality when probability is considered, making the stochastic solution a better choice for simulating these types of problems.

Using a deterministic solution can sometimes result in a false representation of the problem. For instance, in the simulation shown in figure 7, the deterministic solution does not exhibit periodic behavior, while the stochastic solution does. This phenomenon occurs when the rate δR is decreased from 0.2 to 0.05, demonstrating that deterministic methods may not always accurately represent reality in such cases. In contrast, stochastic methods are better suited for simulating processes involving variability and uncertainty.

To illustrate the practical application of these theories, consider the use of deterministic methods in engineering disciplines like structural design, where predictable outcomes and precision are paramount. For example, in designing a bridge, deterministic models are often used to ensure stability and safety under specific, predictable loads. On the other hand, stochastic methods find extensive application in fields like finance and economics, where they are used to model stock market fluctuations and risk management, accounting for the inherent unpredictability and variability in these domains.

In summary, we can conclude that the stochastic method is preferred when simulating a system with randomness and probabilities. However, deterministic solutions are more suitable in scenarios

where efficiency or exact outcomes are crucial. The choice between deterministic and stochastic methods depends on the problem's specifics. For well-defined problems with predictable outcomes, deterministic methods are often preferred for simplicity and efficiency. In contrast, stochastic methods are better for problems characterized by uncertainty and variability, offering a way to incorporate these elements. Therefore, the preference for one method over the other mainly depends on the nature of the problem being addressed.

5 References

[1] Vilar, Jose M. G., Kueh, Hao Y., Barkai, Naama, Leibler, Stanislas. (2002). Mechanisms of Noise-Resistance in Genetic Oscillators. *Proceedings of the National Academy of Sciences of the United States of America*, 99, 5988-5992. DOI: 10.1073/pnas.092133899

A Appendix

Option 1 Code

```
# Import necessary libraries
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt

# Function to generate N random numbers from exponential distribution
def random_exponential(lam, N):
    U = np.random.rand(N) # Generate uniform random numbers
    X = -1 / lam * np.log(1 - U) # Convert to exponential distribution
    return X

# Function to generate N random numbers from discrete distribution
def random_discrete(x, p, N):
    cdf = np.cumsum(p) # Compute cumulative distribution function
    U = np.random.rand(N) # Generate uniform random numbers
    idx = np.searchsorted(cdf, U) # Find indices where random numbers fall
    return x[idx]

# Function to calculate propensity of each reaction in the model
def propensity_func(state, num_reactions):
    w = np.zeros(num_reactions)
    DA, DR, DA_prime, DR_prime, MA, A, MR, R, C = state
    # Calculate reaction propensities
    w[0] = gammaC * A * R
    w[1] = deltaA * A
    w[2] = deltaA * C
    w[3] = deltaR * R
    w[4] = gammaA * DA * A
    w[5] = gammaR * DR * A
    w[6] = thetaA * DA_prime
    w[7] = alphaA * DA
    w[8] = alphaA_prime * DA_prime
    w[9] = deltaMA * MA
    w[10] = betaA * MA
    w[11] = thetaR * DR_prime
    w[12] = alphaR * DR
    w[13] = alphaR_prime * DR_prime
    w[14] = deltaMR * MR
    w[15] = betaR * MR
    return w
```

```
# ODE function defining the model's dynamics for deterministic solution
def ode_function(t, y):
    DA, DR, DA_prime, DR_prime, MA, A, MR, R, C = y
    yprime = np.zeros(9)
    yprime[0] = thetaA * DA_prime - gammaA * DA * A
    yprime[1] = thetaR * DR_prime - gammaR * DR * A
    yprime[2] = gammaA * DA * A - thetaA * DA_prime
    yprime[3] = gammaR * DR * A - thetaR * DR_prime
    yprime[4] = alphaA_prime * DA_prime + alphaA * DA - deltaMA * MA
    yprime[5] = betaA * MA + thetaA * DA_prime + thetaR * DR_prime - A * (gammaA * DA
        + gammaR * DR + gammaC * R + deltaA)
    yprime[6] = alphaR_prime * DR_prime + alphaR * DR - deltaMR * MR
    yprime[7] = betaR * MR - gammaC * A * R + deltaA * C - deltaR * R
    yprime[8] = gammaC * A * R - deltaA * C
    return yprime

# Stochastic Simulation Algorithm (SSA)
def ssa(initial, state_change_matrix, final_time):
    m, n = state_change_matrix.shape
    state = initial
    states = [state.copy()]
    t = 0
    times = [t]

    while t < final_time:
        w = propensity_func(state, m)
        a = np.sum(w)
        tau = random_exponential(a, 1)
        t += tau.item()

        if t > final_time:
            break

        which_reaction = random_discrete(np.arange(m), w / a, 1)
        state = state + state_change_matrix[which_reaction.item(), :]
        times.append(t)
        states.append(state.copy())
    # Implement the SSA logic
    return np.array(times), np.array(states).T

# Model parameters and initial conditions
alphaA, alphaA_prime, alphaR, alphaR_prime = 50, 500, 0.01, 50
betaA, betaR = 50, 5
deltaMA, deltaMR, deltaA, deltaR = 10, 0.5, 1, 0.2
# deltaMA, deltaMR, deltaA, deltaR = 10, 0.5, 1, 0.05
gammaA, gammaR, gammaC = 1, 1, 2
thetaA, thetaR = 50, 100
initial = [1, 1, 0, 0, 0, 0, 0, 0, 0]
final_time = 400

state_change_matrix = np.array([
    [0, 0, 0, 0, 0, -1, 0, -1, 1],
    [0, 0, 0, 0, 0, -1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, -1],
    [0, 0, 0, 0, 0, 0, 0, -1, 0],
    [-1, 0, 1, 0, 0, -1, 0, 0, 0],
    [0, -1, 0, 1, 0, -1, 0, 0, 0],
    [1, 0, -1, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, -1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, -1 ,0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, -1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0]
])

# Solve ODEs for deterministic solution
teval = np.linspace(0, final_time, 400)
sol = solve_ivp(ode_function, [0, final_time], initial, t_eval=teval)

# Plotting deterministic solution
fig, axs = plt.subplots(2, 1, figsize=(8, 6))
axs[0].plot(sol.t, sol.y[5], 'b-', label='A')
axs[0].set_ylabel('A (molecules)')
axs[0].set_xlabel('Time (hr)') # X-axis label for A
axs[0].set_xlim([0, 400]) # Setting x-axis range for A
axs[0].set_title('Deterministic Solution of A and R')
axs[0].legend(loc='upper right')
axs[0].grid(True)

axs[1].plot(sol.t, sol.y[7], 'r-', label='R')
axs[1].set_ylabel('R (molecules)')
axs[1].set_xlabel('Time (hr)') # X-axis label for R
axs[1].set_xlim([0, 400]) # Setting x-axis range for R
axs[1].legend(loc='upper right')
axs[1].grid(True)

plt.tight_layout()
plt.show()

# Perform SSA simulation
times, states = ssa(initial, state_change_matrix, final_time)

# Plotting SSA solution
fig, axs = plt.subplots(2, 1, figsize=(8, 6))
axs[0].plot(times, states[5, :], 'b-', label='A')
axs[0].set_ylabel('A (molecules)')
axs[0].set_xlabel('Time (hr)') # X-axis label for A
axs[0].set_xlim([0, 400]) # Setting x-axis range for A
axs[0].set_title('Stochastic Simulation of A and R')
axs[0].legend(loc='upper right')
axs[0].grid(True)

axs[1].plot(times, states[7, :], 'r-', label='R')
axs[1].set_ylabel('R (molecules)')
axs[1].set_xlabel('Time (hr)') # X-axis label for R
axs[1].set_xlim([0, 400]) # Setting x-axis range for R
axs[1].legend(loc='upper right')
axs[1].grid(True)

plt.tight_layout()
plt.show()
```