Convex Optimization
Homework #05

Submitted By: Rifat Bin Rashid

RUID: 237000174

Problem 1:

Given function:

$$f(x) = \frac{1}{2} x^T P x + q^T x + r$$

where P is a positive definite matrix

So, $\nabla f(x) = Px + q$

and, $\nabla^2 f(x) = P$

① Exact minimizer of $f(x)$:

$f(x)$ will be minimized when,

$$\nabla f(x) = 0$$

$$\Rightarrow Px + q = 0$$

$$\Rightarrow x^* = -P^{-1}q$$

From Newton's method: for any arbitrary starting point $x^{(0)} = x^{(k)}$,

$$x^{(k+1)} = x^{(k)} - \left(\nabla^2 f x^{(k)}\right)^{-1} \nabla f\left(x^{(k)}\right)$$

$$= x^{(k)} - P^{-1}\left(Px^{(k)} + q\right)$$

$$x^{(k+1)} = x^{(k)} - x^{(k)} - P^{-1}q = -P^{-1}q = x^*$$

So, only one iteration of full Newton's method finds the exact minimizer of $f(x)$!

(2) From the class lecture, Newton's method assumes a quadratic approximation around x. Namely, Newton's method minimizes this second order approximation around x:

$$\hat{f}(y) = f(x) + \nabla f(x)^T (y-x) + \frac{1}{2}(y-x)^T \nabla^2 f(x) (y-x)$$

But in our case, $f(x)$ is quadratic itself!

So, the approximation is exact!

So, Newton's method finds the exact minimizer irrespective of $x^{(0)}$ in just one full step.

Problem 2:

I have solved the problem in these steps :

Step_1: Defining the Newton's method

Newton's method is defined as a function with the inputs, namely:

-A function computing the value of the objective function.
-A function computing the gradient of the objective function.
-A function computing the hessian of the objective function.
– An initialization point x(0)
– A maximum number of iterations.
– A tolerance ϵ for the stopping criterion.

```
# Step_1: Newton's method with fixed step size
def newton_method_fixed_step(f, f_grad, f_hess, x0, tol,
max_iters):
```

Here, Newton decrement (λ^2) was checked against tolerance to stop the iteration.

Step_2: Function, gradient of the function and hessian of the function were defined.

Step_3: Initial points were defined as in the question.

Step_4: Hyperparameters

The remaining hyperparameters in the design were maximum iterations and the stopping criteria tolerance. They were chosen reasonably.

Step_5: Run Newton's method and Plot Results

Finally the function was run on the given step size and the result (, Function value and Gradient) were plotted in accordance.

Full code for my implementation:

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

# Step_1: Newton's method with fixed step size
def newton_method_fixed_step(f, f_grad, f_hess, x0, tol, max_iters):
    iterates = [x0]  # Store iterates
    grad_values = [f_grad(x0)]  # Store gradient values
    l = x0  # Store initial point for printing

    for _ in range(max_iters):
        grad = f_grad(x0)
        hess = f_hess(x0)

        # Avoid division by very small Hessian values
        if abs(hess) < 1e-6:
            print("Hessian is too small. Stopping early.")
            break

        # Compute Newton decrement λ²
        lambda2 = grad**2 / hess

        # Stopping criterion: λ²/2 ≤ tolerance
        if lambda2 / 2 <= tol:
            print(f"Stopping criterion met: λ²/2 = {lambda2 / 2:.6f} ≤ {tol}")
            break

        dx_nt = -grad / hess  # Newton direction
        x0 += dx_nt  # Fixed step size t = 1

        # Store iterate and gradient value
        iterates.append(x0)
        grad_values.append(f_grad(x0))

    print(f'Initial point x0 = {l}; Final converged point x* = {x0:.6f}, Number
of iterations = {len(iterates) - 1}')
    return np.array(iterates), np.array(grad_values)
```

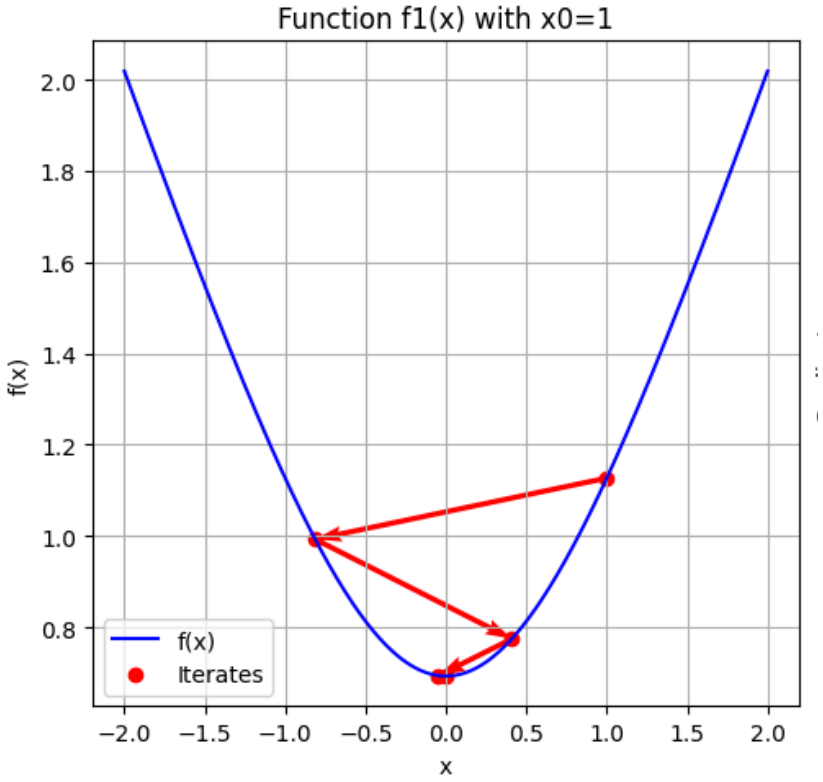
```
# Step_2: Define the functions and their derivatives
def f1(x):
    return np.log(np.exp(x) + np.exp(-x))

def f1_grad(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))

def f1_hess(x):
    return (4 * np.exp(x) * np.exp(-x)) / (np.exp(x) + np.exp(-x))**2

def f2(x):
    return -np.log(x) + x

def f2_grad(x):
    return -1/x + 1

def f2_hess(x):
    return 1/x**2

# Step_3: Initial points
x0_1, x0_1_1, x0_2 = 1, 1.1, 3

# Step_4: Hyperparameters
tol = 1e-6
max_iters = 100

# Step_5: Run and plot
# Run Newton's method for f1(x) with x0 = 1
print("Running Newton's method for f1(x) with x0 = 1")
iterates_f1_x0_1, grad_values_f1_x0_1 = newton_method_fixed_step(f1, f1_grad,
f1_hess, x0_1, tol, max_iters)

# Run Newton's method for f1(x) with x0 = 1.1
print("\nRunning Newton's method for f1(x) with x0 = 1.1")
iterates_f1_x0_1_1, grad_values_f1_x0_1_1 = newton_method_fixed_step(f1,
f1_grad, f1_hess, x0_1_1, tol, max_iters)

# Run Newton's method for f2(x) with x0 = 3
```

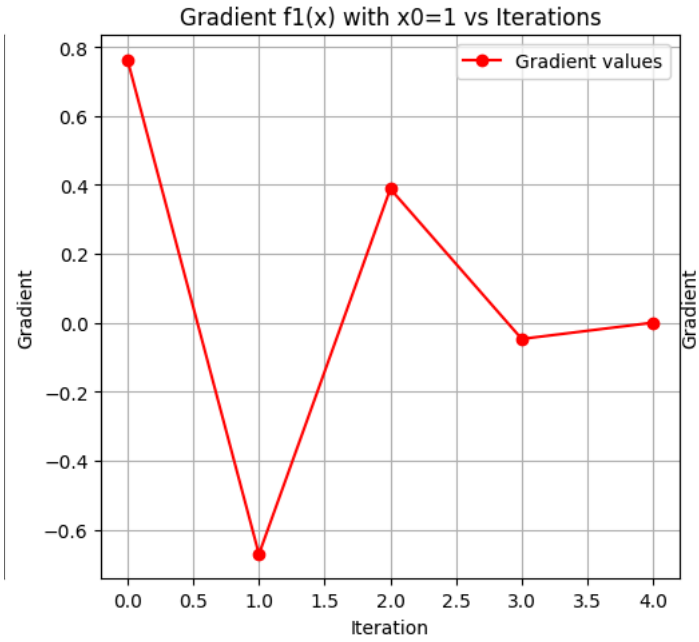
```
print("\nRunning Newton's method for f2(x) with x0 = 3")
iterates_f2_x0_2, grad_values_f2_x0_2 = newton_method_fixed_step(f2, f2_grad,
f2_hess, x0_2, tol, max_iters)

# Generate x values for plotting
x_f1 = np.linspace(-2, 2, 400)
x_f2 = np.linspace(0.1, 5, 400)

# Function to plot all three required graphs
def plot_case(title, x_vals, f, f_grad, iterates, grad_values, color):
   plt.figure(figsize=(18, 5))

   # (1) Plot f(x) with iterates
   plt.subplot(1, 3, 1)
   plt.plot(x_vals, f(x_vals), 'b', label='f(x)')
   plt.scatter(iterates, f(iterates), c=color, label='Iterates')
   plt.quiver(iterates[:-1], f(iterates[:-1]),
              np.diff(iterates), f(iterates[1:]) - f(iterates[:-1]),
              angles='xy', scale_units='xy', scale=1, color=color)
   plt.title(f'Function {title}')
   plt.xlabel('x')
   plt.ylabel('f(x)')
   plt.legend()
   plt.grid()

   # (2) Plot gradient vs iterations
   plt.subplot(1, 3, 2)
   plt.plot(range(len(grad_values)), grad_values, f'{color}-o',
label="Gradient values")
   plt.title(f'Gradient {title} vs Iterations')
   plt.xlabel('Iteration')
   plt.ylabel('Gradient')
   plt.legend()
   plt.grid()

   # (3) Plot gradient vs x
   plt.subplot(1, 3, 3)
   plt.plot(x_vals, f_grad(x_vals), 'r', label="Gradient vs x")
```

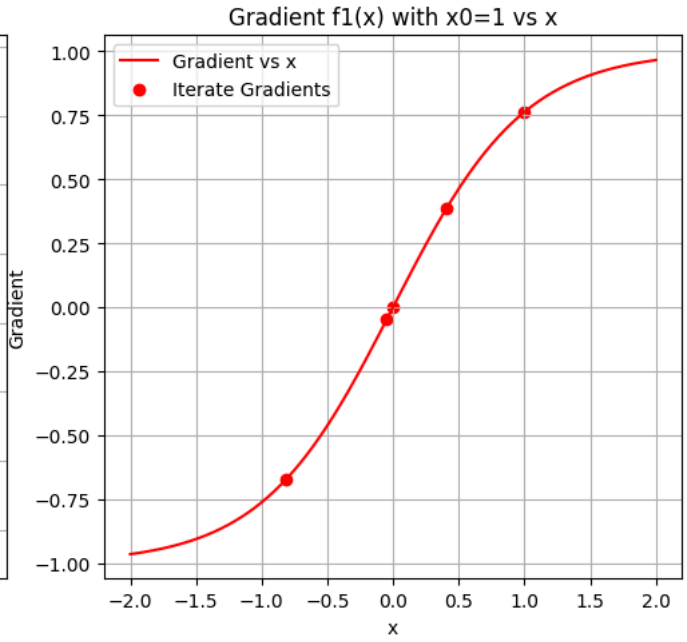
```
    plt.scatter(iterates, f_grad(iterates), c=color, label="Iterate Gradients")
    plt.title(f'Gradient {title} vs x')
    plt.xlabel('x')
    plt.ylabel("Gradient")
    plt.legend()
    plt.grid()

    plt.tight_layout()
    plt.show()

# Plot for f1 with x0=1
plot_case("f1(x) with x0=1", x_f1, f1, f1_grad, iterates_f1_x0_1,
grad_values_f1_x0_1, 'r')

# Plot for f1 with x0=1.1
plot_case("f1(x) with x0=1.1", x_f1, f1, f1_grad, iterates_f1_x0_1_1,
grad_values_f1_x0_1_1, 'g')

# Plot for f2 with x0=3
plot_case("f2(x) with x0=3", x_f2, f2, f2_grad, iterates_f2_x0_2,
grad_values_f2_x0_2, 'b')
```

Output Graphs (function and gradient of function) :

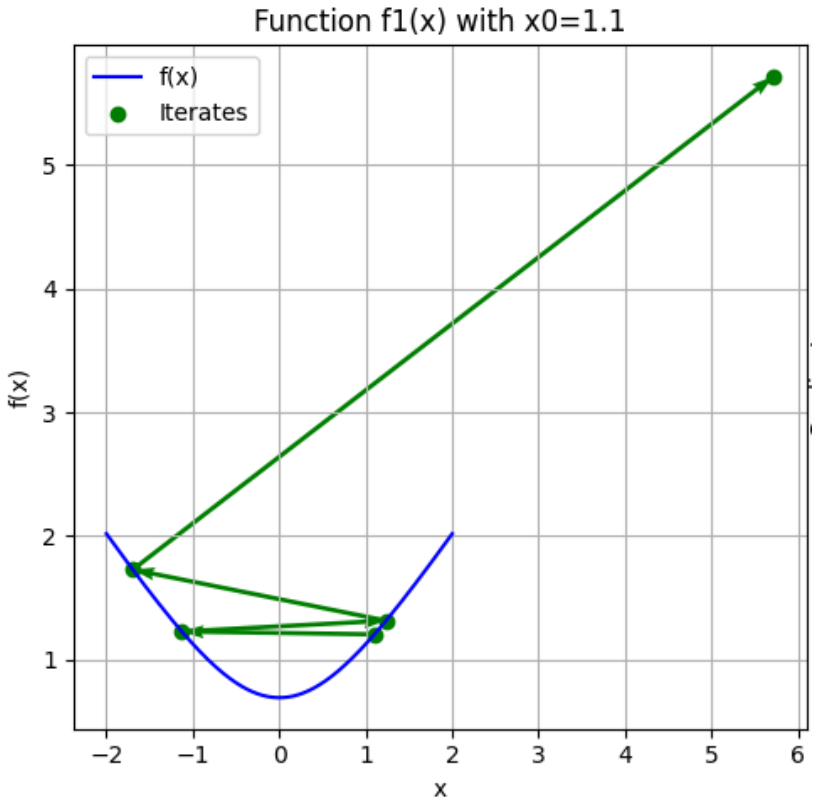
Case 01: $f_1(x)$ with $x_0 = 1$

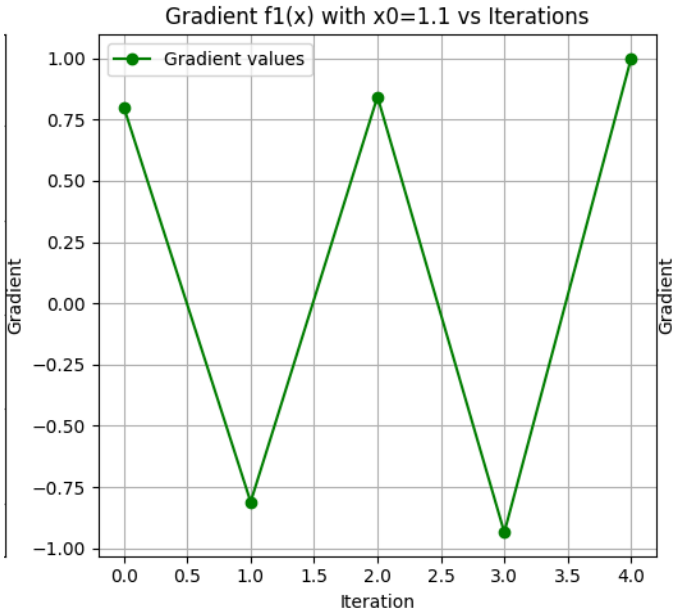


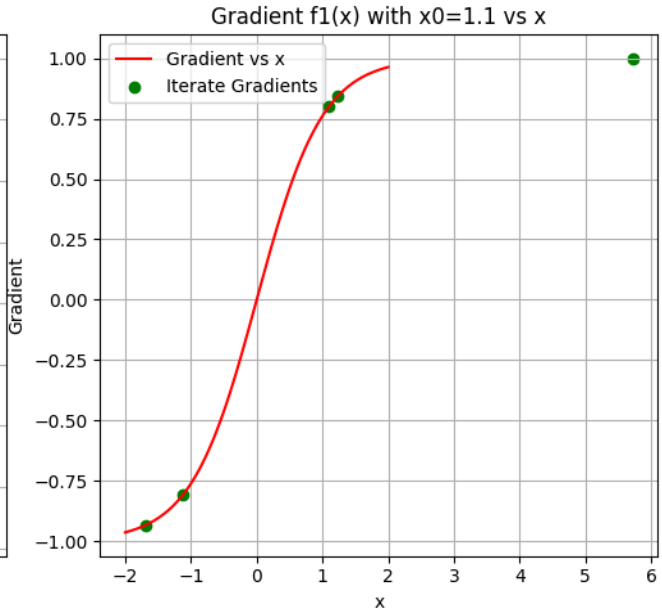




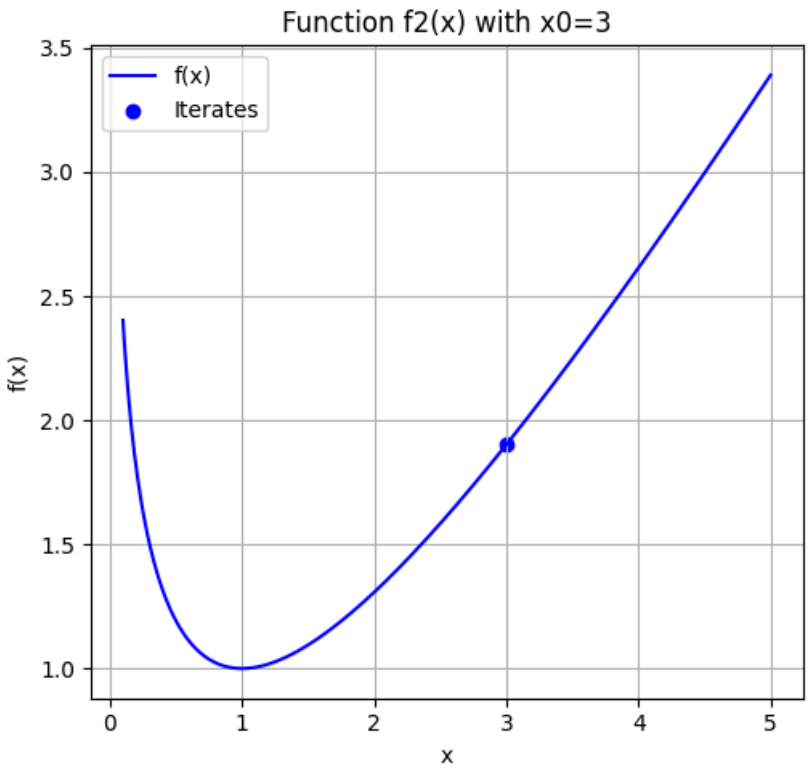
Case 02: $f_1(x)$ with $x_0 = 1.1$

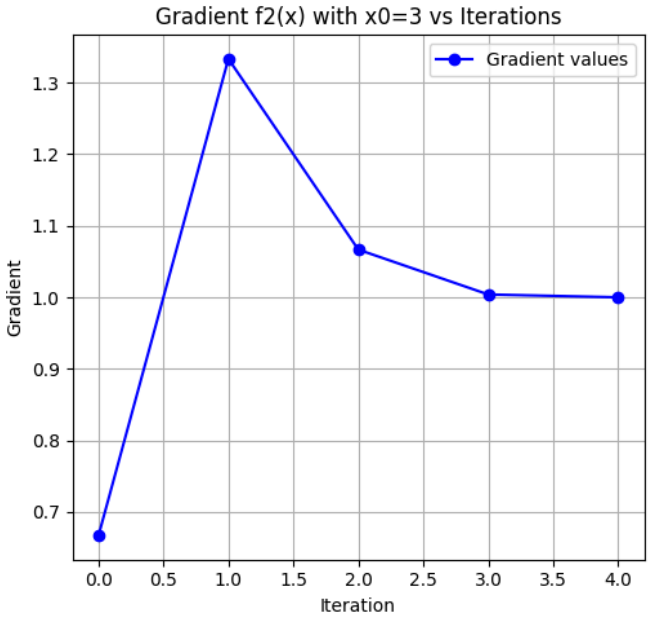


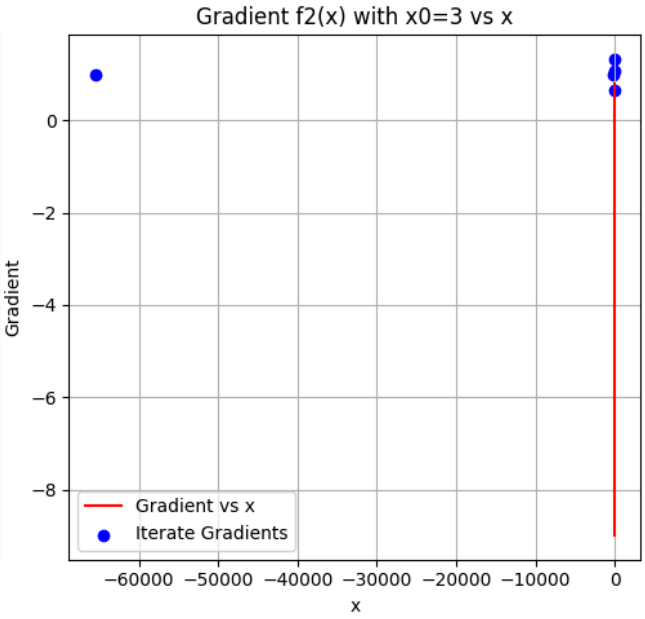




Case 03: $f_2(x)$ with $x_0 = 3$







Explanations:

Case 01: Running Newton's method for **f1(x) with x0 = 1**
Stopping criterion met: $\lambda^2/2 = 0.000000 \leq$ 1e-06
Initial point x0 = 1;
Final converged point x* = 0.000071, Number of iterations = 4

Case 02: Running Newton's method for **f1(x) with x0 = 1.1**
Initial point x0 = 1.1;
Final converged point x* = nan, Number of iterations = 100

Case 03: Running Newton's method for **f2(x) with x0 = 3**
Hessian is too small. Stopping early.
Initial point x0 = 3;
Final converged point x* = -65535.000000, Number of iterations = 4

For the first case , Newton's method converges with a small number of iterations(4), because the initial point (1) is close enough to the actual minimizer.

For the second case, Newton's method diverges, because now the initial point (1.1) is farther from the global minimizer than that of case 01.

For the third case, the function is not defined for x<0 , so the new iterate value of x is being out of bound after a few iterations and Newton's method does not converge to the global minimizer eventually.

Problem 3:

I have solved the problem in these steps :

Step 1: The gradient descent function with backtracking line search is implemented.
– Gradient descent method is defined as in HW4, with the backtracking step size approach.

Step 2: The Newton's method function with backtracking line search is implemented.

– Newton's method is defined as in problem 2, with adjustment for the backtracking step size .

– Newton's method was implemented in a modular way, allowing it to take different input parameters, listed as :

```
# Step_2: Newton's method function with backtracking line search
def newton_method(grad_func, hess_func, func, x0, tol, max_iter, A,
step_type,alpha,beta):
```

Step 3: The objective function, f(x) is defined.

Step 4: The gradient of the objective function is defined.

Step 5: The hessian of the objective function is defined.

Step 6: Problem instances are generated.

– size of A and x was taken reasonably
– matrix A was created randomly

Step 7: Hyperparameters are defined.
– tolerance, maximum iteration was taken reasonably

Step 8: Execution and plot

Here , Gradient descent with backtracking is executed.

Newton's method with backtracking is executed.

Results are plotted.

Ful code for my implementation :

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

# Step_1: Gradient descent function with backtracking line search
def gradient_descent(f_grad, func, x0, step_type, alpha, beta, max_iter, tol, A):
   X = [x0]  # List to store iterates
   x = x0.copy()

   for k in range(max_iter):
       grad = f_grad(x, A)

       # Stopping criterion based on gradient norm
       if np.linalg.norm(grad, 2) <= tol:
           break

       # Step Size Selection
       if step_type == 'fixed':
           step_size = alpha
           x_new = x - step_size * grad
       elif step_type == 'variable':
           # Backtracking line search for step size
           t = 1
           while func(x - t * grad, A) > func(x, A) - alpha * t * np.linalg.norm(grad,
2)**2:
               t *= beta
           x_new = x - t * grad
       else:
           raise ValueError('Invalid step type. Choose either "fixed" or "variable".')
```

```
        # Project onto the feasible region
        x_new = np.clip(x_new, -0.99, 0.99)  # Maintain |x_i| < 1
        while np.any(A @ x_new >= 1):
            t *= beta
            x_new = x - t * grad
            x_new = np.clip(x_new, -0.99, 0.99)

        # Save new iterate
        x = x_new
        X.append(x)

    return np.array(X).T  # Convert to column-wise storage


# Step_2: Newton's method function with backtracking line search
def newton_method(f_grad, f_hess, func, x0, tol, max_iter, A, step_type, alpha, beta):
    X = [x0]  # List to store iterates
    x = x0.copy()

    for k in range(max_iter):
        grad = f_grad(x, A)
        hess = f_hess(x, A)

        # Ensure Hessian is positive definite
        if np.min(np.linalg.eigvals(hess)) < 1e-6:
            break

        dx_nt = -np.linalg.solve(hess, grad)  # Compute Newton direction
        lambda2 = grad.T @ dx_nt  # Compute Newton decrement

        # Stopping condition based on Newton decrement
        if abs(lambda2) / 2 <= tol:
            break

        # Step size selection
        if step_type == 'fixed':
            t = 1  # Use a fixed step size
        elif step_type == 'variable':
            # Backtracking line search for Newton step
            t = 1
            while func(x + t * dx_nt, A) > func(x, A) + alpha * t * lambda2:
                t *= beta
```

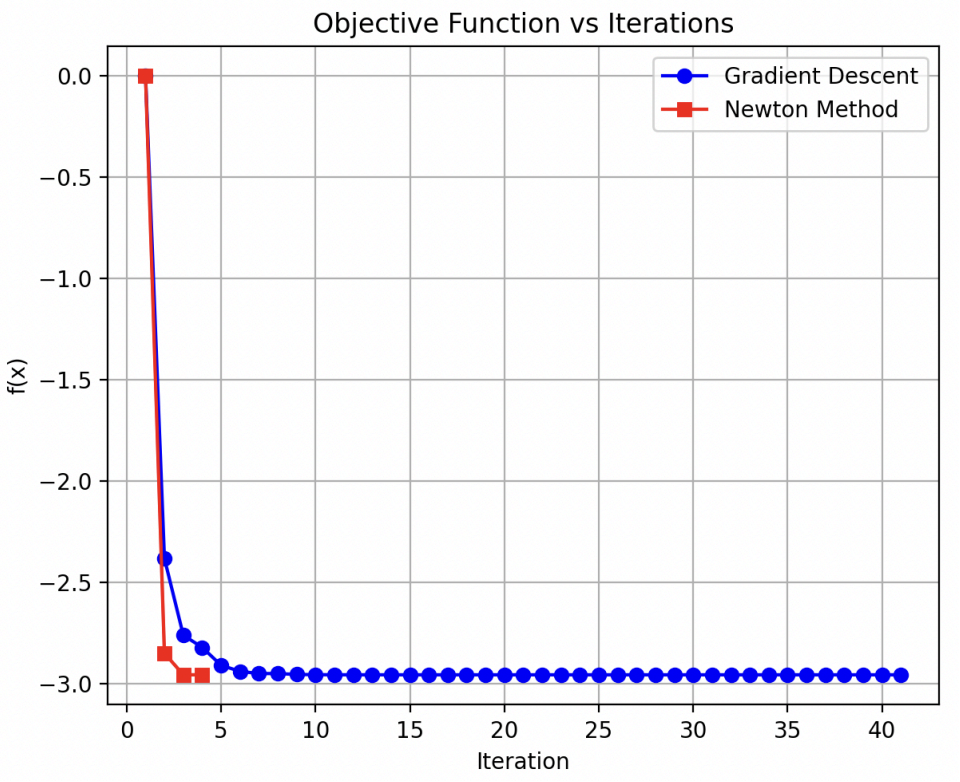
```
        else:
            raise ValueError('Invalid step type. Choose either "fixed" or "variable".')

        # Apply the step update
        x_new = x + t * dx_nt

        # Project onto the feasible region
        x_new = np.clip(x_new, -0.99, 0.99)  # Maintain |x_i| < 1
        while np.any(A @ x_new >= 1):
            t *= beta
            x_new = x + t * dx_nt
            x_new = np.clip(x_new, -0.99, 0.99)

        x = x_new
        X.append(x)

    return np.array(X).T  # Convert to column-wise storage


# Define the objective function
def f(x, A):
    if np.any(np.abs(x) >= 1) or np.any(A @ x >= 1):
        return np.inf
    return -np.sum(np.log(1 - A @ x)) - np.sum(np.log(1 - x**2))

# Define the gradient function
def f_grad(x, A):
    if np.any(np.abs(x) >= 1) or np.any(A @ x >= 1):
        return np.zeros_like(x)
    grad_term1 = A.T @ (1 / (1 - A @ x))
    grad_term2 = 2 * x / (1 - x**2)
    return grad_term1 + grad_term2

# Define the Hessian function for Newton's Method
def f_hess(x, A):
    n = len(x)
    H = np.zeros((n, n))
    if np.any(np.abs(x) >= 1) or np.any(A @ x >= 1):
        return H
    for i in range(A.shape[0]):
        ai = A[i, :]
        H += np.outer(ai, ai) / (1 - ai @ x) ** 2
```

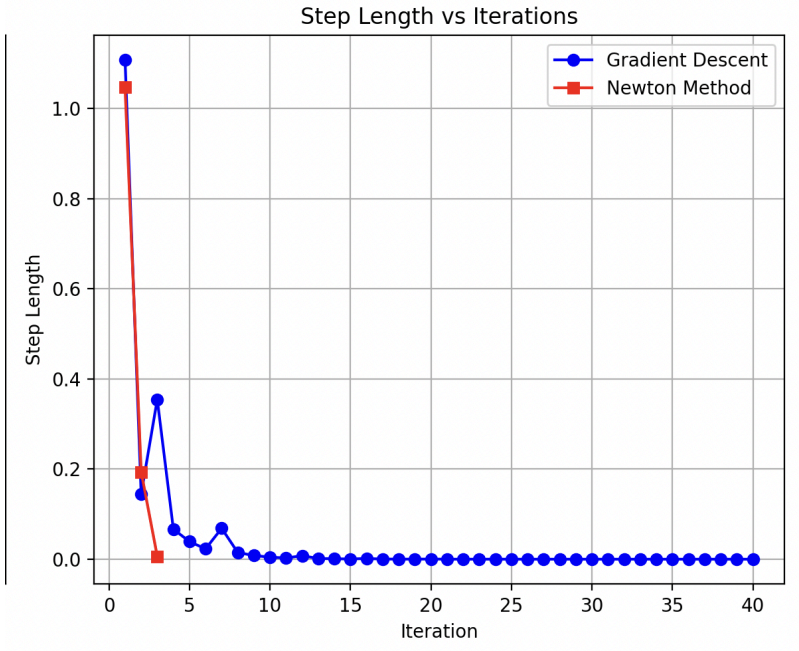
```
    H += np.diag((2 * (1 + x**2)) / (1 - x**2) ** 2)
    return H

# Step_6: Generate problem instances

n = 10  # Number of variables
m = 20  # Number of constraints
np.random.seed(42)  # Fix random seed for reproducibility
A = np.random.randn(m, n) * 0.5  # Generate A matrix

# Step_7: Hyperparameters
# Set Initial Point
x0 = np.zeros(n)  # x(0) = 0 as given in the problem

step_type = 'variable'
alpha = 0.01
beta = 0.05
max_iter = 500
tol = 1e-6

# Step_8: Run Gradient Descent and Newton's Method

X_gd = gradient_descent(f_grad, f, x0, step_type, alpha, beta, max_iter, tol, A)
num_iters_gd = X_gd.shape[1]

X_newton = newton_method(f_grad, f_hess, f, x0, tol, max_iter, A, step_type, alpha,
beta)
num_iters_newton = X_newton.shape[1]

# Compute Optimal Value

p_star = min(f(X_gd[:, -1], A), f(X_newton[:, -1], A))


# Plot Results

plt.figure(figsize=(12, 10))

# (1) **Objective Function vs Iterations**
plt.subplot(2, 2, 1)
plt.plot(range(1, num_iters_gd+1), [f(X_gd[:, i], A) for i in range(num_iters_gd)],
'b-o', linewidth=1.5, label='Gradient Descent')
```

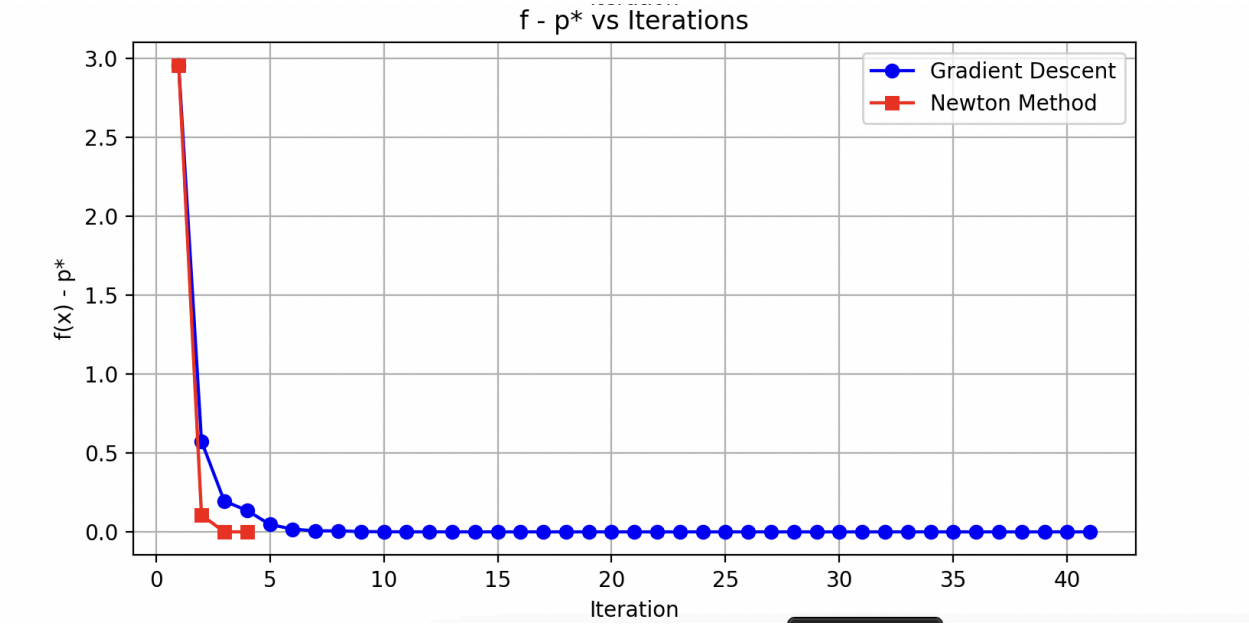
```
plt.plot(range(1, num_iters_newton+1), [f(X_newton[:, i], A) for i in
range(num_iters_newton)], 'r-s', linewidth=1.5, label='Newton Method')
plt.title('Objective Function vs Iterations')
plt.xlabel('Iteration')
plt.ylabel('f(x)')
plt.legend()
plt.grid()

# (2) **Step Length vs Iterations**
step_lengths_gd = np.linalg.norm(np.diff(X_gd, axis=1), axis=0)
step_lengths_nt = np.linalg.norm(np.diff(X_newton, axis=1), axis=0)

plt.subplot(2, 2, 2)
plt.plot(range(1, len(step_lengths_gd)+1), step_lengths_gd, 'b-o', linewidth=1.5,
label='Gradient Descent')
plt.plot(range(1, len(step_lengths_nt)+1), step_lengths_nt, 'r-s', linewidth=1.5,
label='Newton Method')
plt.title('Step Length vs Iterations')
plt.xlabel('Iteration')
plt.ylabel('Step Length')
plt.legend()
plt.grid()

# (3) **f - p* vs Iterations**
plt.subplot(2, 2, 3)
plt.plot(range(1, num_iters_gd+1), [abs(f(X_gd[:, i], A) - p_star) for i in
range(num_iters_gd)], 'b-o', linewidth=1.5, label='Gradient Descent')
plt.plot(range(1, num_iters_newton+1), [abs(f(X_newton[:, i], A) - p_star) for i in
range(num_iters_newton)], 'r-s', linewidth=1.5, label='Newton Method')
plt.title('f - p* vs Iterations')
plt.xlabel('Iteration')
plt.ylabel('f(x) - p*')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()

print(f'Gradient Descent Iterations: {num_iters_gd}')
print(f'Newton Method Iterations: {num_iters_newton}')
```

Output Graphs: ** In all the cases, A was kept consistent with the same random seed.

Case 01: alpha= 0.01, beta= 0.2



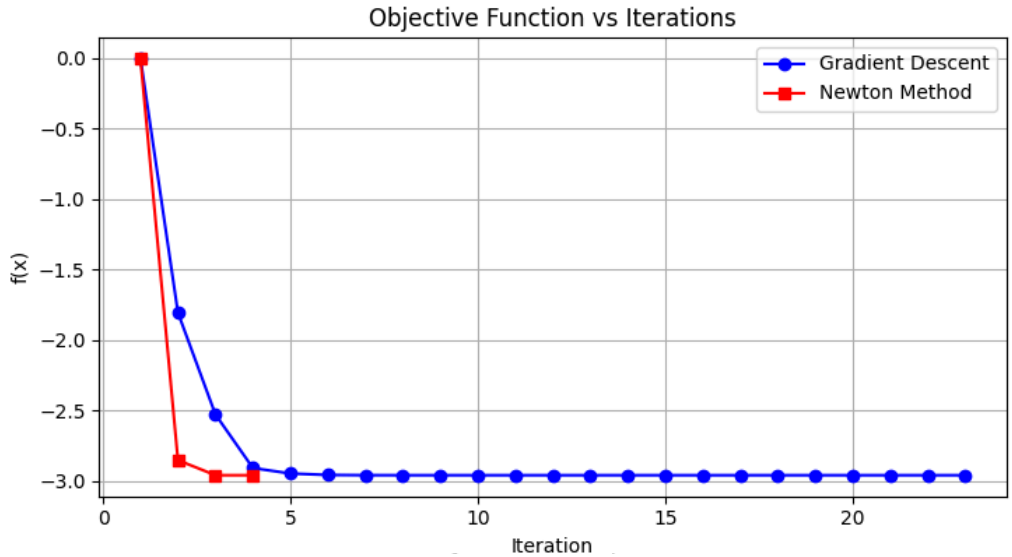


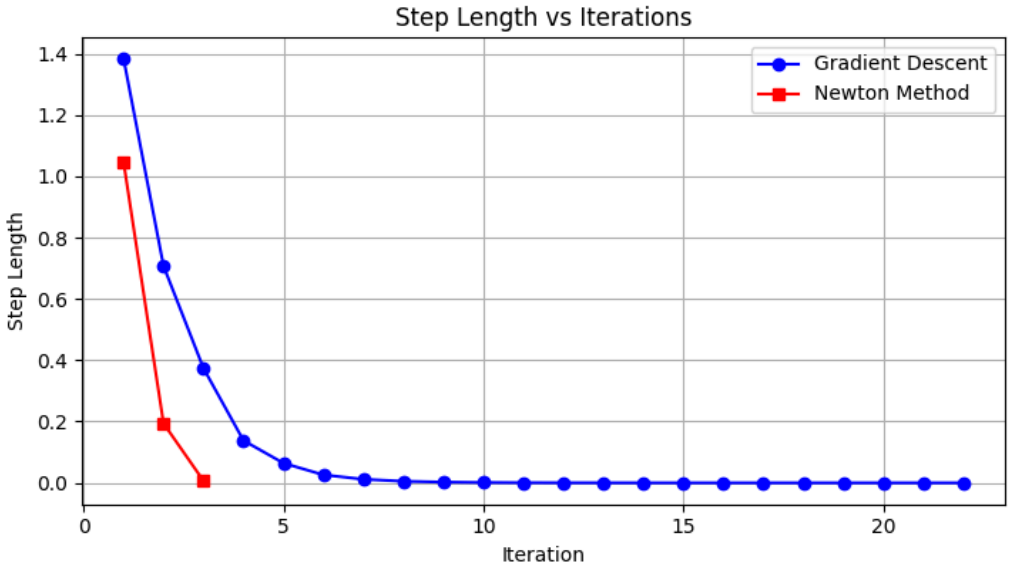


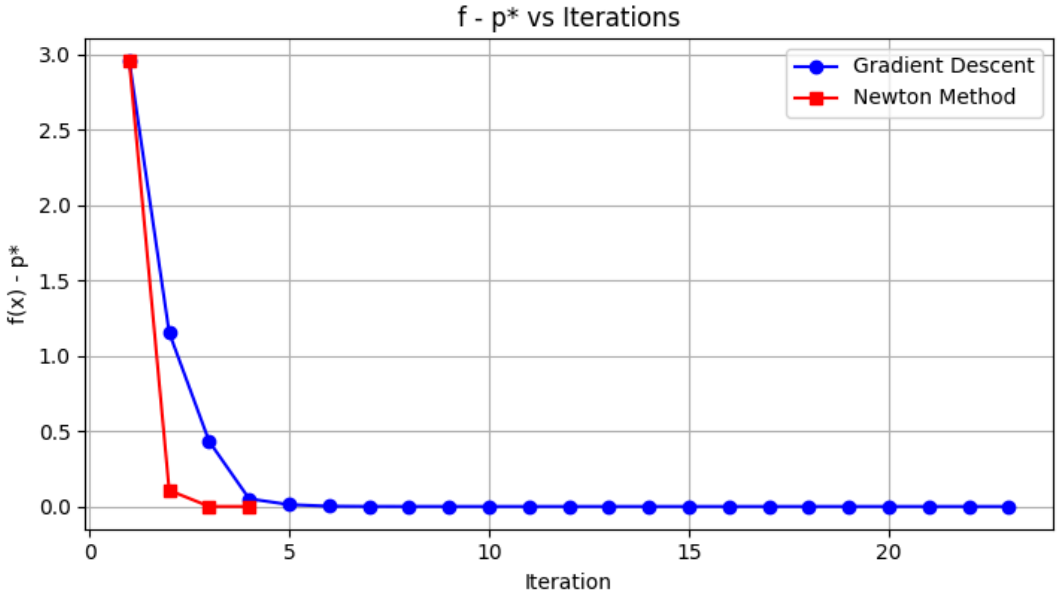
Gradient Descent Iterations: 41

Newton Method Iterations: 4

Case 02: alpha= 0.01, beta= 0.5

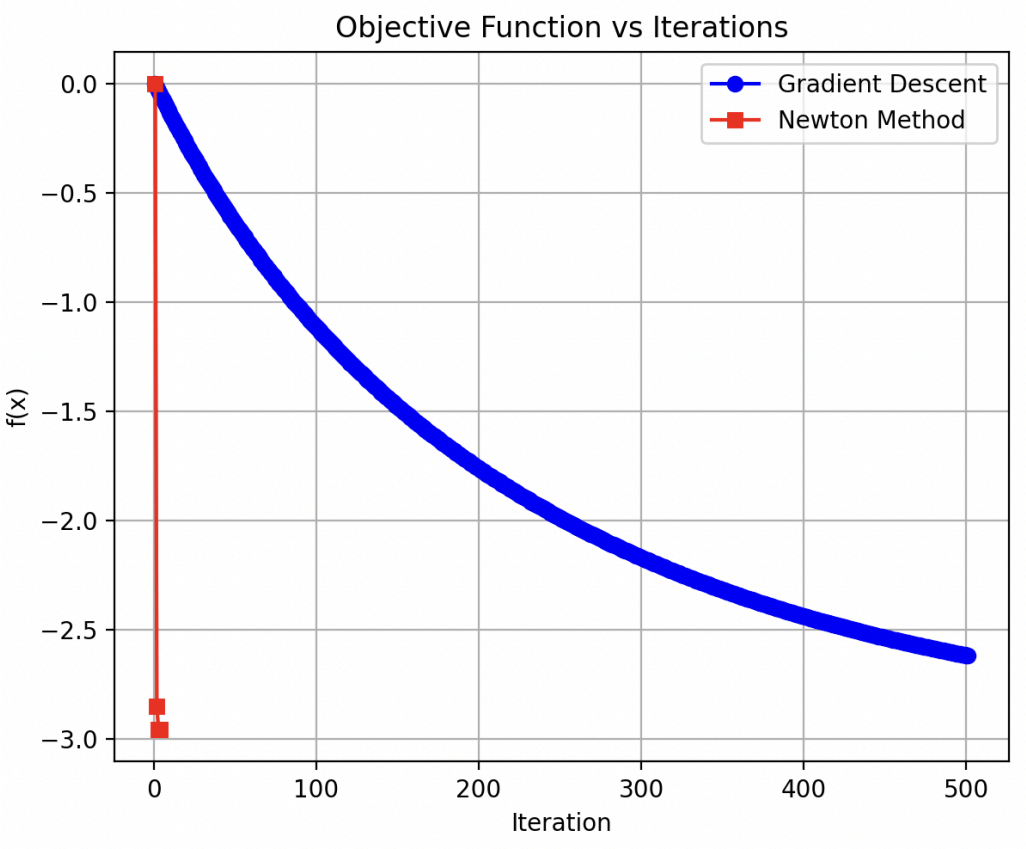


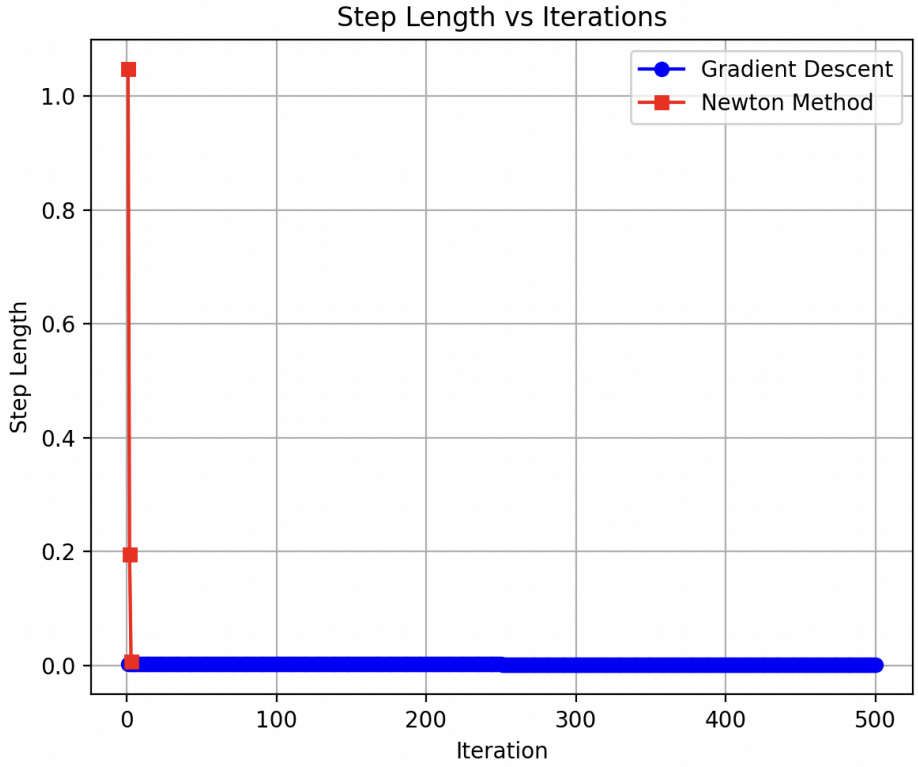


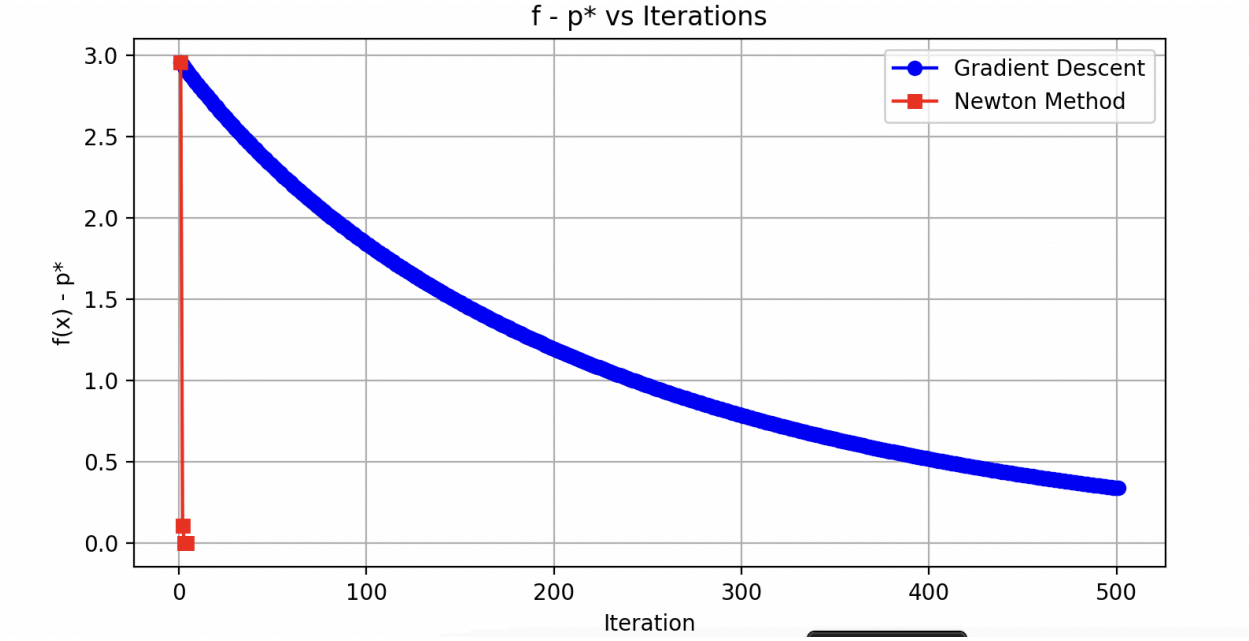


Gradient Descent Iterations: 23
Newton Method Iterations: 4

Case 03: alpha= 0.01, beta= 0.0005

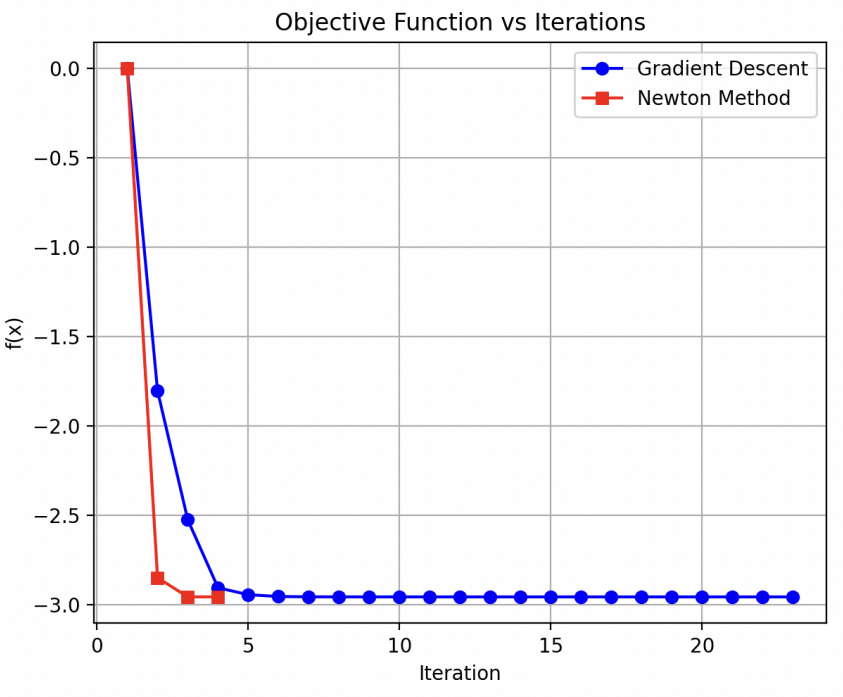


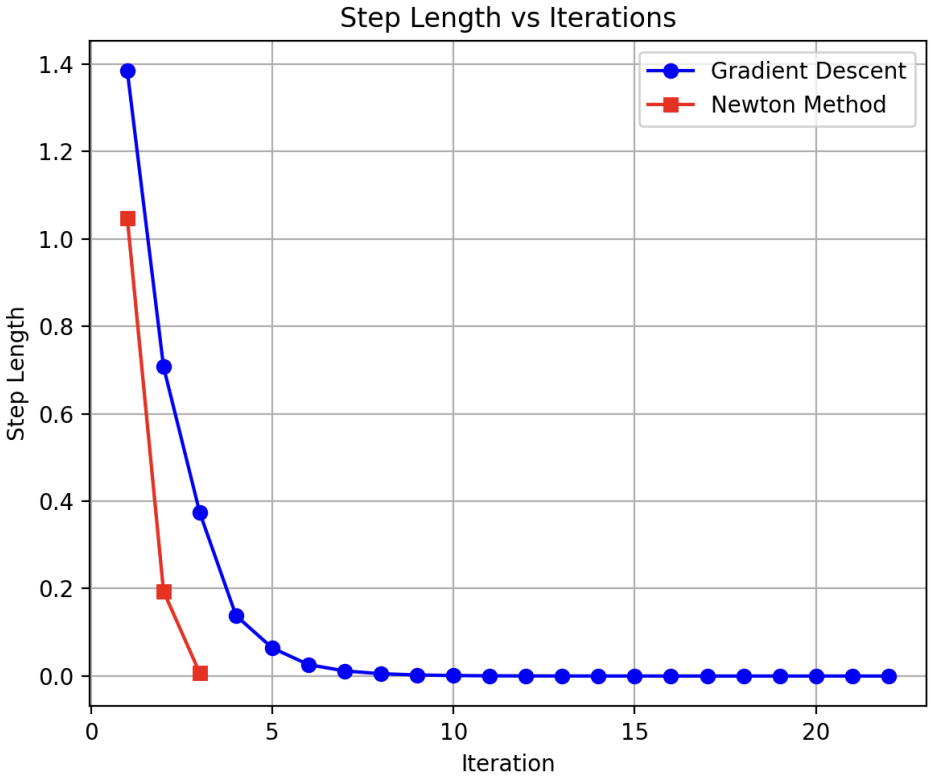


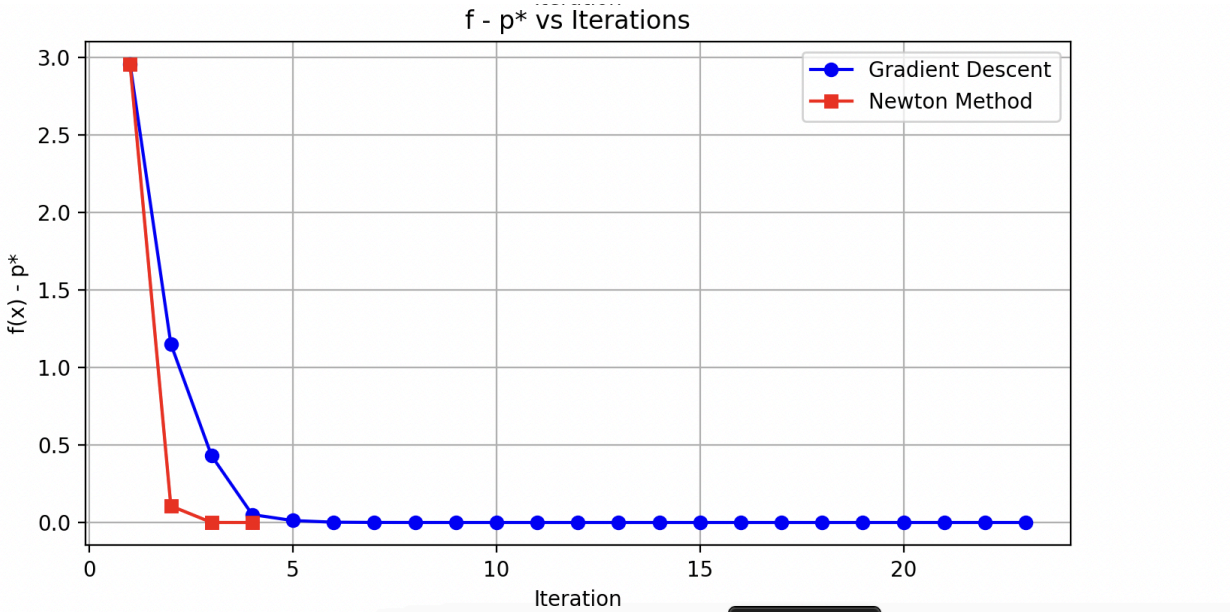


Gradient Descent Iterations: 501
Newton Method Iterations: 4

Case 04: alpha= 0.1, beta= 0.5







Gradient Descent Iterations: 23

Newton Method Iterations: 4

Experimentation with the backtracking parameters α and β to see their effect on the total number of iterations required:

α	β	Number of iterations required(grad_descent)
0.01	0.2	41
0.01	0.5	23
0.01	0.0005	501
0.1	0.5	23

Here , I have not seen any effect of alpha and beta on Newton method step numbers.

On the gradient descent, Beta controls step size reduction.

So a smaller beta means smaller steps, more accurate convergence, but large iteration is needed .

On the other hand, larger beta means larger steps, faster convergence, so, smaller steps are needed.

Problem 4: (Exercise 2.1)

From the definition of convex set,

for $k=2$,

if $x_1, x_2 \in C$, $\theta_1, \theta_2 \geq 0$ & $\theta_1 + \theta_2 = 1$

then, $\theta_1 x_1 + \theta_2 x_2 \in C$ --- (I)

Let's assume,

this definition holds for $k=m$.

So, if, $x_1, \ldots, x_m \in C$; $\theta_1, \ldots, \theta_m \geq 0$ & $\theta_1 + \ldots + \theta_m = 1$

then, $\theta_1 x_1 + \ldots + \theta_m x_m \in C$ --- (II)

Now, for $k = m+1$,

Let, $x_1, \ldots x_{m+1} \in C$; $\theta_1 + \ldots + \theta_{m+1} = 1$ with

$\theta_1, \ldots, \theta_{m+1} \geq 0$

then,

$$\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_m x_m + \theta_{m+1} x_{k+1}$$

Can be written as

$$(\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_m x_m) + \theta_{m+1} x_{k+1} \quad \text{---- (III)}$$

Let, $\theta = \theta_1 + \theta_2 + \dots + \theta_m$

So, $\frac{\theta_1}{\theta} + \frac{\theta_2}{\theta} + \dots + \frac{\theta_m}{\theta} = 1$

So,

$y = \frac{\theta_1}{\theta} x_1 + \frac{\theta_2}{\theta} x_2 + \dots + \frac{\theta_m}{\theta} x_m \in C$; from (ii)

So, Combination in (iii) can be written as

$$\theta y + \theta_{m+1} x_{m+1}$$

Now, $y \in C$; $x_{m+1} \in C$

$$\theta + \theta_{m+1} = \theta_1 + \dots + \theta_m + \theta_{m+1} = 1$$

So, from definition of convexity at (i)

$$\theta y + \theta_{m+1} x_{m+1} \in C$$

So, by induction, the definition holds for all k.

Problem 5 (Exercise 2.2)

'A set is convex iff its intersection with any line is convex'

This can be proved in two steps:

Convex set implies line intersection is convex:

Let $x, y \in C \cap L$

where C is a convex set

& L be any line.

As C is convex,

$$\lambda x + (1-\lambda) y \in C \quad , \text{where } \lambda \in [0,1]$$

So, the line segment between x & y, belongs to both line L and set C.

So, the intersection of C and L contains the whole line segment between x & y, making the intersection convex.

Line intersection is convex implies convex set:

let, $x, y \in C \cap L$; C any set
is convex L any line

Now, by the assumption,

the intersection of the line with C is convex
$(C \cap L)$

⇓

Any convex combination of x & y (which lies in the line) must also be in $C \cap L \rightarrow$ hence in C

⇓

So, C is convex as it contains all convex combinations of its points.

Proof for Affine sets:

Affine set implies line intersection is affine:

let, A is an affine set
L is any line.

let, $x, y \in A \cap L$

Now, A is affine.

So. $\lambda x + (1-\lambda) y \in A$, $\forall \lambda \in \mathbb{R}$

but this is also the line passing through x & y.

So, the intersection of A & L contains the whole line passing through x & y. making the intersection affine.

Line intersection is affine implies affine set:

Let, $x, y \in A \cap L$; A any set
L any line

Now,

the intersect of L with A is affine

⇓

the point $\lambda x + (1-\lambda) y$ (which lies on the line) must also be in $A \cap L$ → hence in L

⇓

So, A is affine as it contains all affine combinations of its points.

Problem 6 (Example 3.1)

Given $f: \mathbb{R} \to \mathbb{R}$ is convex and $a, b \in \text{dom} f$

(a) As f is convex function, so dom$f$ is convex

Now, all $x \in [a, b]$ can be written as

$$x = \theta a + (1-\theta) b, \quad \theta \in [0,1] \quad \text{--- (1)}$$

Then $x \in \text{dom} f$

Now, from definition of convex function,

$$f(\theta a + (1-\theta) b) \leq \theta f(a) + (1-\theta) f(b) \quad \text{--- (2)}$$

From (1)

$$x = \theta a + (1-\theta) b$$

$$\Rightarrow x = \theta a + b - \theta b$$

$$\Rightarrow \theta = \frac{x-b}{a-b} \quad \text{--- (3)}$$

subsituting (1) & (3) into (2)

$$f(x) \leq \frac{x-b}{a-b} f(a) + \left(1 - \frac{x-b}{a-b}\right) f(b)$$

$$\boxed{f(x) \leq \frac{b-x}{b-a} f(a) + \frac{x-a}{b-a} f(b)}$$

as $b > a$
$b - a \neq 0$

(b) from (a)

$$f(x) \le \frac{b-x}{b-a} f(a) + \frac{x-a}{b-a} f(b)$$

$$\Rightarrow f(x) \le \frac{b-a-(x-a)}{b-a} f(a) + \frac{x-a}{b-a} f(b)$$

$$\Rightarrow f(x) \le \left(1 - \frac{x-a}{b-a}\right) f(a) + \frac{x-a}{b-a} f(b)$$

$$\Rightarrow \frac{f(x)-f(a)}{x-a} \le \frac{f(b)-f(a)}{b-a} \quad \text{--- (4)}$$

as $x \in (a,b)$
$x - a \neq 0$

Again,

$$f(x) \le \frac{b-x}{b-a} f(a) + \frac{x-a}{b-a} f(b)$$

$$\Rightarrow f(x) \le \frac{b-x}{b-a} f(a) + \frac{b-a+(x-b)}{b-a} f(b)$$

$$\Rightarrow f(x) \le \frac{b-x}{b-a} f(a) + \left(1 - \frac{b-x}{b-a}\right) f(b)$$

$$\Rightarrow \frac{f(x)-f(b)}{b-x} \le \frac{f(a)-f(b)}{b-a}$$

$$\Rightarrow \frac{f(b)-f(x)}{b-x} \ge \frac{f(b)-f(a)}{b-a} \quad \text{-- (5)}$$

as $x \in (a,b)$
$b - x \neq 0$

(c) f is differentiable.

So, at 'a', from the definition of derivative,

$$f'(a) = \lim_{x \to a^+} \frac{f(x) - f(a)}{x - a}$$

From (b),

$$\frac{f(x) - f(a)}{x - a} \leq \frac{f(b) - f(a)}{b - a}$$

Taking the limit $x \to a^+$ on both side

$$f'(a) \leq \frac{f(b) - f(a)}{b - a}, \quad \forall x \in (a, b) \quad \text{---} (6)$$

similarly, f is differentiable at b.

$$\lim_{x \to b^-} \frac{f(b) - f(x)}{b - x} = f'(b), \quad \forall x \in (a, b)$$

So, $$\frac{f(b) - f(a)}{b - a} \leq f'(b), \quad \forall x \in (a, b) \quad \text{--} (7)$$

Combining ⑥ & ⑦,

$$f'(a) \leq \frac{f(b) - f(a)}{b - a} \leq f'(b)$$

d

from c

$f'(x)$ is non-decreasing

$$f'(a) \leq f'(b)$$

So, the derivative is an increasing function from a to b.

Now, from definition of derivative

$$f''(x) = \lim_{h \to 0} \frac{f'(x+h) - f'(x)}{h}$$

As $f(x)$ is non-decreasing,

$$f'(x+h) \geq f'(x)$$

$$\Rightarrow \frac{f'(x+h)}{h} \geq \frac{f'(x)}{h}$$

$$\Rightarrow \frac{f'(x+h)}{h} - \frac{f'(x)}{h} \geq 0$$

$$\Rightarrow \frac{f'(x+h)}{h} - \frac{f'(x)}{h} \geq 0$$

$$\Rightarrow \frac{f'(x+h) - f'(x)}{h} \geq 0$$

$$\Rightarrow \lim_{h \to 0} \frac{f'(x+h) - f'(x)}{h} \geq 0$$

$$\Rightarrow f''(x) \geq 0 \qquad \forall x \in a, b$$

So,

$$f''(a) \geq 0$$

$$f''(b) \geq 0$$