Numerical Methods for Dynamical Systems
Assignment 3
UPC, FME

Written by
Ole Gunnar Røsholt Hovland

2023

Abstract

Assignment 3 delves into the computational aspects of dynamical systems. In Part A, we focus on the Poincare section y=0 within the harmonic oscillator problem, examining various crossings and initial conditions. Part B shifts to the integration of a 2D linear system, exploring different parameterized systems and their phase spaces. We'll also analyze eigenvalues, eigenvectors, and system manifolds. Part C and D are theoretical sections centered on the Pendulum and Lotka-Volterra models respectively. All findings, codes, and answers are below.

Note: Writing everything in a latex document took much longer than expected this time. Therefore the pdf looks a little rushed, and rough around the edges.

Contents

1 Introduction

Some liberties have been taken in the code presentation, so the lines are short enough. If needed, don't hesitate to ask for the raw jupyter notebook file.

2 Part A

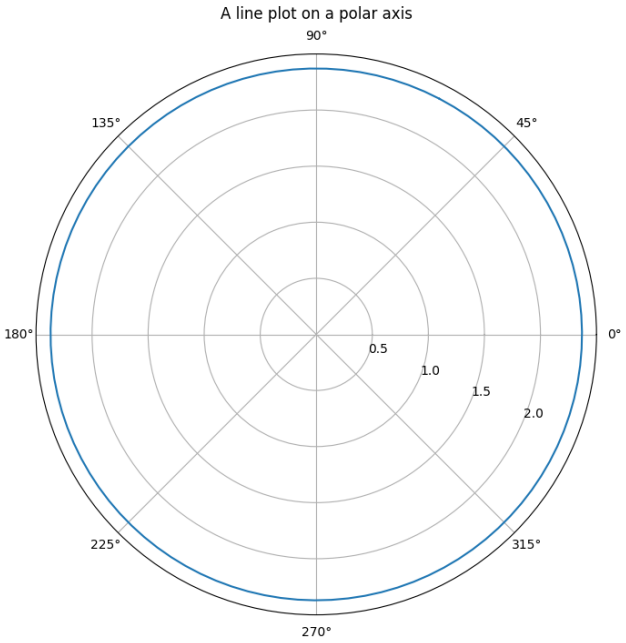
2.1 Poincare section of harmonic oscillator

Implementing function PoincareMap computes the Poincaré map for a harmonic oscillator. Beginning by creating the orbits of the system. A more in depth plot is found later in the text.

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

def f(t, x):
    # x is a vector of size n
    return x[1], -x[0]

x0 = [1.1, 2.1]
t_max = 2*np.pi

# solve the system of ODEs
t_eval = np.linspace(0, t_max, 100)
sol = sp.integrate.solve_ivp(f, [0, t_max],
          x0, method='DOP853', t_eval=t_eval,
          atol=1e-16, rtol=1e-16)

print(np.linalg.norm(sol.y[:, -1] - x0))
print(sol.y[:, -1], x0)

r = np.sqrt(sol.y[0]**2 + sol.y[1]**2)
theta = np.arctan2(sol.y[1], sol.y[0])

# Making the plot look nice
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='polar')
ax.plot(theta, r)
ax.set_rmax(2.5)
ax.set_rticks([0.5, 1, 1.5, 2])  # less radial ticks
ax.set_rlabel_position(-22.5)  # get radial labels away from plotted line
ax.grid(True)
ax.set_title("A line plot on a polar axis", va='bottom')
plt.show()
```

Now creating the function to solve the system

```
import numpy as np
import scipy as sp
```




```
3
4  def poincare_map_solve_ivp(harmonic_oscillator,
5                             initial_conditions, dir, step, t_span):
6      # Tolerance settings for the ODE solver
7      abs_tol = 1e-13
8      rel_tol = 1e-13
9
10     # Procedure to compute when the x-axis is crossed
11     # for the first time
12     product = 1
13     time = 0
14     startPoint = np.array(initial_conditions)
15     startPoint = startPoint
16
17     while product >= 0:
18         solution = sp.integrate.solve_ivp(harmonic_oscillator, t_span,
19                                           initial_conditions, method='RK45',
20                                              rtol=rel_tol,atol=abs_tol,
21                                           t_eval=np.linspace(t_span[0],
22                                                              t_span[1], 1000))
23         Y = solution.y.T  # Transposing to match previous structure
24         product = Y[1, 1] * Y[-1, 1] # in order to enshure crossing
25         #the x-axis, and getting the right sign, dont start at the first
26         #index in the list
27         initial_conditions = [Y[-1, 0], Y[-1, 1]]
28         t_span = [t_span[0]  + dir * step, t_span[1] + dir * step]
29         #t_span = [t_span[0], t_span[1] + dir * step]
30         time += step*dir
31
32     # Procedure to compute the exact time of the crossing
33     for i in range(100):
```

```
        t_eval = np.linspace(0, time, 1000)
        solution = sp.integrate.solve_ivp(harmonic_oscillator, [0, time],
                                          startPoint, method='RK45',
                                          rtol=rel_tol, atol=abs_tol,
                                          t_eval=t_eval)
        Y = solution.y.T
        scalar_product = -Y[-1, 0]
        difference = Y[-1, 1] / scalar_product
        time -= difference
        if abs(difference) < 1e-12:
            break

    TimeDuration = time
    newInitial = [Y[-1, 0], Y[-1, 1]]

    return newInitial, TimeDuration
```

Defining the system, initially values, and a loop that calculates time for a chosen n-crossings

```
def harmonic_oscillator(t, X):
    #print(f"t: {t}, Y: {Y}")
    return [X[1], -X[0]]

# Initial conditions: xval=0, yval=1
initial_conditions = [0, 1]
overallTime = 0

# Set other parameters
dir = 1
step = 0.3
t_span = [0, dir * step]

# Number of crossings and overall time initialization
numberOfCrossings = 6

# Compute the PoincareMap three times
for i in range(numberOfCrossings):
    newInitial, timeDuration = poincare_map_solve_ivp(harmonic_oscillator,
                                                      initial_conditions,
                                                      dir, step, t_span)
    initial_conditions = newInitial
    overallTime += timeDuration
    # Print overall time pretty
    print("The overall time is:", overallTime,
          " The error is:", 0.5 * np.pi + i*np.pi - overallTime)
```

The output of the code is

```
The overall time is: 1.5707963267948954  The error is: 1.1102230246251565e-15
The overall time is: 4.712388980384686  The error is: 3.552713678800501e-15
```

```
The overall time is: 7.853981633974477  The error is: 6.217248937900877e-15
The overall time is: 10.995574287564267  The error is: 8.881784197001252e-15
The overall time is: 14.137166941154057  The error is: 1.2434497875801753e-14
The overall time is: 17.278759594743846  The error is: 1.4210854715202004e-14
```

We can see that the more crossings we do, the time computed is slightly more inaccurate.

2.2 Different initial conditions

Here is some testing with different starting conditions. All had number of crossings set to 2.

```python
def harmonic_oscillator(t, X):
    return [X[1], -X[0]]

# Initial conditions: 90 degrees, 0 degrees and 60 degrees
initial_conditions_lst = [[0, 1], [1, 0], [1/2, np.sqrt(3)/2]]
expected_time_lst = [0.5 * np.pi, np.pi, np.pi/3]

# Set other parameters
dir = 1
step = 0.3
t_span = [0, dir * step]

# Number of crossings and overall time initialization
numberOfCrossings = 2

# Compute the PoincareMap three times
for i in range(len(initial_conditions_lst)):
    initial_conditions = initial_conditions_lst[i]
    overallTime = 0
    for j in range(numberOfCrossings):
        newInitial, timeDuration =
        poincare_map_solve_ivp(harmonic_oscillator,
                               initial_conditions,
                               dir, step, t_span)
        initial_conditions = newInitial
        overallTime += timeDuration
        # Print overall time pretty
        print(f"The time for initial conditions
        {initial_conditions_lst[i]} is {overallTime}
        \n and the error is {expected_time_lst[I]
        + j*np.pi - overallTime}")
```

```
The time for initial conditions [0, 1] is 1.5707963267948954
 and the error is 1.1102230246251565e-15
The time for initial conditions [0, 1] is 4.712388980384686
 and the error is 3.552713678800501e-15
The time for initial conditions [1, 0] is 3.1415926535897905
 and the error is 2.6645352591003757e-15
The time for initial conditions [1, 0] is 6.283185307179581
```

```
 and the error is 5.329070518200751e-15
The time for initial conditions [0.5, 0.8660254037844386] is 1.0471975511965972
 and the error is 4.440892098500626e-16
The time for initial conditions [0.5, 0.8660254037844386] is 4.188790204786388
 and the error is 2.6645352591003757e-15
```

We see that the code also handles various starting points.

2.3 The orbit

Checking that the orbit is well computed by plotting the orbit in (x,y) coordinates. In order to do this, we had to create a new function, for cleaner code. We made the code general enough that it computes orbits anywhere on the vector field.

```python
import matplotlib.pyplot as plt
from tqdm import tqdm # for debugging

number_of_points = 5
xval = np.linspace(-10, 10,number_of_points)
yval = np.linspace(-3,3, number_of_points)

# define the harmonic oscillator
def harmonic_oscillator(t, Y):
    return [Y[1], -np.sin(Y[0])]

# Time span for the integration
t_span = [0, 10]

# Tolerance settings for the ODE solver
abs_tol = 1e-13
rel_tol = 1e-13

# To color the arrows according to the velocity
"""cmap = plt.cm.jet
from matplotlib import colors
norm = colors.Normalize(vmin=0, vmax=1)"""


# Compute the orbit for each initial condition
for x0 in tqdm(xval):
    for y0 in yval:
        initial_conditions = [x0, y0]
        t = np.linspace(t_span[0], t_span[1], 100)
        Y = sp.integrate.solve_ivp(harmonic_oscillator,
                                   t_span, initial_conditions,
                                   method='RK45', rtol=rel_tol,
                                   atol=abs_tol, t_eval=t).y.T
        # Map the orbit to -pi < x < pi
        Y[:, 0] = np.mod(Y[:, 0] + np.pi, 2 * np.pi) - np.pi
        # Plot the orbit
```

```
        plt.plot(Y[:, 0], Y[:, 1], 'b.', zorder=1)

        ### Plot the direction of the flow at each point on
        ### the line x = 0

        # Find the point closest to x = 0
        idx_closest = np.argmin(np.abs(Y[:, 0]))
        # Evaluate the differential equation at that point to get the
        # direction
        dy = harmonic_oscillator(0, Y[idx_closest])
        # Normalize the direction for better visualization
        norm = np.linalg.norm(dy)
        if norm != 0:
            dy = [d/norm for d in dy]
        # Color the arrows according to the velocity
        #color = cmap(norm)
        # Plot the direction at the point closest to x = 0
        plt.quiver(Y[idx_closest, 0], Y[idx_closest, 1], dy[0],
                   dy[1], angles='xy', scale_units='xy', scale=1,
                   color="red", zorder=2)

# Making the plot look nice
plt.xlabel('x')
plt.ylabel('y')
plt.xlim([-np.pi, np.pi])
plt.title('Orbits of the harmonic oscillator')
plt.show()
```

The plot from the code is

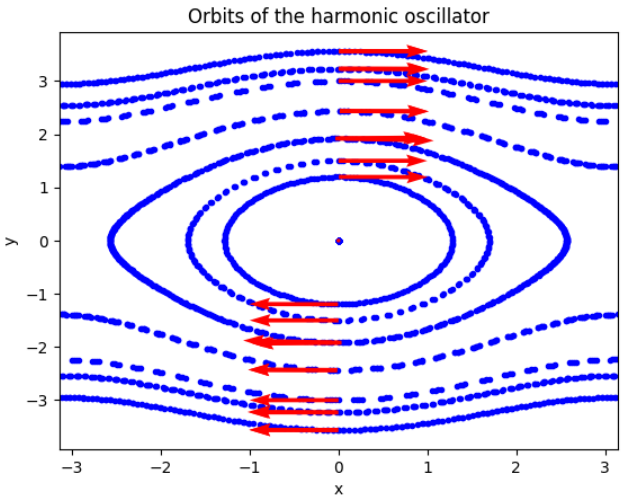


Figure 2.1: Plot of the orbits of the harmonic oscillator

2.4 Direction testing

Changing the initial values to only be $(0, 1)$ and modifying the print statement in the loop to

```
print(f"The time for initial conditions {initial_conditions_lst[i]} is
{overallTime} \n and the time error is
{expected_time_lst[i] + j*np.pi - abs(overallTime)}")
print(f"The final crossing point is {newInitial}")
print(f"The error of the final crossing is [{abs(newInitial[0]) -1},
{newInitial[1]}]")
```

With the new output being

```
The time for initial conditions [0, 1] is 4.712388980384686
 and the time error is 3.552713678800501e-15
The final crossing point is [-0.9999999999998098, 1.0430024899310553e-16]
The error of the final crossing is [-1.9018120411828932e-13, 1.0430024899310553e-16]
```

changing direction to -1

```
dir = -1
```

The output becomes

```
The time for initial conditions [0, 1] is -4.712388980384686
 and the time error is 3.552713678800501e-15
The final crossing point is [0.9999999999998098, 1.0430024899310553e-16]
The error of the final crossing is [-1.9018120411828932e-13, 1.0430024899310553e-16]
```

3 Integration of linear system

We aim to understand the phase portraits of linear systems with constant coefficients. For this purpose, we examined three distinct sets of coefficients, each leading to a unique phase portrait.

Below is a general solver and plotter of a linear system of four unknown constant coefficients. Direction vectors are plotted on the x=y axis for no special reason other than esthetics. The red square marks the box in which the initial conditions are generated.

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

# Coefficients
C = np.array([
    [2, -5, 1, -2],
    [3, -2, 4, -1],
    [-1, 0, 3, 2]
])

t_span = (0, 10)
rel_tol = 1e-6
abs_tol = 1e-6

generator = [5, 5]
number_of_points = 20

# Create arrays of initial conditions for x and y
x_initial_conditions = np.linspace(-generator[0], generator[1],
                        number_of_points)
y_initial_conditions = np.linspace(-generator[0], generator[1],
                        number_of_points)


# The axes for the plots
axes = [[-40, 40, -20, 20], [-20, 20, -20, 20], [-5.2, 5.2, -50, 50]]

# Arrow lengths
arrow_length = [7, 7, 4]

# Function to get the point closest to the line x=y from a trajectory
def get_point_closest_to_line_x_equals_y(X):
    # Get the absolute difference between x and y for each point in the
    # trajectory
    differences = np.abs(X[0] - X[1])
    min_difference = np.min(differences)
    # Return the index of the point with the smallest difference, i.e.
    # the point closest to the line x=y given the threshold
    if min_difference <= 0.5:
        return np.argmin(differences)
```

```
    else:
        return None

for i, coef in enumerate(C):
    plt.figure()
    plt.axis(axes[i])

    # Define the system using the current coefficients
    def system(t, X):
        return [
            coef[0] * X[0] + coef[1] * X[1],
            coef[2] * X[0] + coef[3] * X[1]
        ]

    counter = 0

    # Plot the linear system for each combination of x and y initial
    # conditions
    for x_initial_condition in x_initial_conditions:
        for y_initial_condition in y_initial_conditions:
            initial_conditions = [x_initial_condition,
                                  y_initial_condition]
            sol = solve_ivp(system, t_span,
                initial_conditions,
                rtol=rel_tol,
                atol=abs_tol, t_eval=
                np.linspace(t_span[0],
                        t_span[1], 300))
            plt.plot(sol.y[0], sol.y[1])
            # If this trajectory is one of the 1/10 chosen for
            # arrow plotting
            if counter % 29 == 0:
                # Get the point closest to the line x=y
                idx_closest = get_point_closest_to_line_x_equals_y(sol.y)

                if idx_closest is not None:
                    # Evaluate the differential equation at that
                    # point to get the direction
                    dy = system(0, sol.y[:, idx_closest])
                    # Normalize the direction for better visualization
                    norm = np.linalg.norm(dy)
                    if norm != 0:
                        dy = [arrow_length[i]*d/norm for d in dy]
                    # Plot the direction at the point closest to x=y
                    #using quiver
                    plt.quiver(sol.y[0, idx_closest],
                        sol.y[1, idx_closest], dy[0],
                        dy[1], angles='xy',
```

```
                        scale_units='xy', scale=1,
                        color='black', zorder=3)
            counter += 1
    plt.gcf().set_dpi(150)

    # Draw a box around the origin, -generator[0] < x < generator[0],
    # -generator[1] < y < generator[1]
    plt.plot([-generator[0], -generator[0], generator[0], generator[0],
    -generator[0]], [-generator[1], generator[1], generator[1],
    -generator[1], -generator[1]], 'r--', zorder=2)

    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title(f'Linear system - Test case {i+1}')
    plt.show()
```

The plots are on the next page.

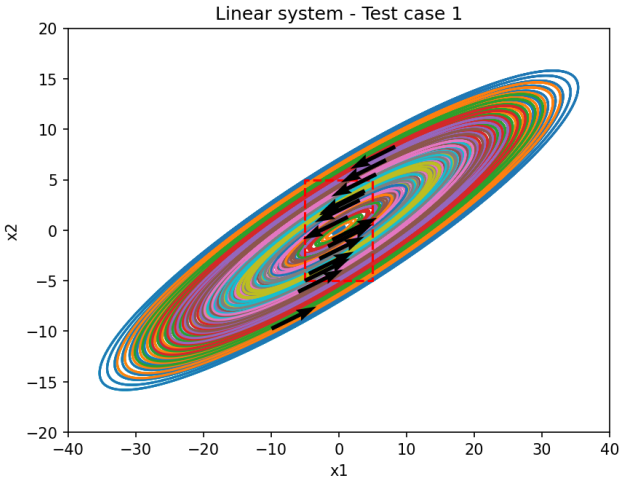


Figure 3.1: Center

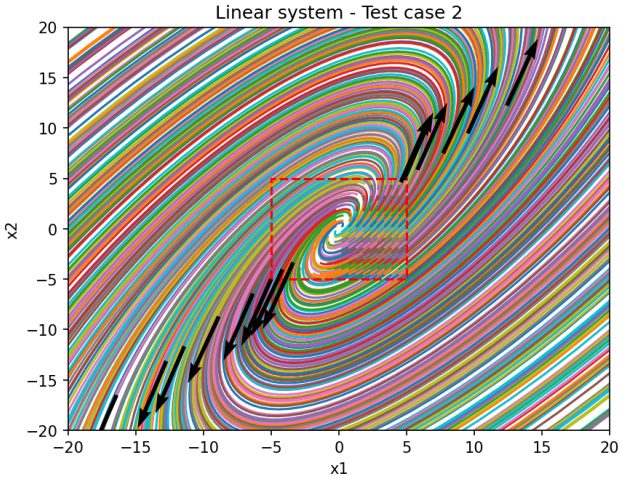


Figure 3.2: Unstable focus

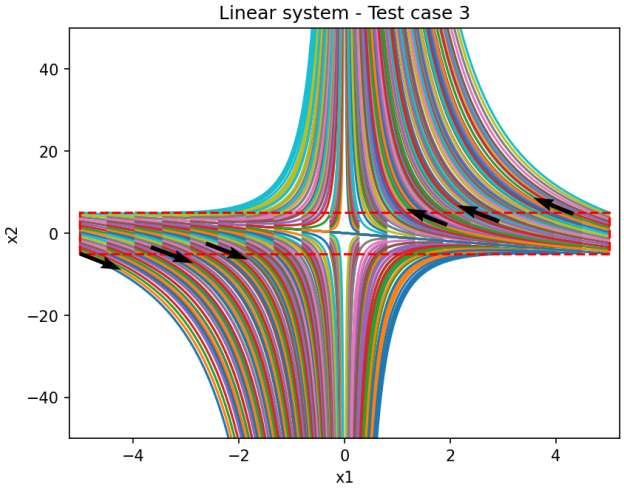


Figure 3.3: Saddle

1. **Center Phase Portrait:** The Jacobian matrix at the origin is:

$$Df(0) = \begin{bmatrix} 2 & -5 \\ 1 & -2 \end{bmatrix}$$

with eigenvalues $\lambda_1, \lambda_2 = \pm i$. Theory suggests that if both eigenvalues possess only imaginary parts, the phase portrait is a center. This confirms our observation from Figure 3.1. The period, given by $T = \frac{2\pi}{b}$ (with $b = 1$), is 2π. The rotation is counterclockwise.

2. **Unstable Focus Phase Portrait:** For the second set, the Jacobian matrix is:

$$Df(0) = \begin{bmatrix} 3 & -2 \\ 4 & -1 \end{bmatrix}$$

The eigenvalues are $\lambda_1, \lambda_2 = 1 \pm 2i$. The presence of complex conjugate eigenvalues with a positive real part characterizes this as an unstable focus (Figure 3.2). Reversing the time direction yields a stable focus. To achieve a stable focus in the original system, the Jacobian's eigenvalues should be complex conjugates with a negative real part.

3. **Saddle Phase Portrait:** Lastly, the Jacobian matrix for the third set is:

$$Df(0) = \begin{bmatrix} -1 & 0 \\ 3 & 2 \end{bmatrix}$$

The eigenvalues are $\lambda_1 = 2$ and $\lambda_2 = -1$, both real and satisfying $\lambda_2 < 0 < \lambda_1$. This confirms the phase portrait as a saddle, evident from Figure 3.3.

3.1 Manifolds

To determine the stable and unstable manifolds, we compute the eigenvectors:

$$v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$v_2 = \begin{bmatrix} 0.7071 \\ -0.7071 \end{bmatrix}$$

From visual inspection of the plot, v_1 is the unstable vector, while v_2 is stable. For computing the stable and unstable invariant manifolds, we bifurcate the computation. The initial condition is given by:

$$p + sv_+, \quad p + sv_-$$

Where p represents the equilibrium point (origin in our context), v is the eigenvector, and s is a suitably small value.

Below is the code and plot of the stable and unstable manifolds

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Manifolds
t_span = [0, 50]
v1 = np.array([0, 1])
v2 = np.array([0.7071, -0.7071])
s = -1
```

```python

def System(t, X):
    return [-1 * X[0] + 0 * X[1], 3 * X[0] + 2 * X[1]]

def plot_trajectory_with_arrows(ax, x, y, color='blue'):
    ax.plot(x, y, color=color)
    num_arrows = len(x) // 20
    for i in range(num_arrows):
        idx = i * 10
        ax.arrow(x[idx], y[idx], x[idx+1]-x[idx], y[idx+1]-y[idx],
                 head_width=0.005, head_length=0.01, fc=color, ec=color)

fig, ax = plt.subplots()

# First initial condition
initial_conditions = s * v2
sol = solve_ivp(System, t_span, initial_conditions, rtol=1e-14,
        atol=1e-11)
plot_trajectory_with_arrows(ax, sol.y[0], sol.y[1])

# Second initial condition
initial_conditions = (-s) * v2
sol = solve_ivp(System, t_span, initial_conditions, rtol=1e-14,
        atol=1e-11)
plot_trajectory_with_arrows(ax, sol.y[0], sol.y[1], color='orange')

ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_title('Manifolds')
ax.axis('equal')
ax.set_xlim([-0.1, 0.1])
ax.set_ylim([-0.1, 0.1])
ax.grid(True)
plt.show()
```

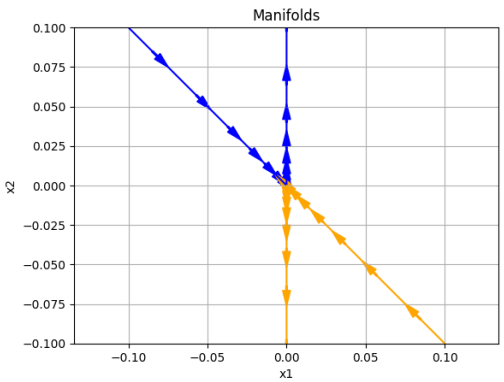


Figure 3.4: Plot of the manifolds

4 Pendulum

Because of time constraints, i did not have the time to write the following up in latex. Therefore, here are the handwritten notes. Refer to Figure 2.1 for a plot of the phase space.

Part C: Pendulum

$\theta'' = -\sin\theta \iff \begin{matrix}\theta \\ \omega = \theta'\end{matrix}, \begin{cases}\theta' = \omega \\ \omega' = -\sin\theta\end{cases} \iff \begin{cases}x' = y \\ y' = -\sin x\end{cases}$ (1)

1) equil. pts.: $\begin{cases}y = 0 \\ \sin x = 0\end{cases}$ $x = n\pi$, $n \in \mathbb{Z}$

$(n\pi, 0)$, $n \in \mathbb{Z}$ we can analyze $(0,0)$, $(\pi, 0)$

2) Stability of equilibrium points

Compute the Jacobi @ eq. pts. and eigenvalues.

$$Df = \begin{pmatrix} 0 & 1 \\ -\cos x & 0 \end{pmatrix}$$

$$Df(\pi,0) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$ $p = (\pi, 0)$; eigenvalues ± 1

$\exists$ stable and unstable

↳ 1d ↪ W^s ↪ W^u

$(\pi, 0)$ unstable for (1), it's a saddle

$p = (0,0)$ $Df(\vec{0}) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$, eigenvals $\pm i$

(center for the linearized system)

$\pm i$ is not in assumption 1 or 2 in Theorem, so we can't glean any information

3) $\exists$ a first integral for sys (1)

$$E(x,y) = \frac{1}{2}y^2 - \cos x,$$

so $E(x(t), y(t)) = \text{const}$. check this numerically (first and after)

$d_t E(x(t), y(t)) = 0$ exercis for ass. 3:

I'm lasy, so here, $x(t) =: x$, $y(t) =: y$, $\frac{d}{dt} = d_t = \dot{}$

$$d_t E(x,y) = y\dot{y} + \dot{x}\sin(x) = d_t \text{ const} = 0$$

5 Lotka-Volterra

Part D: Lotka-Volterra

Model system of ODE:

$$\begin{cases} x' = x(3 - x - 2y) \\ y' = y(2 - x - y) \end{cases}, \quad x, y \geq 0$$

1) Compute equilibrium points:

$$x = 0 \quad \vee \quad 3 - x - 2y = 0 \Rightarrow x + 2y = 3$$
$$y = 0 \quad \vee \quad 2 - x - y = 0 \Rightarrow x + y = 2$$

4 equil points: (i) $(0,0)$, (ii) $(0,2)$, (iii) $(3,0)$, (iv) $(1,1)$

2.1) Classify each equilibrium point according to stability of the linearized system.

$$\begin{cases} x' = -x^2 - 2xy + 3x \\ y' = -y^2 - xy + 2y \end{cases}$$

(i) $(0,0) = p$, $x' = A(x - p)$ where $A = Df(p)$

$$A = Df(p) = \begin{bmatrix} -2x - 2y + 3 & -2x \\ -y & -2y - x + 2 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\Rightarrow (3 - \lambda)(2 - \lambda) = 0 \Rightarrow \lambda_1 = 3, \lambda_2 = 2$$

$\Rightarrow$ All eigenvalues of the Jacobian of A have $\mathrm{Re} > 0$. This point is unstable ("source")

(ii) $(0,2) = p$

$$A = \begin{bmatrix} -1 & 0 \\ -2 & -2 \end{bmatrix} \quad \Rightarrow (-1-\lambda)(-2-\lambda) = 0$$
$$\Rightarrow \operatorname{Re}\lambda_i < 0, \; i \in \{1,2\}$$

This point is **stable**

(iii) $(3,0) = p$

$$A = \begin{bmatrix} -3 & -6 \\ 0 & -1 \end{bmatrix} \quad \Rightarrow (-3-\lambda)(-1-\lambda) = 0$$
$$\Rightarrow \operatorname{Re}\lambda_i < 0, \; i \in \{1,2\}$$

This point is **stable**

(iv) $(1,1) = p$

$$A = \begin{bmatrix} -1 & -2 \\ -1 & -1 \end{bmatrix} \quad \Rightarrow \operatorname{Re}\lambda_i < 0, \; i \in \{1,2\}$$

$$\Rightarrow (-1-\lambda)^2 - 2 = 1 + 2\lambda + \lambda^2 - 2 = -1 + 2\lambda + \lambda^2$$

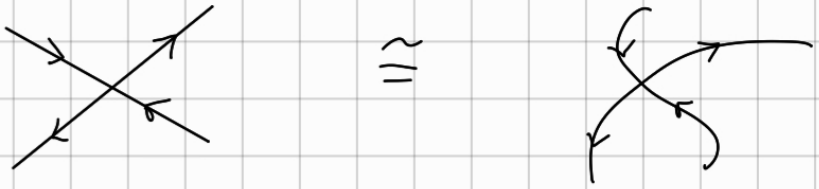
$$\lambda = \frac{-2 \pm \sqrt{4+4}}{-2} \qquad \lambda_1 = \frac{2\sqrt{2} - 2}{-2} = 1 - \sqrt{2} < 0$$
$$\lambda_2 = 1 + \sqrt{2} > 0$$

$\Rightarrow$ This point is an **unstable** saddlepoint

2.2) Can we use 2.1 to gain insight into the original nonlinear system?

Defining the linearized system as local manifolds around their respective equilibrium points, we relate it to the global manifold, for example,

Local, "micro" Global, "macro"



We can therefore extend the insights in 2.1 to their respective points in the non-linear system.

3), 4) programming

5) Write in Latex

Part 4 and 5: I did not have time to plot local phase portraits. Below is the code and plot of the global phase portrait.

The global plot provides insights into the overarching dynamics of our system. It's pivotal to focus on the plot's region where both values are positive since a species cannot have a negative population. Setting aside edge cases, the solutions bifurcate into two dominant trajectories:

- Solutions that converge to the fixed point $(0, 2)$.
- Solutions that head towards the fixed point $(3, 0)$.

Solutions above the line $y = x$ $(0, 2)$, while those below move to $(3, 0)$.

From a biological vantage point, the system's behavior becomes clear. When initiated with two non-zero populations, as time elapses, only one species will prevail. The simultaneous coexistence of both species is unfeasible, except at the equilibrium $(1, 1)$. However, any slight deviation from this balance inevitably results in the extinction of one species. Nature's inherent unpredictability makes sustaining this exact balance improbable, leading to the unfortunate outcome where one species invariably faces extinction. When species y have a bigger starting population (being above y=x), they drive x to extinction, and vice versa.

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

number_of_points = 19

# Initial conditions
x_initial_conditions = np.linspace(0, 3, number_of_points)
y_initial_conditions = np.linspace(0, 2, number_of_points)

t_span = (0, 5)
rel_tol = 1e-6
abs_tol = 1e-11

plt.figure()
plt.axis([0, 3, 0, 2])

# Define the Lotka-Volterra system
def system(t, X):
    return [
        X[0] * (3 - X[0] - 2 * X[1]),
        X[1] * (2 - X[0] - X[1])
    ]

def get_point_closest_to_circle(X, center=(3,3), radius=2.3):
    # Compute the distance of each point in the trajectory to the
    # circle's center
    distances_to_center = np.sqrt((X[0] - center[0])**2 +
    (X[1] - center[1])**2)
    # Compute the absolute difference between each distance and the radius
    differences = np.abs(distances_to_center - radius)
    # Return the index of the point with the smallest difference
```

```
    min_difference = np.min(differences)
    # Return the index of the point with the smallest difference, i.e.
    # the point closest to the line x=y given the threshold
    if min_difference <= 0.05:
        return np.argmin(differences)
    else:
        return None

counter = 0

# Iterate over initial conditions and plot trajectories
for x_initial_condition in x_initial_conditions:
    for y_initial_condition in y_initial_conditions:
        initial_conditions = [x_initial_condition, y_initial_condition]
        sol = solve_ivp(system, t_span, initial_conditions, rtol=rel_tol,
                        atol=abs_tol, t_eval=np.linspace(t_span[0],
                        t_span[1], 100))
        plt.plot(sol.y[0], sol.y[1])

        if counter % 7 == 0:
            # Get the point closest to the circle
            idx_closest = get_point_closest_to_circle(sol.y)

            if idx_closest is not None:
                # Evaluate the differential equation at that point to
                # get the direction
                dy = system(0, sol.y[:, idx_closest])

                # Normalize the direction for better visualization
                norm = np.linalg.norm(dy)
                if norm != 0:
                    dy = [0.2*d/norm for d in dy]

                # Plot the direction at the point closest to the circle
                # using quiver
                plt.quiver(sol.y[0, idx_closest], sol.y[1, idx_closest], dy[0],
                           dy[1], angles='xy', scale_units='xy', scale=1,
                           color='black', zorder=3)

        plt.gcf().set_dpi(150)
        counter += 1

# making the plot look nice
plt.xlabel('x')
plt.ylabel('y')
plt.title('Lotka-Volterra system')

plt.show()
```

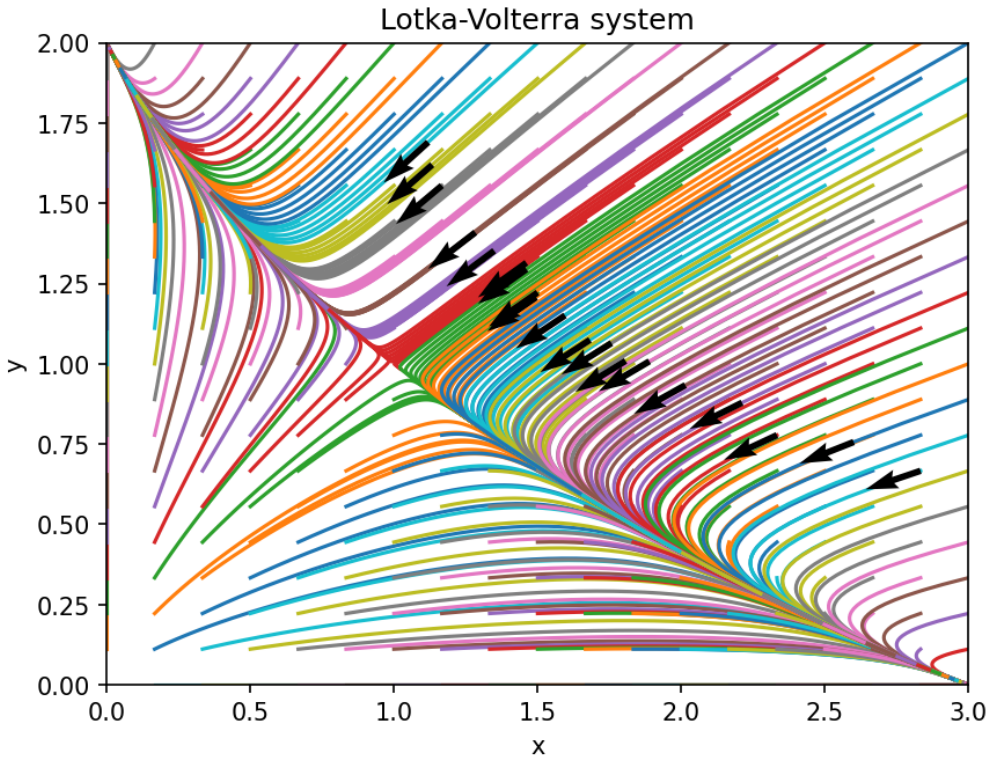


Figure 5.1: Plot of the model. Directions are plotted on the right-hand side

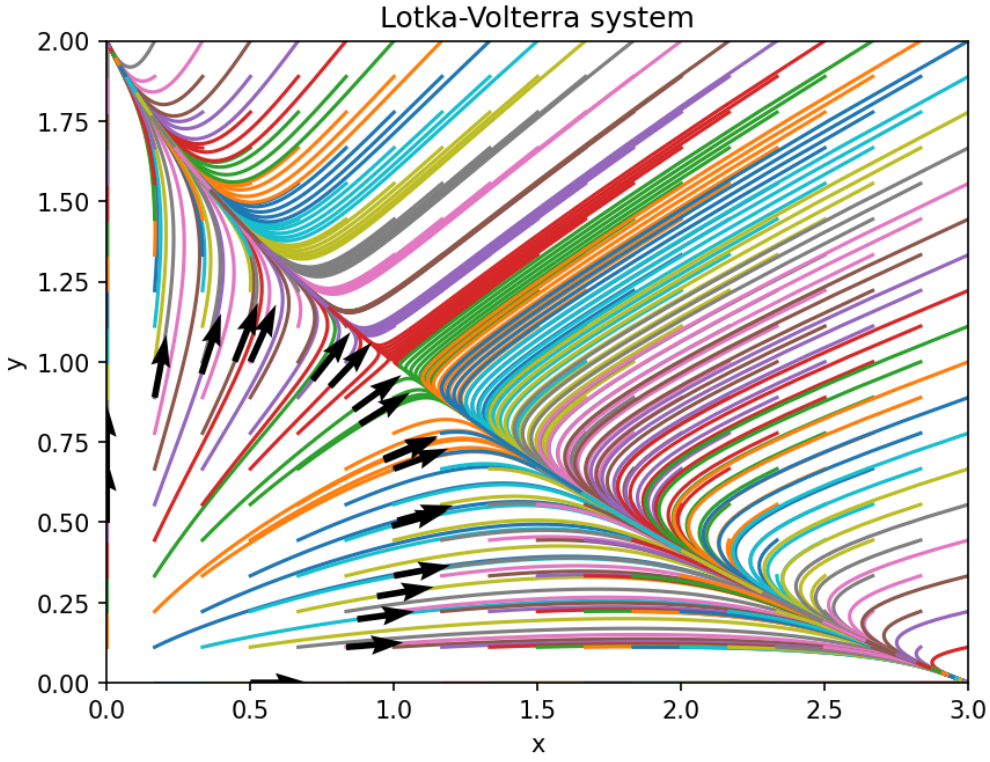


Figure 5.2: Plot of the model. Directions are plotted on the left-hand side

Comment on 7) In addition to the two heteroclinic orbits described, there are also an infinite number of heteroclinic orbits from respectively $(0, 0)$ to $(3, 0)$ as well as $(0, 0)$ to $(0, 2)$

6) Prove that the axes are invariant

x-axis: If trajectory starts in the x-axis, must show that $y(t) = 0 \ \forall \ t > 0$

y-axis: If trajectory starts in the y-axis, must show that $x(t) = 0 \ \forall \ t > 0$

When $x = 0$, $y' = y(2 - y)$. $\begin{cases} x' = x(3 - x - 2y) \\ y' = y(2 - x - y) \end{cases}$

So when $y = 0$, $y' = 0$

we're looking at the 1st quadrant

if $x \geq 0$, $y' = y(2 - x - y)$

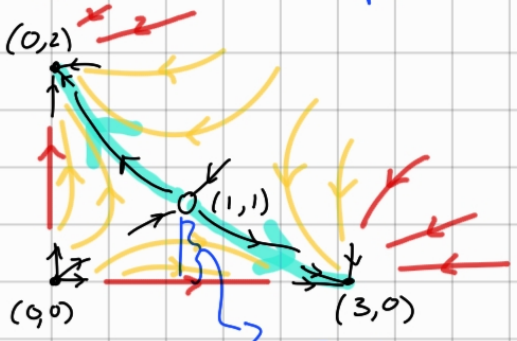
So when $y = 0$, $\forall \ x \geq 0$, $y' = 0$. This means that we're never "accelerated" into the y direction, $\forall \ t$. => x-axis is invariant

When $y \geq 0$, $x' = x(3 - x - 2y)$

So when $x = 0$, $\forall \ y \geq 0$, $x' = 0$. This means that we're never "accelerated" into the x direction, $\forall \ t$. => y-axis is invariant

7) Are there homoclinic / heteroclinic orbits

Intuition: From the plot we see no homoclinic orbits. There are two heteroclinic orbits. (1,1) -> (3,0) and (1,1) -> (0,2).



+

∞ here as well