CS 4100 Final Project

By Mahika Sharma

NU ID: 002229314

Part 1: Gradient Descent

1.1: Quadratic Functions

1.1 Question 1:

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

def f1(x):
    """
    First quadratic function: f1(x) = x^2

    Args:
        x: Input value

    Returns:
        Function value at x
    """
    return x**2

def deriv_f1(x):
    """
    Derivative of f1(x) = x^2
    f1'(x) = 2x

    Args:
        x: Input value

    Returns:
        Derivative value at x
    """
    return 2 * x

def f2(x):
    """
    Second quadratic function: f2(x) = x^2 - 2x + 3
    Args:
        x: Input value

    Returns:
        Function value at x
    """
    return x**2 - 2*x + 3

def deriv_f2(x):
    """
    Derivative of f2(x) = x^2 - 2x + 3
    f2'(x) = 2x - 2

    Args:
        x: Input value

    Returns:
        Derivative value at x
    """
    return 2*x - 2

# Gradient Descent Algorithm
def gradient_descent(f, deriv, x0, alpha=0.1, epsilon=0.001,
iter_max=1000):
    """
    Implementing gradient descent algorithm

    Args:
        f: function to minimize
        deriv: derivative of function
        x0: initial starting point
        alpha: step size
        epsilon: tolerance
        iter_max: max iterations

    Returns:
        x: optimal x value found
        iter_count: # iterations performed
    """
    x = x0
    iter_count = 0
    while iter_count < iter_max:
        # while in loop, compute next x using the update rule
        x_new = x - alpha * deriv(x)

        # break from loop if the change in x is less than epsilon
(converged)
        if abs(x_new - x) < epsilon:
            break
```

```python
        # increment and update x
        x = x_new
        iter_count += 1

    return x, iter_count

def plot_opt(f, optimal_x, title):
    """
    Plots function along with optimal point found by gradient descent

    Args:
        f: function to plot
        optimal_x: optimal x value found
        title: Name of the function for title
    """
    # range for plotting
    x_vals = np.linspace(-5, 5, 400)
    # computing f(x) values for all x values
    y_vals = f(x_vals)

    plt.figure()
    # plotting the function curve
    plt.plot(x_vals, y_vals, label='Function')
    # plotting the minimum point
    plt.scatter(optimal_x, f(optimal_x), color='red', label='Minimum')
    plt.title(title)
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()

def main():
    """
    Main function to test gradient descent on quadratic functions with
different parameters
    """
    # For question 2
    opt_f1_pos = gradient_descent(f1, deriv_f1, x0=3)
    opt_f1_neg = gradient_descent(f1, deriv_f1, x0=-3)

    opt_f2_pos = gradient_descent(f2, deriv_f2, x0=3)
    opt_f2_neg = gradient_descent(f2, deriv_f2, x0=-3)

    print("f1: x0=3 =>", opt_f1_pos, "x0=-3 =>", opt_f1_neg)
    print("f2: x0=3 =>", opt_f2_pos, "x0=-3 =>", opt_f2_neg)

    # For question 3
    alphas = [1, 0.001, 0.0001]
    for alpha in alphas:
        x_f1, iters_f1 = gradient_descent(f1, deriv_f1, x0=3,
alpha=alpha, epsilon=0.001)
        print(f"f1 with alpha={alpha}: x = {x_f1}, iterations =
{iters_f1}")

    # For question 4
    epsilons = [0.1, 0.01, 0.0001]

    for eps in epsilons:
        x_f1, iters_f1 = gradient_descent(f1, deriv_f1, x0=3,
alpha=0.1, epsilon=eps)
        print(f"f1 with epsilon={eps}: x = {x_f1}, iterations =
{iters_f1}")

# Testing with the main function + printing outputs
main()
```

```
f1: x0=3 => (0.004642275147320177, 29) x0=-3 => (-
0.004642275147320177, 29)
f2: x0=3 => (1.0048357032784585, 27) x0=-3 => (0.9950482398428585, 30)
f1 with alpha=1: x = 3, iterations = 1000
f1 with alpha=0.001: x = 0.4999835397587718, iterations = 895
f1 with alpha=0.0001: x = 3, iterations = 0
f1 with epsilon=0.1: x = 0.40265318400000005, iterations = 9
f1 with epsilon=0.01: x = 0.04323455642275677, iterations = 19
f1 with epsilon=0.0001: x = 0.0004984604984193437, iterations = 39
```

1.1 Question 2:

When starting at X0 = 3 and X0 = -3, the gradient descent values end up closer to the actual minimum value for both functions. For F1(x) = x^2, the value approaches a small number near 0, where its positive if starting at 3, and negative if starting at -3. For F2(x), it gets closer to 1 either way. This makes sense and I would have expected this because both functions have only one minimum/lowest point so regardless of the starting point, the algorithm will find it, and so different starting points would not affect the final answer by much.

1.1 Question 3:

For values X0 = 3 and alpha = 0.001, I tested gradient descent on F1(x) = x^2 using different values of alpha. For alpha = 1, the algorithm returned x = 3 and reached the maximmum of 10000 iterations. This means that it didn't converge at all, and the large step size is the likely cause of overshooting. Then, for alpha = 0.001, the algorithm returned x = 0.49998 in 895 iterations. It did converge eventually but took much longer, meaning that the small step size required many steps to approach the minimum. For alpha = 0.0001, the algorithm returned x = 3 in 0 iterations. This shows that the step size was so small that the first update didn't change the value of x enough to exceed the convergence threshold. As a result, the algorithm stopped immediately and resulted in 0 iterations. These are the variations I observed in the output: a learning rate that is too big may end up not converging, while a learning rate that is too small will cause very slow or no progress or iterations.

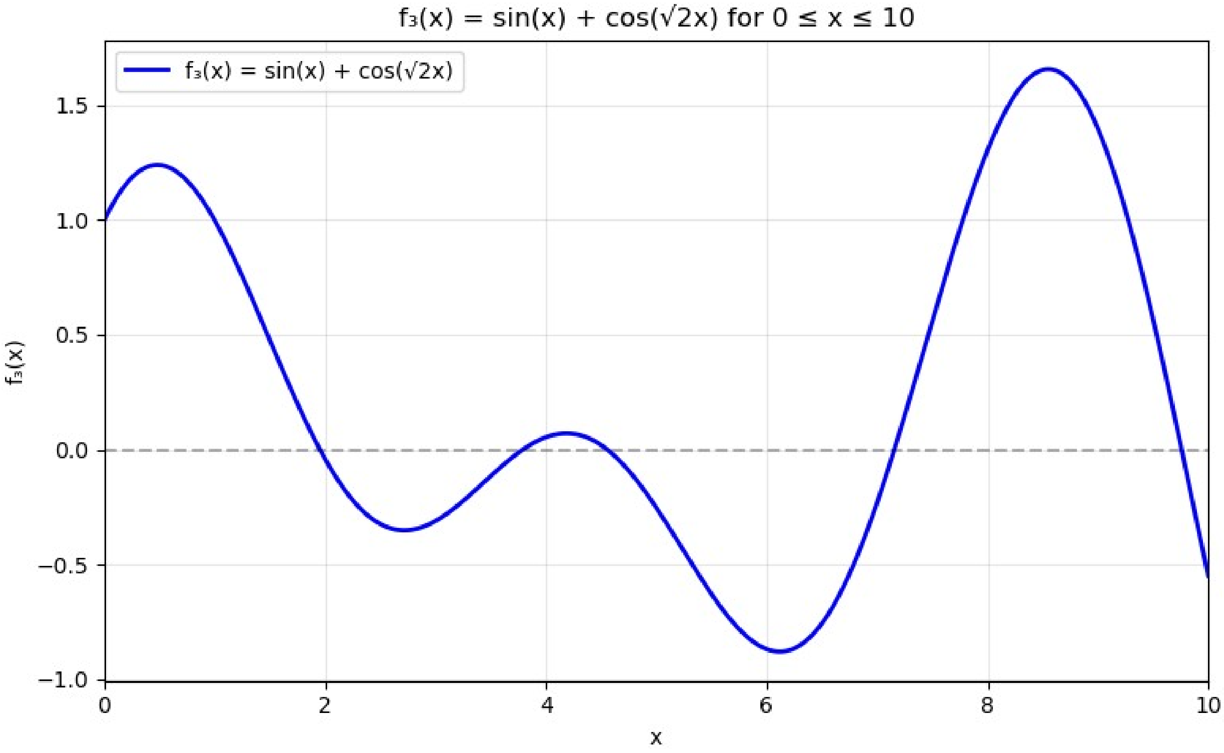
1.1 Question 4:

For values x0 = 3 and α = 0.1, I tested gradient descent on f1(x) = x^2 using different values of ε. For ε = 0.1, the algorithm returned x = 0.4027 in 9 iterations. This shows that it stopped earlier when it was still far from the true minimum value. For ε = 0.01, the algorithm returned x = 0.0432 in 19 iterations, meaning it got closer to the minimum but needed more than double the iterations to do so. For ε = 0.0001, the algorithm returned x = 0.0005 in 39 iterations. This was the most accurate value, and closest to the true minimum of x = 0, but required over four times as many iterations as the largest ε. These are the significant variations I observed in the outputs: larger tolerances stop the algorithm too early with low accuracy, whereas a smaller tolerance forces it to continue until it finds a very precise solution but its computationally expensive.

1.2 More Complex Functions

1.2 Question 1:

```python
def f3(x):
    """
    Complex function: f3(x) = sin(x) + cos(√2 * x)

    Args:
        x: Input value (should be in range 0 ≤ x ≤ 10)

    Returns:
        Function value at x
    """
    return np.sin(x) + np.cos(np.sqrt(2) * x)

def plot_f3():
    """
    Plots f3(x) = sin(x) + cos(√2 * x) for 0 ≤ x ≤ 10
    """
    # creating the x values
    x_plot = np.linspace(0, 10, 2000)
    y_plot = f3(x_plot)

    # creating the plot
    plt.figure(figsize=(8, 5))
    plt.plot(x_plot, y_plot, 'b-', linewidth=2, label='f₃(x) = sin(x) + cos(√2x)')
    plt.xlabel('x')
    plt.ylabel('f₃(x)')
    plt.title('f₃(x) = sin(x) + cos(√2x) for 0 ≤ x ≤ 10')
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.xlim(0, 10)

    # adding horizontal line at y = 0 as reference line
    plt.axhline(y=0, color='k', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()

plot_f3()
```



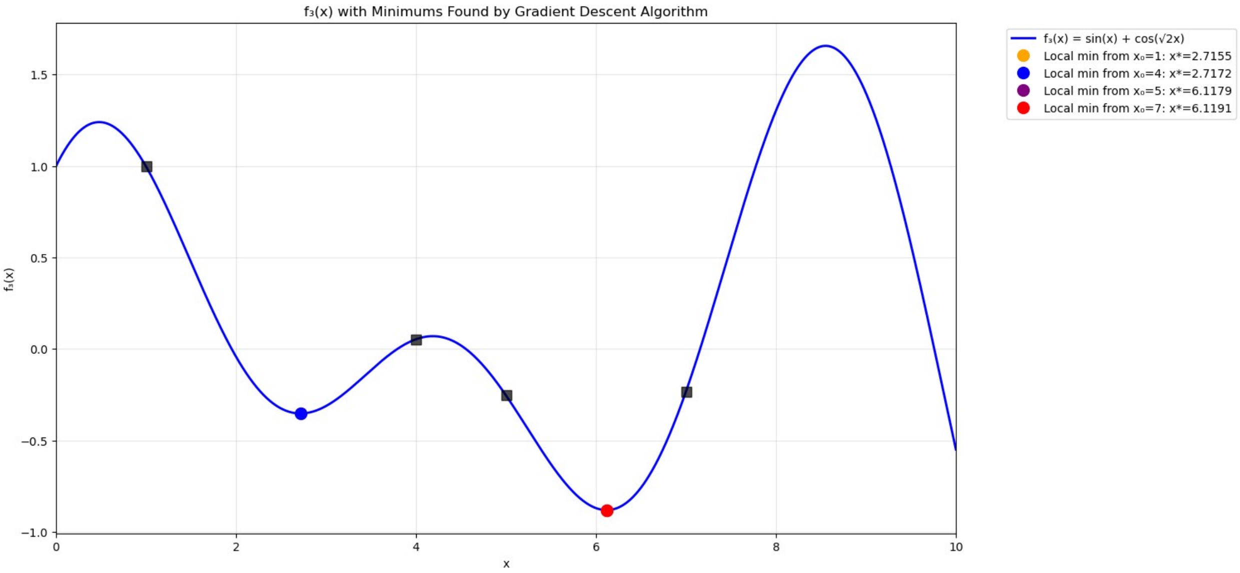
1.2 Question 2:

```
def gradient_descent(func, deriv_func, x0, alpha, epsilon,
iter_max=1000):
    """
    Gradient descent algorithm (using same as the one I had before)
    """
    x_current = x0
    iter_count = 0

    while iter_count < iter_max:
        gradient = deriv_func(x_current)
        x_new = x_current - alpha * gradient

        if abs(x_new - x_current) < epsilon:
            break

        x_current = x_new
        iter_count += 1
    return x_current, iter_count

def f3(x):
    """
    f3(x) = sin(x) + cos(√2 * x)
    """
    return np.sin(x) + np.cos(np.sqrt(2) * x)

def deriv_f3(x):
    """
    Derivative: f3'(x) = cos(x) - √2 * sin(√2 * x)
    """
    return np.cos(x) - np.sqrt(2) * np.sin(np.sqrt(2) * x)

def plot_opt(func, x_opts, starting_points, func_name="f₃(x) = sin(x)
+ cos(√2x)", x_range=(0, 10)):
    """
    Plotting function with several optimal/starting points
    """
    # create x values for plotting
    x_plot = np.linspace(x_range[0], x_range[1], 2000)
    y_plot = [func(x) for x in x_plot]

    # create the plot
    plt.figure(figsize=(14, 8))
    plt.plot(x_plot, y_plot, 'b-', linewidth=2, label=func_name)

    # plot optimal points with varying colors
    colors = ['orange', 'blue', 'purple', 'red']
    for i, x_opt in enumerate(x_opts):
        color = colors[i % len(colors)]
        plt.plot(x_opt, func(x_opt), 'o', color=color, markersize=10,
                label=f'Local min from x₀={starting_points[i]}:
x*={x_opt:.4f}')

    # ploting starting points
    for i, x0 in enumerate(starting_points):
        plt.plot(x0, func(x0), 's', color='black', markersize=8,
alpha=0.7)

    plt.xlabel('x')
    plt.ylabel('f₃(x)')
    plt.title('f₃(x) with Minimums Found by Gradient Descent
Algorithm')
    plt.grid(True, alpha=0.3)
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.xlim(x_range[0], x_range[1])
    plt.show()

def test_multiple_starting_points():
```

```python
    """
    Testing gradient descent on f3(x) with different starting points
    """
    # specifying the parameters and their values
    alpha = 0.1
    epsilon = 0.0001
    starting_points = [1, 4, 5, 7]

    # storing results
    results = []
    optimal_points = []

    for x0 in starting_points:
        x_opt, iterations = gradient_descent(f3, deriv_f3, x0, alpha,
epsilon)
        f3_opt = f3(x_opt)

        results.append((x0, x_opt, f3_opt, iterations))
        optimal_points.append(x_opt)

        # printing results for each starting point
        print(f"Starting point x₀ = {x0}:")
        print(f"  → Converged to x* = {x_opt:.6f}")
        print(f"  → Function value f₃(x*) = {f3_opt:.6f}")
        print(f"  → Iterations: {iterations}")
        print()

    # creating the plot
    plot_opt(f3, optimal_points, starting_points)

    return results

# running the test
if __name__ == "__main__":
    results = test_multiple_starting_points()
```

```
Starting point x₀ = 1:
  → Converged to x* = 2.715480
  → Function value f₃(x*) = -0.352360
  → Iterations: 65

Starting point x₀ = 4:
  → Converged to x* = 2.717223
  → Function value f₃(x*) = -0.352360
  → Iterations: 79

Starting point x₀ = 5:
  → Converged to x* = 6.117920
  → Function value f₃(x*) = -0.880520
  → Iterations: 49
```

```
Starting point x₀ = 7:
  → Converged to x* = 6.119081
  → Function value f₃(x*) = -0.880520
  → Iterations: 41
```



After I implemented the gradient descent function on f_3(x) with α = 0.1 and ε = 0.0001, the algorithm had different results depending on the starting point. Starting from x_0 = 1, the algorithm converged to x* = 2.715480 with function value f_3(x*) = -0.352360 after 65 iterations. Starting from x_0 = 4, it converged to x* = 2.717223 with the same function value f_3(x*) = -0.352360 after 79 iterations. Starting from x_0 = 5, the algorithm found x* = 6.117920 with f_3(x*) = -0.880520 after 49 iterations. Finally, starting from x_0 = 7, it converged to x* = 6.119081 with f_3(x*) = -0.880520 after 41 iterations. According to these results, the gradient descent algorithm found two local minimum points: one around x ≈ 2.716 with function value ≈ -0.352 and another around x ≈ 6.118 with function value ≈ -0.881. Even though the second local minimum point looks lower with a lower function value, gradient descent didn't determine it to be the global minimum without testing all possible starting points.

Part 2: Derivative Approximation for Functions of One Variable

Question 1:

```
def approx_deriv(f, x, h=1e-8):
    """
    Approximating the derivative of function f at point x using the
difference quotient rule
```

```
    Args:
        f (function): function
        x (float): point at which to approximate the derivative
        h (float): step size of 1e-8

    Returns:
        float: approximate derivative of f at point x
    """
    return (f(x + h) - f(x)) / h

# Test points
x_values = [0, 1, 2, 3]

# Compare approximate and exact derivatives (expected vs. actual)
print("Testing derivative approximation:\n")
for x in x_values:
    approx1 = approx_deriv(f1, x)
    exact1 = deriv_f1(x)
    print(f"f1 at x={x}: approx = {approx1:.5f}, exact = {exact1}")

print()

for x in x_values:
    approx2 = approx_deriv(f2, x)
    exact2 = deriv_f2(x)
    print(f"f2 at x={x}: approx = {approx2:.5f}, exact = {exact2}")
```

```
Testing derivative approximation:

f1 at x=0: approx = 0.00000, exact = 0
f1 at x=1: approx = 2.00000, exact = 2
f1 at x=2: approx = 4.00000, exact = 4
f1 at x=3: approx = 6.00000, exact = 6

f2 at x=0: approx = -2.00000, exact = -2
f2 at x=1: approx = 0.00000, exact = 0
f2 at x=2: approx = 2.00000, exact = 2
f2 at x=3: approx = 4.00000, exact = 4
```

Question 2:

```
def gradient_descent_approx(f, x0, alpha=0.1, epsilon=0.001,
iter_max=1000):
    """
    Gradient descent using derivative approximation
    """
    x = x0
    iter_count = 0
    # Using same logic as from the previously written gradient descent
    while iter_count < iter_max:
        grad_approx = approx_deriv(f, x)
        x_new = x - alpha * grad_approx
        if abs(x_new - x) < epsilon:
            break
        x = x_new
        iter_count += 1
    return x, iter_count

# Testing f1
x1, iters1 = gradient_descent_approx(f1, x0=3)
print(f"f1 using approx derivative: x = {x1}, iterations = {iters1}")

# Testing f2
x2, iters2 = gradient_descent_approx(f2, x0=3)
print(f"f2 using approx derivative: x = {x2}, iterations = {iters2}")
```

```
f1 using approx derivative: x = 0.004642270164397316, iterations = 29
f2 using approx derivative: x = 1.0048356995609993, iterations = 27
```

I modified the gradient_descent() function from earlier to use the approx_deriv() function instead of the exact derivative. When I tested it on f1(x) = x^2 it returned x = 0.00464227 in 29 iterations and for f2(x) = x^2 - 2x + 3, it returned x = 1.00483570 in 27 iterations. I found these values to be similar to the results from the earlier gradient descent algorithm using exact derivatives, which returned x = 0.00464228 for f1 and x = 1.00483570. While there is a small difference, I expected this because of the numerical approximation as well as differences in rounding. But this confirms that this method for approximating derivatives is accurate and works in gradient descent.

Part 3: Gradient Descent for Functions of Two Variables

Question 1:

```
def approx_partial_derivs(f, x, y, h=1e-5):
    """
    Approximating partial derivatives of f(x, y) at point (x, y)

    Args:
        f (function): function of two variables f(x, y)
        x (float): x-coordinate
        y (float): y-coordinate
        h (float): step size = 1e-5

    Returns:
        (df_dx, df_dy): partial derivatives at (x, y)
    """
    df_dx = (f(x + h, y) - f(x, y)) / h
    df_dy = (f(x, y + h) - f(x, y)) / h
    return df_dx, df_dy

# Function for testing
def test_f(x, y):
    return x**2 + y**2

# At point (3, 3), we expect it to print something close to (6,6)
print(approx_partial_derivs(test_f, 3, 3))
```

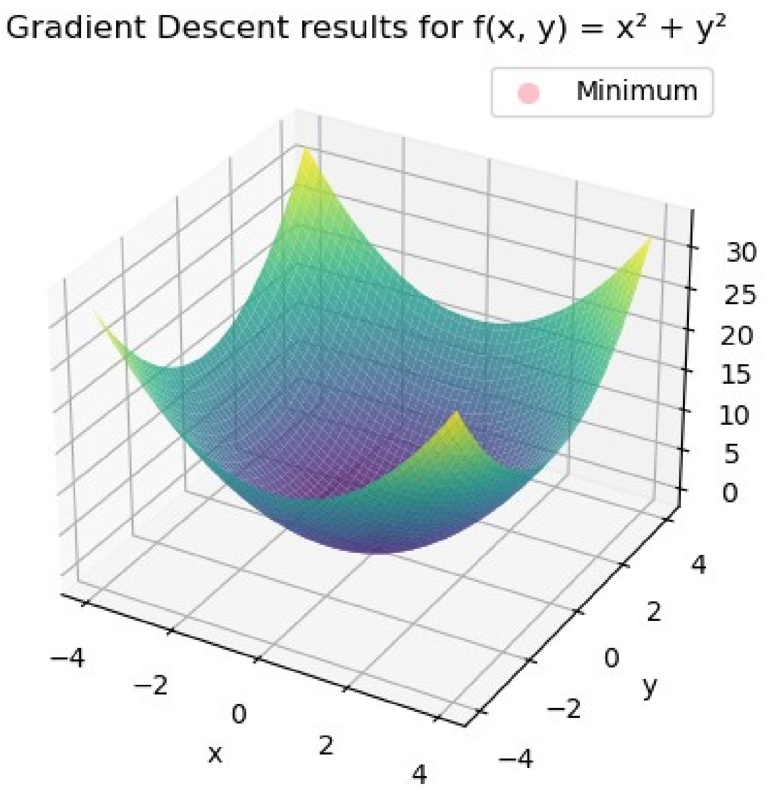
```
(6.000009999951316, 6.000009999951316)
```

Question 2:

```
# importing to be able to create a 3D graph/plot
from mpl_toolkits.mplot3d import Axes3D

# Function for optimizing f(x, y) = x^2 + y^2
def f_2d(x, y):
    """
    Computeing the function f(x, y) = x^2 + y^2.

    Args:
        x (float): x-coordinate
        y (float): y-coordinate

    Returns:
        float: values of the function
    """
    return x**2 + y**2

# approximating partial derivatives of 2-variable functions
def approx_partial_derivs(f, x, y, h=1e-5):
    """
    approximating the partial derivatives of function f at (x, y).

    Args:
        f (function): function f(x, y)
        x (float): for evaluating df/dx
        y (float): for evaluating df/dy
        h (float): step size

    Returns:
        tuple: (df/dx, df/dy) at point (x, y)
    """
    df_dx = (f(x + h, y) - f(x, y)) / h
    df_dy = (f(x, y + h) - f(x, y)) / h
    return df_dx, df_dy

# Gradient descent but with 2 variables
```

```
def gradient_descent_2d(f, x0, y0, alpha=0.1, epsilon=0.001,
iter_max=1000):
    """
    gradient descent to find the minimum of a 2-variable function

    Args:
        f (function): function f(x, y)
        x0 (float): x value (initial)
        y0 (float): y value (initial)
        alpha (float): learning rate
        epsilon (float): tolerance
        iter_max (int): Maximum number of iterations

    Returns:
        tuple: (x_min, y_min, iteration_count)
    """
    x, y = x0, y0
    iter_count = 0

    while iter_count < iter_max:
        # calculate partial derivatives at the current (x, y)
        df_dx, df_dy = approx_partial_derivs(f, x, y)

        # Gradient descent update rule from before
        x_new = x - alpha * df_dx
        y_new = y - alpha * df_dy

        # Checking if updates are small enough to break the loop
(checking if converged)
        if abs(x_new - x) < epsilon and abs(y_new - y) < epsilon:
            break

        # Update (x, y), increment, and continue with loop
        x, y = x_new, y_new
        iter_count += 1

    return x, y, iter_count

def plot_surface_with_min(f, x_opt, y_opt):
    """
    3D plot of a two-variable function f(x, y).

    Args:
        f (function): function to plot
        x_opt (float): x-coordinate of minimum
        y_opt (float): y-coordinate of minimum
    """
    # Creating a grid of x and y values
    X = np.linspace(-4, 4, 100)
    Y = np.linspace(-4, 4, 100)
    X, Y = np.meshgrid(X, Y)
    Z = f(X, Y)

    # Setting up 3D plot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Plotting the surface
    ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)

    # Plotting the minimum point
    ax.scatter(x_opt, y_opt, f(x_opt, y_opt), color='pink', s=50,
label='Minimum')

    # Labeling and style
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('f(x, y)')
    ax.set_title('Gradient Descent results for f(x, y) = x² + y²')
    ax.legend()
    plt.show()

# Running gradient descent starting at (3, 3)
x_min, y_min, iters = gradient_descent_2d(f_2d, x0=3, y0=3)
print(f"Minimum found at x = {x_min}, y = {y_min} in {iters}
iterations")

# Plotting the result
plot_surface_with_min(f_2d, x_min, y_min)
```

```
Minimum found at x = 0.004637282884370288, y = 0.004637282884370288 in
29 iterations
```



Part 4: Cost Function to Evaluate the Performance of a Predictive Model

Question 1:

```
def load_data(filename):
    """
    Load the data into two arrays.

    Args:
        filename (str): string representing the name of the file 
containing the data (x,y)

    Returns:
        Array containing x values, array containing y values
    """
    data = np.loadtxt(filename)
    return data[:, 0], data[:, 1]

x, y = load_data('data_chol_dias_pressure.txt')
print("x:", x)
print("y:", y)
```

```
x: [100.         110.52631579 121.05263158 131.57894737 142.10526316
 152.63157895 163.15789474 173.68421053 184.21052632 194.73684211
 205.26315789 215.78947368 226.31578947 236.84210526 247.36842105
 257.89473684 268.42105263 278.94736842 289.47368421 300.        ]
y: [58.94864536 61.42955826 65.66140653 68.15337824 67.46602089
69.37306594
 71.13458153 75.79139407 75.65104923 79.03581803 82.94976901
84.33271918
 86.13602117 87.3336793  90.14540532 89.99395303 94.16292736
97.16380543
 95.97742045 99.45334313]
```

Question 2:

```python
def cost_function(a, b, x, y):
    """
    Calculate the total squared error cost function

    Args:
        a (float): Slope
        b (float): Intercept
        x (np.ndarray): cholesterol values
        y (np.ndarray): blood pressure values

    Returns:
        float: Value of the cost function g(a, b)
    """
    predictions = a * x + b
    errors = predictions - y
    return np.sum(errors ** 2)
```

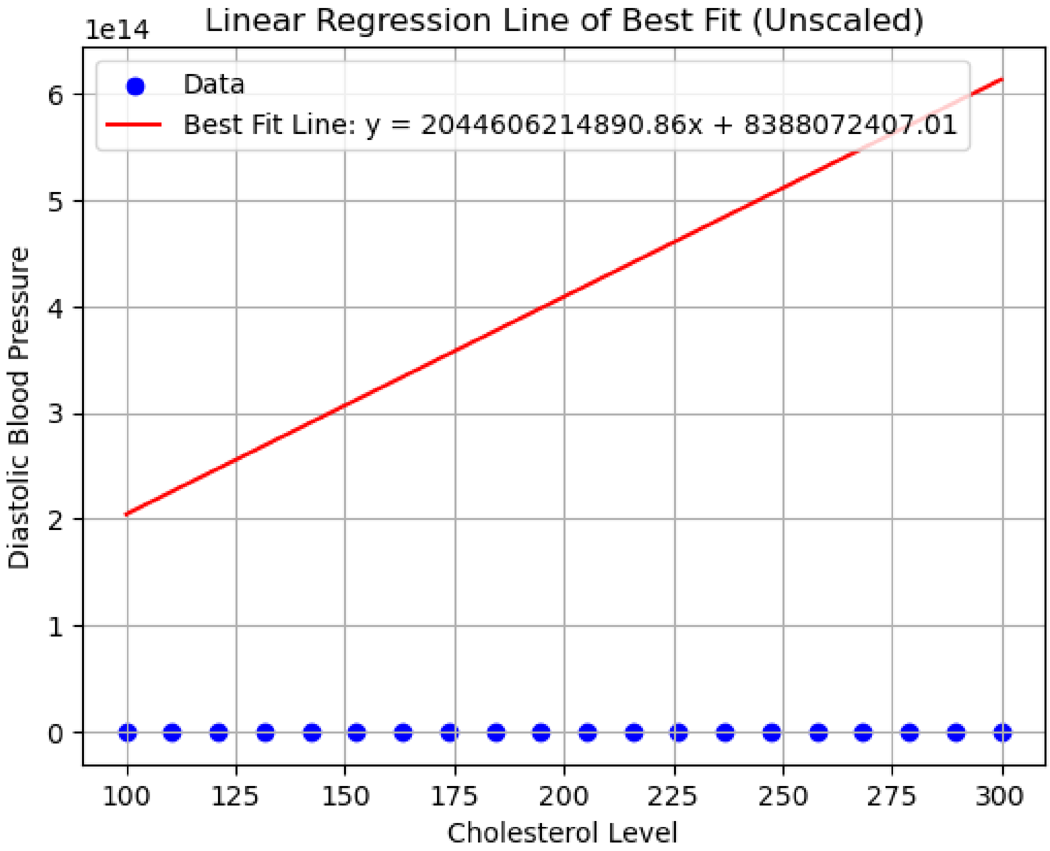
Question 3:

```python
# g(a, b) becomes f(x, y) = g(x, y)
def g_wrapper(a, b):
    return cost_function(a, b, x, y)

# Running gradient descent
a_star, b_star, iters = gradient_descent_2d(g_wrapper, x0=0, y0=0,
alpha=0.01, epsilon=0.001)

# (unscaled)
print(f"Optimal parameters: a = {a_star}, b = {b_star}, in {iters}
iterations")
```

```
Optimal parameters: a = 2044606214890.8635, b = 8388072407.005877, in
3 iterations
```

Question 4:

```python
def plot_linear_fit(x, y, a, b):
    """
    Plotting data points and line of best fit at y = ax + b.

    Args:
        x (np.ndarray): cholesterol values
        y (np.ndarray): blood pressure values
        a (float): slope
        b (float): intercept
    """
    plt.figure()
    plt.scatter(x, y, color='blue', label='Data')
    y_pred = a * x + b
    plt.plot(x, y_pred, color='red', label=f'Best Fit Line: y =
{a:.2f}x + {b:.2f}')
    plt.title('Linear Regression Line of Best Fit (Unscaled)')
    plt.xlabel('Cholesterol Level')
    plt.ylabel('Diastolic Blood Pressure')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_linear_fit(x, y, a_star,b_star)
```

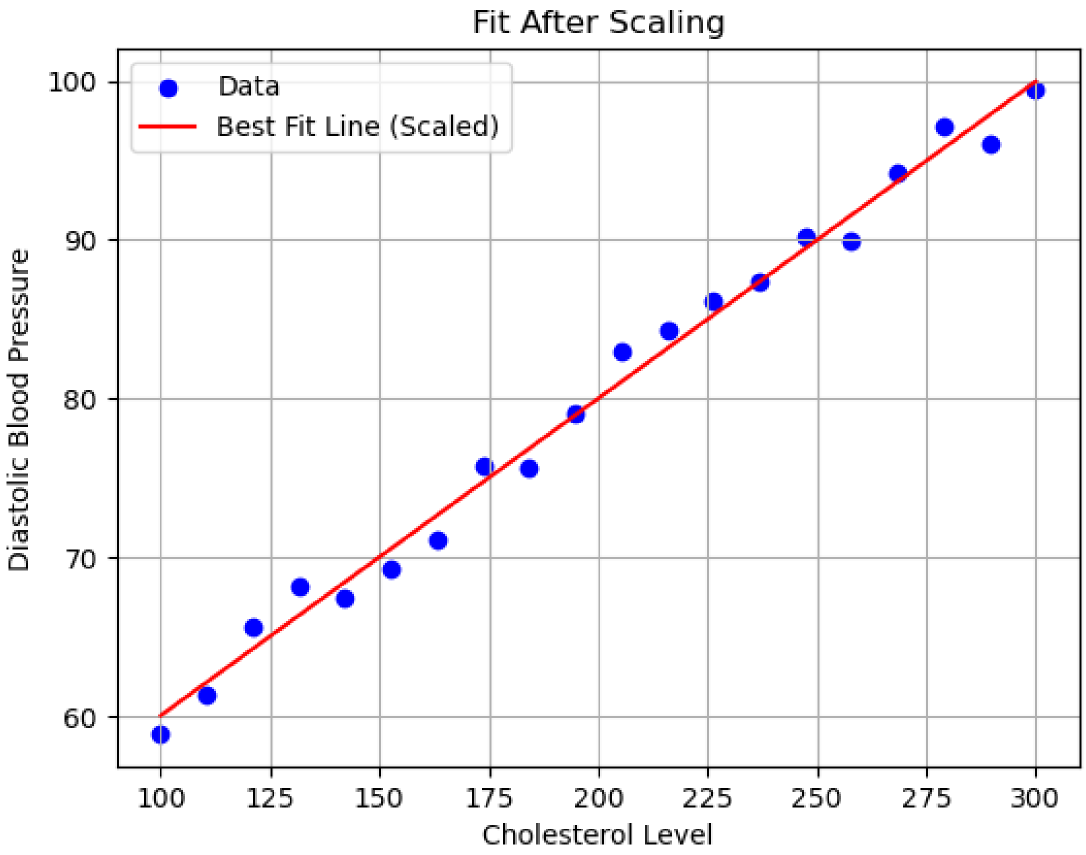


Question 5:

```
# Scale x and y using hint from problem 5
x_mean, x_std = np.mean(x), np.std(x)
y_mean, y_std = np.mean(y), np.std(y)

x_scaled = (x - x_mean) / x_std
y_scaled = (y - y_mean) / y_std

# Cost function (scaled version)
def cost_function_scaled(a, b, x, y):
    predictions = a * x + b
    errors = predictions - y
    return np.sum(errors ** 2)

# Scaled version
def g_wrapper_scaled(a, b):
    return cost_function_scaled(a, b, x_scaled, y_scaled)

# Using gradient descent on scaled values
a_scaled, b_scaled, iters_scaled =
gradient_descent_2d(g_wrapper_scaled, x0=0, y0=0, alpha=0.01)
print(f"Scaled Fit: a = {a_scaled}, b = {b_scaled}, iterations =
{iters_scaled}")
```

```
# Calculating the prediction with scaled values
def predict_unscaled(x_input, a_s, b_s, x_mean, x_std, y_mean, y_std):
    x_norm = (x_input - x_mean) / x_std
    y_norm = a_s * x_norm + b_s
    return y_norm * y_std + y_mean

# Plotting everything (scaled)
def plot_scaled_fit_on_original_data(x_orig, y_orig, a_s, b_s, x_mean,
x_std, y_mean, y_std):
    y_pred = predict_unscaled(x_orig, a_s, b_s, x_mean, x_std, y_mean,
y_std)

    plt.figure()
    plt.scatter(x_orig, y_orig, color='blue', label='Data')
    plt.plot(x_orig, y_pred, color='red', label='Best Fit Line
(Scaled)')
    plt.title('Fit After Scaling')
    plt.xlabel('Cholesterol Level')
    plt.ylabel('Diastolic Blood Pressure')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_scaled_fit_on_original_data(x, y, a_scaled, b_scaled, x_mean,
x_std, y_mean, y_std)
```

```
Scaled Fit: a = 0.992960086926481, b = -4.989115953701883e-06,
iterations = 12
```



No, I did not find it easy to find the correct values for the initial parameters for gradient descent to converge on the unscaled data. The unscaled version produced very large and unrealistic parameter values and converged in only 3 iterations, which shows that the algorithm stopped early on rather than finding the actual minimum. I think this is happening because the features have different scales. On one hand, the cholesterol levels range from 100-300 while blood pressure ranges from 58-99. This scale difference must be what is causing gradient descent to oscillate and overshoot the minimum. By scaling (subtracting mean and dividing by standard deviation), the variables were normalized and the model was more accurate, with the line fitting the data better as shown in the graph above.

Part 5: Non-linear Data

Question 1:

```
def load_data(filename):
    """
    Loading the data into two arrays
    Args:
        filename: string representing the name of the file containing
the data (x,y)
    Return:
        array containing the values of x
        array containing the values of y
    """
    data = np.loadtxt(filename)
    return data[:, 0], data[:, 1]

# Loading the second (non-linear) dataset
x_nonlin, y_nonlin = load_data('data_chol_dias_pressure_non_lin.txt')
```

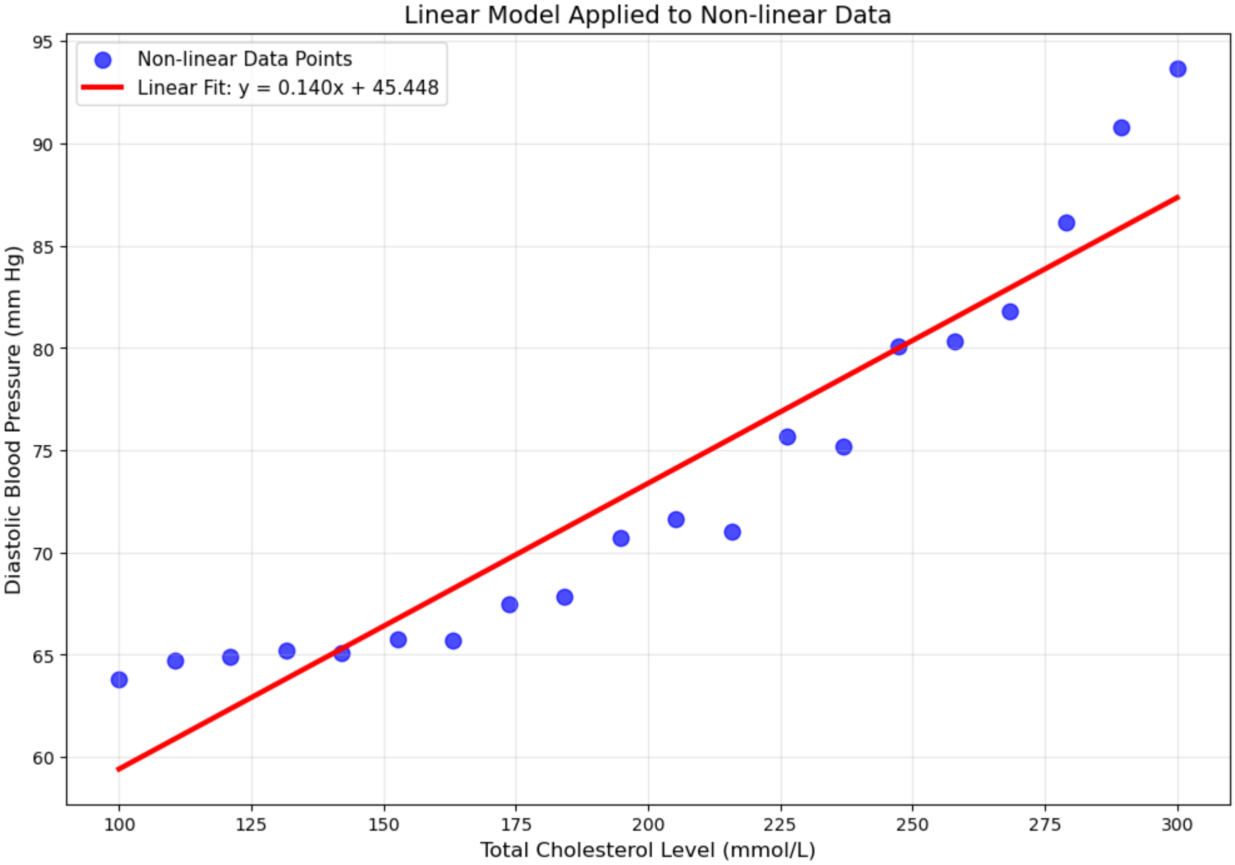
Question 2:

```
def cost_function_nonlinear(a, b, x, y):
    """
    Cost function g(a, b) for the non linear dataset
    """
    predictions = a * x + b
    errors = predictions - y
    return np.sum(errors ** 2)

def g_wrapper_nonlinear(a, b):
    return cost_function_nonlinear(a, b, x_nonlin, y_nonlin)

# Calculating values
a_star_nonlin, b_star_nonlin, iterations_nonlin = gradient_descent_2d(
    g_wrapper_nonlinear,
    x0=0,
    y0=0,
    alpha=0.001,
    epsilon=0.001
)

# Printing results
print(f"Results for non-linear dataset:")
print(f"Optimal parameters: a* = {a_star_nonlin:.6f}, b* =
{b_star_nonlin:.6f}")
print(f"Iterations: {iterations_nonlin}")
print(f"Final cost: {cost_function_nonlinear(a_star_nonlin,
b_star_nonlin, x_nonlin, y_nonlin):.6f}")

# Doing the same but with scaling for better convergence
x_mean, x_std = np.mean(x_nonlin), np.std(x_nonlin)
y_mean, y_std = np.mean(y_nonlin), np.std(y_nonlin)
x_scaled = (x_nonlin - x_mean) / x_std
y_scaled = (y_nonlin - y_mean) / y_std

def g_wrapper_scaled(a, b):
    return cost_function_nonlinear(a, b, x_scaled, y_scaled)

a_scaled, b_scaled, iters_scaled =
gradient_descent_2d(g_wrapper_scaled, x0=0, y0=0, alpha=0.01)
```

```
# Calculating back to original scale
a_final = a_scaled * y_std / x_std
b_final = y_mean - a_final * x_mean

print(f"\nWith scaling:")
print(f"Final parameters: a* = {a_final:.6f}, b* = {b_final:.6f}")
print(f"Iterations: {iters_scaled}")

Results for non-linear dataset:
Optimal parameters: a* = -3273060183882.839844, b* = -
107365726455.841446
Iterations: 4
Final cost: 93625231915564706771299477749 76.000000

With scaling:
Final parameters: a* = 0.139685, b* = 45.447663
Iterations: 12
```

Question 3:

```
def plot_linear_fit_nonlinear_data(x, y, a_star, b_star):
    """
    Plotting the line y = a*x + b* along with the non-linear dataset
values

    Args:
        x: cholesterol data
        y: blood pressure data
        a_star: slope from gradient descent
        b_star: intercept from gradient descent
    """
    plt.figure(figsize=(12, 8))

    # Plotting the data points
    plt.scatter(x, y, color='blue', s=80, alpha=0.7, label='Non-linear
Data Points')

    # linear fit line
    x_line = np.linspace(min(x), max(x), 100)
    y_line = a_star * x_line + b_star

    # Plotting the linear fit
    plt.plot(x_line, y_line, color='red', linewidth=3,
             label=f'Linear Fit: y = {a_star:.3f}x + {b_star:.3f}')

    # Formatting, legend
    plt.xlabel('Total Cholesterol Level (mmol/L)', fontsize=12)
    plt.ylabel('Diastolic Blood Pressure (mm Hg)', fontsize=12)
    plt.title('Linear Model Applied to Non-linear Data', fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.legend(fontsize=11)
    plt.show()

# Using results from Problem 2
a_star = 0.139685
b_star = 45.447663

# Plotting the linear fit on non-linear data
plot_linear_fit_nonlinear_data(x_nonlin, y_nonlin, a_star, b_star)
```



I think the plotted linear line does not perform well on this dataset. The straight line does not follow the pattern of the data because it's non-linear and follows more of a curve than a line. The line is performing differently because the relationship between cholesterol and blood pressure has changed between the two datasets. The first dataset was linear, but this one is non-linear and can't be fitted with a straight line y = ax + b.

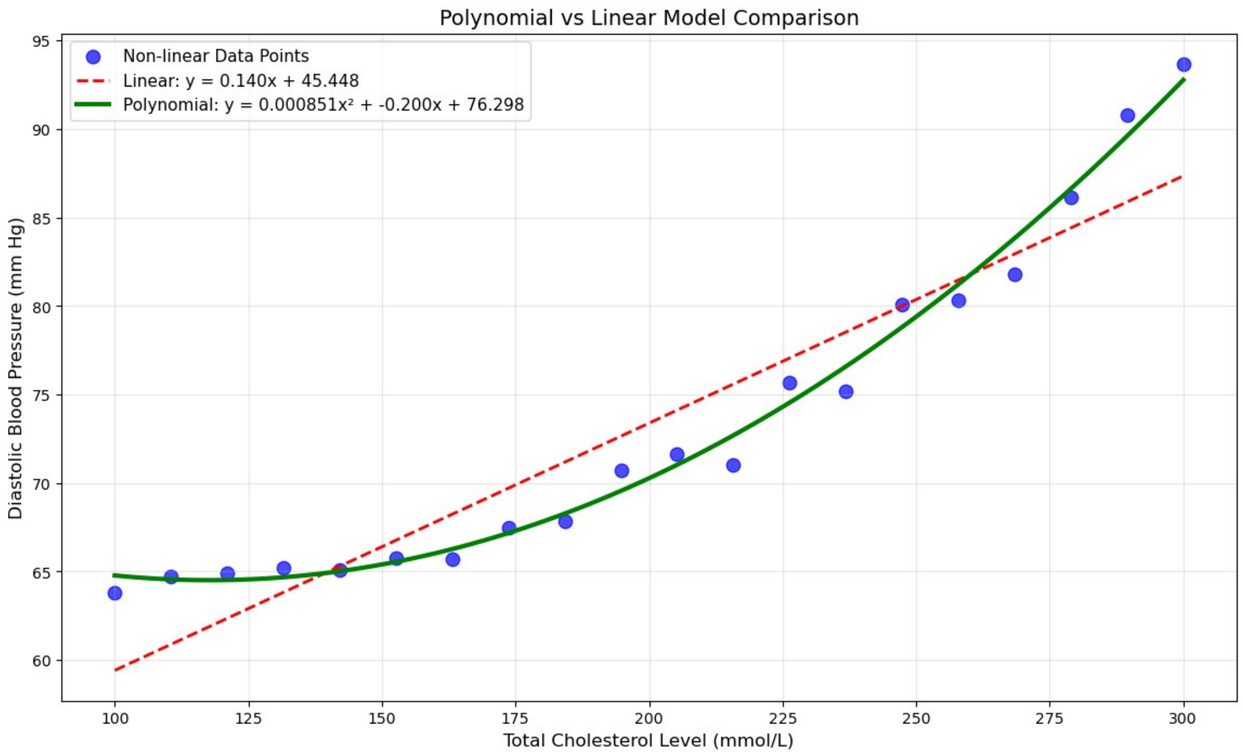
Question 4:

I think that the linear function f(x) = ax + b for this dataset is not suitable because the data clearly shows a non-linear and curved relationship rather than a straight-line pattern. When looking at the plot above, the blue data points show an upward curving trend where blood pressure increases at an accelerating rate with higher cholesterol levels. To improve performance, a modification I would make it implementing a polynomial model f(x) = ax² + bx + c because it can model the curve in the data.

```python
def cost_function_polynomial(a, b, c, x, y):
    """
    Cost function for quadratic model: y = ax^2 + bx + c

    Args:
        a: coefficient of x^2 term
        b: coefficient of x term
        c: constant term
        x: cholesterol values
        y: blood pressure values

    Returns:
        Sum of squared errors
    """
    predictions = a * x**2 + b * x + c
    errors = predictions - y
    return np.sum(errors ** 2)

def approximate_partial_derivative_a_poly(func, a, b, c, h=1e-8):
    """
    Partial derivative with respect to 'a' for polynomial function
    """
    return (func(a + h, b, c) - func(a, b, c)) / h

def approximate_partial_derivative_b_poly(func, a, b, c, h=1e-8):
    """
    Partial derivative with respect to 'b' for polynomial function
    """
    return (func(a, b + h, c) - func(a, b, c)) / h

def approximate_partial_derivative_c_poly(func, a, b, c, h=1e-8):
    """
    Partial derivative with respect to 'c' for polynomial function
    """
    return (func(a, b, c + h) - func(a, b, c)) / h

def gradient_descent_3d(func, x0, y0, z0, alpha, epsilon=0.001,
iter_max=1000):
    """
    Gradient descent algorithm for functions of three variables (a, b,
c)

    Args:
        func: function to minimize
        x0, y0, z0: initial parameters
        alpha:lLearning rate
        epsilon: tolerance
        iter_max: Max iterations

    Returns:
        Optimal parameters and iteration count
    """
    x_current, y_current, z_current = x0, y0, z0
    iter_count = 0

    while iter_count < iter_max:
        # Calculate partial derivatives
        grad_x = approximate_partial_derivative_a_poly(func,
x_current, y_current, z_current)
        grad_y = approximate_partial_derivative_b_poly(func,
x_current, y_current, z_current)
        grad_z = approximate_partial_derivative_c_poly(func,
x_current, y_current, z_current)

        # Update the parameters
        x_new = x_current - alpha * grad_x
        y_new = y_current - alpha * grad_y
        z_new = z_current - alpha * grad_z

        # Check for convergence
        if (abs(x_new - x_current) < epsilon and
            abs(y_new - y_current) < epsilon and
            abs(z_new - z_current) < epsilon):
            break

        x_current, y_current, z_current = x_new, y_new, z_new
        iter_count += 1

    return x_current, y_current, z_current, iter_count

# Applying polynomial regression to non-linear data
print("Implementing Polynomial Regression (Quadratic Model)")
print("Model: y = ax² + bx + c")

# Scaling the data for better convergence (as I did in previous
sections)
x_mean, x_std = np.mean(x_nonlin), np.std(x_nonlin)
y_mean, y_std = np.mean(y_nonlin), np.std(y_nonlin)
x_scaled = (x_nonlin - x_mean) / x_std
y_scaled = (y_nonlin - y_mean) / y_std

def cost_function_poly_scaled(a, b, c):
    return cost_function_polynomial(a, b, c, x_scaled, y_scaled)

a_poly, b_poly, c_poly, iterations_poly = gradient_descent_3d(
    cost_function_poly_scaled,
    x0=0,
    y0=0,
    z0=0,
    alpha=0.01,
    epsilon=0.001
)

print(f"Scaled polynomial parameters:")
print(f"a = {a_poly:.6f}, b = {b_poly:.6f}, c = {c_poly:.6f}")
print(f"Iterations: {iterations_poly}")

# Transforming back to the original scale
# For polynomial: y_scaled = a*x_scaled^2 + b*x_scaled + c
# Need to convert back to : y = a'*x^2 + b'*x + c'
def transform_poly_to_original_scale(a_s, b_s, c_s, x_mean, x_std,
y_mean, y_std):
    """
    Transform scaled polynomial parameters back to original scale
    """
    # y = y_std * (a_s * ((x - x_mean)/x_std)^2 + b_s * ((x -
x_mean)/x_std) + c_s) + y_mean
    a_orig = a_s * y_std / (x_std**2)
    b_orig = b_s * y_std / x_std - 2 * a_s * y_std * x_mean /
(x_std**2)
    c_orig = (a_s * (x_mean**2) / (x_std**2) - b_s * x_mean / x_std +
c_s) * y_std + y_mean

    return a_orig, b_orig, c_orig

a_final_poly, b_final_poly, c_final_poly =
transform_poly_to_original_scale(
    a_poly, b_poly, c_poly, x_mean, x_std, y_mean, y_std
)


# Calculating final cost
final_cost_poly = cost_function_polynomial(a_final_poly, b_final_poly,
c_final_poly, x_nonlin, y_nonlin)
print(f"Final cost (polynomial): {final_cost_poly:.6f}")

# Comparing with the linear model cost
linear_cost = cost_function_nonlinear(a_final, b_final, x_nonlin,
y_nonlin)  # From Problem 2
print(f"Linear model cost: {linear_cost:.6f}")
print(f"Improvement: {((linear_cost - final_cost_poly) / linear_cost *
100):.2f}% reduction in cost")

def plot_polynomial_vs_linear(x, y, a_lin, b_lin, a_poly, b_poly,
c_poly):
    """
    Plotting both linear and polynomial models for comparison in
```

```
accuracy
    """
    plt.figure(figsize=(14, 8))

    # Plotting data points
    plt.scatter(x, y, color='blue', s=80, alpha=0.7, label='Non-linear
Data Points')

    # Creating smooth line for plotting
    x_line = np.linspace(min(x), max(x), 100)

    # Linear model plot
    y_linear = a_lin * x_line + b_lin
    plt.plot(x_line, y_linear, color='red', linewidth=2,
linestyle='--',
             label=f'Linear: y = {a_lin:.3f}x + {b_lin:.3f}')

    # Polynomial model plot
    y_poly = a_poly * x_line**2 + b_poly * x_line + c_poly
    plt.plot(x_line, y_poly, color='green', linewidth=3,
             label=f'Polynomial: y = {a_poly:.6f}x² + {b_poly:.3f}x +
{c_poly:.3f}')

    plt.xlabel('Total Cholesterol Level (mmol/L)', fontsize=12)
    plt.ylabel('Diastolic Blood Pressure (mm Hg)', fontsize=12)
    plt.title('Polynomial vs Linear Model Comparison', fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.legend(fontsize=11)
    plt.show()

# Plotting comparison
plot_polynomial_vs_linear(x_nonlin, y_nonlin, a_final, b_final,
                         a_final_poly, b_final_poly, c_final_poly)
```

```
Implementing Polynomial Regression (Quadratic Model)
Model: y = ax² + bx + c
Scaled polynomial parameters:
a = 0.347842, b = 0.943177, c = -0.345423
Iterations: 29
Final cost (polynomial): 18.715944
Linear model cost: 179.239901
Improvement: 89.56% reduction in cost
```



Part 6: Report

CS 4100 Project Report: Gradient Descent Implementation and Analysis

For this project, I began by implementing gradient descent for simple functions like $f(x) = x^2$. The basic idea was to start at some point, look at the slope, and take a step in the opposite direction to go downhill towards the minimum point. The algorithm I wrote followed these steps: start at an initial point x_0, calculate the slope (derivative) at the current point, then move in the opposite direction using x_new = x_old - α × slope, and keep repeating this until the change becomes very small. As an example test case, testing on $f_1(x) = x^2$ with starting point $x_0 = 3$, the algorithm found the minimum at $x = 0$, took a reasonable amount of time/iterations to converge, and showed that different starting points ($x_0 = 3$ vs $x_0 = -3$) had the same result because the function only had one minimum.

Then, I moved on to functions with two variables like g(a,b) which followed the same logic. Instead of one slope, I calculated two partial derivatives, one for each variable direction. Then, for the 3-variable polynomial case (a, b, c), I followed the same pattern to handle three partial derivatives. I organized my code into small, focused functions, each one having a docstring explaining the purpose of the function, as well as arguments taken in and what the function is suppose to return. This included gradient_descent() for single variables, gradient_descent_2d() for two variables, gradient_descent_3d() for three variables, and approximate_derivative() for numerical derivative calculation, and separate functions for each test function. I organized my code this way in order to make it easier to test different parts and then reuse code snippets in later sections.

I tested my code implementations on several types of functions to make sure they were accurate and consistent. For simpler functions like $f_1(x) = x^2$ and $f_2(x) = x^2 - 2x + 3$, I was able to check for accuracy since they both converged to the expected minimum values ($x = 0$ for f_1, $x = 1$ for f_2). For more complex functions in later sections like $f_3(x) = \sin(x) + \cos(\sqrt{2}x)$ with multiple local minimums, the results showed that different starting points led to different local minimum values. For the datasets provided (one linear, one non-linear), I tested the algorithm and verified accuracy by comparing with known solutions, plotting results to see if its visually accurate and making sense, testing multiple starting points, and using derivative approximation and comparing with exact derivatives.

Through my testing process, I learned how each parameter affects the performance of the gradient descent algorithm. For starting points, simpler quadratic functions showed that $x_0 = 3$ and $x_0 = -3$ both led to the same minimum value, which made sense because they have only one minimum. However, for more complicated functions like $f_3(x)$, different starting points led to completely different results, with $x_0 = 1$ finding a minimum at $x \approx 2.716$, but $x_0 = 5$ finding a minimum at $x \approx 6.118$. I learned that for functions with multiple minimums, the starting point can determine which minimum will be found.

I also tested the learning rates to see how the results changed. When $\alpha = 1$ (a large value), the algorithm hit the maximum iterations without ever converging because the steps were too big, causing it to overshoot the minimum. On the other hand, when $\alpha = 0.001, 0.0001$ (too small of a value), the algorithm converged correctly but took a longer time, and required significantly more iterations. When $\alpha = 0.1$, it provided a good balance with fast convergence and more accurate results. I learned that the learning rate is more of a trial-and-error process, and I have to make sure that the value isn't too big to overshoot, but also not too small to be slow and inefficient.

I also tested the tolerance values. For example, $\varepsilon = 0.1$ gave $x = 0.403$ in 9 iterations, $\varepsilon = 0.01$ gave $x = 0.043$ in 19 iterations, and $\varepsilon = 0.0001$ gave $x = 0.0005$ in 39 iterations. I learned that a smaller tolerance value will give more accurate results but the tradeoff is that it is more computationally expensive (takes more time). When it came to working with the two datasets, I learned the importance of scaling. Without scaling the data, the algorithm produced unrealistic results like a = 2 trillion, b = 8 billion and failed to converge properly. The line also did not fit the data at all and the graph visually did not look accurate. With scaling (normalizing data by subtracting the mean and dividing by the standard deviation) the algorithm found more reasonable parameters like $a = 0.14$, $b = 45.45$.

For the first dataset I used, the linear model $f(x) = ax + b$ worked well because the data points followed a straight-line pattern, so it made sense to use a linear model. However, for the second dataset, the same linear model failed and did not visually make sense because the data showed a curved pattern, and so a straight line could not fit the curved data. In part 5 I chose to implement a polynomial model $f(x) = ax^2 + bx + c$ which resulted in a 89.56% reduction in prediction error. The curved line of a polynomial function followed the data pattern much better. I learned that the type of function you choose to implement should match the pattern as seen when visualizing the dataset you are working with. A linear model worked for the first dataset because the pattern was linear, but it didn't work for a curved pattern, and the same thing can be said for applying a polynomial model to a linear dataset (it wouldn't work because the model does not match the pattern of the data).

Overall, the things I learned from this project include implementing gradient descent, derivative approximation, data scaling, tuning of parameters like learning rate, tolerance, and starting points, algorithms for finding local minimums, and model selection for fitting data.