Introduction to Dynamical Systems in Biology

Dynamical systems are mathematical models that describe how a system changes over time. In biology, they are crucial for understanding complex processes like gene regulation, population dynamics, and cellular metabolism. A key concept in dynamical systems is the steady state, where the system's variables remain constant over time. Mathematically, we can express this as the derivative

$$\frac{dx}{dt} = 0$$

where x represents the system's variables and t is time.

Broadly, although steady-state itself is not a particularly interesting area of study in biological dynamics, it is useful to analyze a biological system "in the neighborhood" of steady-states. Take for instance the simple derivative that describes logistic growth of a bacterial culture:

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right)$$

which describes growth of some species at rate r in a system with finite carrying capacity K. By just looking at this equation, we can see that there are two values of x for which $dx/dt = 0$.

First, is when $x = 0$, such that there is no member of the species to replicate/grow at all. The second is where $x = K$, at which point the carrying capacity has been reached exactly. Neither of these describes quantitative dynamics. However when x *approaches* 0

$$\frac{dx}{dt} \approx rx$$

and when x *approaches* K

$$\frac{dx}{dt} \approx rK(1 - x/K)$$

meaning that the time derivative of x appears to have a linear relationship to x, meaning that we can approximate its behavior by exponential growth/decay. Because of this, linearity – and linear algebra – plays a crucial role in analyzing these systems. We will see this concretely in the next section.

The Lotka-Volterra Model

The Lotka-Volterra model, also known as the predator-prey equations, is a pair of first-order nonlinear differential equations frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. While the model is simple, assuming constant rates of predation and reproduction, it allows us to understand how species interact, and how these interactions can lead to complex dynamics. These dynamics have implications well beyond ecology.

In its simplest form, the Lotka-Volterra model is described by the following system of equations:

$$\begin{aligned} \frac{dx}{dt} &= ax - bxy \\ \frac{dy}{dt} &= -cy + dxy \end{aligned}$$

Here, $x(t)$ represents the prey population and $y(t)$ represents the predator population at time t. The parameters a, b, c, and d are positive real numbers that describe the interaction of the two species:

a: the growth rate of the prey in the absence of predators
b: the rate at which predators eat prey
c: the death rate of predators in the absence of prey
d: the rate at which predators increase by consuming prey

Note that some of the terms in the equations are "first-order", meaning that they contribute a linear term to the time-derivative of the predator or prey, whereas some are "second-order", meaning that the *contribution* of one population explicitly depends on the population of the other population.

The two first-order terms, ax and $-cy$, represent the natural growth of the prey population in the absence of predators and the natural death rate of predators in the absence of prey, respectively. For the two second-order terms, $-bxy$ and dxy, represent the rate at which prey are eaten by predators and the growth of the predator population due to consuming prey, respectively.

The behavior of this system can vary dramatically depending on the values of the parameters and initial conditions. One of the most interesting features of the Lotka-Volterra model is its ability to produce oscillatory behavior under certain conditions. Specifically, when all parameters are positive and non-zero, the system will generally exhibit periodic solutions. These solutions trace closed orbits in the phase plane, representing cyclical fluctuations in both predator and prey populations.

The oscillatory nature of the system can be understood intuitively: as the prey population increases, it provides more food for predators, leading to an increase in the predator population. The increased predator population then consumes more prey, causing the prey population to decrease. With less prey available, the predator population begins to decline, which then allows the prey population to recover, and the cycle continues.

However, not all parameterizations lead to oscillatory behavior. If we set $a = 0$ and $c = 0$, for instance, we get a system where both populations will exponentially decay to zero, regardless of the initial conditions. This represents a scenario where prey have no natural growth and predators have no natural death rate, leading to mutual extinction.

On the other hand, if we set $b = 0$ and $d = 0$, we decouple the equations, resulting in exponential growth for the prey population and exponential decay for the predator population. This could represent a situation where predators are unable to catch prey, leading to unchecked prey growth and predator extinction.

In the following we'll walk through a simulated example of how these dynamics can work using some simple Python scripts.

Python script

We can begin by importing some relevant libraries.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

These Python libraries are as follows: NumPy for numerical computations, Matplotlib for plotting, and SciPy's odeint for solving ordinary differential equations. We can now define a function that will return the time-derivatives of the predator –

```python
def lotka_volterra(state, t, a, b, c, d):
    x, y = state
    dxdt = ax - bxy
    dydt = -cy + dxy
    return [dxdt, dydt]
```

This function defines the Lotka-Volterra equations. It takes the current state (prey and predator populations) and parameters as inputs, and returns the rate of change for both populations.

```python
a, b, c, d = 1, 0.5, 0.75, 0.25
t = np.linspace(0, 100, 1000)
max_x, max_y = 8, 6
```

Here we set the model parameters (a, b, c, d), create a time array from 0 to 100 with 1000 points, and set the maximum values for plotting.

```python
x0, y0 = 3, 4
state0 = [x0, y0]
epsilon = 1E-10
```

These lines set the initial populations for prey and predator, combine them into an initial state vector, and define a small epsilon value to avoid division by zero later.

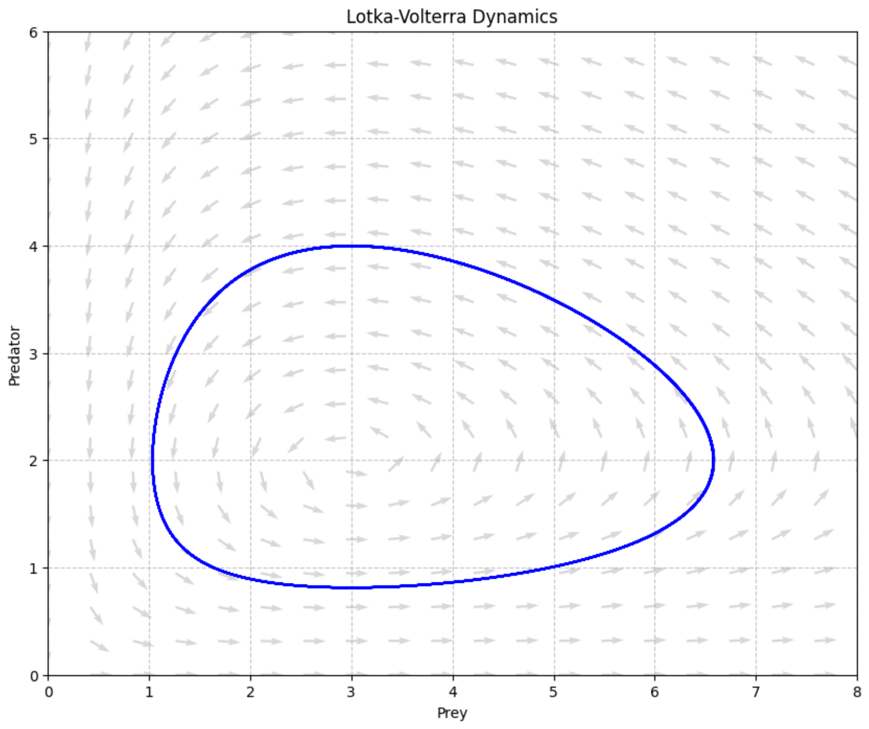

Figure 1: $a, b, c, d = 1, 0.5, 0.75, 0.25$ with stable orbit at $state0 = [3, 4]$.

```python
solution = odeint(lotka_volterra, state0, t, args=(a, b, c, d))
x, y = solution.T
```

Here we use SciPy's odeint to solve the Lotka-Volterra equations over the specified time range, starting from the initial conditions. The solution is then unpacked into separate arrays for prey (x) and predator (y) populations.

```python
X, Y = np.meshgrid(np.linspace(0, max_x, 20), np.linspace(0, max_y, 20))
u, v = lotka_volterra([X, Y], 0, a, b, c, d)
norm = np.sqrt(u2 + v2) + epsilon
u, v = u/norm, v/norm
```

These lines create a grid for the vector field, calculate the direction of change at each point and normalize the vectors.

```python
plt.figure(figsize=(10, 8))
plt.quiver(X, Y, u, v, color='gray', alpha=0.3)
plt.plot(x, y, color='blue')
plt.xlabel('Prey')
plt.ylabel('Predator')
plt.title('Lotka-Volterra Dynamics')
plt.xlim(0, max_x)
plt.ylim(0, max_y)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Finally, we create the plot, showing both the vector field and the solution trajectory (see Fig 1).

Now what happens we change the parameters? Let's say, for example that we increase the predator appetite, so that we simply re-set $a = 0.1$. Then, we get what we see in Fig 2.

Exercise: Describe in detail how and why the shape of the orbit changes due to this specific parameter change.

Exercise: The new orbit is still an orbit. Would this be true in a real population of predator/prey? Why?

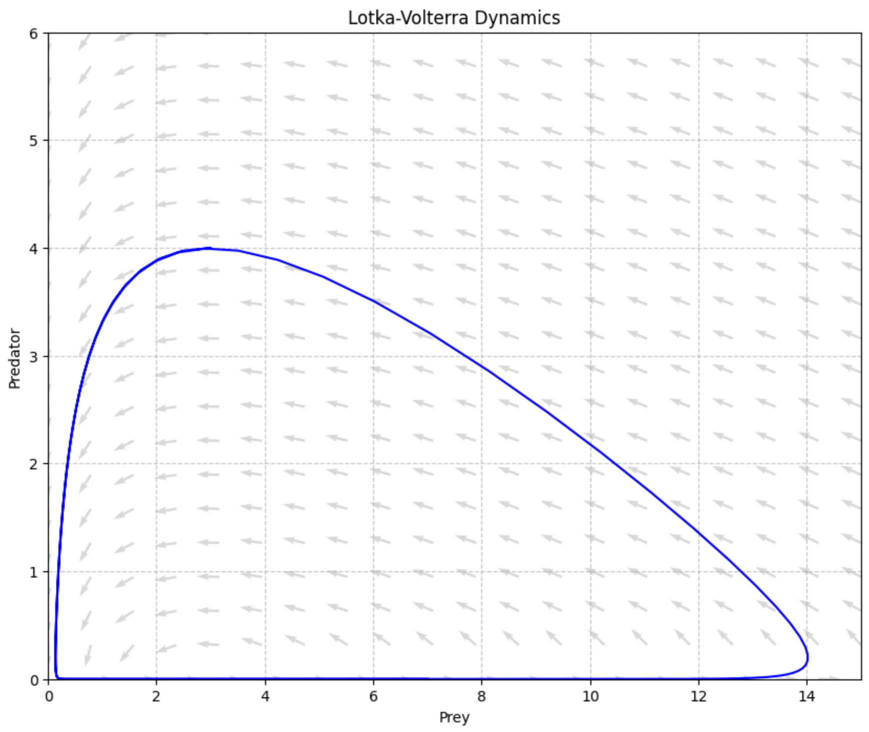

Figure 2: $a, b, c, d = 0.1, 0.5, 0.75, 0.25$ with stable orbit at $state0 = [3, 4]$.

Simplified Repressilator Model

Model Description

The repressilator is an early example of a synthetic genetic regulatory network that exhibits oscillatory behavior.

The simplified repressilator model focuses on the dynamics of three proteins in a cyclic repression network. Each protein represses the production of the next protein in the cycle. The model assumes that mRNA dynamics are fast compared to protein dynamics, allowing us to describe the system using only protein concentrations. The deterministic version of this model can be described by the following system of time-derivatives like the systems we looked at earlier:

$$\begin{aligned}
\frac{dx_1}{dt} &= \frac{\alpha}{1 + x_3^n} + \alpha_0 - \beta x_1 \\
\frac{dx_2}{dt} &= \frac{\alpha}{1 + x_1^n} + \alpha_0 - \beta x_2 \\
\frac{dx_3}{dt} &= \frac{\alpha}{1 + x_2^n} + \alpha_0 - \beta x_3
\end{aligned}$$

Where:

- x_i is the concentration of protein i
- α is the maximum production rate
- α_0 is the basal production rate
- n is the Hill coefficient, describing the cooperativity of repression
- β is the degradation rate

Stochastic simulation

To capture the fact the stochastic counting-error in gene expression, we will use a simplified form of something called the Gillespie algorithm. This algorithm treats each reaction as a discrete event that occurs with a certain probability in a simulated form of continuous time.

```python
def simplified_stochastic_repressilator(initial_state, params, tmax):
    alpha, alpha0, n, beta = params
    t = 0
    state = initial_state.astype(np.float64)
    times = [t]
    states = [state.copy()]
    while t < tmax:
        x1, x2, x3 = state

        # Calculate propensities (vectorized)
        production = alpha / (1 + np.array([x3, x1, x2])**n) + alpha0
        degradation = beta * state

        propensities = np.concatenate([production, degradation])

        a0 = np.sum(propensities)

        # Time to next reaction
        tau = np.random.exponential(1/a0)

        # Choose next reaction
        reaction = np.random.choice(6, p=propensities/a0)

        # Update state (vectorized)
        state_change = np.zeros(3, dtype=np.float64)
        if reaction < 3:
            state_change[reaction] = 1  # protein production
        else:
            state_change[reaction - 3] = -1  # protein degradation

        state += state_change
        t += tau
        times.append(t)
        states.append(state.copy())

    return np.array(times), np.array(states)
```

Simulation and Visualization

We'll set parameters as follows:

```python
# Parameters
alpha = 100  # Maximum production rate
alpha0 = 0.01    # Basal production rate
n = 4         # Hill coefficient
beta = 1      # Degradation rate
params = [alpha, alpha0, n, beta]
```

Running the simulation as follows generates Fig 3:

```python
# Initial conditions (x1, x2, x3)
initial_state = np.array([50, 50, 50], dtype=np.float64)

# Run simulation
tmax = 100
times, states = simplified_stochastic_repressilator(initial_state, params, tmax)

# Set up the figure and axes
plt.rcParams.update({'font.size': 12})  # Set global font size to 12
fig = plt.figure(figsize=(18, 6))

```

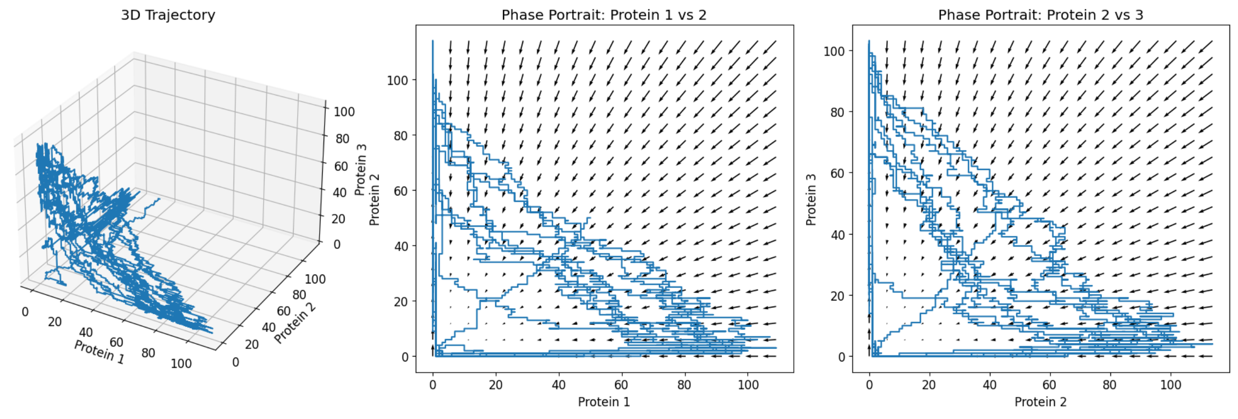

Figure 3: Repressilator simulation.

```python
# 3D trajectory plot
ax1 = fig.add_subplot(131, projection='3d')
ax1.plot(states[:, 0], states[:, 1], states[:, 2])
ax1.set_xlabel('Protein 1')
ax1.set_ylabel('Protein 2')
ax1.set_zlabel('Protein 3')
ax1.set_title('3D Trajectory')

# Function to calculate the derivative for quiver plots
def repressilator_derivative(X, Y, protein_idx):
    Z = np.mean(states[:, (protein_idx+1)%3])  # Use mean of the third protein
    dX = alpha / (1 + (Z**n)) + alpha0 - beta * X
    dY = alpha / (1 + (X**n)) + alpha0 - beta * Y
    return dX, dY

# 2D quiver plot (Protein 1 vs Protein 2)
ax2 = fig.add_subplot(132)
X, Y = np.meshgrid(np.linspace(0, np.max(states[:, 0]), 20),
                   np.linspace(0, np.max(states[:, 1]), 20))
dX, dY = repressilator_derivative(X, Y, 0)
ax2.quiver(X, Y, dX, dY)
ax2.plot(states[:, 0], states[:, 1])
ax2.set_xlabel('Protein 1')
ax2.set_ylabel('Protein 2')
ax2.set_title('Phase Portrait: Protein 1 vs 2')

# 2D quiver plot (Protein 2 vs Protein 3)
ax3 = fig.add_subplot(133)
X, Y = np.meshgrid(np.linspace(0, np.max(states[:, 1]), 20),
                   np.linspace(0, np.max(states[:, 2]), 20))
dX, dY = repressilator_derivative(X, Y, 1)
ax3.quiver(X, Y, dX, dY)
ax3.plot(states[:, 1], states[:, 2])
ax3.set_xlabel('Protein 2')
ax3.set_ylabel('Protein 3')
ax3.set_title('Phase Portrait: Protein 2 vs 3')

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

Exercise: Try running the simulation with $n = 1$, such that there is no cooperativity. Is there still oscillatory behavior? Why is cooperativity necessary/unnecessary?
Exercise: Let's assume that the trajectories in Fig 3 show the distribution of cells in gene expression space. Frequently, PCA – which highlights the dimensions in gene expression space that preserve the greatest total variance – is used to reduce the dimensionality of such a scenario. How might one expect for PCA to fail in this case?