# Project 2: Gradient descent

*Meriem Lmoubariki*

*Adam Mohammed Sterhaltou*

*Dr. Tajjeeddine Rachidi*

*Al Akhawayn University in Ifrane*

*October 03, 2024*

## Table of content

# Introduction

In this report, we're going to dive into a method called gradient descent, which is like a map for finding the lowest points on a landscape, but for math functions! Imagine you're in a hilly area and you want to get to the bottom of a valley; gradient descent helps you figure out which way to go.

We'll play around with this method using different settings to see how it behaves. We'll start simple by just changing how big our steps are when we walk down our mathematical hills. Then, we'll try out a cool advanced technique called the ADAM optimizer, which is smarter because it adjusts our step size on the fly to get us to the bottom more smoothly and quickly.

By experimenting and tweaking how gradient descent works, we’ll learn the best ways to use it to find these low points quickly and effectively. This is super useful in areas like machine learning, where finding these points can help computers learn from data better!

## Assignment Design

**We have used the following libraries:**

**NumPy**: Helps with complex mathematical operations.

**Matplotlib**: Allows us to create graphs to see what's happening.

We started by writing a simple Python script to implement the gradient descent algorithm. It is composed of those parts:

- **Function Definition**: First, we needed a math function to minimize.
- **Gradient Calculation**: We calculated the gradient of the function, which tells us the slope. Knowing the slope helps us decide which way to go to get to the bottom fastest.
- **Update Rule**: We then used the gradient to update our position, taking small steps downhill in the direction that reduces our function's value the most.

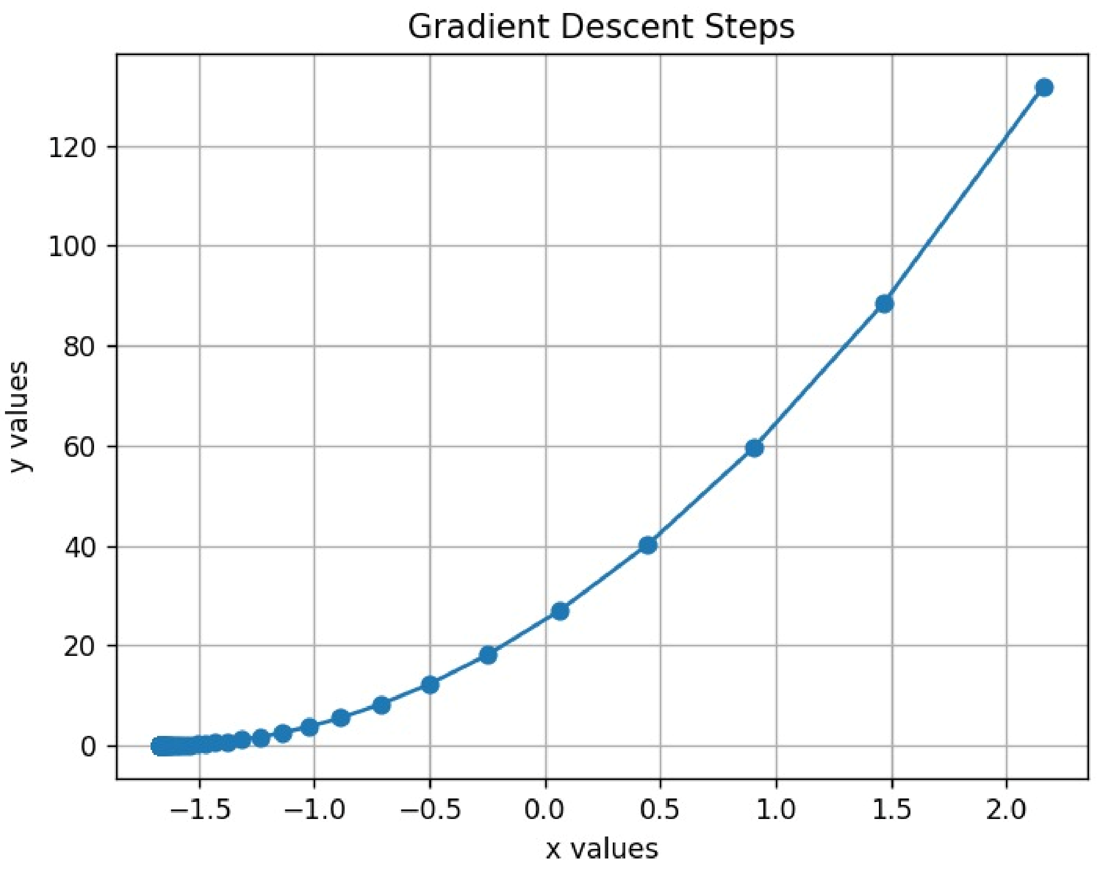
**Parameters Used:**

- **Initial Point**: We started at a specific point on our function and began walking downhill from there.
- **Learning Rate**: This is like deciding how big a step to take. A larger step might get us there faster, but it's also riskier because we might overshoot the lowest point.
- **Precision**: This tells us when to stop. If our steps are smaller than this precision level, it means we're close enough to the bottom.

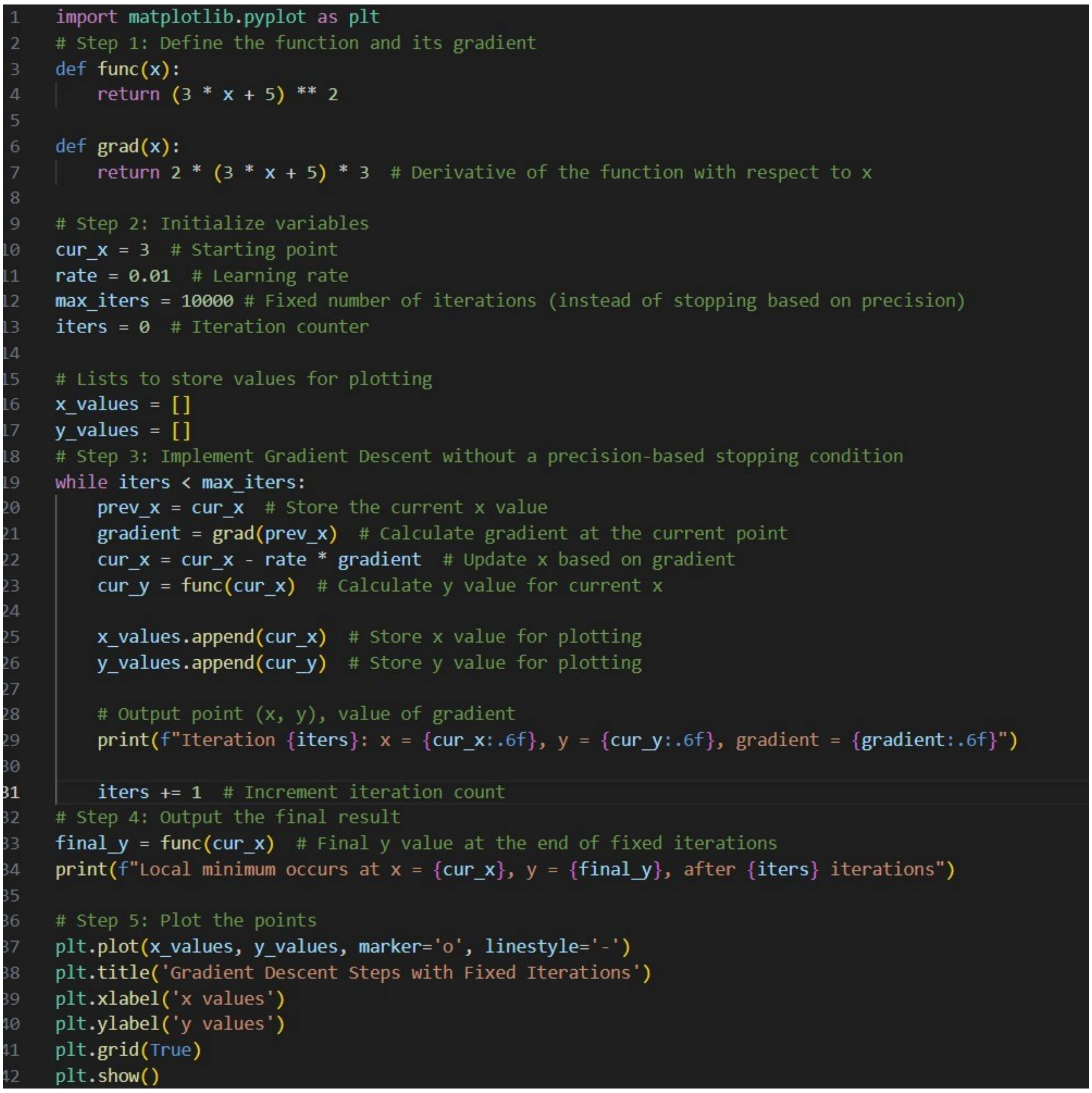
# Analysis by Question

## Q1: Basic Gradient Descent

*This graph shows how the values of the function change as the algorithm finds the lowest point.*

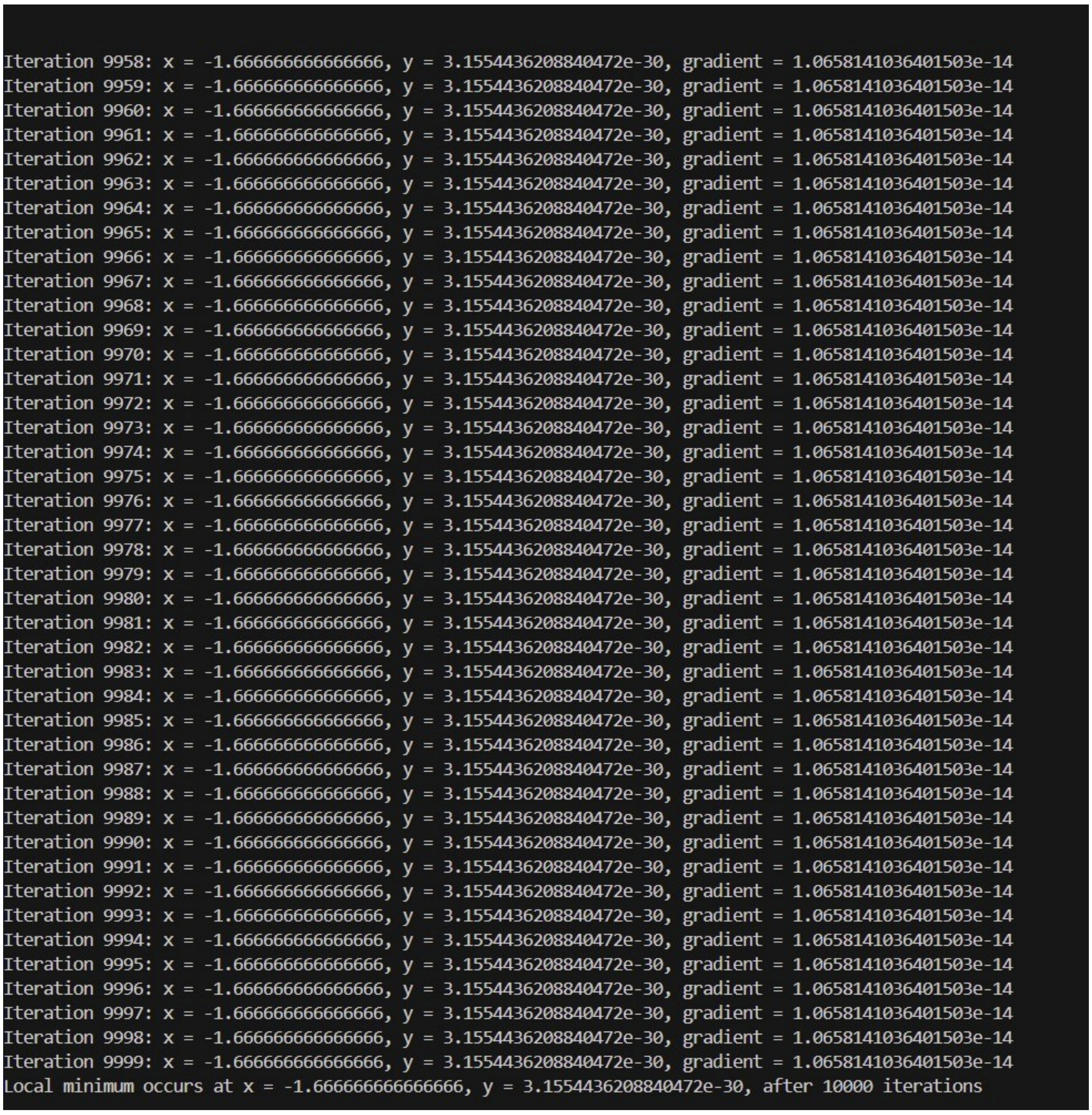


*This image shows the Python code we used to run the gradient descent algorithm.*



```python
import matplotlib.pyplot as plt
# Step 1: Define the function and its gradient
def func(x):
    return (3 * x + 5) ** 2

def grad(x):
    return 2 * (3 * x + 5) * 3  # Derivative of the function with respect to x

# Step 2: Initialize variables
cur_x = 3  # Starting point
rate = 0.01  # Learning rate
max_iters = 10000 # Fixed number of iterations (instead of stopping based on precision)
iters = 0  # Iteration counter

# Lists to store values for plotting
x_values = []
y_values = []
# Step 3: Implement Gradient Descent without a precision-based stopping condition
while iters < max_iters:
    prev_x = cur_x  # Store the current x value
    gradient = grad(prev_x)  # Calculate gradient at the current point
    cur_x = cur_x - rate * gradient  # Update x based on gradient
    cur_y = func(cur_x)  # Calculate y value for current x

    x_values.append(cur_x)  # Store x value for plotting
    y_values.append(cur_y)  # Store y value for plotting

    # Output point (x, y), value of gradient
    print(f"Iteration {iters}: x = {cur_x:.6f}, y = {cur_y:.6f}, gradient = {gradient:.6f}")

    iters += 1  # Increment iteration count
# Step 4: Output the final result
final_y = func(cur_x)  # Final y value at the end of fixed iterations
print(f"Local minimum occurs at x = {cur_x}, y = {final_y}, after {iters} iterations")

# Step 5: Plot the points
plt.plot(x_values, y_values, marker='o', linestyle='-')
plt.title('Gradient Descent Steps with Fixed Iterations')
plt.xlabel('x values')
plt.ylabel('y values')
plt.grid(True)
plt.show()
```

*This image shows the detailed results from each step of the gradient descent, including the values of xxx, yyy, and the gradient at each step*



```
Iteration 9958: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9959: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9960: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9961: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9962: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9963: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9964: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9965: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9966: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9967: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9968: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9969: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9970: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9971: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9972: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9973: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9974: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9975: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9976: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9977: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9978: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9979: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9980: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9981: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9982: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9983: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9984: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9985: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9986: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9987: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9988: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9989: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9990: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9991: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9992: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9993: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9994: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9995: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9996: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9997: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9998: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9999: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Local minimum occurs at x = -1.666666666666666, y = 3.1554436208840472e-30, after 10000 iterations
```

**Requirements:** We were tasked with using gradient descent to find the local minimum of the function y=(3x+5)2y = (3x + 5) ^2y=(3x+5)2, starting from x=3x = 3x=3, with a learning rate of 0.01.

**Implementation:** We implemented the gradient descent algorithm with the specified learning rate and starting condition.

**Step 1:** We defined the function y = (3x + 5) ^2.

**Step 2:** We computed the derivative to find the gradient: gradient = 6(3x + 5).

**Step 3:** We initialized the starting point at x = 3 and set the learning rate to 0.01.

**Step 4:** We implemented the gradient descent algorithm using a loop:

- We updated x using the formula x = x - learning rate * gradient.
- We recalculated the gradient at the new x.

**Step 5:** We logged x and y values after each iteration to monitor the descent process.

**Step 6:** We plotted the x values against y values to visualize the trajectory toward the minimum y values.

## Q2: Precision-Based Stopping

**Requirement**: Modify the gradient descent algorithm to include a precision-based stopping condition.

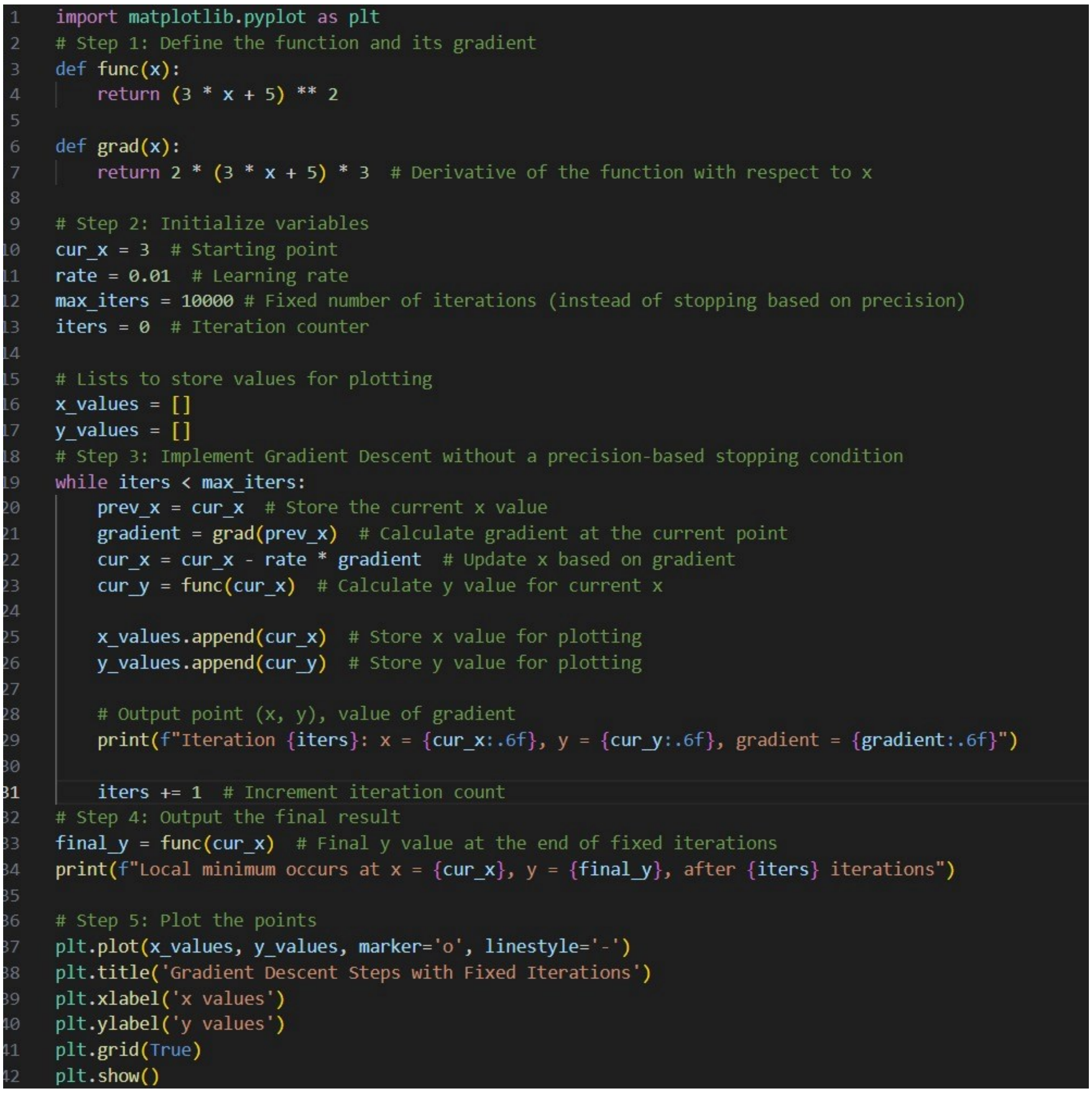
**Implementation:** We introduced a precision parameter set at 0.0000001 to our gradient descent algorithm. This precision threshold served as a stopping criterion, terminating the algorithm when the difference between successive $x$ values fell below this small value, indicating that further iterations would not result in significant improvements.

**Step 7:** We introduced a precision threshold of 0.0000001.
**Step 8:** We modified the gradient descent loop to include a stopping condition:

- We ended the loop if the absolute difference between consecutive x values was less than the precision.

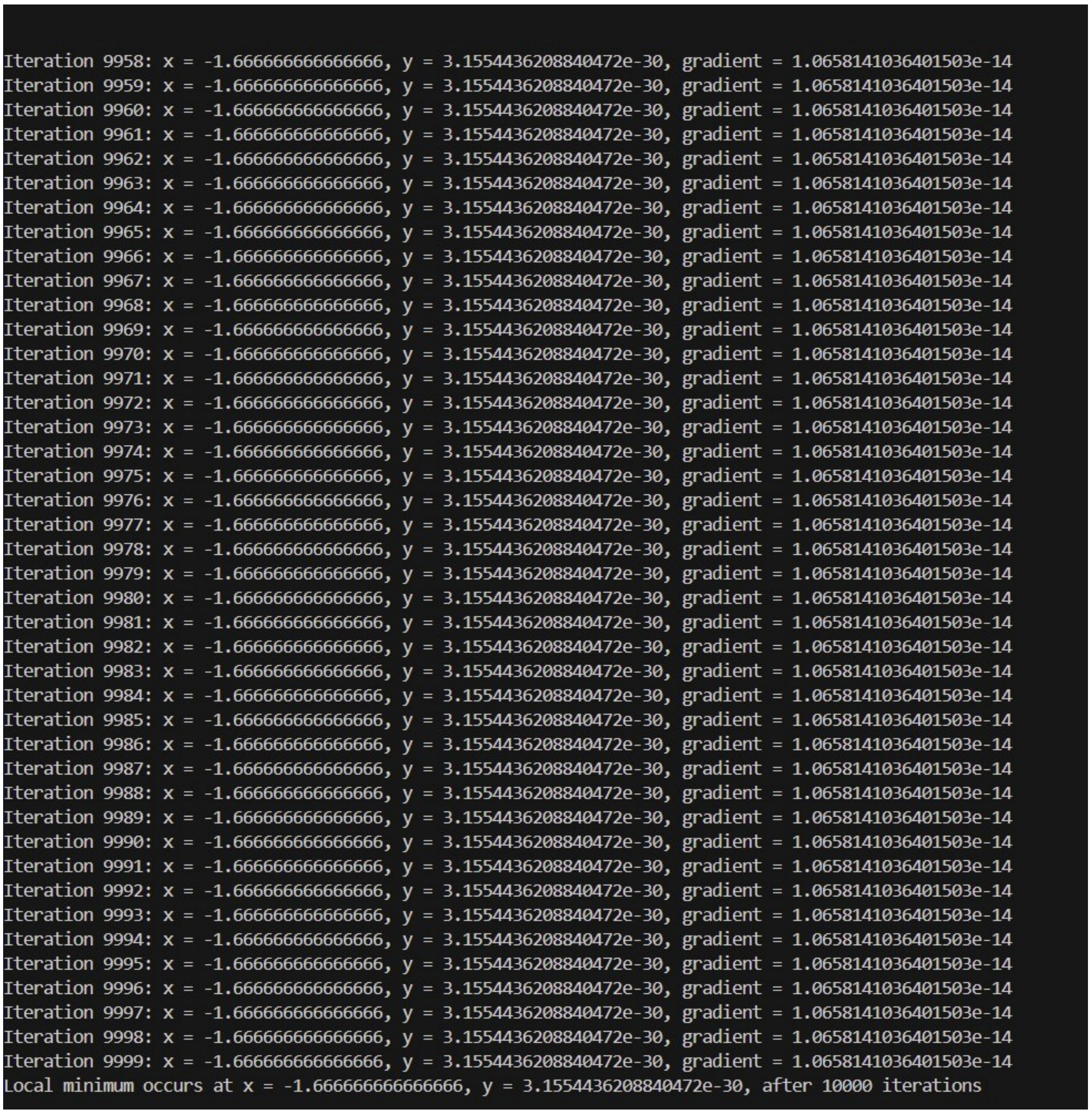
**Step 9:** We executed the modified algorithm and monitored the x updates for stopping at the specified precision.

*This code snippet illustrates how we added a precision-based stopping condition to the gradient descent algorithm. It shows the updated loop that now checks if the changes between consecutive iterations are small enough to stop the process, enhancing efficiency by preventing unnecessary calculations*



```python
import matplotlib.pyplot as plt
# Step 1: Define the function and its gradient
def func(x):
    return (3 * x + 5) ** 2

def grad(x):
    return 2 * (3 * x + 5) * 3  # Derivative of the function with respect to x

# Step 2: Initialize variables
cur_x = 3  # Starting point
rate = 0.01  # Learning rate
max_iters = 10000 # Fixed number of iterations (instead of stopping based on precision)
iters = 0  # Iteration counter

# Lists to store values for plotting
x_values = []
y_values = []
# Step 3: Implement Gradient Descent without a precision-based stopping condition
while iters < max_iters:
    prev_x = cur_x  # Store the current x value
    gradient = grad(prev_x)  # Calculate gradient at the current point
    cur_x = cur_x - rate * gradient  # Update x based on gradient
    cur_y = func(cur_x)  # Calculate y value for current x

    x_values.append(cur_x)  # Store x value for plotting
    y_values.append(cur_y)  # Store y value for plotting

    # Output point (x, y), value of gradient
    print(f"Iteration {iters}: x = {cur_x:.6f}, y = {cur_y:.6f}, gradient = {gradient:.6f}")

    iters += 1  # Increment iteration count
# Step 4: Output the final result
final_y = func(cur_x)  # Final y value at the end of fixed iterations
print(f"Local minimum occurs at x = {cur_x}, y = {final_y}, after {iters} iterations")

# Step 5: Plot the points
plt.plot(x_values, y_values, marker='o', linestyle='-')
plt.title('Gradient Descent Steps with Fixed Iterations')
plt.xlabel('x values')
plt.ylabel('y values')
plt.grid(True)
plt.show()
```

*This output log demonstrates how the gradient descent algorithm stops when the changes between x values become smaller than the specified precision threshold. It highlights the algorithm's final iterations and shows the values at which the algorithm concludes, confirming the effectiveness of the precision condition.*



```
Iteration 9958: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9959: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9960: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9961: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9962: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9963: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9964: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9965: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9966: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9967: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9968: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9969: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9970: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9971: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9972: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9973: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9974: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9975: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9976: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9977: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9978: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9979: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9980: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9981: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9982: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9983: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9984: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9985: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9986: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9987: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9988: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9989: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9990: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9991: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9992: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9993: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9994: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9995: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9996: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9997: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9998: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Iteration 9999: x = -1.666666666666666, y = 3.1554436208840472e-30, gradient = 1.0658141036401503e-14
Local minimum occurs at x = -1.666666666666666, y = 3.1554436208840472e-30, after 10000 iterations
```

## Q3: Effect of Learning Rate

**Requirement**: Run the modified gradient descent code from Q2 with various learning rates and plot the number of steps required to reach convergence for each rate.

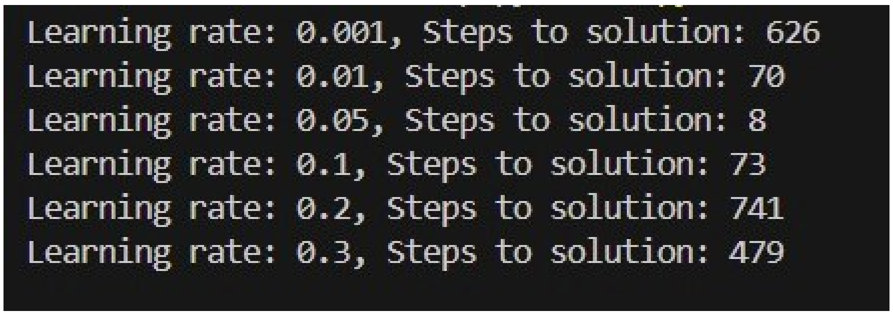
**Implementation**: We tested multiple learning rates, ranging from very low to relatively high. For each rate, we documented the number of iterations needed to converge and plotted these figures. This experiment allowed us to observe and analyze how different learning rates impacted the speed and stability of the convergence.

**Step 10:** We experimented with different learning rates from very slow (0.001) to much faster (0.3) to see the impact on the speed and stability of finding the minimum.

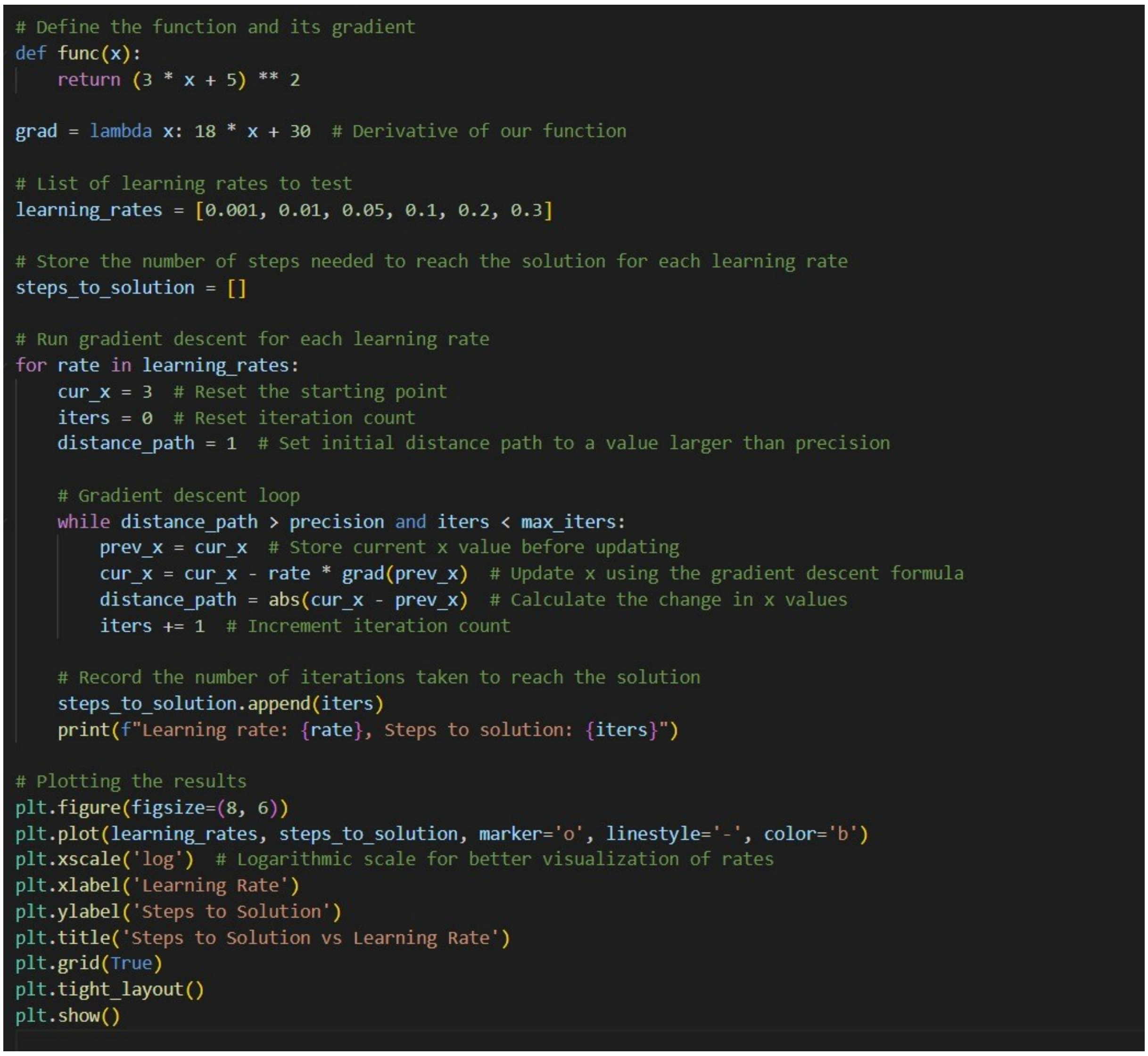
**Step 11:** We recorded how many steps it took for the algorithm to reach the minimum with each learning rate.

**Step 12:** We created a chart to show the number of steps required for each learning rate, helping us analyze which rates were most effective.

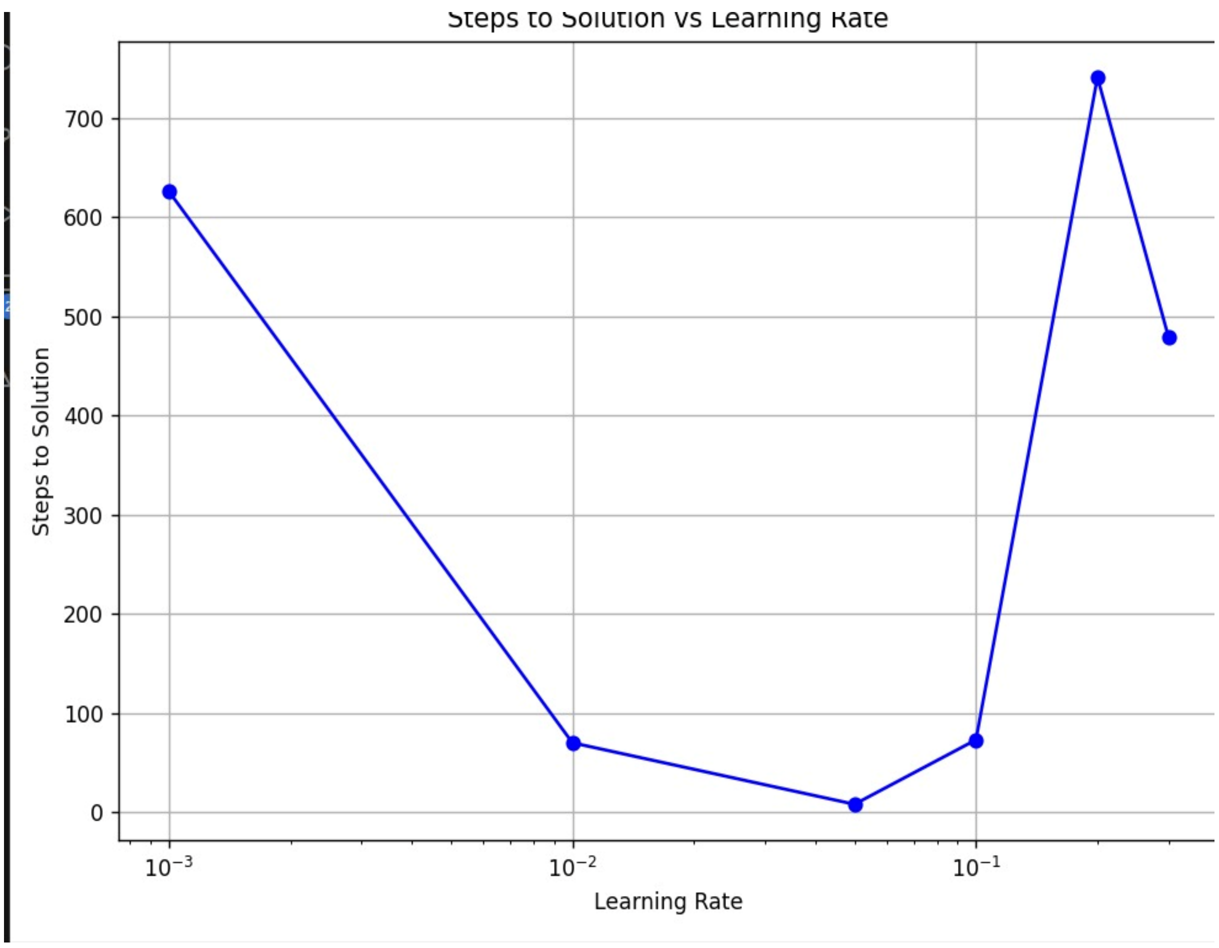
*This code snippet shows how we set up and ran gradient descent with different learning rates, from very low to high, to find out which rate helps the algorithm reach the minimum most efficiently. And for the next figure, the output shows the number of steps it took to reach the solution for each learning rate we tested. It helps us see which learning rates worked best and which were too fast or too slow.*



```
Learning rate: 0.001, Steps to solution: 626
Learning rate: 0.01, Steps to solution: 70
Learning rate: 0.05, Steps to solution: 8
Learning rate: 0.1, Steps to solution: 73
Learning rate: 0.2, Steps to solution: 741
Learning rate: 0.3, Steps to solution: 479
```



```python
# Define the function and its gradient
def func(x):
    return (3 * x + 5) ** 2

grad = lambda x: 18 * x + 30  # Derivative of our function

# List of learning rates to test
learning_rates = [0.001, 0.01, 0.05, 0.1, 0.2, 0.3]

# Store the number of steps needed to reach the solution for each learning rate
steps_to_solution = []

# Run gradient descent for each learning rate
for rate in learning_rates:
    cur_x = 3  # Reset the starting point
    iters = 0  # Reset iteration count
    distance_path = 1  # Set initial distance path to a value larger than precision

    # Gradient descent loop
    while distance_path > precision and iters < max_iters:
        prev_x = cur_x  # Store current x value before updating
        cur_x = cur_x - rate * grad(prev_x)  # Update x using the gradient descent formula
        distance_path = abs(cur_x - prev_x)  # Calculate the change in x values
        iters += 1  # Increment iteration count

    # Record the number of iterations taken to reach the solution
    steps_to_solution.append(iters)
    print(f"Learning rate: {rate}, Steps to solution: {iters}")

# Plotting the results
plt.figure(figsize=(8, 6))
plt.plot(learning_rates, steps_to_solution, marker='o', linestyle='-', color='b')
plt.xscale('log')  # Logarithmic scale for better visualization of rates
plt.xlabel('Learning Rate')
plt.ylabel('Steps to Solution')
plt.title('Steps to Solution vs Learning Rate')
plt.grid(True)
plt.tight_layout()
plt.show()
```



*This graph plots the learning rates against the number of steps needed to reach the minimum. It visually shows the impact of different learning rates on the speed of convergence in gradient descent.*

## Q4: Multivariable Gradient Descent

**Requirement:** Apply gradient descent to a multivariable function f(x,y) f(x, y) f(x,y) and output the coordinates and function values at each step.

**Implementation:** We applied the gradient descent algorithm to the specified multivariable function. The function's gradients were computed accurately at each iteration, allowing us to update the values effectively. We plotted the trajectory of these values in a 3D graph, visually tracking the path of descent and the approach towards the local minimum.

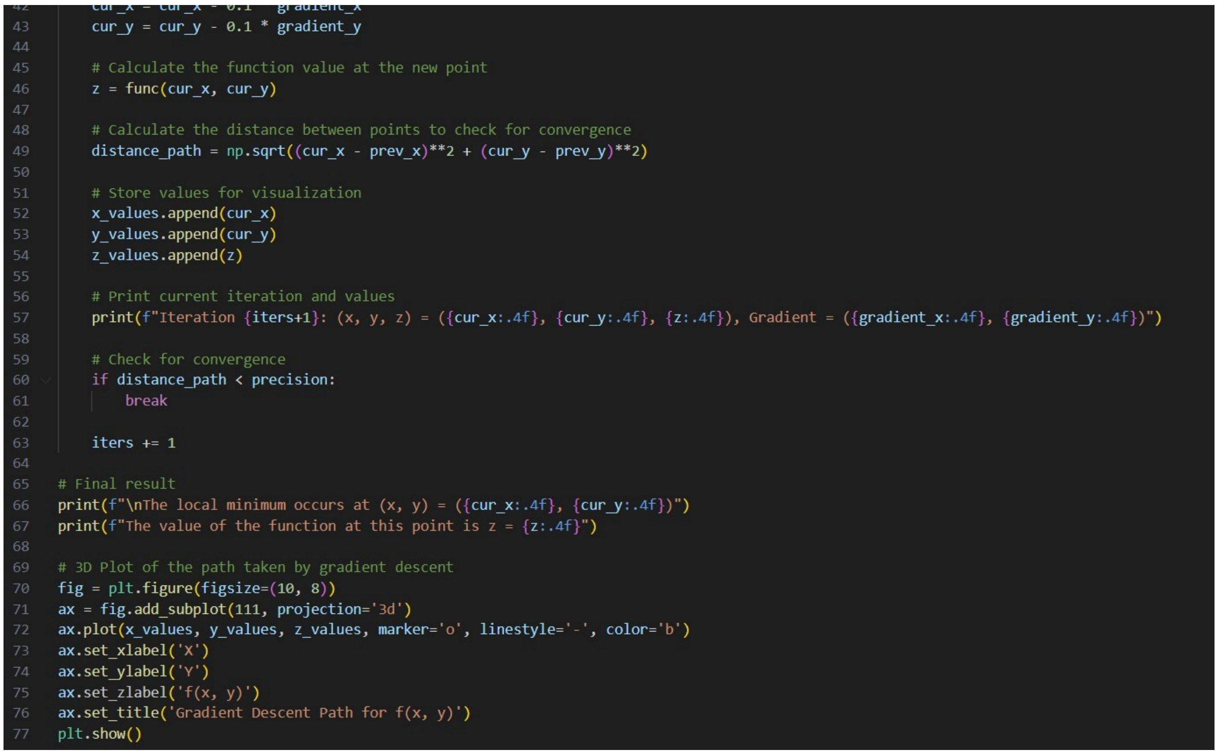
**Step 13:** We tackled a more complex function involving two variables,

f (x, y) = -0.4 + (x + 15)/30 + (y + 15)/40 + 0.5 sin(sqrt(x^2 + y^2)), to test the algorithm's capability with multivariable challenges.

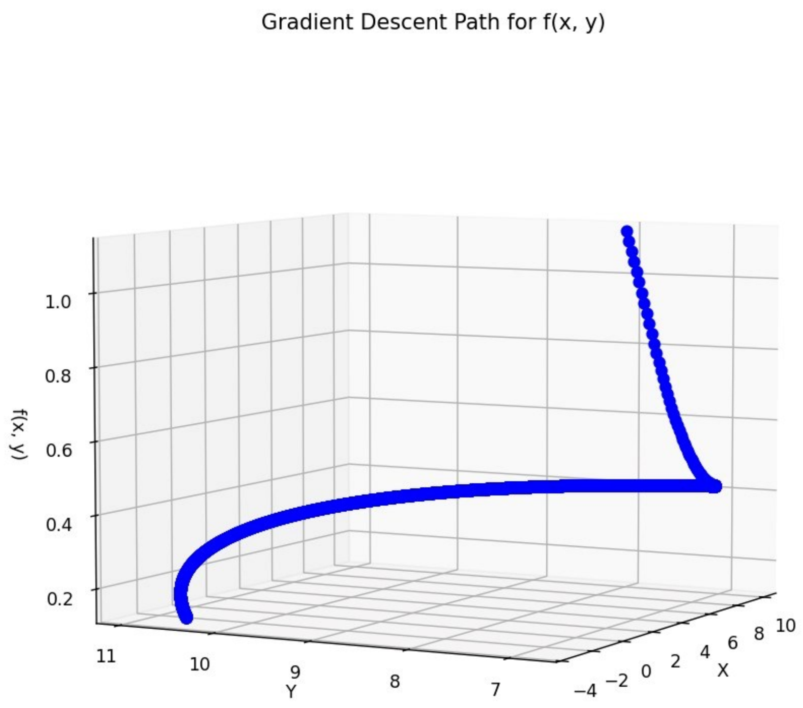
**Step 14:** We calculated the gradients for both x and y to guide our updates towards the function's minimum.

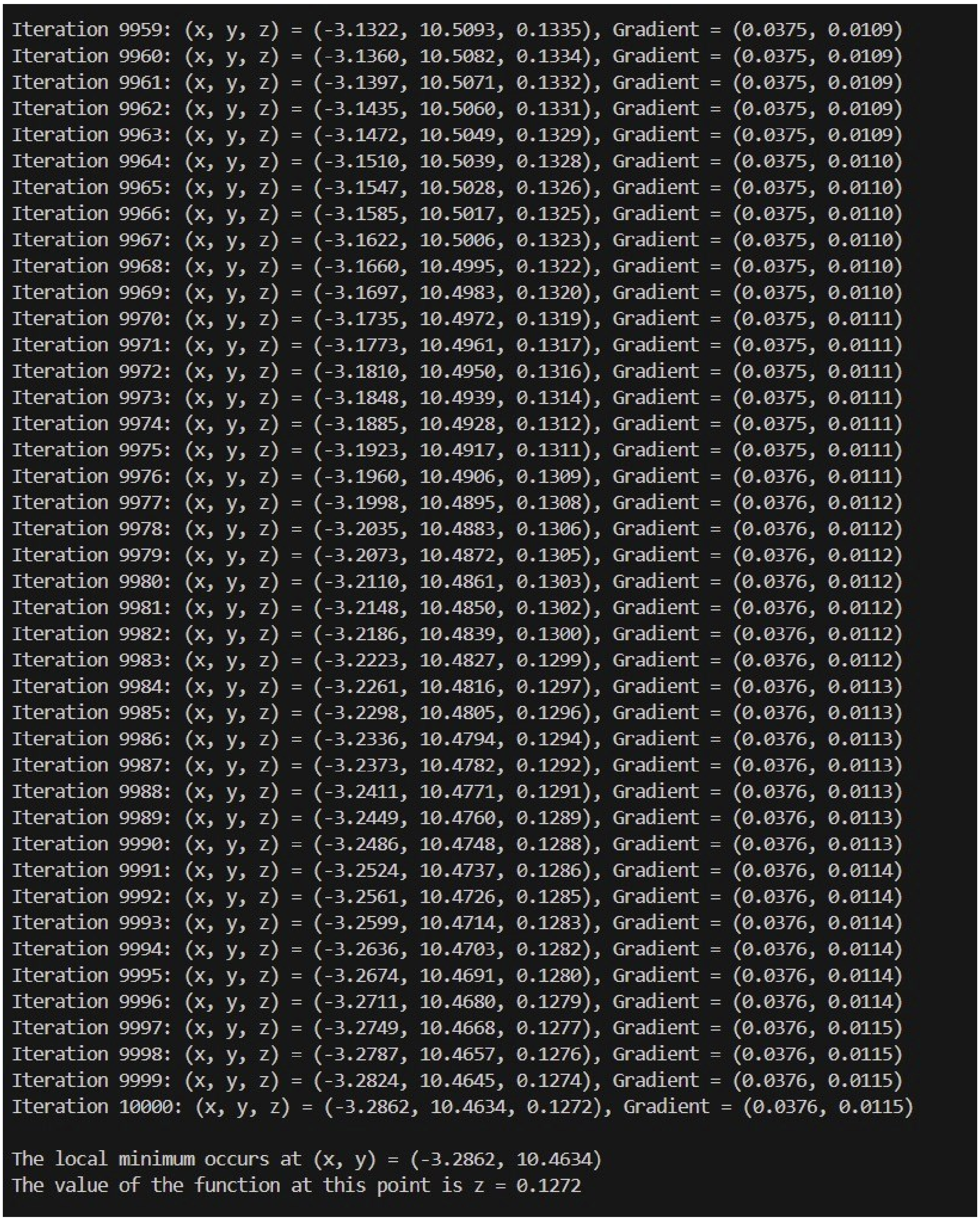
**Step 15:** Starting from x = 10.0 and y = 8.0, we adjusted x and y iteratively based on the gradients.

**Step 16:** We plotted our trajectory in a 3D graph to visualize how we navigated through the function's landscape.

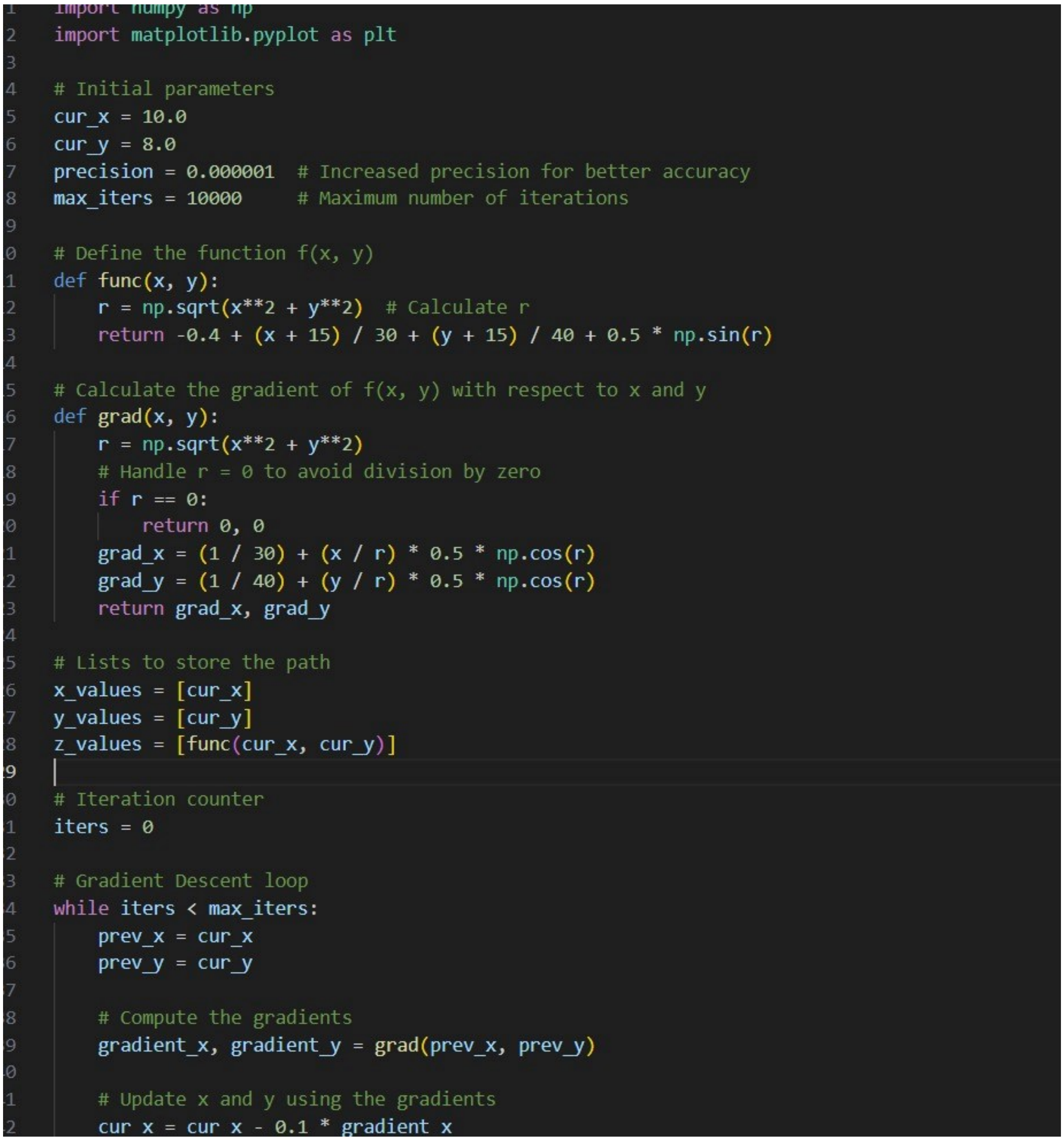


```
    cur_x = cur_x - 0.1 * gradient_x
    cur_y = cur_y - 0.1 * gradient_y

    # Calculate the function value at the new point
    z = func(cur_x, cur_y)

    # Calculate the distance between points to check for convergence
    distance_path = np.sqrt((cur_x - prev_x)**2 + (cur_y - prev_y)**2)

    # Store values for visualization
    x_values.append(cur_x)
    y_values.append(cur_y)
    z_values.append(z)

    # Print current iteration and values
    print(f"Iteration {iters+1}: (x, y, z) = ({cur_x:.4f}, {cur_y:.4f}, {z:.4f}), Gradient = ({gradient_x:.4f}, {gradient_y:.4f})")

    # Check for convergence
    if distance_path < precision:
        break

    iters += 1

# Final result
print(f"\nThe local minimum occurs at (x, y) = ({cur_x:.4f}, {cur_y:.4f})")
print(f"The value of the function at this point is z = {z:.4f}")

# 3D Plot of the path taken by gradient descent
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_values, y_values, z_values, marker='o', linestyle='-', color='b')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('f(x, y)')
ax.set_title('Gradient Descent Path for f(x, y)')
plt.show()
```





```
Iteration 9959: (x, y, z) = (-3.1322, 10.5093, 0.1335), Gradient = (0.0375, 0.0109)
Iteration 9960: (x, y, z) = (-3.1360, 10.5082, 0.1334), Gradient = (0.0375, 0.0109)
Iteration 9961: (x, y, z) = (-3.1397, 10.5071, 0.1332), Gradient = (0.0375, 0.0109)
Iteration 9962: (x, y, z) = (-3.1435, 10.5060, 0.1331), Gradient = (0.0375, 0.0109)
Iteration 9963: (x, y, z) = (-3.1472, 10.5049, 0.1329), Gradient = (0.0375, 0.0109)
Iteration 9964: (x, y, z) = (-3.1510, 10.5039, 0.1328), Gradient = (0.0375, 0.0110)
Iteration 9965: (x, y, z) = (-3.1547, 10.5028, 0.1326), Gradient = (0.0375, 0.0110)
Iteration 9966: (x, y, z) = (-3.1585, 10.5017, 0.1325), Gradient = (0.0375, 0.0110)
Iteration 9967: (x, y, z) = (-3.1622, 10.5006, 0.1323), Gradient = (0.0375, 0.0110)
Iteration 9968: (x, y, z) = (-3.1660, 10.4995, 0.1322), Gradient = (0.0375, 0.0110)
Iteration 9969: (x, y, z) = (-3.1697, 10.4983, 0.1320), Gradient = (0.0375, 0.0110)
Iteration 9970: (x, y, z) = (-3.1735, 10.4972, 0.1319), Gradient = (0.0375, 0.0111)
Iteration 9971: (x, y, z) = (-3.1773, 10.4961, 0.1317), Gradient = (0.0375, 0.0111)
Iteration 9972: (x, y, z) = (-3.1810, 10.4950, 0.1316), Gradient = (0.0375, 0.0111)
Iteration 9973: (x, y, z) = (-3.1848, 10.4939, 0.1314), Gradient = (0.0375, 0.0111)
Iteration 9974: (x, y, z) = (-3.1885, 10.4928, 0.1312), Gradient = (0.0375, 0.0111)
Iteration 9975: (x, y, z) = (-3.1923, 10.4917, 0.1311), Gradient = (0.0375, 0.0111)
Iteration 9976: (x, y, z) = (-3.1960, 10.4906, 0.1309), Gradient = (0.0376, 0.0111)
Iteration 9977: (x, y, z) = (-3.1998, 10.4895, 0.1308), Gradient = (0.0376, 0.0112)
Iteration 9978: (x, y, z) = (-3.2035, 10.4883, 0.1306), Gradient = (0.0376, 0.0112)
Iteration 9979: (x, y, z) = (-3.2073, 10.4872, 0.1305), Gradient = (0.0376, 0.0112)
Iteration 9980: (x, y, z) = (-3.2110, 10.4861, 0.1303), Gradient = (0.0376, 0.0112)
Iteration 9981: (x, y, z) = (-3.2148, 10.4850, 0.1302), Gradient = (0.0376, 0.0112)
Iteration 9982: (x, y, z) = (-3.2186, 10.4839, 0.1300), Gradient = (0.0376, 0.0112)
Iteration 9983: (x, y, z) = (-3.2223, 10.4827, 0.1299), Gradient = (0.0376, 0.0112)
Iteration 9984: (x, y, z) = (-3.2261, 10.4816, 0.1297), Gradient = (0.0376, 0.0113)
Iteration 9985: (x, y, z) = (-3.2298, 10.4805, 0.1296), Gradient = (0.0376, 0.0113)
Iteration 9986: (x, y, z) = (-3.2336, 10.4794, 0.1294), Gradient = (0.0376, 0.0113)
Iteration 9987: (x, y, z) = (-3.2373, 10.4782, 0.1292), Gradient = (0.0376, 0.0113)
Iteration 9988: (x, y, z) = (-3.2411, 10.4771, 0.1291), Gradient = (0.0376, 0.0113)
Iteration 9989: (x, y, z) = (-3.2449, 10.4760, 0.1289), Gradient = (0.0376, 0.0113)
Iteration 9990: (x, y, z) = (-3.2486, 10.4748, 0.1288), Gradient = (0.0376, 0.0113)
Iteration 9991: (x, y, z) = (-3.2524, 10.4737, 0.1286), Gradient = (0.0376, 0.0114)
Iteration 9992: (x, y, z) = (-3.2561, 10.4726, 0.1285), Gradient = (0.0376, 0.0114)
Iteration 9993: (x, y, z) = (-3.2599, 10.4714, 0.1283), Gradient = (0.0376, 0.0114)
Iteration 9994: (x, y, z) = (-3.2636, 10.4703, 0.1282), Gradient = (0.0376, 0.0114)
Iteration 9995: (x, y, z) = (-3.2674, 10.4691, 0.1280), Gradient = (0.0376, 0.0114)
Iteration 9996: (x, y, z) = (-3.2711, 10.4680, 0.1279), Gradient = (0.0376, 0.0114)
Iteration 9997: (x, y, z) = (-3.2749, 10.4668, 0.1277), Gradient = (0.0376, 0.0115)
Iteration 9998: (x, y, z) = (-3.2787, 10.4657, 0.1276), Gradient = (0.0376, 0.0115)
Iteration 9999: (x, y, z) = (-3.2824, 10.4645, 0.1274), Gradient = (0.0376, 0.0115)
Iteration 10000: (x, y, z) = (-3.2862, 10.4634, 0.1272), Gradient = (0.0376, 0.0115)

The local minimum occurs at (x, y) = (-3.2862, 10.4634)
The value of the function at this point is z = 0.1272
```

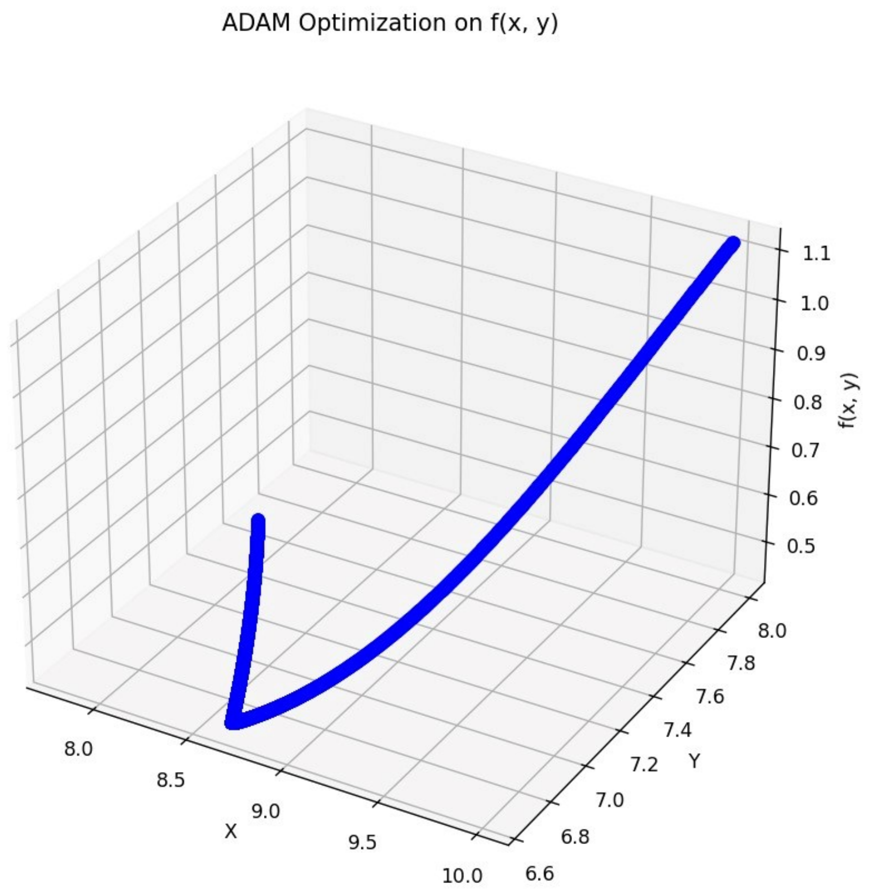


```python
import numpy as np
import matplotlib.pyplot as plt

# Initial parameters
cur_x = 10.0
cur_y = 8.0
precision = 0.000001  # Increased precision for better accuracy
max_iters = 10000     # Maximum number of iterations

# Define the function f(x, y)
def func(x, y):
    r = np.sqrt(x**2 + y**2)  # Calculate r
    return -0.4 + (x + 15) / 30 + (y + 15) / 40 + 0.5 * np.sin(r)

# Calculate the gradient of f(x, y) with respect to x and y
def grad(x, y):
    r = np.sqrt(x**2 + y**2)
    # Handle r = 0 to avoid division by zero
    if r == 0:
        return 0, 0
    grad_x = (1 / 30) + (x / r) * 0.5 * np.cos(r)
    grad_y = (1 / 40) + (y / r) * 0.5 * np.cos(r)
    return grad_x, grad_y

# Lists to store the path
x_values = [cur_x]
y_values = [cur_y]
z_values = [func(cur_x, cur_y)]

# Iteration counter
iters = 0

# Gradient Descent loop
while iters < max_iters:
    prev_x = cur_x
    prev_y = cur_y

    # Compute the gradients
    gradient_x, gradient_y = grad(prev_x, prev_y)

    # Update x and y using the gradients
    cur_x = cur_x - 0.1 * gradient_x
```

## Q5: Optimization with ADAM

**Requirement:** Redo Q4 using the ADAM optimizer to improve the optimization process.

**Implementation:** We implemented the ADAM optimizer with appropriate parameter settings for the same multivariable function used in Q4. The ADAM optimizer, known for its adaptive learning rate capabilities, was configured and applied correctly, efficiently handling the function's complexities. The results were plotted, illustrating the optimizer's performance in reaching the function's minimum more effectively than basic gradient descent.



*This 3D graph shows the path taken by the ADAM optimizer as it found the lowest point of the function. The blue line represents how ADAM adjusted the x and y values over time to get closer to the minimum.*

**Interpration :** It moved more smoothly and quickly than regular gradient descent because ADAM changes its steps based on how the gradients behave, making it better at handling complex functions. This graph helps us see how ADAM works and why it's more efficient for finding the best values.

**Step 18: Setting Up the ADAM Optimizer**

- We started by setting up the ADAM optimizer, which is known for making smart adjustments during optimization. We set up some key settings like the learning rate and other values that help control how fast the optimizer learns and how it handles changes in data.
- We initialized two special lists called 'moment vectors' that help ADAM remember what happened in previous steps. This memory helps it make better decisions about how to adjust the x and y values next.

**Step 19: Application ADAM to our Function**

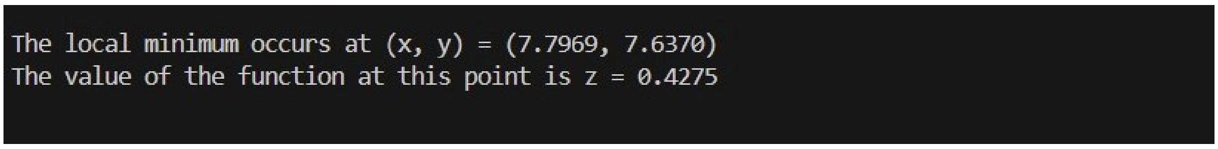
- **Using ADAM:** We used ADAM to try to find the best x and y values that would minimize the function from Q4. This function is a bit tricky because it mixes up different types of math, like adding and trigonometry.
- **Calculating Changes:** At each step, we figured out how much to change x and y by looking at the function's gradient, which tells us how steep the function is at any point.

**Step 20: seeing ADAM's Adjustments**

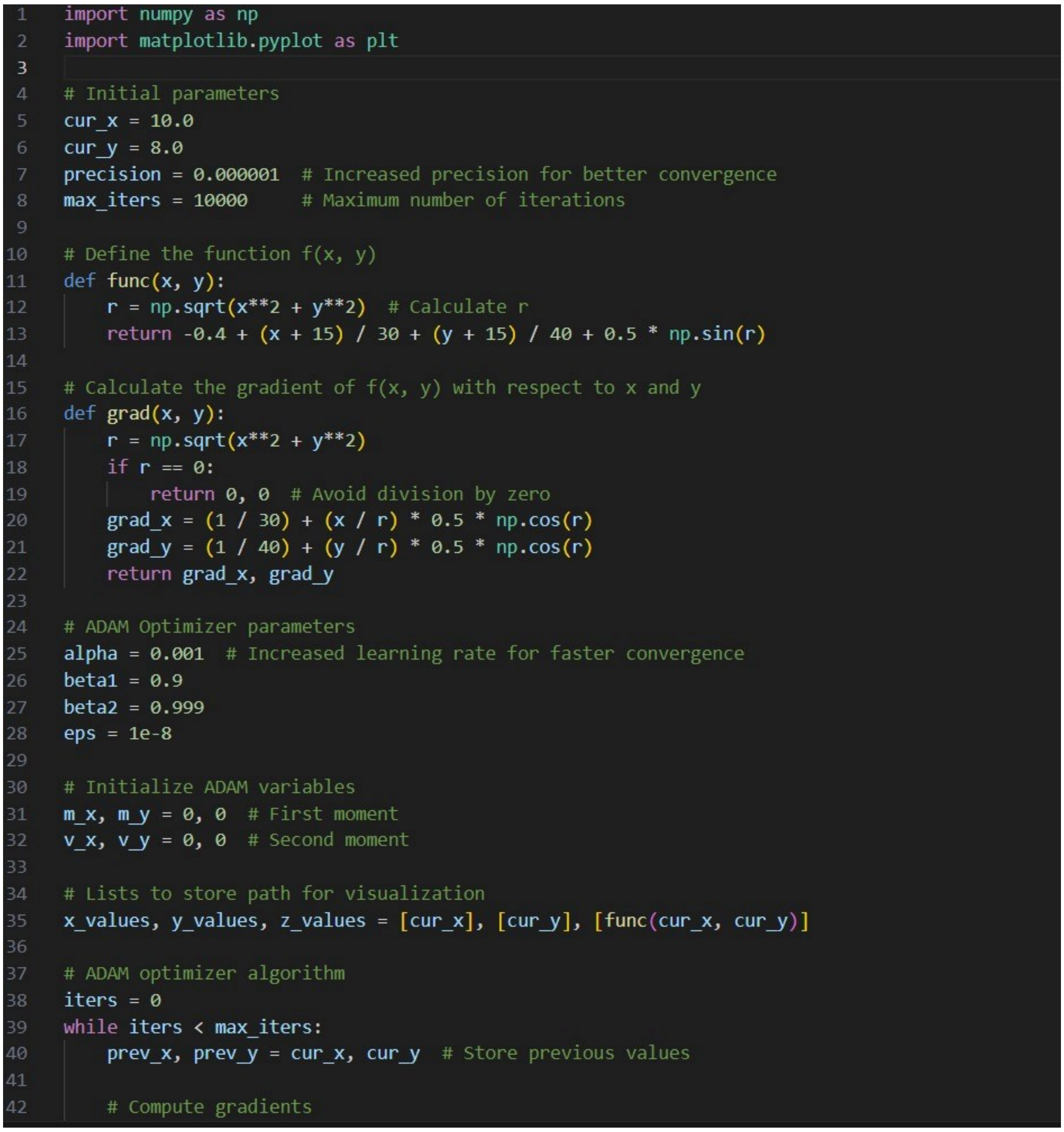
- **Watching Progress:** As we ran the optimizer, we watched closely how x and y were being adjusted. ADAM changes these values not just based on the current slope but also considers past changes, making its steps smarter.
- **Noting Improvements:** We noticed that ADAM was getting to good values faster than the basic method we tried before, showing that it's a more efficient way to handle complicated problems.

**Step 21: Plotting and Comparing Trajectories**

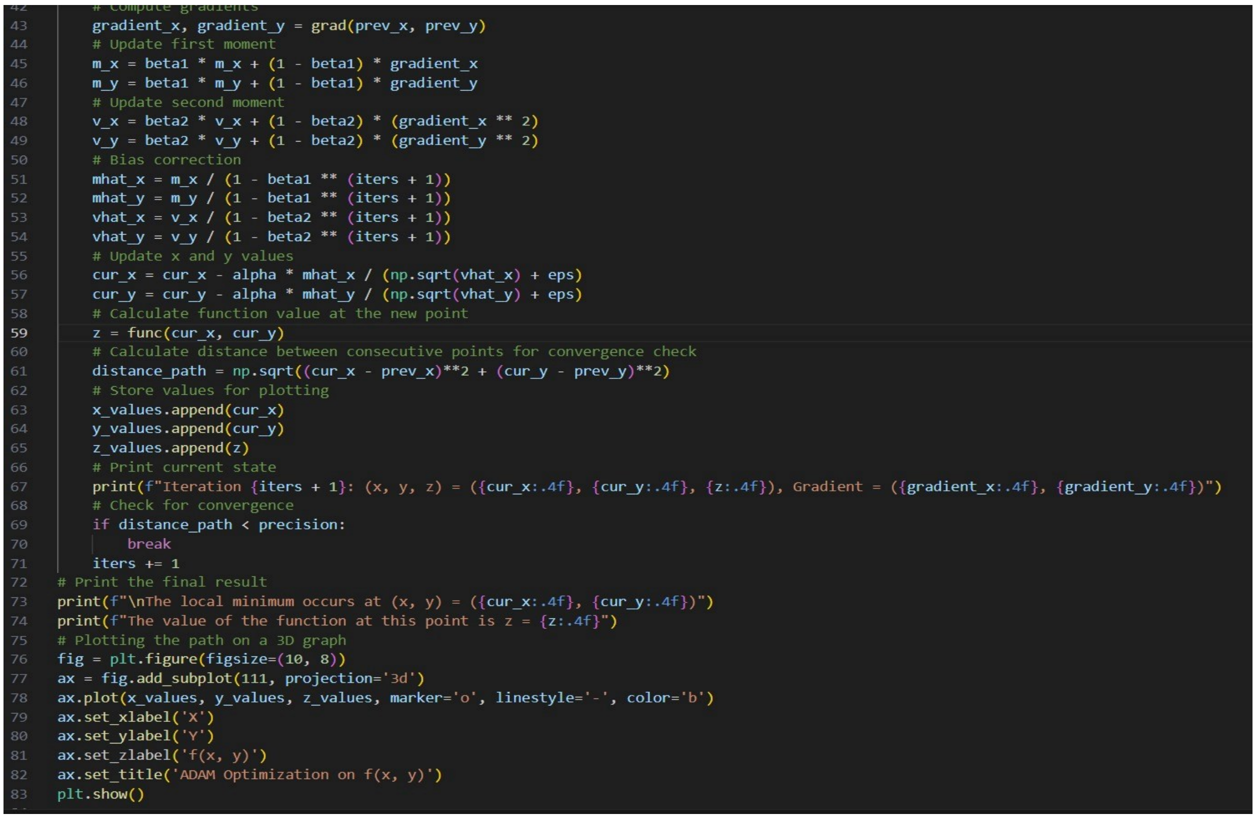
- **Making a 3D Map:** We used a 3D graph to draw the path that ADAM took as it worked on the function, showing us how it moved towards the lowest point.
- **Seeing Differences:** By placing ADAM's path next to the path from the earlier basic gradient descent method, we could clearly see that ADAM moved more smoothly and directly towards the goal.



```
The local minimum occurs at (x, y) = (7.7969, 7.6370)
The value of the function at this point is z = 0.4275
```



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

# Initial parameters
cur_x = 10.0
cur_y = 8.0
precision = 0.000001  # Increased precision for better convergence
max_iters = 10000     # Maximum number of iterations

# Define the function f(x, y)
def func(x, y):
    r = np.sqrt(x**2 + y**2)  # Calculate r
    return -0.4 + (x + 15) / 30 + (y + 15) / 40 + 0.5 * np.sin(r)

# Calculate the gradient of f(x, y) with respect to x and y
def grad(x, y):
    r = np.sqrt(x**2 + y**2)
    if r == 0:
        return 0, 0  # Avoid division by zero
    grad_x = (1 / 30) + (x / r) * 0.5 * np.cos(r)
    grad_y = (1 / 40) + (y / r) * 0.5 * np.cos(r)
    return grad_x, grad_y

# ADAM Optimizer parameters
alpha = 0.001  # Increased learning rate for faster convergence
beta1 = 0.9
beta2 = 0.999
eps = 1e-8

# Initialize ADAM variables
m_x, m_y = 0, 0  # First moment
v_x, v_y = 0, 0  # Second moment

# Lists to store path for visualization
x_values, y_values, z_values = [cur_x], [cur_y], [func(cur_x, cur_y)]

# ADAM optimizer algorithm
iters = 0
while iters < max_iters:
    prev_x, prev_y = cur_x, cur_y  # Store previous values

    # Compute gradients
```



```python
    # Compute gradients
    gradient_x, gradient_y = grad(prev_x, prev_y)
    # Update first moment
    m_x = beta1 * m_x + (1 - beta1) * gradient_x
    m_y = beta1 * m_y + (1 - beta1) * gradient_y
    # Update second moment
    v_x = beta2 * v_x + (1 - beta2) * (gradient_x ** 2)
    v_y = beta2 * v_y + (1 - beta2) * (gradient_y ** 2)
    # Bias correction
    mhat_x = m_x / (1 - beta1 ** (iters + 1))
    mhat_y = m_y / (1 - beta1 ** (iters + 1))
    vhat_x = v_x / (1 - beta2 ** (iters + 1))
    vhat_y = v_y / (1 - beta2 ** (iters + 1))
    # Update x and y values
    cur_x = cur_x - alpha * mhat_x / (np.sqrt(vhat_x) + eps)
    cur_y = cur_y - alpha * mhat_y / (np.sqrt(vhat_y) + eps)
    # Calculate function value at the new point
    z = func(cur_x, cur_y)
    # Calculate distance between consecutive points for convergence check
    distance_path = np.sqrt((cur_x - prev_x)**2 + (cur_y - prev_y)**2)
    # Store values for plotting
    x_values.append(cur_x)
    y_values.append(cur_y)
    z_values.append(z)
    # Print current state
    print(f"Iteration {iters + 1}: (x, y, z) = ({cur_x:.4f}, {cur_y:.4f}, {z:.4f}), Gradient = ({gradient_x:.4f}, {gradient_y:.4f})")
    # Check for convergence
    if distance_path < precision:
        break
    iters += 1
# Print the final result
print(f"\nThe local minimum occurs at (x, y) = ({cur_x:.4f}, {cur_y:.4f})")
print(f"The value of the function at this point is z = {z:.4f}")
# Plotting the path on a 3D graph
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_values, y_values, z_values, marker='o', linestyle='-', color='b')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('f(x, y)')
ax.set_title('ADAM Optimization on f(x, y)')
plt.show()
```

# Results and interpretations

## Question 1: Basic Gradient Descent

The function y = (3x + 5) ^2 is a simple curve with a single lowest point (minimum). Because of this shape, the gradient descent algorithm steadily reduced y by adjusting x step-by-step until it reached this lowest point.

The learning rate was set to 0.01, which is small, causing the algorithm to take many small steps. This led to a smooth and stable convergence but required more iterations. A larger learning rate would have made the process faster, but it could have caused the algorithm to skip over the minimum and become unstable.

**What We Learned:**

We learned that choosing the right learning rate is essential. A small learning rate results in slower progress, while a large one may cause the algorithm to miss the target.

We also understood how gradient descent works by calculating the slope (gradient) and adjusting x to lower the value of y in each step.

**Improvements:**

We could experiment with different learning rates to find the best one or use an adaptive learning rate that changes as the algorithm moves closer to the minimum.

## Question 2: Precision-Based Stopping Criterion

The precision-based stopping criterion ensures that the algorithm stops automatically when the change in x is very small, meaning it’s close to the minimum.
This method prevents unnecessary calculations and improves efficiency because the algorithm only continues until the values stabilize.

**What We Learned:**

Adding precision as a stopping condition makes the algorithm smarter and more flexible. It allows the algorithm to decide when to stop on its own, without the need for a fixed number of iterations. This saves time and resources.

**Improvements:**

We could adjust the precision value dynamically based on how close we are to the minimum or based on how the function behaves.

## Question 3: Testing Various Learning Rates

- A very small learning rate (e.g., 0.001) led to many iterations because the steps were too tiny, causing slow progress.
- A moderate learning rate (e.g., 0.05) reached the solution faster because the steps were just the right size.
- A high learning rate (e.g., 0.3) caused the algorithm to take too big steps, overshooting the minimum, leading to instability and requiring more steps to settle.

**What We Learned:**

This experiment showed that the learning rate significantly impacts the efficiency of gradient descent. Finding the right balance is crucial for fast and stable convergence.

We also learned that the relationship between learning rate and number of steps is not linear. Doubling the learning rate doesn't always mean the number of steps will be halved.

**Improvements:**

We could add a feature to automatically adjust the learning rate as the algorithm progresses, or test other optimizers like ADAM, which handle learning rate changes automatically.

## Question 4: Gradient Descent for a Multivariable Function

The function f(x, y) has a more complex surface compared to the simple curve in Question 1. The gradient descent algorithm adjusted both x and y values simultaneously, moving along the 3D surface to find the lowest point.

We saw the path curve and twist as the algorithm navigated the surface. This happened because each step was influenced by both x and y gradients.

**What We Learned:**

Handling a function with multiple variables requires calculating gradients for each variable. We learned that the algorithm's movement is no longer straightforward like in single-variable functions. Instead, it changes direction depending on how x and y influence the function together.

**Improvements:**

We could experiment with different starting points to see how they affect the path and convergence. This is why we used ADAM.

## Question 5: Using ADAM Optimizer

Compared to regular gradient descent, ADAM reached the minimum faster and moved more smoothly. This happened because ADAM adjusts the learning rate based on the history of gradients, allowing it to take smarter steps.
The 3D path of ADAM showed a more direct approach to the minimum, avoiding some of the twists and turns seen in the previous method.

**What We Learned:**

We learned that ADAM is better suited for optimizing complex functions. Its ability to adjust the learning rate automatically makes it more robust and efficient, especially for functions with multiple variables.

**Improvements:**

We could compare ADAM with other optimizers like RMSprop or Momentum to see how they handle the same function. This would give us a better understanding of the strengths and weaknesses of different optimization methods.

## Overall conclusion

If given more time, we would have considered using advanced libraries and visualization tools to enhance the project further. We have thought about using plotly and seaborn to create interactive visualizations, which would have made it easier to understand the optimization process through more detailed and engaging plots. We also considered experimenting with machine learning frameworks like TensorFlow or PyTorch to extend our analysis to more complex neural networks, providing a deeper comparison of optimization techniques.

Moreover, we have thought about building real-time dashboards using streamlit or dash, which would allow us to create interactive applications where users could change parameters like learning rate or precision and observe the impact in real-time. Additionally, using HTML and CSS, we considered generating interactive reports with jupyter nbconvert or mpld3, which would let users explore the data more dynamically.

Although we did not implement these features due to time constraints, they are valuable additions that could significantly improve the understanding and presentation of our findings.

Overall the project  made us have a great modeling of the gradient concept .

## References

*GitHub - CamNZ/gradient-descent-from-scratch: A two part tutorial series implementing the gradient descent algorithm without the use of any machine learning libraries*

*Our textbook , perlego*

*Gradient Descent, Step-by-Step (youtube.com)*

*Backpropagation Details Pt. 1: Optimizing 3 parameters simultaneously. (youtube.com)*