Introduction to Mathematics for Data Science
Personal Assignnment 1

Zehao Qian

November 3, 2023

1 Question 1

1.1 Modelling Bird Population Decline Due to Invasive Snakes

In the modeling of bird populations after snake invasion, I utilized **Formula 1**. This choice was made to satisfy the requirements specified in the problem statement, where the population starts decreasing gradually after the snake invasion, and then, over time, the bird population experiences a sharp decline. Additionally, the modeling process should also account for the principles of **population dynamics** (in the later stages, due to a reduced availability of birds for the snakes to prey on, the snake population would also decrease, resulting in a slowdown in the rate of decline of the bird population).

$$P(t) = \frac{A}{e^{kt^2}} \tag{1}$$

Where:

- $\mathbf{P(t)}$ is the population of birds at time 't'
- **A**: Maximum capacity, representing the constant bird population before snake invasion.
- **k** is a constant that determines the rate of decline.

Here is a brief proof where I applied SymPy to differentiate Formula 1. The code is provided below, and the result is (I preset A=1000, k=0.1):

$$P'(t) = -200.0te^{-0.1t^2} \tag{2}$$

```python
import sympy

# Define symbolic variable
t = sympy.symbols('t')
```

```

# Define constants
# Define the maximum capacity A, the value of k,
# and the invasion time
A = 1000
k = 0.1

# Define the function
P = A * sympy.exp(-k * t**2)

# Calculate the derivative with resympyect to t
P_derivative = sympy.diff(P, t)

# Print the elegant mathematical expression
P_derivative
```

After I computed the derivative of $P(t)$ using SymPy, I employed the Python library Matplotlib to create a graphical representation of the derivative (**Figure 1**). I found that it meets the specified criteria, with the derivative values always being less than or equal to 0. Initially, the derivative is far from the x-axis, indicating an increasing rate of decline, while in the later portion, it starts to exhibit a deceleration in the decline rate.

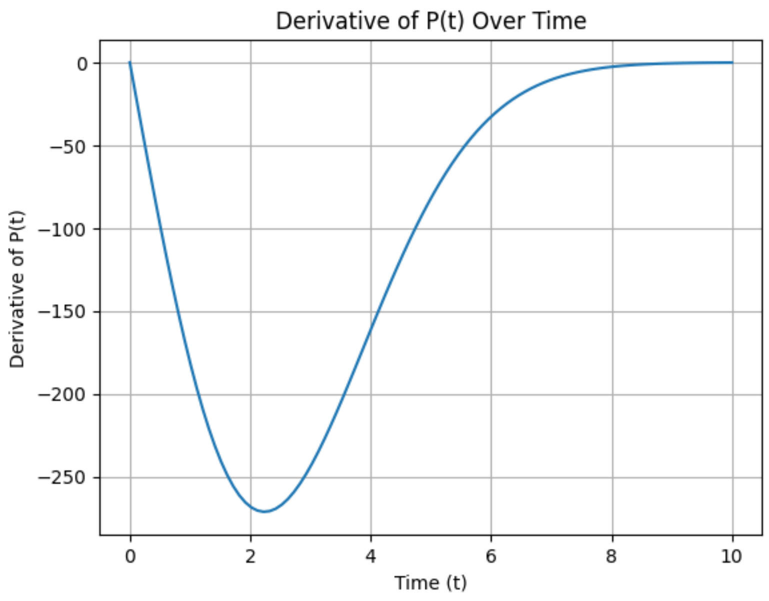


Figure 1: Derivative of P(t)

Finally, I took into account the event of snake invasion by introducing an invasion event. Before this event, the bird population remains constant.

```python
# import numpy as np
# import matplotlib.pyplot as plt
import numpy as np

def bird_population_model(t, A, k, invade_time):
    P = np.zeros_like(t)  # Initialize the population array

    for i in range(len(t)):
        if t[i] < invade_time:
            P[i] = A  # Bird population remains constant before
                invasion
        else:
            P[i] = A * np.exp(-k * (t[i] - invade_time)**2)

    return P


invade_time = 5  # Time of snake invasion

# Generate a time range for plotting
t = np.linspace(0, 15, 100)  # Adjust the time range as needed

# Calculate the population over time
P = bird_population_model(t, A, k, invade_time)

# Plot the population over time
plt.plot(t, P)
plt.xlabel("Time")
plt.ylabel("Bird␣Population")
plt.title("Bird␣Population␣Over␣Time")
plt.grid(True)
plt.show()
```

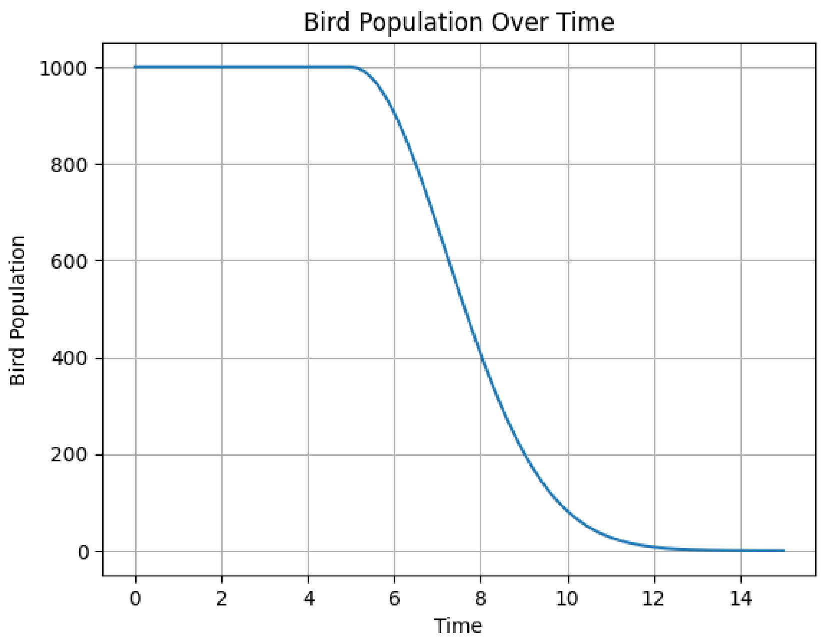


Figure 2: Figure of Bird Popullation Model

Figure 2 represents the final graph I created.

1.2 Seasonal Epidemiological Models

In this code, the abscissa shows the summer and winter of each year, with the number of cases increasing in the summer and remaining the same in the winter. To achieve this, we use the integer part to determine the year in order to switch between summer and winter. Custom labels were also created for the x-axis to represent the seasons of the year.

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

def pandemic_model(t):
    # Define the peak year and the duration of the pandemic
    peak_year = 5
    duration = 10

    # Calculate the number of cases based on the given
       conditions
    cases = np.zeros_like(t)
    for i in range(len(t)):
```

```python
        year = int(t[i])
        if 0 <= year < peak_year:
            cases[i] = (year + t[i] - year) / peak_year * (duration / 2)
        elif peak_year <= year < peak_year + duration / 2:
            cases[i] = (1 - (year + t[i] - year - peak_year) / (duration / 2)) * (duration / 2)

    return cases

# Generate a time range for plotting (a decade)
t = np.linspace(0, 10, 100)

# Calculate the number of cases over the decade
cases = pandemic_model(t)

# Create custom labels for x-axis to represent summer and winter
x_labels = [f'Year {int(year)} {"Summer" if int(year) % 1 == 0 else "Winter"}' for year in t]

# Plot the number of cases over time
plt.plot(x_labels, cases)
plt.xlabel("Time (Year and Season)")
plt.ylabel("Number of Cases")
plt.title("Pandemic Cases Over a Decade with Seasonal Variation")
plt.grid(True)
plt.xticks(rotation=45)
plt.show()
```

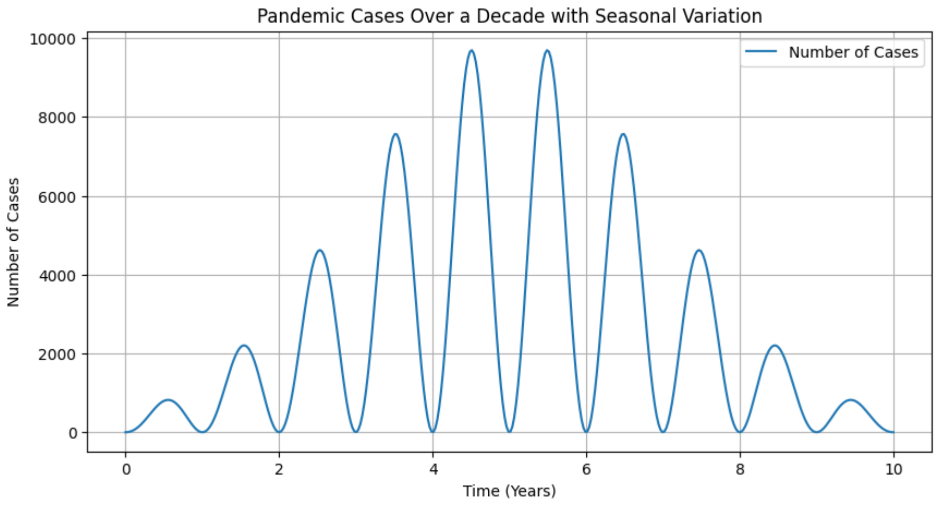


Figure 3: Figure of Pandemic Cases

The result of this code is **Figure 3**. The number of cases changes only when the Season is summer.

2 Question 2

2.1 Derivative $f'(x)$ at $x = 7$

From the definition of the derivative, it can be seen that:

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{3}$$

In the diagram I plotted at $x = 7, 7.2$, while $y = 3.8, 4.4$. So we let $\Delta x = 0.2$

$$\begin{aligned} f'(x) &= \frac{f(7 + \Delta x) - f(7)}{\Delta x} \\ &= \frac{f(7.2) - f(7)}{0.2} \\ &= \frac{4.4 - 3.8}{0.2} \\ &= 3.0 \end{aligned}$$

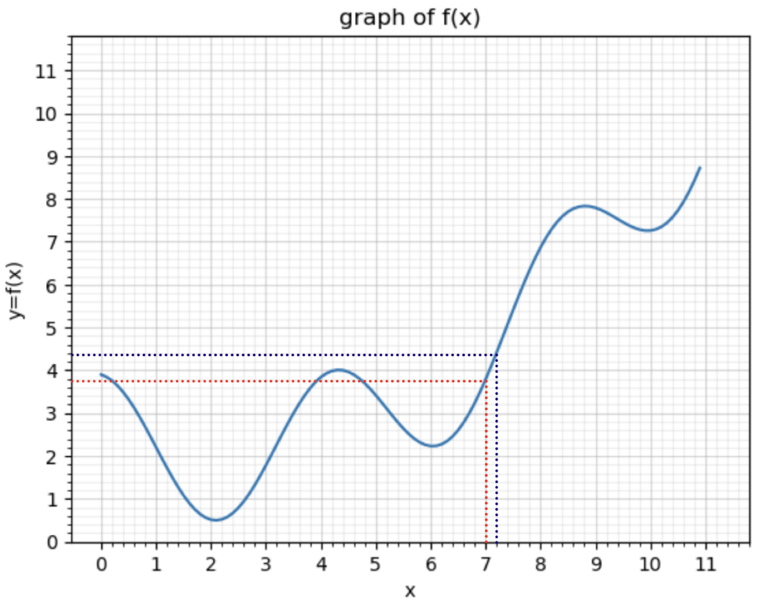


Figure 4: $f(x)$'s value when $x = 7, 7.2$

2.2 The Estimated Value of $\int_2^6 f(x)dx$

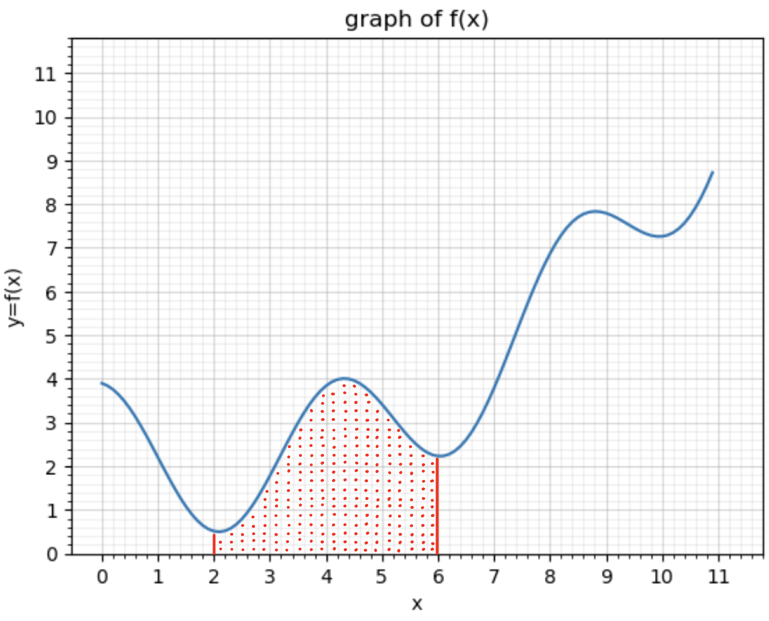
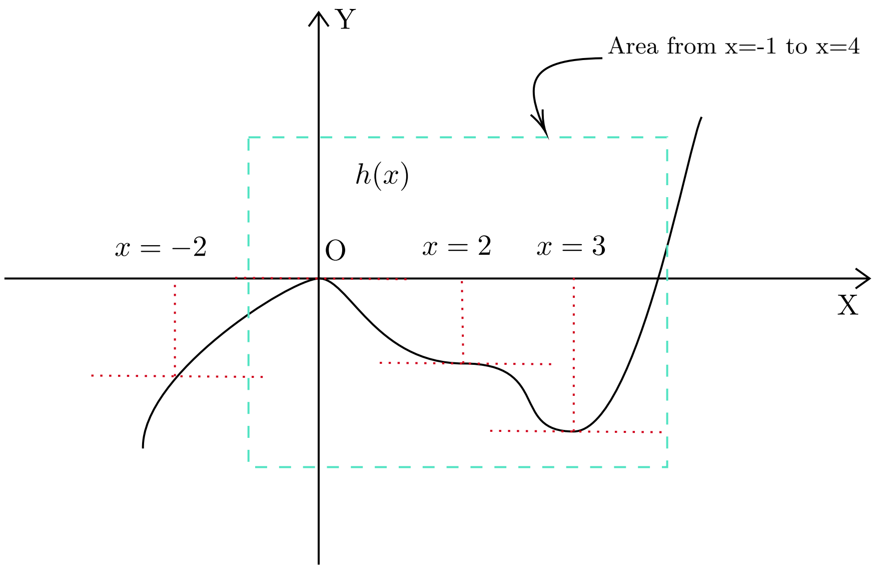


Figure 5: It is about 258 blocks from $x = 2$ to 6

The area of each block is: $0.2 \times 0.2 = 0.04$, so the value of $\int_2^6 f(x)dx$ is $258 \times 0.04 = 10.32 \approx \mathbf{10.3}$

2.3 Draw the plot of function h



I draw this plot above with LaTeX Tikz package via Mathcha (https://www.mathcha.io/editor)

Analytics:

- For $x < 0$, the graph increases.
- At $x = 0$, there is a local maximum.
- For $0 < x < 2$, the graph decreases.
- At $x = 2$, there is an inflection point.
- For $2 < x < 3$, the graph decreases.
- At $x = 3$, there is a local minimum.
- For $x > 3$, the graph increases.

3 Question 3

3.1 Gradient Estimation at $(0.5, 0.3)$ Using Linear Interpolation

In this question I use the **finite difference method**. It is a commonly used numerical computational technique for estimating derivatives or partial derivatives of a function, especially when dealing with discrete data points.

The finite difference method is typically applied in the following scenarios:

- A set of discrete data points but do not possess an analytical expression for the function.
- Compute the derivative of a function at a specific point, but lack an analytical expression for the derivative.

I apply Central Difference Method (Here, h is a small step size):

$$\nabla g(x,y) = (\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}) \tag{4}$$

$$\frac{\partial g}{\partial x} = \frac{g(x+h,y) - g(x-h,y)}{2h} \tag{5}$$

$$\frac{\partial g}{\partial y} = \frac{g(x,y+h) - g(x,y-h)}{2h} \tag{6}$$

$$\begin{aligned}\frac{\partial g_{x=0.5,y=0.3}}{\partial x_{x=0.5}} &= \frac{g(0.6,0.3) - g(0.4,0.3)}{2 \times 0.1} \\ &= \frac{0.003 - (-0.337)}{0.2} \\ &= 1.7\end{aligned}$$

$$\begin{aligned}\frac{\partial g_{x=0.5,y=0.3}}{\partial y_{y=0.3}} &= \frac{g(0.5,0.4) - g(0.5,0.2)}{2 \times 0.1} \\ &= \frac{-0.234 - (-0.038)}{0.2} \\ &= -0.98\end{aligned}$$

So, $\nabla g(x_{x=0.5}, y_{y=0.3}) = (1.7, -0.98)$.

3.2 Estimating Directional Derivative of g at $(0.5, 0.3)$ along the Vector $u = (1, -4)$

To estimate the directional derivative of g at the point $(x, y) = (0.5, 0.3)$ along the vector $\mathbf{u} = (1, -4)$, the directional derivative of g along the vector $\mathbf{u} = (a, b)$ can be calculated using the gradient ∇g as follows ((a,b) is the unitized $\vec{u}$.):

$$D_{\mathbf{u}} g = \nabla g \cdot \mathbf{u} = \frac{\partial g}{\partial x} \cdot a + \frac{\partial g}{\partial y} \cdot b \tag{7}$$

$\mathbf{u} = (1, -4)$ and I have estimated the gradient components as $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y}$.

$$\vec{u} = (\frac{u_x}{|u|}, \frac{u_y}{|u|}) = (\frac{1}{\sqrt{17}}, \frac{-4}{\sqrt{17}})$$

$$D_{\mathbf{u}}g = \left(\frac{\partial g}{\partial x}\right) \cdot \frac{1}{\sqrt{17}} + \left(\frac{\partial g}{\partial y}\right) \cdot (\frac{-4}{\sqrt{17}})$$

Use the values for $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y}$ to calculate $D_{\mathbf{u}}g$.

$$D_u g = 1.7 \times \frac{1}{\sqrt{17}} - 0.98 \times (\frac{-4}{\sqrt{17}}) = 1.3507$$

4 Question 4

4.1 Gradient Descent from $(2, 4)$ for $F(x, y) = x^2 + y^2 - 6sin(x - y)$

```python
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp

# Define symbolic variables
x, y = sp.symbols('x y')

# Define the function pandemic_model(x, y)
F = x**2 + y**2 - 6*sp.sin(x - y)

# Initialize the starting point
x_current, y_current = 2, 4

# Choose the step size parameter
alpha = 0.1  # Change different values 0.1, 0.5, 1

# Create lists to store the trajectory
x_trajectory = [x_current]
y_trajectory = [y_current]

# Iterate for gradient descent
for step in range(3):
    # Calculate the gradient
    gradient_x = sp.diff(F, x).subs({x: x_current, y: y_current
        })
    gradient_y = sp.diff(F, y).subs({x: x_current, y: y_current
        })

    # Update the point
    x_current -= alpha * gradient_x
    y_current -= alpha * gradient_y

    # Add to the trajectory lists
    x_trajectory.append(x_current)
    y_trajectory.append(y_current)

# Create a contour plot
x_vals = np.linspace(-2, 4, 400)
y_vals = np.linspace(-2, 6, 400)
X, Y = np.meshgrid(x_vals, y_vals)
```

```
Z = X**2 + Y**2 - 6*np.sin(X - Y)
plt.contour(X, Y, Z, levels=50)
plt.colorbar()

plt.plot(x_trajectory, y_trajectory, marker='o', label='
    Gradient Descent Path')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Contour Plot of F(x, y) with Gradient Descent Path')
plt.legend()
plt.grid(True)
plt.show()
```

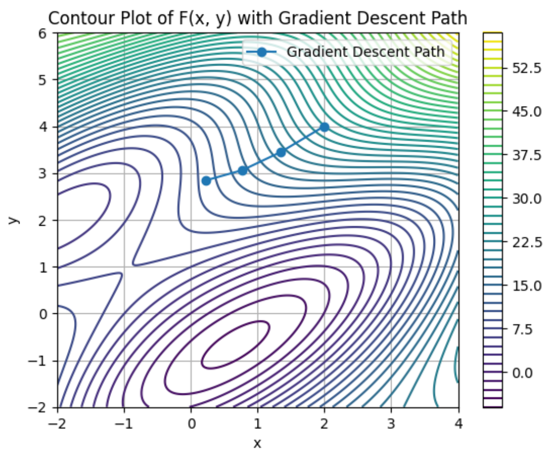


Figure 6: step size parameter = 0.1

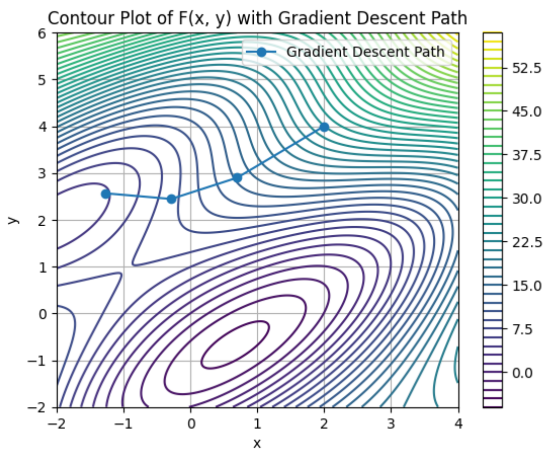


Figure 7: step size parameter = 0.2

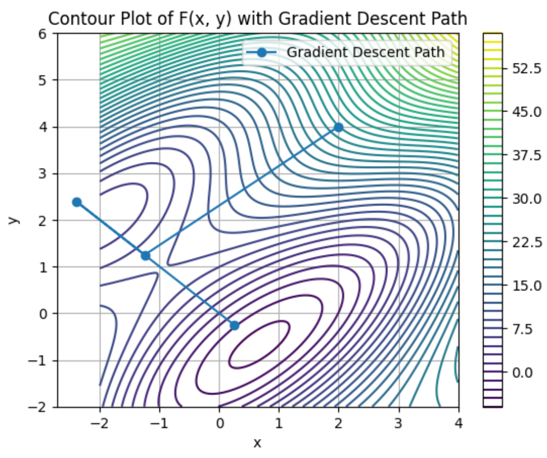


Figure 8: step size parameter = 0.5

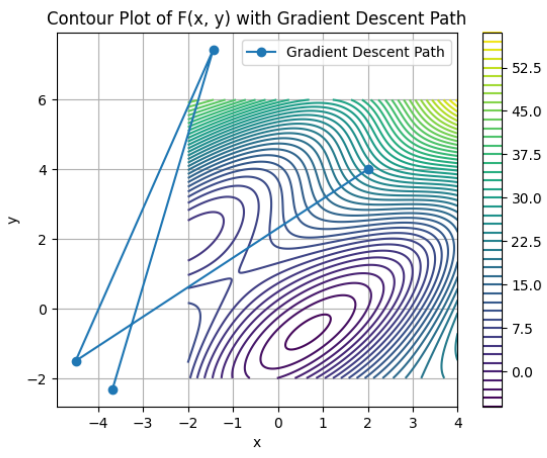


Figure 9: step size parameter = 1.0

4.2 Gradient Descent with more steps

Analytics:

1. Convergence to a Local Minimum: If the learning rate is chosen appropriately and the optimization problem is convex or has a single global minimum, gradient descent will converge to that minimum. may also be the global minimum.
2. Convergence to a Saddle Point: In non-convex optimization problems, gradient descent may converge to a saddle point rather than a local minimum. Saddle points are points where the gradient is zero, but they are not necessarily optimal.
3. Oscillation or Divergence: If the learning rate is too high, gradient descent may oscillate or even diverge. The model parameters may bounce back and forth, failing to converge.
4. Slowing Down Near the Minimum: As gradient descent gets closer to the minimum, the step sizes become smaller because the gradient becomes smaller.
5. Flat Regions: In some optimization landscapes, there may be regions where the gradient is close to zero, but it is not a minimum.
6. Longer Matplotlib Randering Time: when I set iteration number to 100, the plot is very complex, with a large number of data points, labels, and graphical elements, it can take longer to render (About 2.5 minutes).

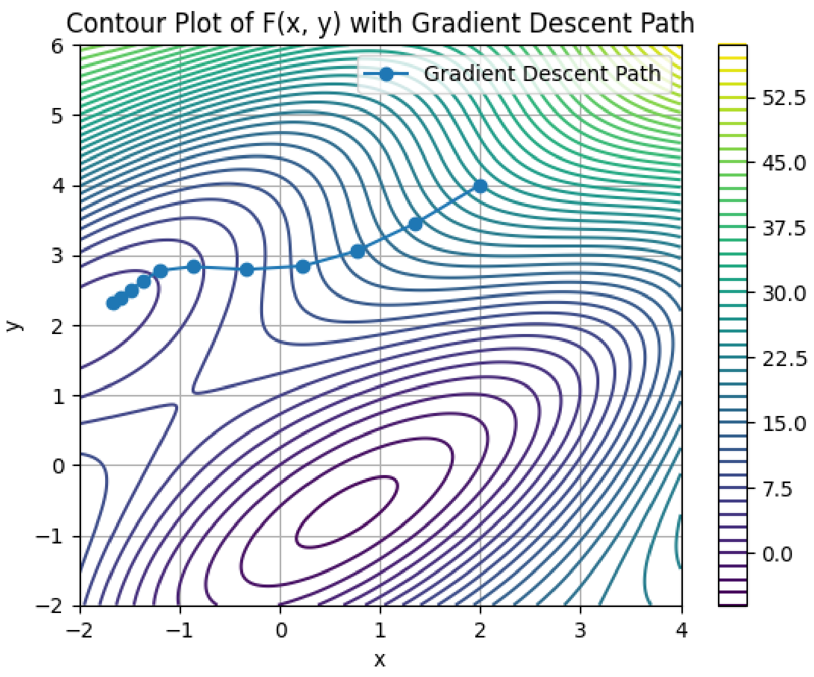


Figure 10: Set iteration for 10 times, learning rate = 0.1

5 Question 5

5.1 Vector Length and Angle Calculation

```python
import numpy as np

# Define the vectors u and v
u = np.array([5, 6])
v = np.array([1, -2])

# Calculate the length of vector u
length_u = np.linalg.norm(u)

# Calculate the length of vector v
length_v = np.linalg.norm(v)

# Calculate the dot product of u and v
dot_product_uv = np.dot(u, v)

# Calculate the angle between u and v (in radians)
cosine_theta = dot_product_uv / (length_u * length_v)
theta = np.arccos(cosine_theta)

# Convert the angle from radians to degrees
theta_degrees = np.degrees(theta)

# Print the results
print("Length␣of␣u:", length_u)
print("Length␣of␣v:", length_v)
print("Angle␣between␣u␣and␣v␣(in␣radians):", theta)
print("Angle␣between␣u␣and␣v␣(in␣degrees):", theta_degrees)
```

The output is:

- Length of u: 7.810249675906654
- Length of v: 2.23606797749979
- Angle between u and v (in radians): 1.983206768392284
- Angle between u and v (in degrees): 113.62937773065683

5.2 Orthogonal Projection and Vector Visualization

```python
import numpy as np
import matplotlib.pyplot as plt

# Define vectors ~u and ~v
u = np.array([5, 6])
v = np.array([1, -2])

# Calculate Proj_v(u)
proj_v_u = (np.dot(u, v) / np.dot(v, v)) * v

# Create a plot to visualize the vectors
plt.figure(figsize=(6, 6))
plt.quiver(0, 0, u[0], u[1], angles='xy', scale_units='xy',
    scale=1, color='b', label='Vec␣U')
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy',
    scale=1, color='r', label='Vec␣V')
plt.quiver(0, 0, proj_v_u[0], proj_v_u[1], angles='xy',
    scale_units='xy', scale=1, color='g', label='Proj_v(u)')

# Set plot limits
plt.xlim(-4, 8)
plt.ylim(-4, 8)

# Add labels and legend
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

# Show the plot
plt.grid(True)
plt.show()
```

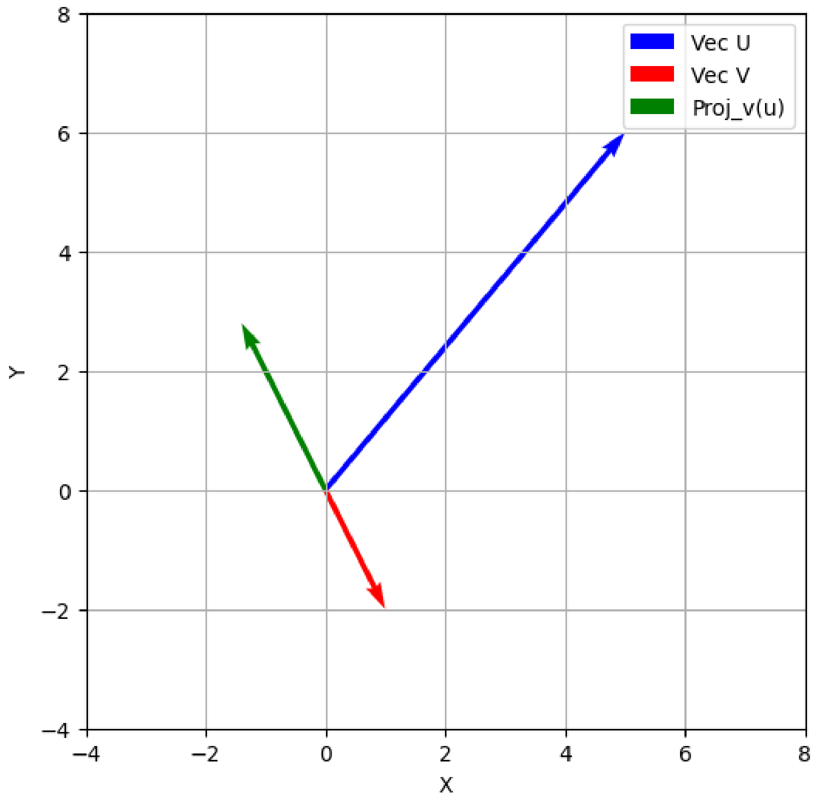


Figure 11: Orthogonal Projection of the Vector $\vec{u}$ onto $\vec{v}$