Assignment 1

Danielle Buggle 20333829

February 11, 2025

Part A

```python
def derivative_expr(var, func):
    """
    Function to get the derivative of an expression.
    :param var: Variable with respect to which differentiation is
        performed.
    :param func: Mathematical expression to differentiate.
    :return: Derivative of the expression.
    """
    return sympy.diff(func, var)

x = sympy.symbols('x', real=True)
f = x ** 4
dfdx = derivative_expr(x, f)
print(dfdx)
```

Listing 1: Function to obtain derivative

(i)

Differentiation is a method used to compute the rate of change of a function $f(x)$ with respect to a variable, for example, x. The code snippet above, Listing 1, shows the function I created to calculate the derivative of a given function. The code defines the function for $y(x) = x^4$. The function takes two inputs, `var` which is the variable with respect to which differentiation is performed (in this case, x). The other argument it takes is `func`, which is the mathematical function to be differentiated - in this case x^4. I use the `sympy.diff()` which allows me to symbolically calculate the function's derivative with respect to the variable. The output to the terminal after running this function is `4*x**3`.

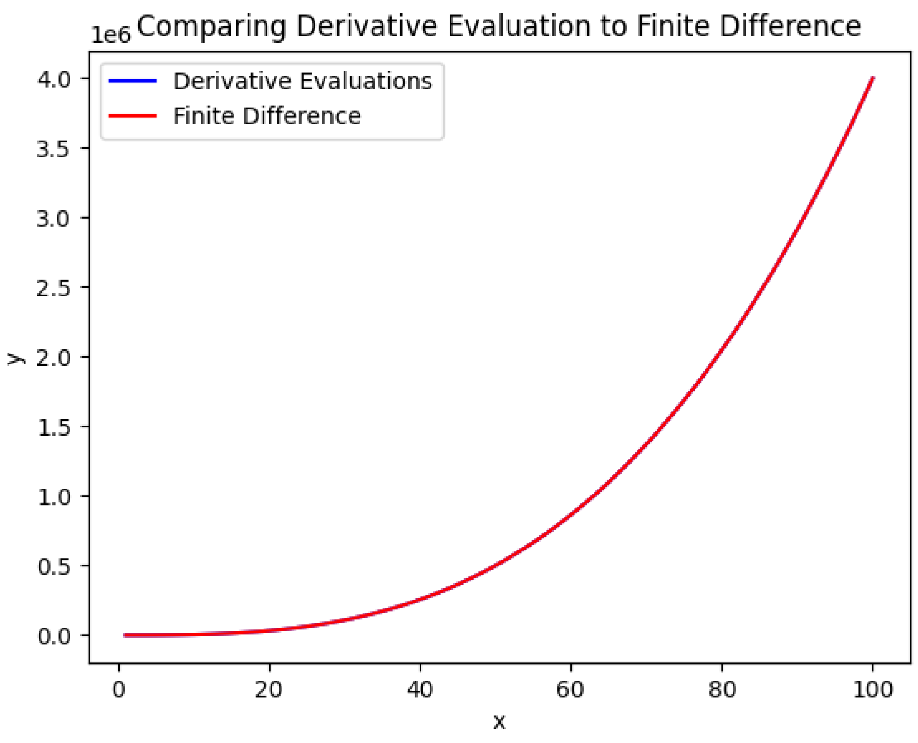

Figure 1: Comparison of Actual Derivates vs. Finite Difference

(ii)

Figure 1, shows the plot of the actual derivative evaluations versus the finite difference estimate, for the x values 1 to 100. From this plot, it is difficult to distinguish any differences between the two values. We can only see the `Finite Difference` line due to the fact it was plotted second and the `Derivative Evaluations` is in blue underneath. The reason they are plotted directly on top of each other is because the actual derivative values and the finite difference estimates are very close to each other. The function $f(x) = x^4$ has a smooth and predictable rate of change, and since we're using a small perturbation of $\delta = 0.01$, the finite difference approximation is very close to the exact derivative.

The listings 2 and 3 below, are code snippets of the functions I created to evaluate the actual derivative and the finite difference estimate. From Listing 2, the `evaluate_derivative()` function takes the symbolic derivative from `derivative_expr()` in Listing 1, and evaluates it at specific x values using `sympy.lambdify` to create a numerical version of the derivative.

```
def evaluate_derivative(var, func, values):
    """
    Compute the derivative of a function and evaluate.
    :param var: Variable with respect to which differentiation is
         performed.
    :param func: Mathematical expression to differentiate.
    :param values: Values to evaluate function with.
    :return: Evaluated function.
    """
    derivative = derivative_expr(var, func)
    derivative_numeric = sympy.lambdify(var, derivative, 'numpy')
    return derivative_numeric(values)
```

Listing 2: Function to evaluate derivative.

Listing 3 provides the function I created to calculate the finite difference estimate. It is based on the finite difference formula below:

$$\frac{f(x+\delta)-f(x)}{\delta} \tag{1}$$

It first uses `sympy.lambdify` to convert a symbolic expression into a numerical function that can be evaluated. Following that it performs the finite difference calculation by computing the difference between the function's value at $x+\delta$ and its value at x. The difference is then divided by δ to approximate the rate of change at that point. The function returns this calculated estimate of the derivative at the points in `x_values`.

```
def finite_difference(func, var, delta, x_values):
    """
    Compute the finite difference estimate of a function.
    :param func: Mathematical expression provided
    :param var: Variable used within the function
    :param delta: Value used to calculate the finite difference -
         step size.
    :param x_values: Values used to evaluate the finite
        difference.
    :return: The calculated finite difference estimate.
    """
    func_numeric = sympy.lambdify(var, func, 'numpy')
    return (func_numeric(x_values + delta) - func_numeric(
        x_values)) / delta
```

Listing 3: Function to calculate finite difference estimate.

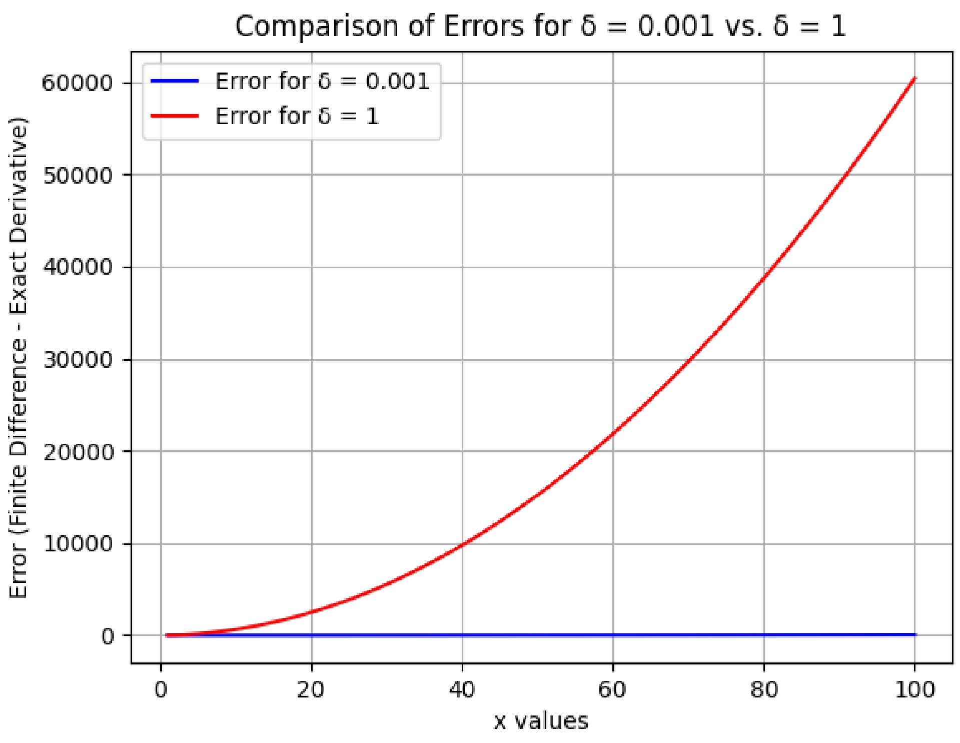

Figure 2: Comparison of errors for varying delta values

(iii)

Figure 2 provides a plot comparing the errors for $\delta = 0.001$ and $\delta = 1$.

From the plot, $\delta = 0.001$ results in a much more accurate approximation of the derivative in comparison to $\delta = 1$. The error is pretty much a flat line at 0 for every x value. For this function, a smaller step size allows for the finite difference estimation to accurately capture the changes in the function's slope.

In comparison, when $\delta = 1$ the error increases as the x values grow. In this case, the finite difference deviates a lot from the actual derivative. This is a result of the step size becoming too large and introducing a significant error in the approximation. For this function, when δ is too large, the finite difference estimation struggles to capture the function's precise behaviour. When δ is large, the two points x and $x+\delta$ are farther apart. This means that the finite difference approximation is capturing a larger segment of the curve i.e. the steps are much bigger. Ultimately, making the step size too large causes the steps to bounce back and forth and completely miss the minimum, causing it to diverge.

Part B

(i)

```python
def grad_descent(fn, x0, alpha, num_iters, max_val=1e6):
    x = x0  # starting point
    X = np.array([x])  # convert starting point into array object
    F = np.array(fn.f(x))  # convert function into array object
    for i in range(num_iters):
        step = alpha * (fn.df(x))
        x = x - step
        # Stop if x explodes
        if abs(x) > max_val:
            print(f"Diverged at iteration {i} with x = {x},
                stopping early.")
            break

        X = np.append(X, [x], axis=0)
        F = np.append(F, fn.f(x))
    return X, F
```

Listing 4: Function to implement gradient descent.

This function takes in the mathematical function to perform gradient descent on (`fn`), the initial starting point (`x0`), the step size (`alpha`) and the number of iterations to be performed (`num_iters`).

The algorithm starts at `x0` and then for the number of iterations to be performed the gradient is caclulated by multiplying `alpha` by the derivative of the function at the point `x`. Following that the next step i.e. point `x`, is calculated by getting the difference between the previous `x` and the step size. There is also a divergence check for when `x` becomes too large meaning the algorithm is diverging instead of converging. At the end of the loop each `x` point is stored, keeping track of all values throughout the iterations, and the function value at `x` is stored to keep track of how the function changes over time.

(ii)

Figure 3 provides two plots showing how x and $y(x)$ vary with each gradient descent iteration when $x = 1$ and step sizes $\alpha = 0.1$. From the figure, we can see both x values and function values are decreasing at every iteration indicating that gradient descent is converging and minimising the function by moving towards a local minimum. From the subplot with the function values, it is clear they are decreasing at every iteration. They are decreasing extremely quickly with nearly a completely vertical line down to 0 within approximately

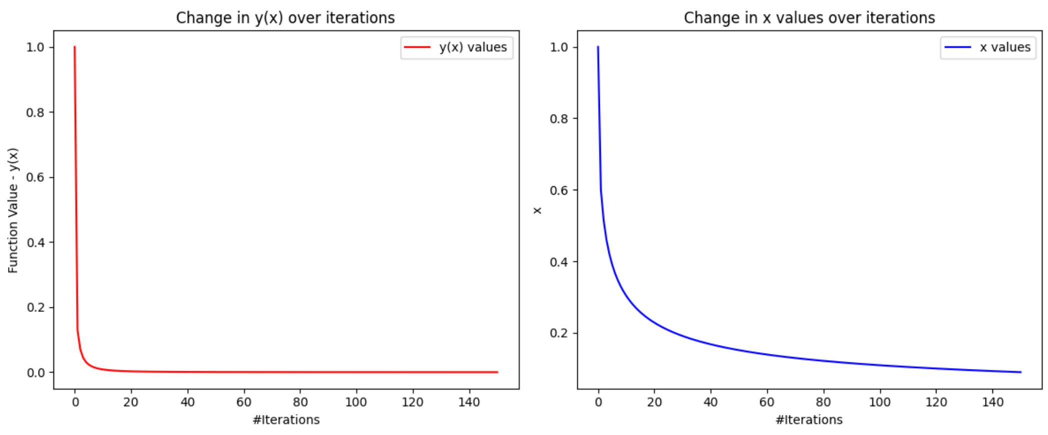

Figure 3: Subplots of x and f(x) values over iterations.

5 iterations. However, after that point we can see the decrease towards 0 gets smaller as the number of iterations increases. The input x also decreases from 1 towards 0 which we know is our optimum value, however, it has a slightly more curved decrease meaning that the x-values are reaching 0 at a slower rate compared to the function values.

(iii)

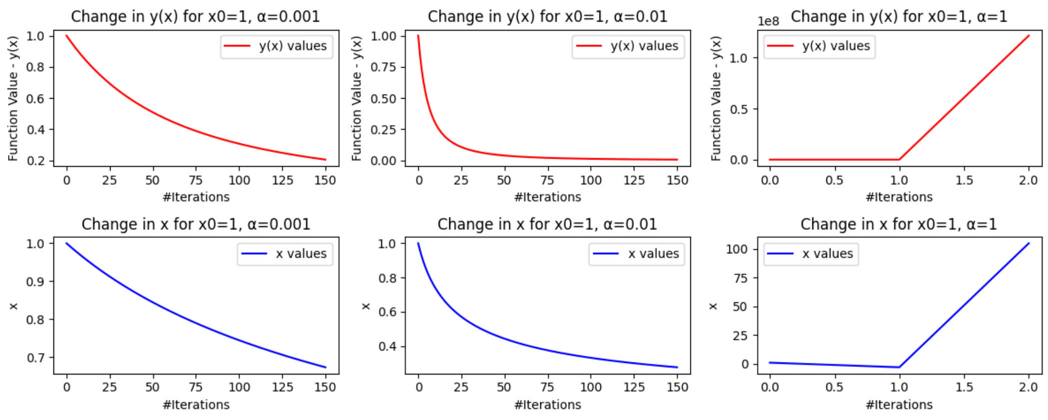

Figure 4: Subplots of varying alpha values for initial point x=1.

Comparing all figures, we can see that when the α values are small, the gradient descent steps will be very small, causing the algorithm to converge slowly. In particular, if we focus on figure 5, it is clear to see that when $\alpha = 0.001$ the x values reach a lowest point of just below 1, where in comparison to $\alpha = 0.01$ it reaches a much lower point of below 0.5 for the same number of iterations. This indicates that to reach convergence with a too-small α value, you would need a larger number of iterations. This is directly because α indicates the step size of the gradient descent and the smaller the step size, ultimately means it'll take smaller steps and need more of them to converge. Overall it will be more stable and converge but it will take a much longer amount of time.

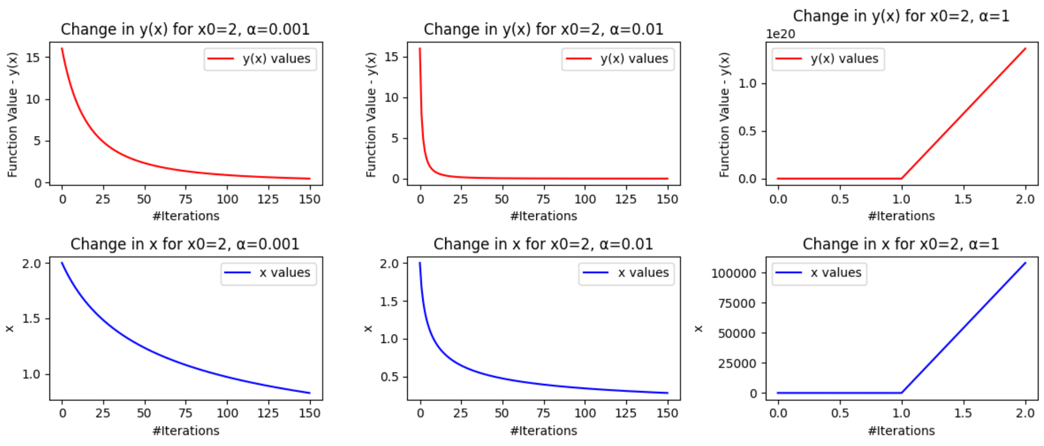

Figure 5: Subplots of varying alpha values for initial point x=2.

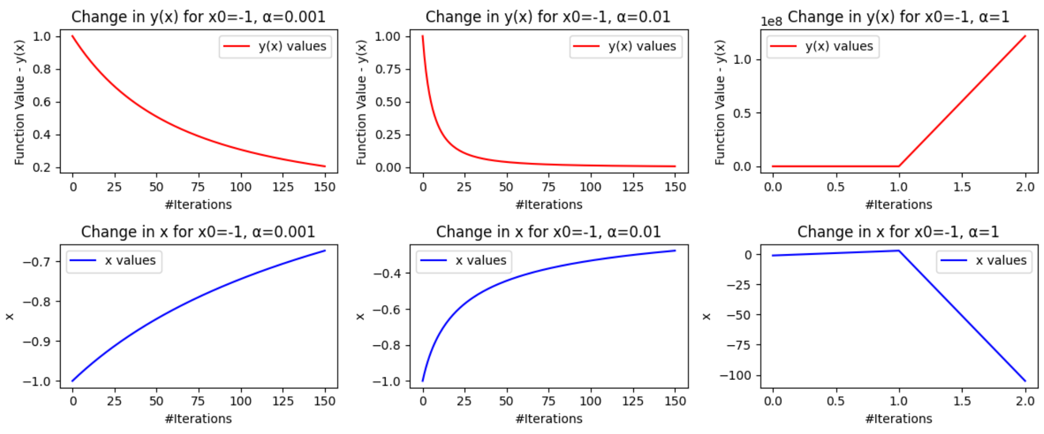

Figure 6: Subplots of varying alpha values for initial point x=-1.

The initial starting point affects how many iterations are needed to complete convergence as we can see when the starting point is 2, it takes a greater number of iterations for the x point to reach 0. This is a result of the distance from the initial point to the function's minimum. The further away this value is from the function's minimum, in this case, $x = 0$ is the minimum of $y = x^4$, the more iterations may be needed for convergence.

Additionally, we can see that every time $\alpha = 1$, for these initial values that step size is far too large. It causes the gradient descent algorithm to diverge and start heading away from the minimum. In this case, using this α value caused the algorithm to overshoot the minimum and start bouncing back and forth. This is because the step size is too big and taking such a large step causes it to jump past the minimum.

Overall a balance is needed between the initial starting point and step size to have a good rate of convergence and accuracy. In this case, a step size of $\alpha = 0.01$ was ideal for this function and choosing an initial point close to 0 was necessary for achieving quick and precise convergence.

Part C

(i)

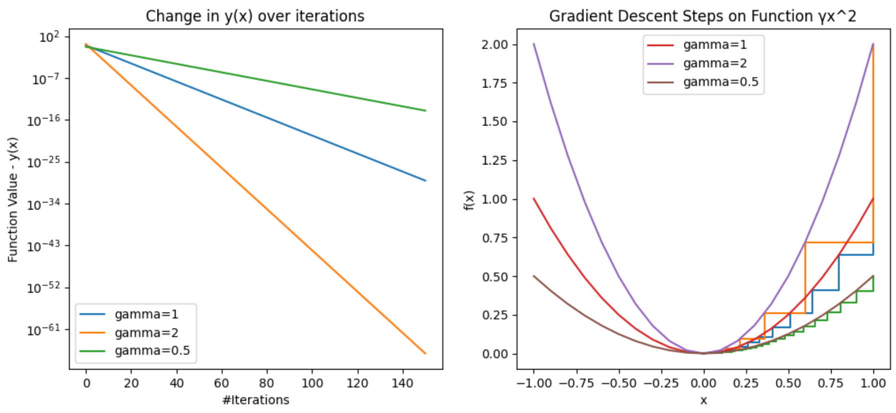

Figure 7: Subplots of changing function values and gradient descent steps.

Figure 7, features subplots for the function γx^2. It has a subplot of the changing function values for varying γ values and another subplot of gradient descent steps with the same γ values, with a fixed step-size $\alpha = 0.1$. As we can see from these subplots as γ gets larger the function becomes steeper. As γ gets smaller the function becomes shallower. When using $\alpha = 0.1$, the larger γ is, the faster it appears to be converging. This is because when γ is bigger it is steeper, so with this small α value it is taking a bigger step without it being too large causing it to overshoot past the minimum. This is evident from the subplot for gradient descent steps where $\gamma = 2$, the step sizes (in orange) begin very large and decrease faster than for the other two γ values. This is also confirmed by the subplot showcasing the change in function values for each γ value, the orange i.e. $\gamma = 2$, decreases the fastest to 0.

Overall, it is important to note that the choice of step size depends on the gradient of the function, if it is shallow it will decrease slower so you will need a bigger step size. If the gradient is steep, it has a larger gradient, then it decreases faster so you will need to use a smaller step size.

(ii)

Figure 8, has subplots for the function $\gamma|x|$. It has a subplot of the changing function values for varying γ values and another subplot of gradient descent steps with the same γ values, with a fixed step-size $\alpha = 0.1$.

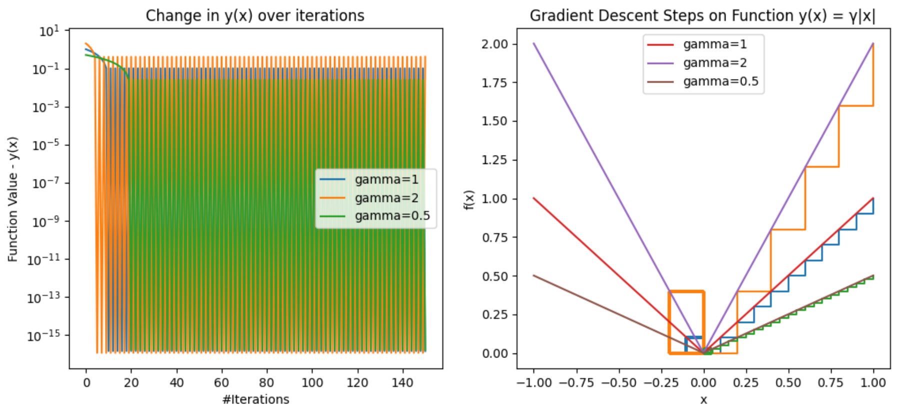

Figure 8: Subplots of changing function values and gradient descent steps.

The reason why these plots look drastically different to figure 7, is because the function we are performing the algorithm on has a constant slope. the function is not smooth when $x = 0$. When $x > 0$ the derivative of $\gamma|x|$ is γ, and when $x < 0$ the derivative is $-\gamma$. When $x = 0$ the derivative doesn't exist at the point causing the gradient descent algorithm to struggle to converge.

Similarly to figure 7, making γ larger causes the function to become steeper, and shallower for smaller γ values. Convergence for this function with $\alpha = 0.1$ is never achieved, as we can see from the plots. For each γ value the step sizes for each function are constant, they decrease steadily until they reach the kink when the function completely changes direction. It decreases until it reaches the vicinity of the minimum and then oscillates forever due to the kink in the function, which causes the derivative sign to flip abruptly.

Code

derivative.py

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


def derivative_expr(var, func):
    """
    Function to get the derivative of an expression.
    :param var: Variable with respect to which differentiation is
       performed.
    :param func: Mathematical expression to differentiate.
    :return: Derivative of the expression.
    """
    return sympy.diff(func, var)


def evaluate_derivative(var, func, values):
    """
    Compute the derivative of a function and evaluate.
    :param var: Variable with respect to which differentiation is
       performed.
    :param func: Mathematical expression to differentiate.
    :param values: Values to evaluate function with.
    :return: Evaluated function.
    """
    derivative = derivative_expr(var, func)
    derivative_numeric = sympy.lambdify(var, derivative, 'numpy')
    return derivative_numeric(values)


def finite_difference(func, var, delta, x_values):
    """
    Compute the finite difference estimate of a function.
    :param func: Mathematical expression provided.
    :param var: Variable used within the function.
    :param delta: Value used to calculate the finite difference - step
        size.
    :param x_values: Values used to evaluate the finite difference.
    :return: The calculated finite difference estimate.
    """
    func_numeric = sympy.lambdify(var, func, 'numpy')
    return (func_numeric(x_values + delta) - func_numeric(x_values)) /
        delta


# (a)(i)
```

```
x = sympy.symbols('x', real=True)
f = x ** 4
dfdx = derivative_expr(x, f)
print(dfdx)

# (ii)
x_values = np.arange(1, 101)
calculations = evaluate_derivative(x, f, x_values)

delta = 0.01
finite_derivatives = finite_difference(f, x, delta, x_values)


def plot_comparison(x, y1, y2, xlim_range, ylim_range, y1_label,
    y2_label, xaxis_label, yaxis_label, title, file_name):
    plt.clf()
    plt.figure()
    plt.plot(x, y1, label=y1_label, color='b')
    plt.plot(x, y2, label=y2_label, color='r')

    if xlim_range or ylim_range:
        plt.xlim(xlim_range)
        plt.ylim(ylim_range)

    plt.xlabel(xaxis_label)
    plt.ylabel(yaxis_label)
    plt.title(title)
    plt.legend()
    plt.savefig("images/" + file_name)


plot_comparison(x_values, calculations, finite_derivatives, None, None
    , 'Derivative Evaluations',
                'Finite Difference', 'x', 'y',
                'Comparing Derivative Evaluation to Finite Difference'
                    , 'part_2.png')


x_values = np.arange(1, 101)
calculations = evaluate_derivative(x, f, x_values)

# (iii)
finite_derivatives = finite_difference(f, x, 0.001, x_values)
error = finite_derivatives - calculations  # Error between finite
    difference and exact derivative
errors_0001 = error

finite_derivatives = finite_difference(f, x, 1, x_values)
error = finite_derivatives - calculations
```

```
errors_1 = error

plt.figure()
plt.plot(x_values, errors_0001, label='Error for δ = 0.001', color='b'
    )  # Plot for δ = 0.001
plt.plot(x_values, errors_1, label='Error for δ = 1', color='r')  #
    Plot for δ = 1

plt.xlabel('x values')
plt.ylabel('Error (Finite Difference - Exact Derivative)')
plt.title('Comparison of Errors for δ = 0.001 vs. δ = 1')
plt.legend()
plt.grid(True)
plt.savefig('images/delta_error_comparison_p3.png')
```

gradient_descent.py

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


class QuadraticFn:
    def f(self, x):
        return x ** 4  # Function value f(x)

    def df(self, x):
        return 4 * x ** 3  # Derivative of f(x)


class PolynomialFn:
    def __init__(self, a):
        self.a = a  # Coefficient for x^2

    def f(self, x):
        return self.a * x ** 2

    def df(self, x):
        return 2 * self.a * x


class AbsoluteFn:
    def __init__(self, gamma):
        self.gamma = gamma

    def f(self, x):
        return self.gamma * np.abs(x)  # Function value y(x) = γ|x|

    def df(self, x):
        # Derivative of γ|x| is γ for x > 0 and -γ for x < 0
        # Handle the derivative at x=0 as 0 or a small number to avoid
            issues
        if x > 0:
            return self.gamma
        elif x < 0:
            return -self.gamma
        else:
            return 0


# (b) (i)
def grad_descent(fn, x0, alpha, num_iters, max_val=1e6):
    """
    Implements gradient descent with step size alpha.
    :param fn: Function to implement gradient descent on.
    :param x0: Initial starting point.
    :param alpha: Step size.
    :param num_iters: Number of iterations.
    :param max_val: Stopping threshold.
    :return: Arrays of X values and function values.
    """
    x = x0  # starting point
    X = np.array([x])  # convert starting point into array object
    F = np.array(fn.f(x))  # convert function into array object
    for i in range(num_iters):
        step = alpha * (fn.df(x))
        x = x - step
        # Stop if x explodes
        if abs(x) > max_val:
            print(f"Diverged at iteration {i} with x = {x}, stopping
                early.")
            break

        X = np.append(X, [x], axis=0)
        F = np.append(F, fn.f(x))
    return X, F


# (ii)
func = QuadraticFn()

(X, F) = grad_descent(func, x0=1, alpha=0.1, num_iters=150)

fig, ax = plt.subplots(1, 2, figsize=(12, 5))
ax[0].plot(F, label="y(x) values", color='r')
ax[0].set_xlabel('#Iterations')
ax[0].set_ylabel('Function Value - y(x)')
ax[0].set_title('Change in y(x) over iterations')
ax[0].legend()
```

```
ax[1].plot(X, label="x values", color='b')
ax[1].set_xlabel('#Iterations')
ax[1].set_ylabel('x')
ax[1].set_title('Change in x values over iterations')
ax[1].legend()

plt.tight_layout()
plt.savefig(f'images/grad_descent_subplot.png')


# (iii)
fig, ax = plt.subplots(2, 3, figsize=(12, 5))
initial_values = [8, 2, -1]
alpha_values = [0.001, 0.01, 1]
for i in initial_values:
    for k in alpha_values:
        (X, F) = grad_descent(func, x0=i, alpha=k, num_iters=150)

        alpha_index = alpha_values.index(k)
        ax[0, alpha_index].clear()
        ax[1, alpha_index].clear()

        ax[0, alpha_index].plot(F, label="y(x) values", color='r')
        ax[0, alpha_index].set_xlabel('#Iterations')
        ax[0, alpha_index].set_ylabel('Function Value - y(x)')
        ax[0, alpha_index].set_title(f'Change in y(x) for x0={i}, α={k
            }')
        ax[0, alpha_index].legend()

        ax[1, alpha_index].plot(X, label="x values", color='b')
        ax[1, alpha_index].set_xlabel('#Iterations')
        ax[1, alpha_index].set_ylabel('x')
        ax[1, alpha_index].set_title(f'Change in x for x0={i}, α={k}')
        ax[1, alpha_index].legend()

    plt.tight_layout()
    plt.savefig(f'images/part2/subplot_x0={i}.png')

plt.figure()
# (c) (i)
poly1 = PolynomialFn(1)
poly2 = PolynomialFn(2)
poly3 = PolynomialFn(0.5)

(X, F) = grad_descent(poly1, x0=1, alpha=0.1, num_iters=150)
(X2, F2) = grad_descent(poly2, x0=1, alpha=0.1, num_iters=150)
(X3, F3) = grad_descent(poly3, x0=1, alpha=0.1, num_iters=150)
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

ax[0].plot(F, label="gamma=1")
ax[0].plot(F2, label="gamma=2")
ax[0].plot(F3, label="gamma=0.5")
ax[0].set_xlabel('#Iterations')
ax[0].set_ylabel('Function Value - y(x)')
ax[0].set_title('Change in y(x) over iterations')
ax[0].set_yscale('log')
ax[0].legend()

# Plot the gradient descent steps for each function
ax[1].step(X, poly1.f(X))
ax[1].step(X2, poly2.f(X2))
ax[1].step(X3, poly3.f(X3))

# Plot the actual functions
xx = np.arange(-1, 1.1, 0.1)
ax[1].plot(xx, poly1.f(xx), label="gamma=1")
ax[1].plot(xx, poly2.f(xx), label="gamma=2")
ax[1].plot(xx, poly3.f(xx), label="gamma=0.5")
ax[1].set_xlabel('x')
ax[1].set_ylabel('f(x)')
ax[1].set_title('Gradient Descent Steps on Function γx^2')
ax[1].legend()
plt.savefig(f'images/gamma_comparison_subplot.png')

# (c)(ii)
plt.figure()
abs_fn1 = AbsoluteFn(1)
abs_fn2 = AbsoluteFn(2)
abs_fn3 = AbsoluteFn(0.5)

(X, F) = grad_descent(abs_fn1, x0=1, alpha=0.1, num_iters=150)
(X2, F2) = grad_descent(abs_fn2, x0=1, alpha=0.1, num_iters=150)
(X3, F3) = grad_descent(abs_fn3, x0=1, alpha=0.1, num_iters=150)

fig, ax = plt.subplots(1, 2, figsize=(12, 5))

ax[0].plot(F, label="gamma=1")
ax[0].plot(F2, label="gamma=2")
ax[0].plot(F3, label="gamma=0.5")
ax[0].set_xlabel('#Iterations')
ax[0].set_ylabel('Function Value - y(x)')
ax[0].set_title('Change in y(x) over iterations')
ax[0].set_yscale('log')
ax[0].legend()

# Plot the gradient descent steps for each function
```

```
ax[1].step(X, abs_fn1.f(X))
ax[1].step(X2, abs_fn2.f(X2))
ax[1].step(X3, abs_fn3.f(X3))

# Plot the actual functions
xx = np.arange(-1, 1.1, 0.1)
ax[1].plot(xx, abs_fn1.f(xx), label="gamma=1")
ax[1].plot(xx, abs_fn2.f(xx), label="gamma=2")
ax[1].plot(xx, abs_fn3.f(xx), label="gamma=0.5")
ax[1].set_xlabel('x')
ax[1].set_ylabel('f(x)')
ax[1].set_title('Gradient Descent Steps on Function y(x) = γ|x|')
ax[1].legend()
plt.savefig(f'images/absolute_comparison_subplot.png')
```