A)

1)For the first step, Sympy library was used for symbolic mathematics to compute derivatives of the function f(x) = x*4, starting by initiating the variable x and defining f(x) as x*4 then calculating derivatives as shown below.

```
#1)
x = sp.symbols('x', real=True)
f = x ** 4
f_x = sp.diff(f, x)
print(f_x)
```

The result obtained was:

```
PS C:\Trinity College dub
nity College dublin/Sem-2
4*x**3
```

2) A finite difference approximation was calculated using above equation with δ= 0.01, then derivative value and finite difference approximation both were plotted across values ranging from -150 to 150 and step size of 10. The plot shows the exact derivative (in red) and the finite difference approximation (in blue, dotted line) for δ = 0.01. It can be observed that both that both lines are coinciding so we can say that finite difference approximation method is effective for derivative in this case.

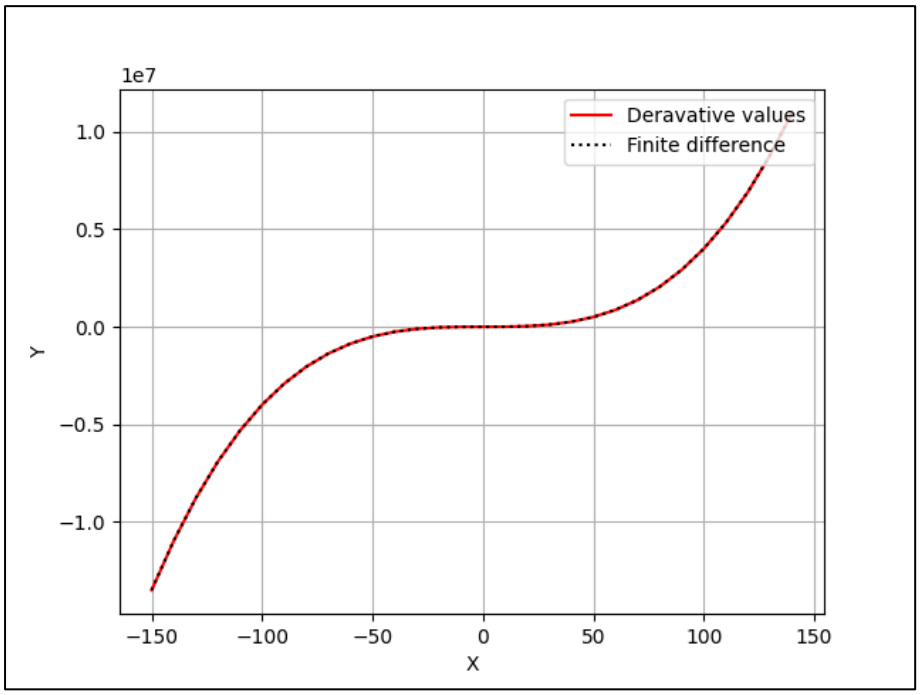

Fig: Comparison of Derivative and Finite Difference for δ=0.01

3) We observe finite difference method to approximate the derivative on different δ values: 0.001, 0.01, 0.1, 0.5, and 1, to understand its impact on the accuracy of the approximation.

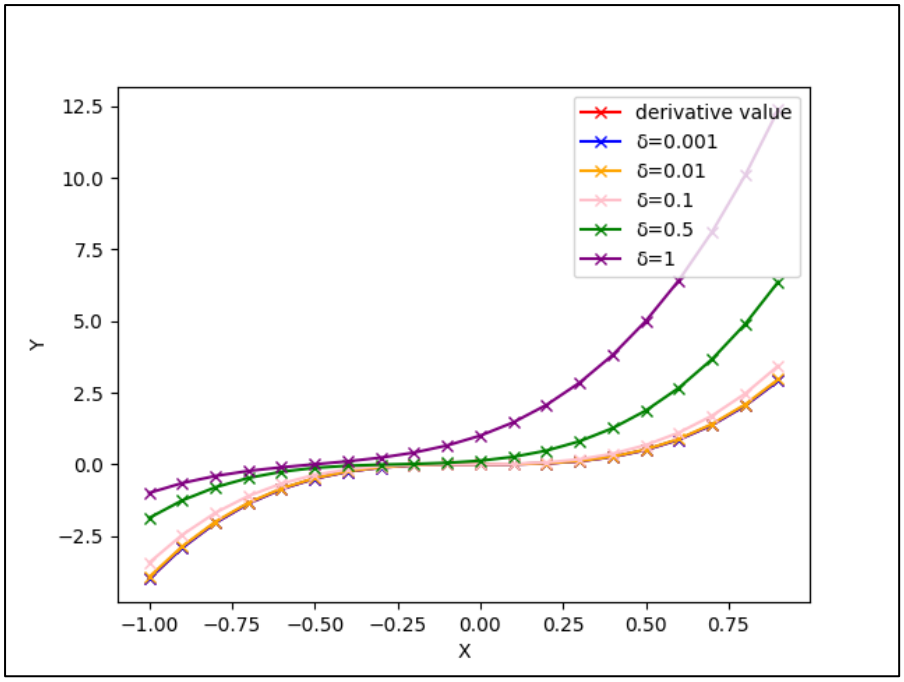

Fig: Comparison of Exact Derivative with Finite Differences for Various δ

It can be observed that when we use smaller δ values, like 0.001 and 0.01, the proximity of the points ensures a slope that closely mimics the actual slope at each point. As a result, these approximations are highly accurate, aligning closely with the exact derivative curve. However, as δ increases, the distance between points widens, leading to a less precise slope that may not capture the subtleties of the function's curvature. This can be seen with larger δ values (0.1, 0.5, 1), where the approximations show a noticeable deviation from the exact derivative, particularly in areas where the function curves significantly.

The visual comparison tells us that the nearer a finite difference curve is to the exact derivative, the more accurate the approximation. This graphical representation not only makes the data accessible but also vividly illustrates the nuanced trade-off between δ size and approximation accuracy.

B)

1) The function starts with an initial value for gradient descent and uses a constant step size, known as alpha, alongside a predetermined count of iterations. It maintains two arrays; one captures the sequence of x-values, and the other logs the corresponding derivative values. Within each iteration, the function adjusts the x0 value in a manner that maximally reduces the given expression, modulated by the factor alpha. Concurrently, it keeps a record of the x-values and their associated derivative values at every step.

```python
#1)
def gradient_descent(derivative_f_x, finite_f_x, x0, alpha=0.15, num_iters=50):
    x = x0
    X = np.array([x])
    F = np.array(derivative_f_x(x))
    for k in range(num_iters):
        step = alpha * finite_f_x(x)
        x = x - step
        X = np.append(X, [x], axis=0)
        F = np.append(F, derivative_f_x(x))
    return (X, F)
```

2) The figure shown below shows the gradient descent algorithm for 50 Iterations. The red line, indicative of the derivative, starts high, reflecting a steep initial slope at x0 = 1. As the algorithm progresses, this line plummets, signifying a move toward flatter terrain, which corresponds to a function's minimum. The blue line, tracking x-values, mirrors this journey, initially descending sharply in response to the steep slope, then moderating as the slope flattens. The convergence of both lines, as they level off, suggests that the algorithm is homing in on a stable point, likely a local minimum, as the derivative nears zero. The learning rate is well-tuned, allowing for swift, but controlled progression toward the minimum without overshooting.

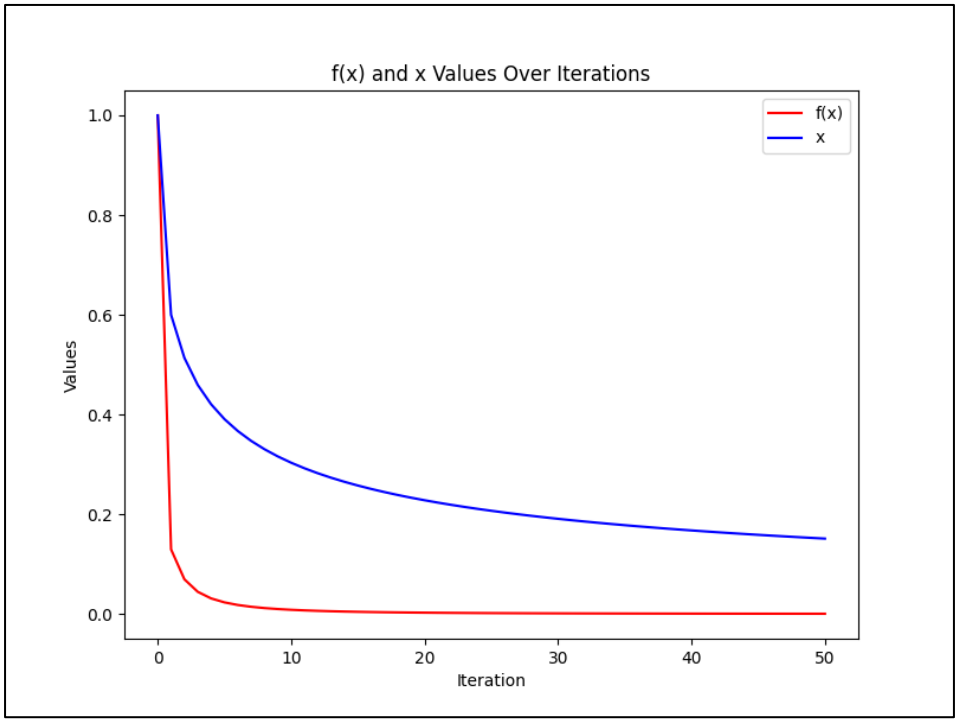

Fig: Gradient descent convergence over 50 iteration

3) The given below plots of gradient descent are for values ranging from 0.5 to 0. The steps show a trend of moving towards the minimum of the function, as indicated by where the derivative curve nears or touches the x-axis. If the steps stop before reaching the x-axis, it means the gradient descent has converged to a local minimum.

Each plot corresponds to a different x value for the algorithm. If the green path leads to the bottom of the magenta curve, it indicates that the gradient descent has likely found the minimum from that starting point. It can be seen that if x value is smaller the gradient descent drops slower and as x values increases the gradient descent drops faster.

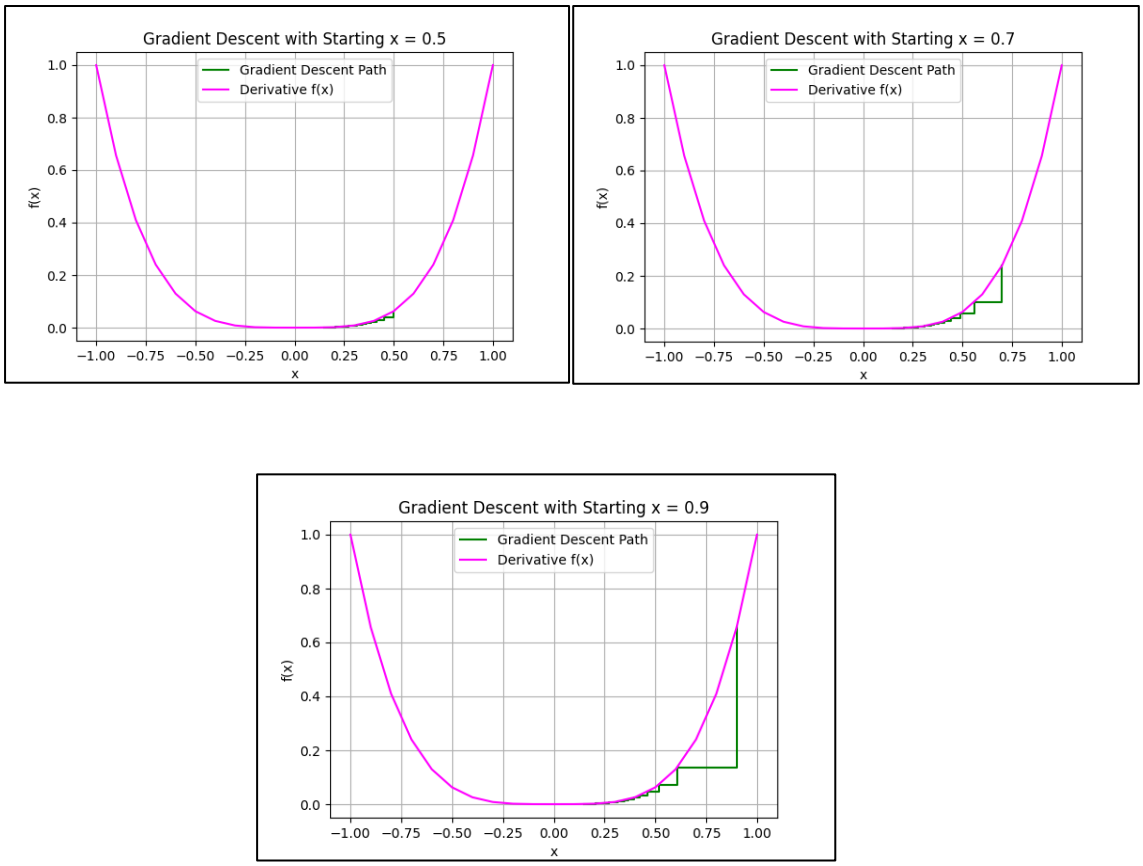

Fig: Gradient descent for different x values

The given below graphs are for different alpha values like 0.01,0.1,0.2,0.3. The Orange curve represents the derivative of the function over the interval. Cyan steps how the trajectory of the gradient descent algorithm for each alpha value. The number of steps and their length reflect how quickly the algorithm moves towards the function's minimum. It can be seen that smaller alpha shows a longer, more gradual path towards the minimum, indicating a slower convergence. As alpha increases, the steps become larger, and the path shows a more direct route to the minimum.

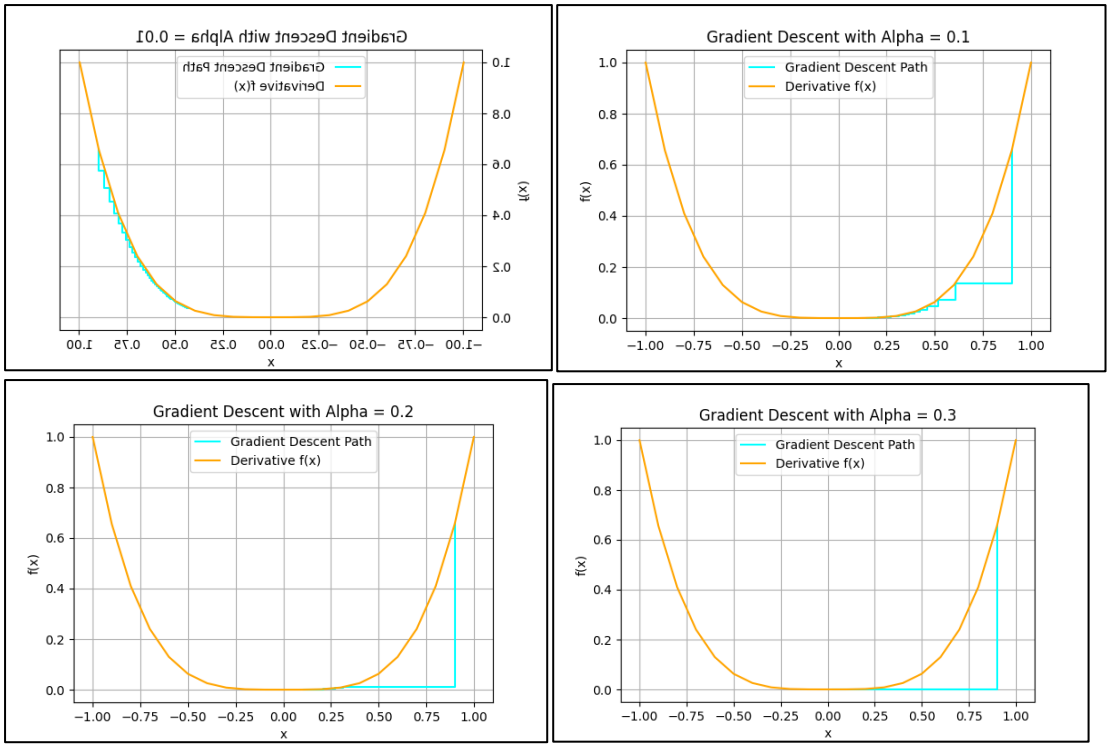

Fig: Gradient descent for different alpha values

C)

1) When using gradient descent on the equation y(x)=γx ,it can be seen that changing the value of γ changes how quickly the function's curve goes up or down. If γ is big, the function gets steeper, and this makes the gradient descent steps bigger too as they're based on the slope of the curve. This means we get to the lowest point faster, but there is also the chance that we also might miss it if we're not careful. If γ is small, the function is gentler, leading to smaller, more careful steps, which makes it easier to find the lowest point accurately.

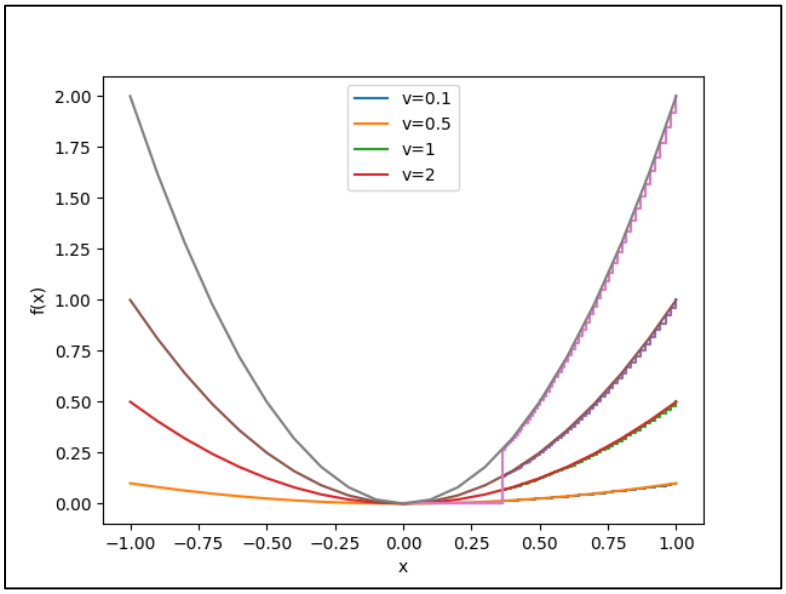

Fig: Effects on function y(x)=γx for different gamma.

2) In gradient descent for the function y(x)=γ|x|, changing γ changes how pointy the V-shaped curve is. A bigger γ means bigger jumps toward the lowest point, but there's a risk of jumping past it. If γ is small, the jumps are smaller and more careful, which helps in getting closer to the lowest point without missing it.

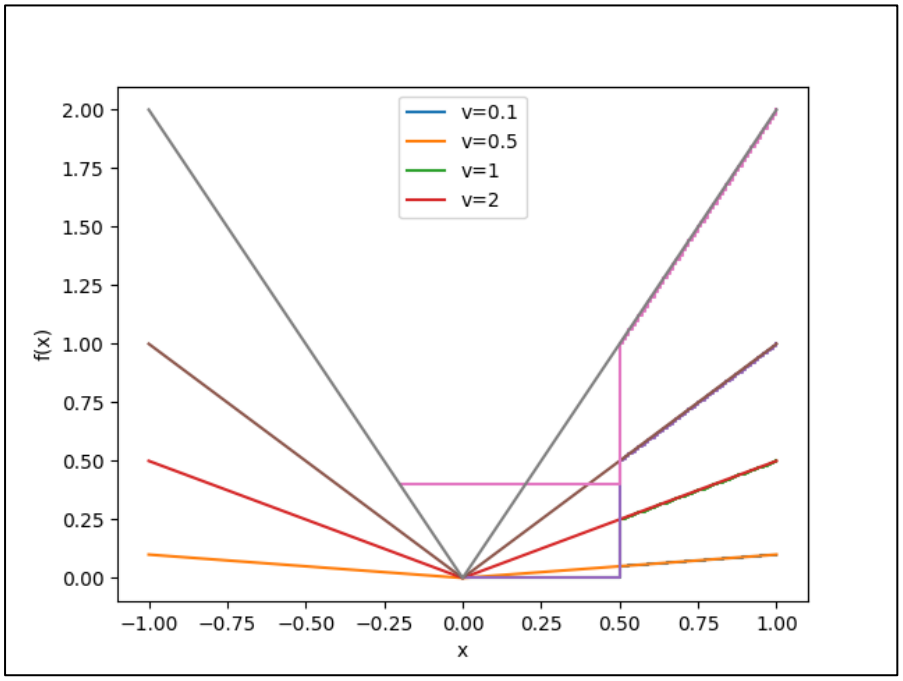

Fig: Effects on function y(x)=γ|x| for different gamma.

CODE APPENDIX

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

#A-Part
#1)
x = sp.symbols('x', real=True)
f = x ** 4
f_x = sp.diff(f, x)
print(f_x)

#2)
x = sp.symbols('x', real=True)
```

```
derivative_f_x = sp.lambdify(x, f_x)  # Derivative function

f = x ** 4
delta = 0.01
finite_difference = ((x + delta) ** 4 - x ** 4) / delta #Formula
finite_f_x = sp.lambdify(x, finite_difference)  # Finite difference function

X = np.arange(-150, 150, 10)
arr_1 = []
arr_2 = []

for x in X:
    arr_1.append(derivative_f_x(x))
    arr_2.append(finite_f_x(x))
plt.xlabel('X')
plt.ylabel('Y')
plt.plot(X, arr_1, label='Derivative value', color='red')
plt.plot(X, arr_2, label='Finite difference',
color='black',linestyle='dotted')
plt.legend(["Deravative values",'Finite difference'],loc='upper right')
plt.grid(True)
plt.show()
#3)
x = sp.symbols('x', real=True)
derivative_f_x = sp.lambdify(x, f_x)  # Derivative function

f = x ** 4
X = np.arange(-1, 1, 0.1)

def get_function(x, delta):
    finite_difference = ((x + delta) ** 4 - x ** 4) / delta
    return sp.lambdify(x, finite_difference)

finite_f_x = get_function(x, delta=0.001)
A = get_function(x, delta=0.01)
B = get_function(x, delta=0.1)
C = get_function(x, delta=0.5)
D = get_function(x, delta=1)

arr_1 = [derivative_f_x(x_val) for x_val in X]
arr_2 = [finite_f_x(x_val) for x_val in X]
Y3 = [A(x_val) for x_val in X]
Y4 = [B(x_val) for x_val in X]
Y5 = [C(x_val) for x_val in X]
Y6 = [D(x_val) for x_val in X]

plt.xlabel('X')
plt.ylabel('Y')
```

```
plt.plot(X, arr_1, 'x-', c='red', label='derivative value')
plt.plot(X, arr_2, 'x-', c='blue', label='δ=0.001')
plt.plot(X, Y3, 'x-', c='orange', label='δ=0.01')
plt.plot(X, Y4, 'x-', c='pink', label='δ=0.1')
plt.plot(X, Y5, 'x-', c='green', label='δ=0.5')
plt.plot(X, Y6, 'x-', c='purple', label='δ=1')
plt.legend(loc='upper right')
plt.show()

def derivative_f_x(x):
    return x ** 4

def finite_f_x(x):
    return 4*x**3

def A(x, gamma):
    return gamma*x**2

def B(x, gamma):
    return gamma*2*x

def C(x, gamma):
    return gamma*abs(x)

def D(x, gamma):
    if x< 0:
        return -gamma
    return gamma
#B-Part
#1)
def gradient_descent(derivative_f_x, finite_f_x, x0, alpha=0.15,
num_iters=50):
    x = x0
    X = np.array([x])
    F = np.array(derivative_f_x(x))
    for k in range(num_iters):
        step = alpha * finite_f_x(x)
        x = x - step
        X = np.append(X, [x], axis=0)
        F = np.append(F, derivative_f_x(x))
    return (X, F)

# b.ii
x0 = 1
alpha = 0.1
(X, F) = gradient_descent(derivative_f_x, finite_f_x, x0=x0, alpha=alpha)
xx = np.arange(-1, 1.1, 0.1)
```

```
plt.figure(figsize=(8, 6))

plt.plot(F, label='f(x)', color='red')

plt.plot(X, label='x', color='blue')
plt.xlabel('Iteration')
plt.ylabel('Values')
plt.title('f(x) and x values Over Iterations')
plt.legend()
plt.show()


# b.iii
# Gradient Descent for Different X Values
for x in np.arange(0.5, 1, 0.1):
    x0 = x
    alpha = 0.1
    (X, F) = gradient_descent(derivative_f_x, finite_f_x, x0=x0, alpha=alpha)
    xx = np.arange(-1, 1.1, 0.1)

    plt.figure(figsize=(6, 4))
    plt.step(X, derivative_f_x(X), color='green', label='Gradient Descent
Path')
    plt.plot(xx, derivative_f_x(xx), color='magenta', label='Derivative f(x)')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title(f'Gradient Descent with Starting x = {x:.1f}')
    plt.legend()
    plt.grid(True)
    plt.show()


# Gradient Descent for Different Alpha Values
for alpha in [0.01, 0.1, 0.2, 0.3]:
    (X, F) = gradient_descent(derivative_f_x, finite_f_x, x0=x0, alpha=alpha)
    xx = np.arange(-1, 1.1, 0.1)

    plt.figure(figsize=(6, 4))
    plt.step(X, derivative_f_x(X), color='cyan', label='Gradient Descent
Path')
    plt.plot(xx, derivative_f_x(xx), color='orange', label='Derivative f(x)')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title(f'Gradient Descent with Alpha = {alpha}')
    plt.legend()
    plt.grid(True)
    plt.show()

#c.i
```

```
gamma= 0.1
x=1
A_1= np.array([x])
B_1 = np. array(A(x, gamma))
for k in range(50):
    step = 0.1 * B(x, gamma)
    x=x-step
    X_1 = np.append(A_1, [x] , axis=0)
    F_1=np.append(B_1, A(x,gamma))
gamma= 0.5
x=1
X_2= np.array([x])
F_2 = np.array(A(x, gamma))
for k in range(50):
    step= 0.1* B(x,gamma)
    x=x-step
    X_2 = np.append(A_1, [x] , axis=0)
    F_2 = np.append(B_1, A(x, gamma))
gamma= 1
x=1
X_3 = np.array([x])
F_3 = np.array(A(x, gamma))
for k in range(50):
    step=0.1* B(x,gamma)
    x=x-step
    X_3 = np.append(A_1, [x] , axis=0)
    F_3 = np.append(B_1, A(x, gamma))
gamma = 2
x = 1
X_4 =np.array([x])
F_4 = np.array(A(x, gamma))
for k in range(50):
    step=0.1* B(x,gamma)
    x=x-step
    X_4 = np.append(A_1, [x] , axis=0)
    F_4 = np.append(B_1, A(x, gamma))
xx=np.arange(-1,1.1,0.1)


plt.step(X_1, A(X_1, 0.1))
plt.plot(xx, A(xx, 0.1))
plt.step(X_2, A(X_2, 0.5 ) )
plt.plot(xx, A(xx, 0.5))
plt.step(X_3, A(X_3, 1 ) )
plt.plot(xx,A(xx,1))
plt.step(X_4, A(X_4, 2 ) )
plt.plot(xx, A(xx, 2 ) )
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
plt.legend(['v=0.1','v=0.5','v=1','=2'])
plt.show( )

##c.ii
gamma = 0.1
x=1
A_1 = np.array([x])
B_1 = np.array(C(x, gamma))
for k in range(50):
    step=0.1* D(x,gamma)
    x=x-step
    X_1= np.append(A_1,[ x ] ,axis=0)
    F_1 = np.append(B_1, C(x, gamma))
gamma= 0.5
x=1
X_2 = np.array([x])
F_2 = np.array(C(x, gamma))
for k in range(50):
    step = 0.1 * D(x, gamma)
    x=x-step
    X_2= np.append(A_1,[ x ] ,axis=0)
    F_2=np.append(B_1, C(x,gamma))
gamma = 1
x=1
X_3 = np.array([x])
F_3=np.array(C(x,gamma))
for k in range(50):
    step=0.1* D(x,gamma)
    x=x-step
    X_3 = np.append(A_1, [ x ] , axis=0)
    F_3 = np.append(B_1, C(x, gamma))
gamma= 2
x=1
X_4= np.array([x])
F_4=np.array(C(x,gamma))
for k in range(50):
    step =0.1 * D(x, gamma)
    x=x-step
    X_4 = np.append(A_1, [ x ] , axis=0)
    F_4=np.append(B_1, C(x,gamma))
xx= np.arange(-1, 1.1 ,0.1)


plt.step(X_1, C(X_1, 0.1))
plt.plot(xx, C(xx, 0.1))
plt.step(X_2, C(X_2, 0.5 ) )
plt.plot(xx, C(xx, 0.5))
plt.step(X_3, C(X_3, 1 ) )
plt.plot(xx,C(xx,1))
```

```
plt.step(X_4, C(X_4, 2 ) )
plt.plot(xx, C(xx, 2 ) )
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend(['v=0.1','v=0.5','v=1','v=2'])
plt.show( )
```