

MA5710 - Assignment 5

Abhinav T K, MA23M002

November 15, 2023

1. Using Matlab's **ode45 solver** or equivalent function in python, evaluate trajectories of $N(t)$ and $P(t)$ for Lotka-Volterra Model(**with and without fishing**) taking $(N(0), P(0)) = (10, 5)$ as initial condition and setting the parameters as $a = 4$, $b = 2$, $c = 1.5$, $d = 3$ and for **with fishing model** set δ as 0.2

Code: **q1_withoutfishing.m**

```
1 % Q1 ode45 solver for Lotka-Volterra Model - without fishing
2 clc;
3 clear;
4
5 %Initial condition
6 y0 = [10;5]; % (Initial Population of prey & predator respectively)
7 % Parameters
8 a = 4; b = 2; c = 1.5; d = 3;
9 soln = ode45(@lotka,[0 20], y0); % Solution of ode
10 t = linspace(0,10,20);
11 y(:,1) = deval(soln,t,1); % Prey population
12 y(:,2) = deval(soln,t,2); % Predator population
13 % Equilibrium point
14 eq = [d/c; a/b];
15
16 % Plots
17 figure;
18 plot(t,y(:,1),'-o', 'LineWidth',1.5);
19 grid on;
20 hold on;
21 plot(t,y(:,2),'-o', 'LineWidth',2);
22 legend('Prey','Predator');
23 xlabel('Time');
24 ylabel('Population');
25 title('Predator/Prey Populations Over Time without Fishing')
26 hold off;
27
28
29 % Phase plot
30 [t1, y1] = ode45(@lotka,[0 20], y0);
31 figure;
32 plot(y1(:,1),y1(:,2))
33 hold on;
34 title('Phase portrait without Fishing')
35 xlabel('Prey Population')
36 ylabel('Predator Population')
37 plot(eq(1), eq(2), 'ro', 'MarkerFaceColor', 'red', 'MarkerSize', 5); % Eq. point
38 label = sprintf('(%1f, %1f)', eq(1), eq(2)); % Eq. point label
39 text(eq(1), eq(2), label, 'VerticalAlignment', 'bottom', 'HorizontalAlignment', '
    right');
40 hold off;
```

```

41 legend('Phase portrait', 'Equilibrium Point'); % Legend
42
43 % Lotka-Volterra (Predator-prey) function
44 function lv = lotka(t,x)
45 lv = [0;0];
46 % Parameters
47 a = 4; b = 2; c = 1.5; d = 3;
48 lv(1) = a*x(1) - b*x(1)*x(2); % Rate of change of prey population
49 lv(2) = c*x(1)*x(2) - d*x(2); % Rate of change of predator population
50 end

```

Output:

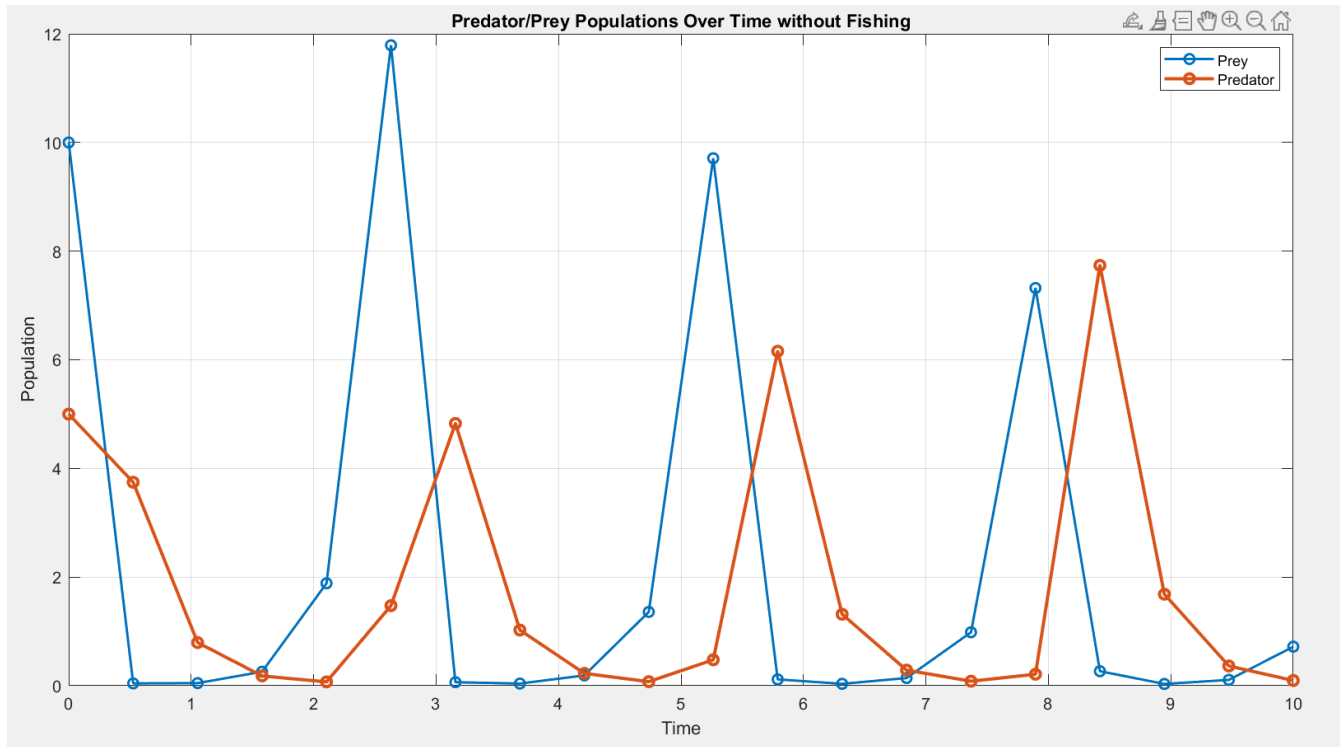


Figure 1: Predator-prey population over time (without fishing)

Code: `q1_withfishing.m`

```

1 % Q1 ode45 solver for Lotka-Volterra Model - with fishing
2 clc;
3 clear;
4
5 %Initial condition
6 y0 = [10;5]; % (Initial Population of prey & predator respectively)
7 % Parameters
8 a = 4; b = 2; c = 1.5; d = 3; delta = 0.2;
9
10 soln = ode45(@lotka,[0 20], y0); % Solution of ode
11 t = linspace(0,10,20);
12 y(:,1) = deval(soln,t,1); % Prey population
13 y(:,2) = deval(soln,t,2); % Predator population
14 % Equilibrium point
15 eq = [d+delta/c; a-delta/b];

```

```

16
17 % Plots
18 figure;
19 plot(t,y(:,1),'-o', 'LineWidth',1.5);
20 grid on;
21 hold on;
22 plot(t,y(:,2),'-o', 'LineWidth',2);
23 legend('Prey','Predator');
24 xlabel('Time');
25 ylabel('Population');
26 title('Predator/Prey Populations Over Time with Fishing')
27 hold off;
28
29 % Phase plot
30 [t1, y1] = ode45(@lotka,[0 20], y0);
31 figure;
32 plot(y1(:,1),y1(:,2));
33 hold on;
34 title('Phase portrait with Fishing');
35 xlabel('Prey Population');
36 ylabel('Predator Population');
37 plot(eq(1), eq(2), 'ro', 'MarkerFaceColor', 'red', 'MarkerSize', 5); % Eq. point
38 label = sprintf('(%0.1f, %0.1f)', eq(1), eq(2)); % Eq. point label
39 text(eq(1), eq(2), label, 'VerticalAlignment', 'bottom', 'HorizontalAlignment', '
    right');
40 hold off;
41 legend('Phase portrait', 'Equilibrium Point'); % Legend
42
43 % Lotka-Volterra (Predator-prey) function
44 function lv = lotka(t,x)
45     lv = [0;0];
46     % Parameters
47     a = 4; b = 2; c = 1.5; d = 3; delta = 0.2;
48     lv(1) = a*x(1) - b*x(1)*x(2) - delta*x(1); % Rate of change of prey population
49     lv(2) = c*x(1)*x(2) - d*x(2) - delta*x(2); % Rate of change of predator
        population
50 end

```

Output:

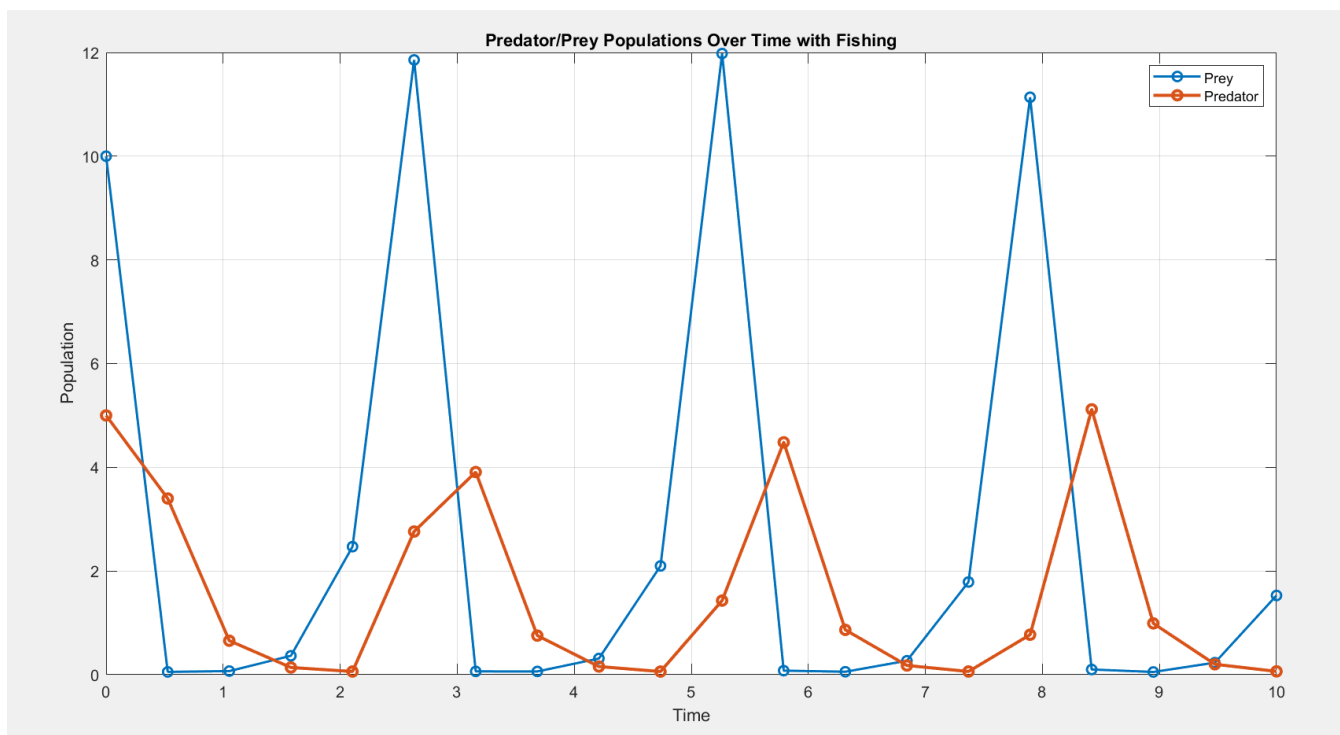


Figure 2: Predator-prey population over time (with fishing)

2. Plot the Phase portrait for both cases of Q1

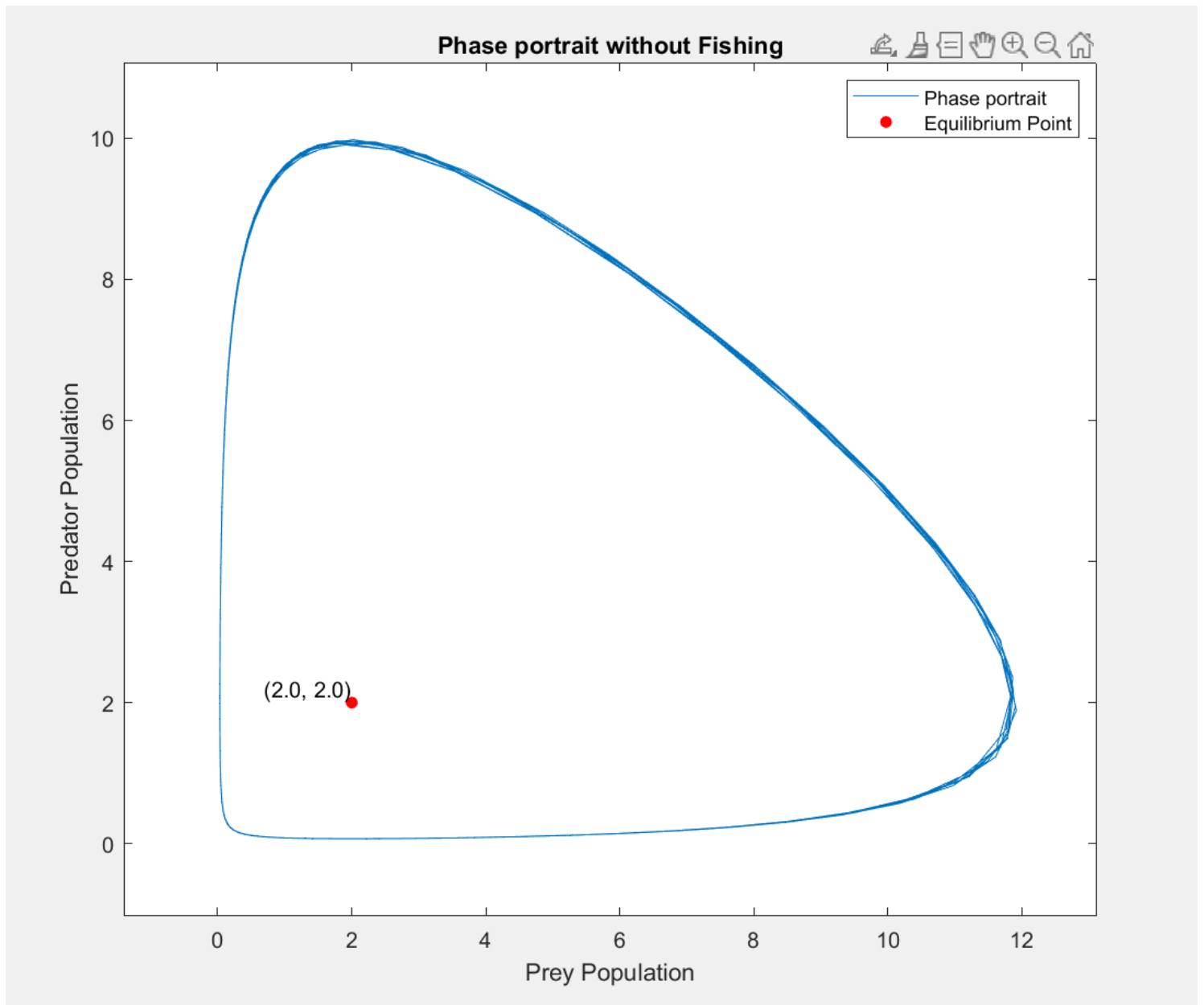


Figure 3: Phase-plot: without fishing

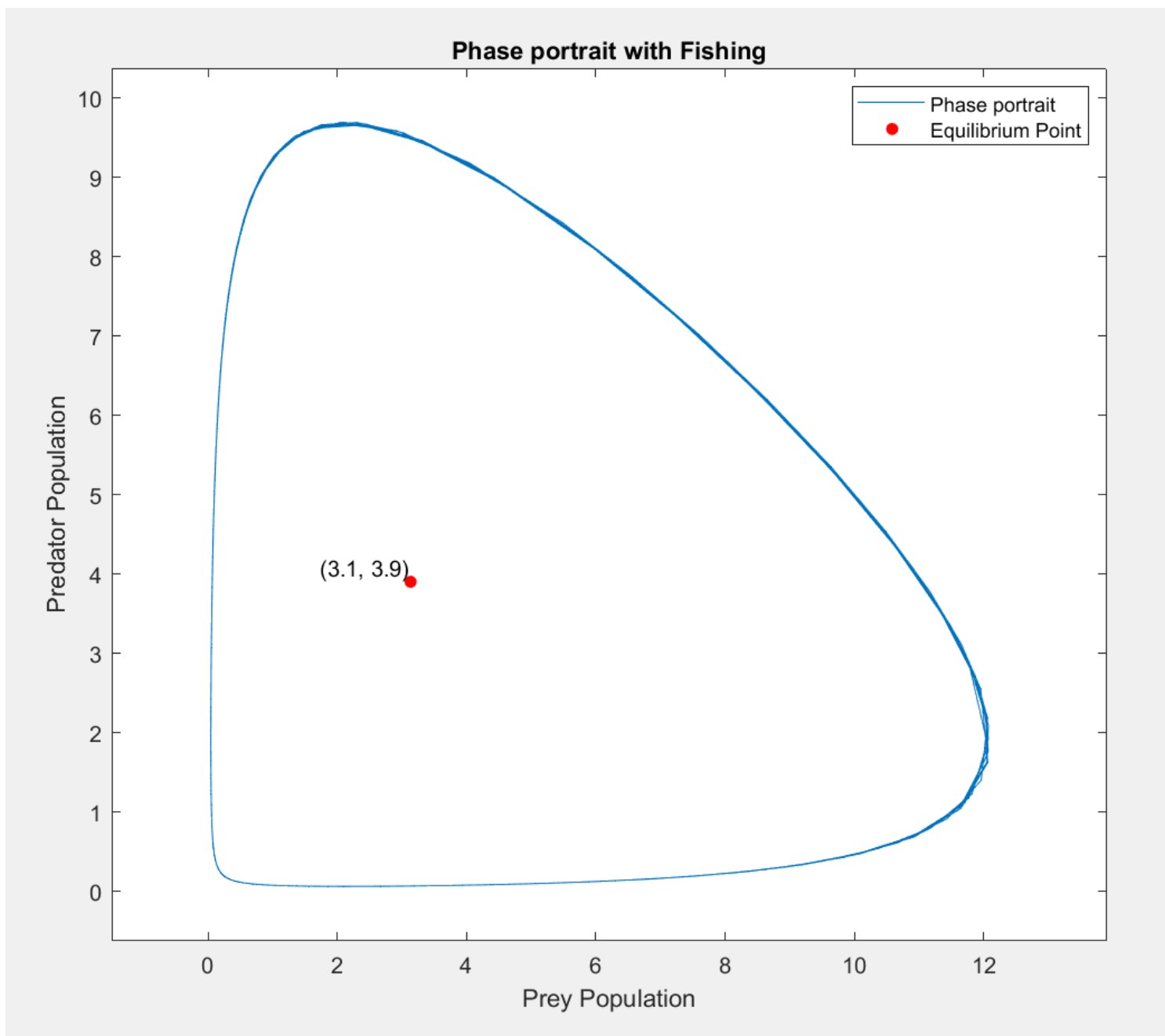


Figure 4: Phase-plot: with fishing

3. Linearize the Lotka Volterra Model(with and without fishing) and solve it by applying ode solver as in Q1. Evaluate the trajectories and draw respective phase portraits.

```

1  % Q3. Linearized Lotka-Volterra (Predator-prey) model (without fishing)
2  clear;
3  clc;
4
5  %Initial condition
6  y0 = [10;5];      % (Initial Population of prey & predator respectively)
7  % Parameters
8  a = 4; b = 2; c = 1.5; d = 3;
9  % Equilibrium point
10 eq = [d/c; a/b];
11 % Solve the linearized system using ODE solver
12 soln = ode45(@lotka_linearize, [0 20], y0);
13 t = linspace(0,10,10);
14 y(:,1) = deval(soln,t,1);    % Prey population
15 y(:,2) = deval(soln,t,2);    % Predator population
16
17 % Plots
18 figure;
19 plot(t,y(:,1),'-o', 'LineWidth',1.5);
20 hold on;
21 plot(t,y(:,2),'-o', 'LineWidth',2);
22 xlabel('Time');
23 ylabel('Population');
24 title('Linearized Lotka-Volterra Model (without fishing)');
25 legend('Prey', 'Predator');
26 grid on;
27 hold off;
28
29 % Phase portrait
30 [t1, y1] = ode45(@lotka_linearize, [0 20], y0);
31 figure;
32 plot(y1(:, 1), y1(:, 2));
33 hold on;
34 title('Phase Portrait - Linearized Lotka-Volterra Model (without fishing)');
35 xlabel('Prey Population');
36 ylabel('Predator Population');
37 grid on;
38 plot(eq(1), eq(2), 'ro', 'MarkerFaceColor', 'red', 'MarkerSize', 5); % Eq. point
39 label = sprintf('(%1f, %1f)', eq(1), eq(2)); % Eq. point label
40 text(eq(1), eq(2), label, 'VerticalAlignment', 'bottom', 'HorizontalAlignment', '
    right');
41 hold off;
42 legend('Phase portrait', 'Equilibrium Point'); % Legend
43
44 function lv_linearized = lotka_linearize(t, x)
45     % Parameters
46     a = 4; b = 2; c = 1.5; d = 3;
47     % Equilibrium point
48     eq = [d/c; a/b];
49     x0 = eq(1);      % Equilibrium point for prey
50     y0 = eq(2);      % Equilibrium point for predator
51
52     % Jacobian matrix at the equilibrium point
53     J = [a - b * y0, -b * x0; c * y0, c * x0 - d];
54
55     % Linearized system

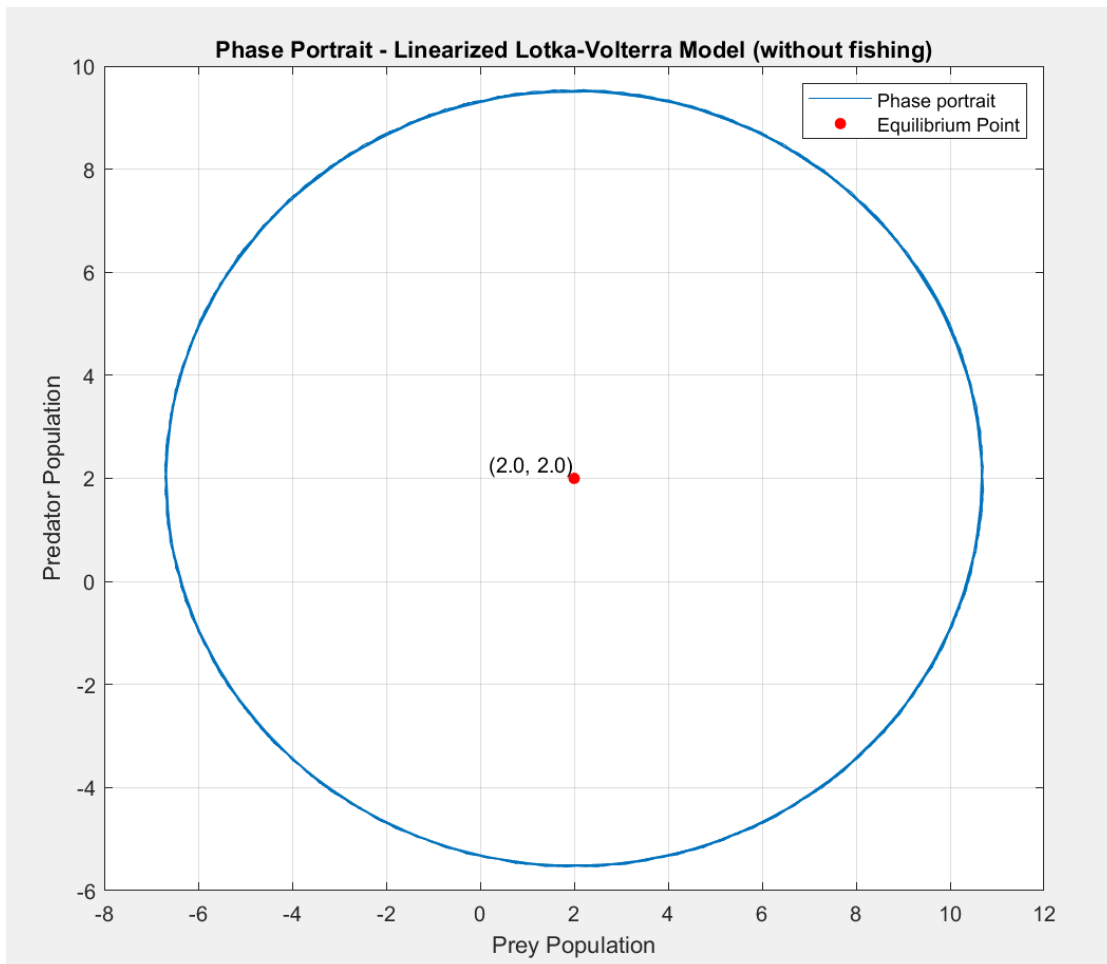
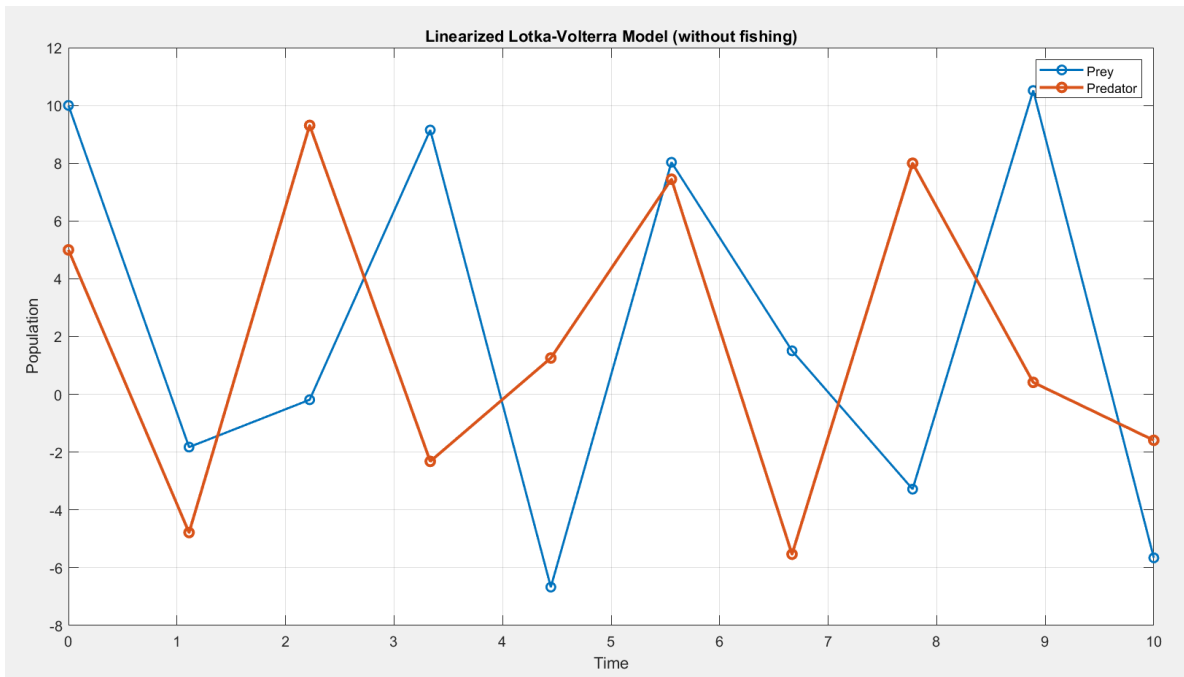
```

```

56 lv_linearized = J * (x - [x0; y0]);
57 end

```

Output:

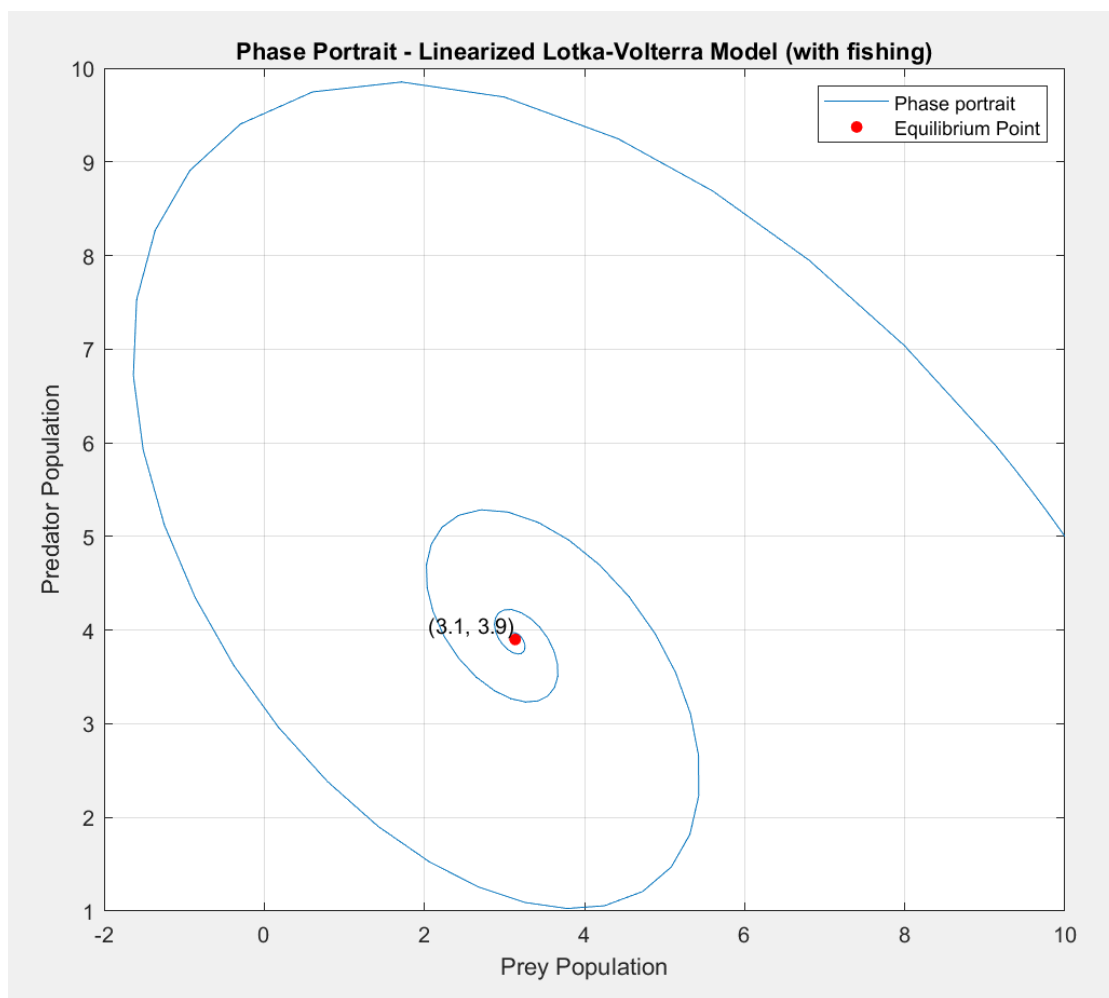
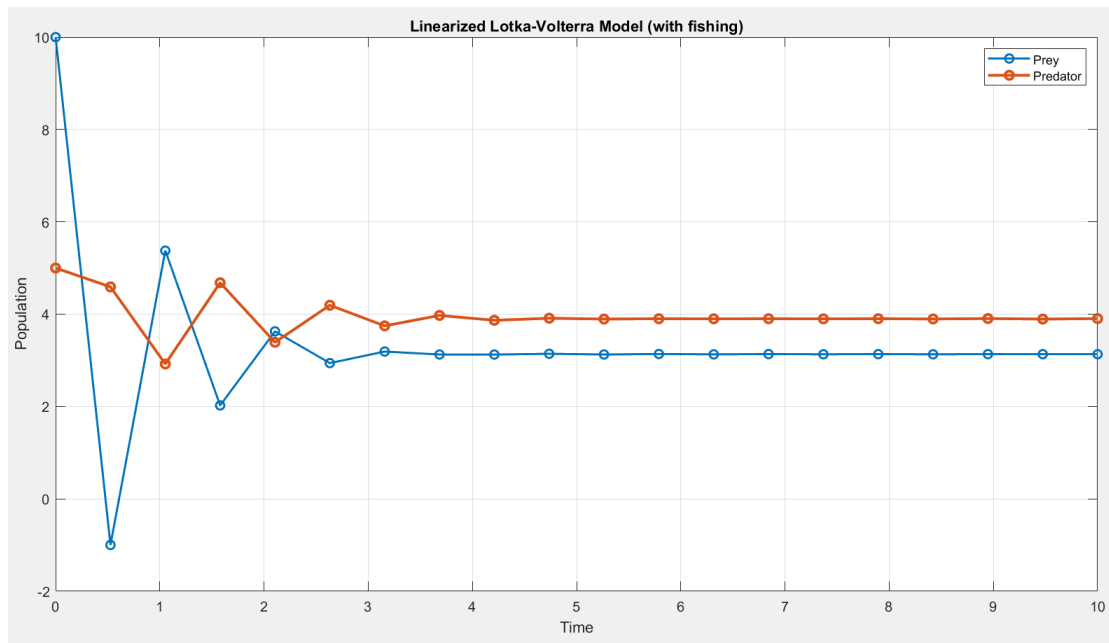



```

1      % Q3. Linearized Lotka-Volterra (Predator-prey) model (with fishing)
2  clear;
3  clc;
4
5  %Initial condition
6  y0 = [10;5];      % (Initial Population of prey & predator respectively)
7  % Parameters
8  a = 4; b = 2; c = 1.5; d = 3; delta = 0.2;
9  % Equilibrium point
10 eq = [d+delta/c; a-delta/b];
11 % Solve the linearized system using ODE solver
12 soln = ode45(@lotka_linearize, [0 20], y0);
13 t = linspace(0,10,20);
14 y(:,1) = deval(soln,t,1);    % Prey population
15 y(:,2) = deval(soln,t,2);    % Predator population
16
17 % Plots
18 figure;
19 plot(t,y(:,1),'-o', 'LineWidth',1.5);
20 grid on;
21 hold on;
22 plot(t,y(:,2),'-o', 'LineWidth',2);
23 xlabel('Time');
24 ylabel('Population');
25 title('Linearized Lotka-Volterra Model (with fishing)');
26 legend('Prey', 'Predator');
27 hold off;
28
29 % Phase portrait
30 [t1, y1] = ode45(@lotka_linearize, [0 20], y0);
31 figure;
32 plot(y1(:, 1), y1(:, 2));
33 hold on;
34 title('Phase Portrait - Linearized Lotka-Volterra Model (with fishing)');
35 xlabel('Prey Population');
36 ylabel('Predator Population');
37 grid on;
38 plot(eq(1), eq(2), 'ro', 'MarkerFaceColor', 'red', 'MarkerSize', 5); % Eq. point
39 label = sprintf('(%0.1f, %0.1f)', eq(1), eq(2)); % Eq. point label
40 text(eq(1), eq(2), label, 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right');
41 hold off;
42 legend('Phase portrait', 'Equilibrium Point'); % Legend
43
44 function lv_linearized = lotka_linearize(t, x)
45     % Parameters
46     a = 4; b = 2; c = 1.5; d = 3; delta = 0.2;
47     % Equilibrium point
48     eq = [d+delta/c; a-delta/b];
49     x0 = eq(1);      % Equilibrium point for prey
50     y0 = eq(2);      % Equilibrium point for predator
51
52     % Jacobian at the equilibrium point
53     J = [a - b * y0 - delta, -b * x0; c * y0, c * x0 - d - delta];
54
55     % Linearized system
56     lv_linearized = J * (x - [x0; y0]);
57 end

```

Output:



4. This question requires use of **symbolic computation**. Look for **sympy** module in python or equivalent computational toolbox in Matlab. Consider the **Four species Model**. It has already been derived in the slides and also in the reference. Using symbolic computation, derive the equilibrium points of the model and classify those points by figuring out respective eigen values and eigen vectors of the linearized system. Please note that, deriving equilibrium points, computing eigen values and eigen vectors of the linearized system, the whole computation process has to be done either in matlab or in python, using the module for symbolic computation.

Python code for finding eq point and finding eigen values and eigen vectors

```
from sympy import symbols, Eq, solve, Matrix, diff

# Define the variables and parameters
n1, n2, n3, n4 = symbols('n1 n2 n3 n4')
k, c1, c2, c3, c4 = symbols('k c1 c2 c3 c4')
d1, d2, d3, d4 = symbols('d1 d2 d3 d4')
e1, e2, e3 = symbols('e1 e2 e3')
f2 = symbols('f2')

# Define the equations for Four-Species Model
n1_dash = c1 * n1 * (1 - n1 / k) - d1 * n1 * n2 - e1 * n1 * n3
n2_dash = c2 * n1 * n2 - d2 * n2 + e2 * n2 * n4 - f2 * n2 * n3
n3_dash = c3 * n3 * n1 + d3 * n3 * n2 - e3 * n3
n4_dash = c4 * n2 * n4 - d4 * n4

# Find equilibrium points by solving the system of equations
equilibrium_points = solve((n1_dash, n2_dash, n3_dash, n4_dash), (n1, n2, n3, n4))

# Display equilibrium points
print("Equilibrium Points:")
for point in equilibrium_points:
    print(point)

# Define the Jacobian matrix
variables = [n1, n2, n3, n4]
equations = [n1_dash, n2_dash, n3_dash, n4_dash]
Jacobian_matrix = Matrix([[diff(eq, var) for var in variables] for eq in equations])

# Compute the Jacobian matrix at each equilibrium point
for point in equilibrium_points:
    J_point = Jacobian_matrix.subs({n1: point[0], n2: point[1], n3: point[2], n4: point[3]})
    eigenvalues = J_point.eigenvals()
    eigenvectors = J_point.eigenvecs()

    print("\nEquilibrium Point:", point)
    print("Eigenvalues:")
    for eigenvalue in eigenvalues:
        print(eigenvalue)
    print("\nEigenvectors:")
    for eigenvector in eigenvectors:
        print(eigenvector)
```

Output: The execution of the above code to find equilibrium points was not possible in local machine or Google colab as the code didn't stop running.

Equilibrium point 1 (k,0,0,0)

```
# Eq point 1: (k,0,0,0)
from sympy import symbols, Eq, solve, Matrix, diff

# Define the variables and parameters
n1, n2, n3, n4 = symbols('n1 n2 n3 n4')
k, c1, c2, c3, c4 = symbols('k c1 c2 c3 c4')
d1, d2, d3, d4 = symbols('d1 d2 d3 d4')
e1, e2, e3 = symbols('e1 e2 e3')
f2 = symbols('f2')

# Define the equations for Four-Species Model
n1_dash = Eq(c1 * n1 * (1 - n1 / k) - d1 * n1 * n2 - e1 * n1 * n3, 0)
n2_dash = Eq(c2 * n1 * n2 - d2 * n2 + e2 * n2 * n4 - f2 * n2 * n3, 0)
n3_dash = Eq(c3 * n3 * n1 + d3 * n3 * n2 - e3 * n3, 0)
```

```

n4_dash = Eq(c4 * n2 * n4 - d4 * n4, 0)

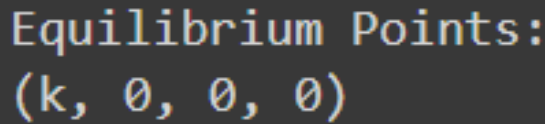
# Substitute numerical values into the equations
n1_dash = n1_dash.subs({n2:0, n3:0})
n2_dash = n2_dash.subs({n2:0, n3:0, n4:0})
n3_dash = n3_dash.subs({n2:0, n3:0})
n4_dash = n3_dash.subs({n2:0, n4:0})

# Find equilibrium points by solving the system of equations
equilibrium_points = solve((n1_dash, n2_dash, n3_dash, n4_dash), (n1, n2, n3, n4))

# Display equilibrium points
print("Equilibrium Points:")
print(equilibrium_points[0])

```

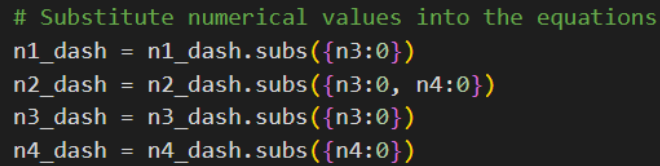
Output:



Equilibrium Points:
(k, 0, 0, 0)

Equilibrium point (k,0,0,0) is unstable.

Equilibrium point 2 (n1,n2,0,0)



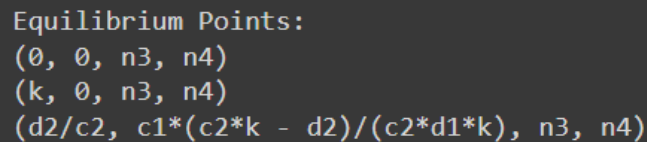
```

# Substitute numerical values into the equations
n1_dash = n1_dash.subs({n3:0})
n2_dash = n2_dash.subs({n3:0, n4:0})
n3_dash = n3_dash.subs({n3:0})
n4_dash = n4_dash.subs({n4:0})

```

Figure 5: Change from above python code

Output:



Equilibrium Points:
(0, 0, n3, n4)
(k, 0, n3, n4)
(d2/c2, c1*(c2*k - d2)/(c2*d1*k), n3, n4)

Figure 6: Equilibrium point when $n3 = 0$ & $n4 = 0$

Equilibrium point 2 (n1,n2,0,0) is locally asymptotically stable point if $n1n2\epsilon < (n1 + n2)\mu$

Equilibrium point 3 (n1,n2,0,n3)

```
# Substitute numerical values into the equations
n1_dash = n1_dash.subs({n3:0})
n2_dash = n2_dash.subs({n3:0})
n3_dash = n3_dash.subs({n3:0})
n4_dash = n4_dash.subs({n3:0})
```

Figure 7: Change from above python code

Output:

```
Equilibrium Points:
(0, 0, n3, 0)
(0, d4/c4, n3, d2/e2)
(k, 0, n3, 0)
(d2/c2, c1*(c2*k - d2)/(c2*d1*k), n3, 0)
(k*(c1*c4 - d1*d4)/(c1*c4), d4/c4, n3, (-c1*c2*c4*k + c1*c4*d2 + c2*d1*d4*k)/(c1*c4*e2))
```

Figure 8: Equilibrium points when $n3 = 0$

5. Consider the **Infectious disease model**. The model is derived and the equilibrium points are evaluated in the slide provided. Verify the equilibrium points using symbolic computation. Derive the conditions for equilibrium points to be **asymptotically stable**. Verify your calculation through symbolic computation.

Equilibrium point for healthy state:

```
# Infectious model
# Equilibrium point (healthy state) when Veq = 0, Meq = 0, Ceq = Cstar
from sympy import symbols, Eq, solve

# Define the variables
# F: Antibody, V: Antigen, C: Plasma cells, M: Organ
F, V, C, M = symbols('F V C M', real=True, positive=True)
alpha, beta, gamma, mu, sigma, eta = symbols('alpha beta gamma mu sigma eta', real=True, positive=True)
a, p, k = symbols('a p k', real=True, positive=True)
Cstar = symbols('Cstar', real=True, positive=True)

# Define the equations
V_dash = Eq(alpha * V - p * V * F, 0)
F_dash = Eq(beta * C - gamma * p * V * F - a * F, 0)
C_dash = Eq(-mu * (C - Cstar) + k * V * F, 0)
M_dash = Eq(alpha * V - eta * M, 0)

# Substitute specific values in the equations
V_dash = V_dash.subs({V: 0})
M_dash = M_dash.subs({M: 0})
C_dash = C_dash.subs({C: Cstar, V: 0})
F_dash = F_dash.subs({C: Cstar, V: 0})
# Solve the equation for F directly to find its equilibrium point
equilibrium_F = solve(F_dash, F)

print("Equilibrium Points:")
print("V: ", 0)
print("F: ", equilibrium_F[0])
print("C: ", Cstar)
print("M: ", 0)
```

```
Equilibrium Points:
V:  0
F:  Cstar*beta/a
C:  Cstar
M:  0
```

Conditions for equilibrium points to be asymptotically stable:

The equilibrium point for healthy state is

$$V = 0, F = \frac{C^* \beta}{a} = F^*, C = C^*, M = 0$$

It is asymptotically stable if $\alpha < pF^*$

At eq. point if the inequality (infected zone of attraction to healthy state)

$$0 < V^0 < V^* = \frac{a(pF^* - \alpha)}{\alpha \gamma p}$$

This estimate V^* is the **immunological barrier** against the germs. If germs cannot get over ($V^0 < V^*$), disease cannot occur as the germ population is removed from the body during the time.

Check for asymptotic stability

```

# Check for asymptotic stability
from sympy import Matrix, diff
# Equations
V_dash = Eq(alpha * V - p * V * F, 0)
F_dash = Eq(beta * C - gamma * p * V * F - a * F, 0)
C_dash = Eq(-mu * (C - Cstar) + k * V * F, 0)
M_dash = Eq(alpha * V - eta * M, 0)
# Jacobian matrix
Jacobian_mat = Matrix([
    [V_dash.lhs.diff(V), V_dash.lhs.diff(F), V_dash.lhs.diff(C), V_dash.lhs.diff(M)],
    [F_dash.lhs.diff(V), F_dash.lhs.diff(F), F_dash.lhs.diff(C), F_dash.lhs.diff(M)],
    [C_dash.lhs.diff(V), C_dash.lhs.diff(F), C_dash.lhs.diff(C), C_dash.lhs.diff(M)],
    [M_dash.lhs.diff(V), M_dash.lhs.diff(F), M_dash.lhs.diff(C), M_dash.lhs.diff(M)],
])
print(Jacobian_mat)
# Substitute the equilibrium values
equilibrium_values = {F: equilibrium_F[0], V: 0, C: Cstar, M: 0}
Jacobian_matrix_eq = Jacobian_mat.subs(equilibrium_values)
print(Jacobian_matrix_eq)
# Compute the eigenvalues
eigenvalues = Jacobian_matrix_eq.eigenvals()
print("\nEigenvalues:")
print(list(eigenvalues.keys()))

```

Output:

```

Jacobian_mat

$$\begin{bmatrix} -Fp + \alpha & -Vp & 0 & 0 \\ -F\gamma p & -V\gamma p - a & \beta & 0 \\ Fk & Vk & -\mu & 0 \\ \alpha & 0 & 0 & -\eta \end{bmatrix}$$


Jacobian_matrix_eq

$$\begin{bmatrix} -\frac{Cstar\beta p}{a} + \alpha & 0 & 0 & 0 \\ -\frac{Cstar\beta\gamma p}{a} & -a & \beta & 0 \\ \frac{Cstar\beta k}{a} & 0 & -\mu & 0 \\ \alpha & 0 & 0 & -\eta \end{bmatrix}$$


Eigenvalues:
[-(Cstar*beta*p - a*alpha)/a, -mu, -a, -eta]

It is asymptotically stable if  $\beta < \gamma F^*$ 

```

Equilibrium point for chronic disease state:

```

# Infectious model
# Equilibrium point (chronic disease state)
from sympy import symbols, Eq, solve

# Define the variables
F, V, C, M = symbols('F V C M', real=True, positive=True)
alpha, beta, gamma, mu, sigma, eta = symbols('alpha beta gamma mu sigma eta', real=True, positive=True)
a, p, k = symbols('a p k', real=True, positive=True)
Cstar = symbols('Cstar', real=True, positive=True)

# Define the equations
V_dash = Eq(alpha * V - p * V * F, 0)
F_dash = Eq(beta * C - gamma * p * V * F - a * F, 0)
C_dash = Eq(-mu * (C - Cstar) + k * V * F, 0)
M_dash = Eq(alpha * V - eta * M, 0)

# Find equilibrium points by solving the system of equations

```

```
equilibrium_points2 = solve((V_dash, F_dash, C_dash, M_dash), (V, F, C, M))

print("Equilibrium Points:")
print(equilibrium_points2)
```

Output:

```
Equilibrium Points:
[(mu*(Cstar*beta*p - a*alpha)/(alpha*(-beta*k + gamma*mu*p)),
 alpha/p,
 (Cstar*gamma*mu*p**2 - a*alpha*k)/(p*(-beta*k + gamma*mu*p)),
 mu*(Cstar*beta*p - a*alpha)/(eta*(-beta*k + gamma*mu*p)))]
```

Conditions for equilibrium points to be asymptotically stable:

The above equilibrium point is asymptotically stable when:

$$\mu_c \tau \leq 1, \quad 0 < \frac{f-d}{a-g-f\tau} < b-g-f\tau$$

where:

$$\begin{aligned}\alpha &= \mu_c + \mu_f + \eta\gamma V_2 \\ b &= \mu_c(\eta\gamma V_2 + \mu_f) - \eta\gamma\beta V_2 \\ d &= \mu_c\eta\gamma\beta V_2 \\ g &= \alpha\rho V_2 \\ f &= \beta\alpha\rho V_2\end{aligned}$$

For highly sensitive immune response $\alpha \rightarrow \infty$.

In this case a stable chronic disease state is possible.

Then we have the condition:

$$0 < \beta - \gamma F^* < \left[\tau + \frac{1}{\mu_c + \mu_f} \right]^{-1}$$

Check for asymptotic stability

```
# Check for asymptotic stability
from sympy import Matrix, diff
# Equations
V_dash = Eq(alpha * V - p * V * F, 0)
F_dash = Eq(beta * C - gamma * p * V * F - a * F, 0)
C_dash = Eq(-mu * (C - Cstar) + k * V * F, 0)
M_dash = Eq(alpha * V - eta * M, 0)
# Jacobian matrix
Jacobian_mat = Matrix([
    [V_dash.lhs.diff(V), V_dash.lhs.diff(F), V_dash.lhs.diff(C), V_dash.lhs.diff(M)],
    [F_dash.lhs.diff(V), F_dash.lhs.diff(F), F_dash.lhs.diff(C), F_dash.lhs.diff(M)],
    [C_dash.lhs.diff(V), C_dash.lhs.diff(F), C_dash.lhs.diff(C), C_dash.lhs.diff(M)],
    [M_dash.lhs.diff(V), M_dash.lhs.diff(F), M_dash.lhs.diff(C), M_dash.lhs.diff(M)],
])
print(Jacobian_mat)
# Substitute the equilibrium values
equilibrium_values2 = {V: equilibrium_points2[0][0], F: equilibrium_points2[0][1], C:
                        equilibrium_points2[0][2], M: equilibrium_points2
                        [0][3]}
Jacobian_matrix_eq2 = Jacobian_mat.subs(equilibrium_values2)
print(Jacobian_matrix_eq2)
# Compute the eigenvalues
eigenvalues2 = Jacobian_matrix_eq2.eigenvals()
print("\nEigenvalues:")
A = list(eigenvalues2.keys())
```

Output:

Jacobian_mat

$$\begin{bmatrix} -Fp + \alpha & -Vp & 0 & 0 \\ -F\gamma p & -V\gamma p - a & \beta & 0 \\ Fk & Vk & -\mu & 0 \\ \alpha & 0 & 0 & -\eta \end{bmatrix}$$

Jacobian_matrix_eq2

$$\begin{bmatrix} 0 & -\frac{\mu p(Cstar\beta p - a\alpha)}{\alpha(-\beta k + \gamma\mu p)} & 0 & 0 \\ -\alpha\gamma & -a - \frac{\gamma\mu p(Cstar\beta p - a\alpha)}{\alpha(-\beta k + \gamma\mu p)} & \beta & 0 \\ \frac{\alpha k}{p} & \frac{k\mu(Cstar\beta p - a\alpha)}{\alpha(-\beta k + \gamma\mu p)} & -\mu & 0 \\ \alpha & 0 & 0 & -\eta \end{bmatrix}$$

6. In Q5 system model, initial conditions are $V(0) = V_0 > 0, F(0) = F^*, C(0) = C^*, m(0) = 0$. The immunological barrier is defined as $V^* = a(pF^* - \alpha)/\alpha\gamma p$. Choose the parameter values and initial condition such that $\alpha < pF^*$. This is **subclinical form** of the disease. Solve the system of ode's for the chosen values and plot $V(t)$ when, $V_0 < V^*$ and $V_0 > V^*$. Conclude how the plot differs for these two subcases.
- Next, Choose the parameter values and initial condition such that $\alpha > pF^*$. This is called **Acute form** of the disease. For this case immunological barrier does not exist. Plot $V(t)$ when $k\beta > \mu\gamma p$ and $k\beta < \mu\gamma p$. The first subcase denotes normal immune response to acute disease thus leading to recovery and the second subcase denotes immunodeficiency response, thus leading to more severe form of the disease. Also plot $V(t)$ when value of σ is gradually increased. See whether with increased value of sigma, you are reaching the **chronic state** or not.

Case 1: Subclinical form

$$\alpha < pF^*$$

Case 1.a: $V_0 < V^*$

Parameter values:

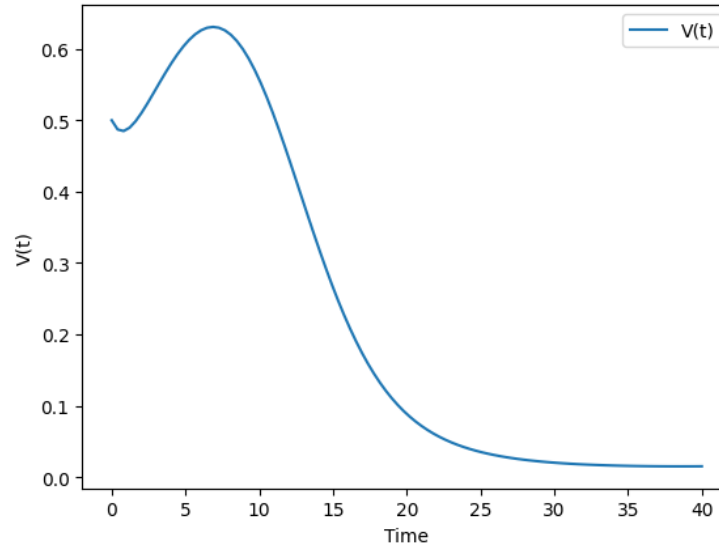
$$\alpha = 0.3$$

$$p = 0.8$$

$$F^* = 0.5$$

$$V_0 = 0.5$$

$$V^* = 0.9$$



Case 1: Subclinical form

$$\alpha < pF^*$$

Case 1.b: $V_0 > V^*$

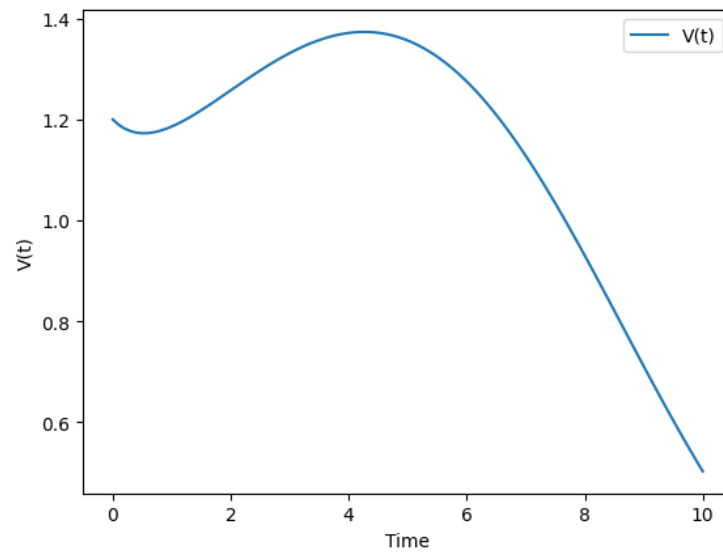
$$\alpha = 0.3$$

$$p = 0.8$$

$$F^* = 0.5$$

$$V_0 = 1.2$$

$$V^* = 0.9$$



Case 2: Acute form

$$\alpha > pF^*$$

Case 2.a: $k\beta > \mu\gamma p$

Parameter values:

$$\alpha = 0.6$$

$$p = 0.8$$

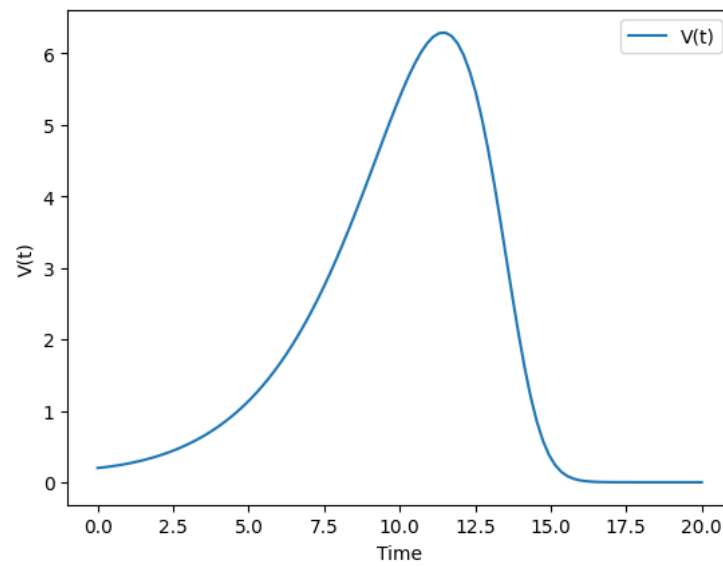
$$F^* = 0.5$$

$$k = 0.9$$

$$\beta = 0.2$$

$$\mu = 0.1$$

$$\gamma = 0.3$$



Case 2: Acute form

$$\alpha > pF^*$$

Case 2.b: $k\beta < \mu\gamma p$

Parameter values:

$\alpha = 0.6$

$p = 0.8$

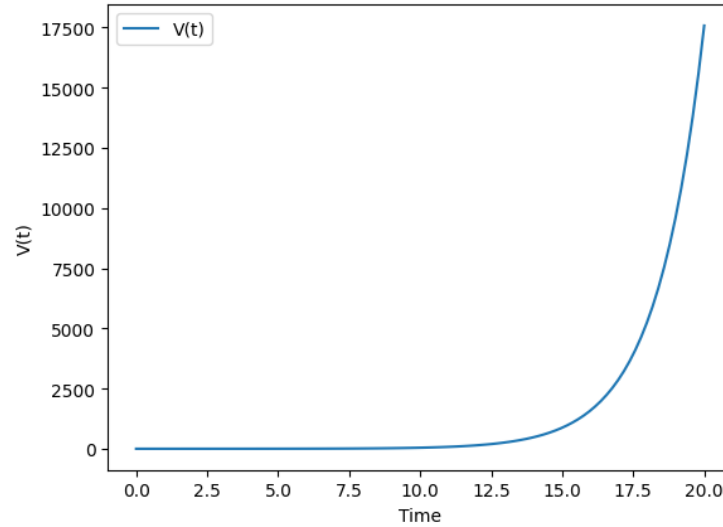
$F^* = 0.5$

$k = 0.8$

$\beta = 0.02$

$\mu = 0.1$

$\gamma = 0.3$

**Generic python code for the above 4 cases.**

(Change parameter values on case by case basis.)

```

# Infectious model
# Case 1: Subclinical form ( $\alpha < pF^*$ ) ( $0.3 < 0.4$ )
# 1.a:  $V_0 < V^*$  ( $0.5 < 0.9$ )
# 1.b:  $V_0 > V^*$  ( $1.2 > 0.9$ )
#  $V^* = a(pF^* - \alpha)/\alpha\gamma p = 0.9$ 

# Case 2: Acute: ( $\alpha > pF^*$ ) ( $0.6 > 0.4$ )
# 2.a  $k\beta > \mu\gamma p$  ( $0.18 > .024$ )
# 2.b  $k\beta < \mu\gamma p$  ( $0.016 < .024$ )

from sympy import symbols, Eq, solve

# Variables
F, V, C, M = symbols('F V C M', real=True, positive=True)
alpha, beta, gamma, mu, sigma, eta = symbols('alpha beta gamma mu sigma eta', real=True, positive=True)
a, p, k = symbols('a p k', real=True, positive=True)
Cstar = symbols('Cstar', real=True, positive=True)

# Equations
V_dash = alpha * V - p * V * F
F_dash = beta * C - gamma * p * V * F - a * F
C_dash = -mu * (C - Cstar) + k * V * F
M_dash = alpha * V - eta * M

# Parameter values
# Case 2:
#  $\alpha > pF^*$  ( $0.6 > 0.4$ )
# Case 2.b  $k\beta < \mu\gamma p$  ( $0.016 < .024$ )

alpha_val = 0.6
beta_val = 0.02
gamma_val = 0.3
mu_val = 0.1
sigma_val = 0.4

```

```

eta_val = 0.6
a_val = 0.7
p_val = 0.8
k_val = 0.8
Cstar_val = 1.0

# Initial conditions
V0_val = 0.2 # Choose V0 < V*
Fstar_val = 0.5 # Initial condition for F
Cstar_init_val = 0.8 # Initial condition for C
M0_val = 0.0 # Initial condition for M

# Substituting parameter values and initial conditions
V_dash_sub = V_dash.subs({alpha: alpha_val, p: p_val, F: Fstar_val})
F_dash_sub = F_dash.subs({beta: beta_val, gamma: gamma_val, a: a_val, p: p_val, C: Cstar_init_val, F:
                          Fstar_val})
C_dash_sub = C_dash.subs({mu: mu_val, Cstar: Cstar_val, k: k_val, V: V0_val, F: Fstar_val})
M_dash_sub = M_dash.subs({alpha: alpha_val, eta: eta_val, V: V0_val})

# Defining system of ODEs and solving
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def model(y, t):
    V, F, C, M = y
    dV = alpha_val * V - p_val * V * F
    dF = beta_val * C - gamma_val * p_val * V * F - a_val * F
    dC = -mu_val * (C - Cstar_val) + k_val * V * F
    dM = alpha_val * V - eta_val * M
    return [dV, dF, dC, dM]

# Initial conditions
initial_conditions = [V0_val, Fstar_val, Cstar_init_val, M0_val]

# Time points
t = np.linspace(0, 20, 100)

# Solve the system of ODEs
soln = odeint(model, initial_conditions, t)

# Extract the solutions for each variable
V_solution, F_solution, C_solution, M_solution = soln.T

# Plot V(t)
plt.plot(t, V_solution, label='V(t)')
plt.xlabel('Time')
plt.ylabel('V(t)')
plt.legend()
plt.show()

```

Plot $V(t)$ when value of σ is gradually increased.

```

# Chronic state
from sympy import symbols
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Variables
F, V, C, M = symbols('F V C M', real=True, positive=True)
alpha, beta, gamma, mu, sigma, eta = symbols('alpha beta gamma mu sigma eta', real=True, positive=True)
a, p, k = symbols('a p k', real=True, positive=True)
Cstar = symbols('Cstar', real=True, positive=True)

# Equations
V_dash = alpha * V - p * V * F
F_dash = beta * C - gamma * p * V * F - a * F
C_dash = -mu * (C - Cstar) + k * V * F
M_dash = alpha * V - eta * M

```

```

# Chosen parameter values
alpha_val = 0.3 # alpha < pF*
beta_val = 0.2
gamma_val = 0.3
mu_val = 0.1
eta_val = 0.6
a_val = 0.7
p_val = 0.8
k_val = 0.9
Cstar_val = 1.0

# Initial conditions
V0_val = 0.2 # Choose V0 < V*
Fstar_val = 0.5 # Initial condition for F
Cstar_init_val = 0.8 # Initial condition for C
M0_val = 0.0 # Initial condition for M

# Time points
t = np.linspace(0, 300, 100)

# Increase in sigma with time
def sigma_function(t):
    return 0.5 + 0.03 * t

# Plot V(t) for sigma as a function of time
# Define the ODE system with sigma as a function of time
def model(y, t):
    V, F, C, M = y
    dV = alpha_val * V - p_val * V * F
    dF = beta_val * C - gamma_val * p_val * V * F - a_val * F
    dC = -mu_val * (C - Cstar_val) + k_val * V * F
    dM = alpha_val * V - eta_val * M - sigma_function(t) * M
    return [dV, dF, dC, dM]

# Solve the system of ODEs
soln = odeint(model, initial_conditions, t)

# Extract the solutions for each variable
V_solution, F_solution, C_solution, M_solution = soln.T

# Plot V(t) with sigma as a function of time
plt.plot(t, V_solution, label='V(t)')
plt.xlabel('Time')
plt.ylabel('V(t)')
plt.legend()
plt.title('Effect of Sigma on V(t) - Chronic state')
plt.show()

```

Output:

