# Convex Optimization
# Homework #04

**Submitted By: Rifat Bin Rashid**

**RUID: 237000174**

## Problem 1:

Given: $f \in C_L^1(\mathbb{R}^n)$

So, $f(y) \leq f(x) + \nabla f(x)^T (y-x) + \frac{L}{2} \|y-x\|_2^2$

$\forall\ x, y \in \text{dom} f$ ----(1)

Considering gradient descent with,

$$y = x^{(k+1)} = x^{(k)} - t^{(k)} \nabla f(x^{(k)})$$

$$x = x^{(k)}$$

Eq(1) gives,

$$f(x^{(k+1)}) \leq f(x^{(k)}) - t^{(k)} \|\nabla f(x^{(k)})\|_2^2 + (t^{(k)})^2 \cdot \frac{L}{2} \|\nabla f(x^{(k)})\|_2^2$$

$$f(x^{(k+1)}) - f(x^{(k)}) \leq -t^{(k)} \left(1 - \frac{L t^{(k)}}{2}\right) \|\nabla f(x^{(k)})\|_2^2$$

Given that,

$$\epsilon \leq t^{(k)} \leq \frac{2-\epsilon}{L} \quad \text{for } \epsilon > 0$$

So, $1 - \frac{L t^{(k)}}{2} \leq 1 - \frac{L\left(\frac{2-c}{L}\right)}{2}$

$= \frac{c}{2} > 0$

So,

$$f(x^{(k+1)}) - f(x^{(k)}) \leq -\frac{c}{2} t^{(k)} \|\nabla f(x^{(k)})\|_2^2$$

Now, adding this above expression for $k = 0, 1 \cdots \infty$

$$\sum_{k=0}^{\infty} \left[f(x^{(k+1)}) - f(x^{(k)})\right] \leq -\frac{c}{2} t^{(k)} \sum_{k=\infty}^{\infty} \|\nabla (f(x^{(k)}))\|_2^2$$

Using telescoping sum,

$$f(x^{(\infty)}) - f(x^{(0)}) \leq -\frac{c}{2} t^{(k)} \sum_{k=0}^{\infty} \|\nabla f(x^{(k)})\|_2^2$$

$\Rightarrow$ ~~$\sum_{k=\infty}^{\infty} \|\nabla f(x^{(k)})\|_2^2 \leq$~~

$\Rightarrow \frac{c}{2} t^{(k)} \sum_{k=0}^{\infty} \|\nabla f(x^{(k)})\|_2^2 \leq \left[f(x^{(0)}) - f(x^{(\infty)})\right]$

$$\leq \left[ f(x^{(0)}) - p^* \right]$$

where, $p^*$ is the global minimum of the function.

So,

$$\frac{\epsilon}{2} t^{(k)} \sum_{k=0}^{\infty} \| \nabla f(x^{(k)}) \|_2^2 < \infty$$

$$\Rightarrow \frac{\epsilon}{2} t^{(k)} \lim_{k \to \infty} \| \nabla f(x^{(k)}) \|_2^2 = 0$$

Now $\epsilon > 0$ and, $t^k > \epsilon$

So, $\frac{\epsilon}{2} \not\to 0$ $t^k \not\to 0$

So,

$$\lim_{k \to \infty} \| \nabla f(x^{(k)}) \|_2^2 = 0$$

$$\Rightarrow \boxed{\lim_{k \to \infty} \nabla f(x^{(k)}) = 0}$$

## Problem -2

Given, $f(x) = \log\left(1 + e^{-a^T x}\right)$

Using the chain Rule:

$$\nabla f(x) = \frac{1}{1+e^{-a^T x}} \cdot \left(-e^{-a^T x}\right) \cdot a$$

$$= \frac{e^{a^T x}}{1+e^{a^T x}} \left(-e^{-a^T x}\right) \cdot a$$

$$= \frac{-a}{1+e^{a^T x}}$$

$$\nabla^2 f(x) = \frac{-a \cdot \nabla\left(1+e^{a^T x}\right)}{-\left(1+e^{a^T x}\right)^2}$$

$$= \frac{-a \cdot \left(e^{a^T x} \cdot a\right)}{-\left(1+e^{a^T x}\right)^2}$$

$$\boxed{\nabla^2 f(x) = \frac{e^{a^T x} \cdot a a^T}{\left(1+e^{a^T x}\right)^2}}$$

## Semi definite: positive

A matrix M is semi definite if for any vector v,

$$v^T M v \geq 0$$

Now, for any $x \in \mathbb{R}^n$

$$x^T \nabla^2 f(x) \cdot x$$

$$= x^T \left( \frac{e^{a^T x}}{(1+e^{a^T x})^2} \cdot a a^T \right) \cdot x$$

$$= \frac{e^{a^T x}}{(1+e^{a^T x})^2} \left( x^T a a^T x \right)$$

$\frac{e^{a^T x}}{(1+e^{a^T x})^2}$ scaler

$$= \frac{e^{a^T x}}{(1+e^{a^T x})^2} \left[ (a^T x)^T \cdot (a^T x) \right]$$

$$= \frac{e^{a^T x}}{(1+e^{a^T x})^2} \left( \| a^T x \|_2^2 \right)$$

Here, for any $x$; $\frac{e^{a^T x}}{(1+e^{a^T x})^2} \geq 0$ and $\| a^T x \|_2^2 \geq 0$

So, $x^T \nabla^2 f(x) \cdot x \geq 0$ for any $x$ → positive semi definite

## Problem 03:

Given, a step of Newton's method applied to $f$ at $v_0$ results in:

$$v_1 = v_0 - \left(\nabla^2 f(v_0)\right)^{-1} \nabla f(v_0)$$

Given,

$$g(x) = f(Ax+b)$$

So, $\nabla g(x) = A^T \nabla f(Ax+b)$ ; using the chain rule

$$\nabla^2 g(x) = A^T \nabla^2 f(Ax+b) A$$

Now, Applying Newton's method to $g$ :-

$$x_1 = x_0 - \left(\nabla^2 g(x_0)\right)^{-1} \nabla g(x_0)$$

$$= x_0 - \left(A^T \nabla^2 f(Ax_0+b) A\right)^{-1} A^T \nabla f(Ax_0+b)$$

Here,

$x_0 = A^{-1}(v_0 - b)$

$\hookrightarrow Ax_0 + b = v_0$

and, $(ABC)^{-1} = C^{-1} B^{-1} A^{-1}$

$$= x_0 - \left(A^T \nabla^2 f(v_0) A\right)^{-1} A^T \nabla f(v_0)$$

$$= x_0 - A^{-1} \left(\nabla^2 f(v_0)\right)^{-1} (A^T)^{-1} A^T \nabla f(v_0)$$

$$= x_0 - A^{-1} \left(\nabla^2 f(v_0)\right)^{-1} \nabla f(v_0)$$

$$= x_0 - A^{-1} (v_0 - v_1)$$

$$= x_0 - A^{-1} (Ax_0 + b - v_1)$$

$$= x_0 - x_0 - A^{-1} (b - v_1)$$

$$\boxed{x_1 = A^{-1} (v_1 - b)}$$

Thus, Newton's method is affine invariant.

## Problem 4:

**Strategy :** I have completed the code in these five steps explained below.

### Step_1: Defining the gradient descent function

I have previously (in HW 3) defined the gradient descent function with the inputs mentioned in the question, namely:

-A function computing the gradient of the objective.
– An initialization point x(0)
– A flag specifying whether to use a fixed step size or a variable step size.
– A step size value (for fixed step size).
– A maximum number of iterations.
– A tolerance $\epsilon$ for the stopping criterion.

New inputs I have taken into the function in HW4:

∗ $\alpha$ (sufficient reduction parameter in Armijo condition).
∗ $\beta$ (backtracking parameter).
and *f (a function computing the value of the given function, to check the condition of variable step size)

First, I have checked if a non-zero step size is provided for the fixed step size case, because zero step size will not make any progress toward the minimum.

For the fixed step size case, 't' remains fixed at alpha. For the variable step size case, based on alpha_armijo and function value in each iteration, 't' takes varying values as $t = \beta, \beta^2, \beta^3 \ldots$ and such.

Then, until we reach the max number of iterations, as defined in the question, the function stops when the $L_2$ norm of the gradient falls below the tolerance value, so the stopping criteria is defined in the function as such.

And the iterate was updated using the gradient descent algorithm:

$$\mathbf{x^{(k+1)} = x^{(k)} - \alpha \cdot \nabla f(x^{(k)})}$$

Also, a global iteration count variable was declared to see exactly how many iterations it took in every case to reach the tolerance limit.

So my function for step 1 looks like this :

```
# Step_1: Defining the gradient descent function with backtracking line search
def gradient_descent(grad_func, f, x0, step_flag, alpha, alpha_armijo,
beta_backtracking, max_iteration, tolerance):
    global iteration_count  # Declare the global variable first

    # Check if a non-zero step size is given for fixed step size
    if step_flag == 'fixed' and alpha <= 0:
        print('Step size alpha must be greater than zero.')
        return None

    # Initialize
    x = x0
    X = x.reshape(-1, 1)  # Store iterates as columns

    for k in range(max_iteration):
        grad = grad_func(x)  # Compute gradient using the provided grad_func

        # Check stopping criterion
        grad_norm = 0.0
        for g in grad:
            grad_norm += g**2
        grad_norm = grad_norm**0.5  # Take the square root

        # Check if the gradient norm is below the tolerance,break if not.
        if grad_norm <= tolerance:
            iteration_count = k + 1  # Update the global variable
            break

        # Step Size Selection
        if step_flag == 'fixed':
            t = alpha  # Use the provided fixed step size
        elif step_flag == 'backtracking':
            t = 1.0  # Start with an initial step size
            while True:
```

```
            if f(x - t * grad) <= f(x) - alpha_armijo * t * grad_norm**2:
                break
            t *= beta_backtracking  # Reduce step size by beta

        # Gradient Descent Update
        x = x - t * grad

        # Store iterate
        X = np.hstack((X, x.reshape(-1, 1)))

    # If the loop completes without breaking, set iteration_count to
max_iteration
    if grad_norm > tolerance:
        iteration_count = max_iteration

    return X
```

**Step_2: Taking the inputs as mentioned in question**

Matrix Q1 and Q2 as mentioned in the question and their corresponding step sizes are taken.

**Step_3: Function and gradient function**

The given quadratic function was :

$$\mathbf{f(x) = \frac{1}{2}\, x^T Q\, x}$$

$$\mathbf{So,} \quad \nabla \mathbf{f(x) = Q\, x}$$

This is implemented in code as:

```
#Step_3: Function and gradient function
def grad_func1(x):
    return Q1 @ x  # Gradient for Q1

def grad_func2(x):
```

```
    return Q2 @ x  # Gradient for Q2

# Functions for quadratic function values
def f1(x):
    return 0.5 * x.T @ Q1 @ x  # Quadratic function for Q1

def f2(x):
    return 0.5 * x.T @ Q2 @ x  # Quadratic function for Q2
```

**Step_4: Hyperparameters**

The remaining hyperparameters in the design were initial point, maximum iterations and the stopping criteria tolerance. They were chosen as:

```
#Step_4: Hyperparameters
# Initial point
x0 = np.array([1, 1])

# Maximum iterations and tolerance
max_iter = 2000
tolerance = 1e-7
```

**Step_5: Run Gradient Descent and Plot Results**

Finally the gradient descent function was run on the given step size and matrix and the result (Contour plot of f(x), Function value (in semi log plot) and Gradient norm) were plotted in accordance.

## Full code for my implementation:

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

# Global variable to store the number of iterations
iteration_count = 0

# Step_1: Defining the gradient descent function with backtracking line search
def gradient_descent(grad_func, f, x0, step_flag, alpha, alpha_armijo,
beta_backtracking, max_iteration, tolerance):
    global iteration_count  # Declare the global variable first

    # Check if a non-zero step size is given for fixed step size
    if step_flag == 'fixed' and alpha <= 0:
        print('Step size alpha must be greater than zero.')
        return None

    # Initialize
    x = x0
    X = x.reshape(-1, 1)  # Store iterates as columns

    for k in range(max_iteration):
        grad = grad_func(x)  # Compute gradient using the provided grad_func

        # Check stopping criterion
        grad_norm = 0.0
        for g in grad:
            grad_norm += g**2
        grad_norm = grad_norm**0.5  # Take the square root

        # Check if the gradient norm is below the tolerance,break if not.
        if grad_norm <= tolerance:
            iteration_count = k + 1  # Update the global variable
            break

        # Step Size Selection
        if step_flag == 'fixed':
            t = alpha  # Use the provided fixed step size
```

```
        elif step_flag == 'backtracking':
            t = 1.0  # Start with an initial step size
            while True:
                if f(x - t * grad) <= f(x) - alpha_armijo * t * grad_norm**2:
                    break
                t *= beta_backtracking  # Reduce step size by beta

        # Gradient Descent Update
        x = x - t * grad

        # Store iterate
        X = np.hstack((X, x.reshape(-1, 1)))

    # If the loop completes without breaking, set iteration_count to
max_iteration
    if grad_norm > tolerance:
        iteration_count = max_iteration

    return X

# Step_2: Taking the inputs as mentioned in question
# Define two different Q matrices
Q1 = np.array([[1, 0], [0, 1]])   # Case 1
Q2 = np.array([[10, 0], [0, 1]])  # Case 2

# Step sizes
alphas_Q1 = [0.1, 0.5]
alphas_Q2 = [0.01, 0.05]

# Backtracking parameters
alpha_armijo = 1e-4
beta_backtracking = 0.5

# Step_3: Function and gradient function
# Functions for gradients
def grad_func1(x):
    return Q1 @ x  # Gradient for Q1
```

```
def grad_func2(x):
    return Q2 @ x  # Gradient for Q2

# Functions for quadratic function values
def f1(x):
    return 0.5 * x.T @ Q1 @ x  # Quadratic function for Q1

def f2(x):
    return 0.5 * x.T @ Q2 @ x  # Quadratic function for Q2

Q_matrices = [Q1, Q2]
grad_funcs = [grad_func1, grad_func2]
funcs = [f1, f2]
alpha_sets = [alphas_Q1, alphas_Q2]
Q_labels = ['Q1', 'Q2']

# Step_4: Hyperparameters
# Initial point
x0 = np.array([1, 1])

# Maximum iterations and tolerance
max_iter = 2000
tolerance = 1e-7

# Step_5: Run Gradient Descent and Plot Results
# Initialize figure counter
figure_number = 1

for i in range(len(Q_matrices)):
    Q = Q_matrices[i]
    grad_func = grad_funcs[i]
    f = funcs[i]
    alphas = alpha_sets[i]
    Q_name = Q_labels[i]

    for j in range(len(alphas)):
        alpha = alphas[j]
        # Reset the global variable before each run
```

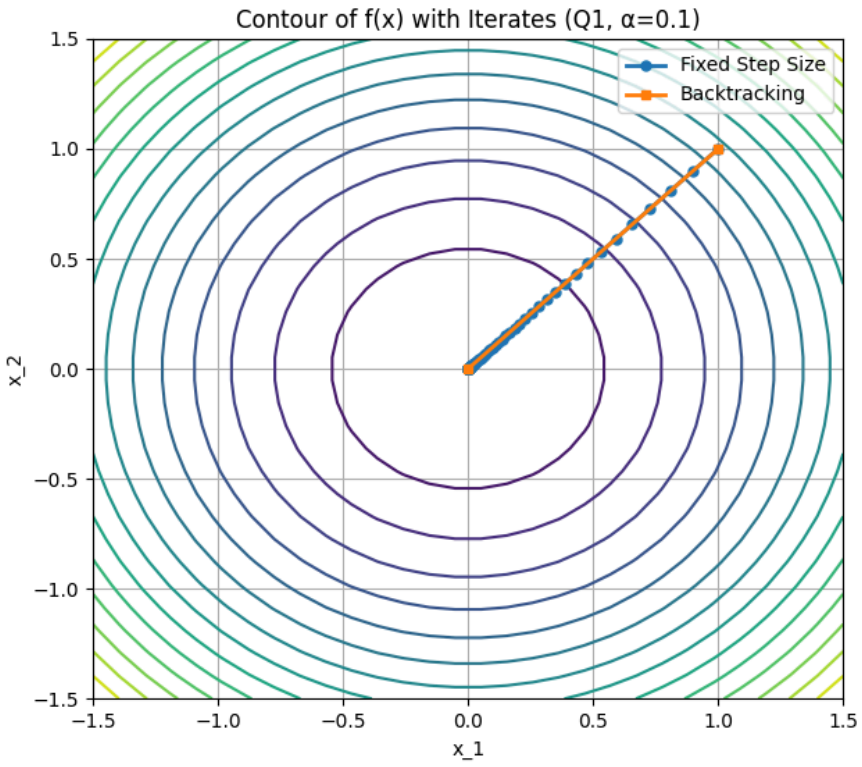
```
        iteration_count = 0

        # Run gradient descent with fixed step size
        X_fixed = gradient_descent(grad_func, f, x0, 'fixed', alpha,
alpha_armijo, beta_backtracking, max_iter, tolerance)
        fixed_iterations = iteration_count

        # Reset the global variable before each run
        iteration_count = 0

        # Run gradient descent with backtracking line search
        X_backtracking = gradient_descent(grad_func, f, x0, 'backtracking',
alpha, alpha_armijo, beta_backtracking, max_iter, tolerance)
        backtracking_iterations = iteration_count

        # Print the number of iterations for this case
        print(f'Case: {Q_name}, Step size (α): {alpha}, Iterations (Fixed):
{fixed_iterations}')
        print(f'Case: {Q_name}, Backtracking, Iterations:
{backtracking_iterations}')

        # Compute function values and gradient norms for fixed step size
        num_iters_fixed = X_fixed.shape[1]
        f_vals_fixed = np.zeros(num_iters_fixed)
        grad_norms_fixed = np.zeros(num_iters_fixed)

        for k in range(num_iters_fixed):
            f_vals_fixed[k] = f(X_fixed[:, k])
            grad_norms_fixed[k] = np.linalg.norm(grad_func(X_fixed[:, k]), 2)

        # Compute function values and gradient norms for backtracking
        num_iters_backtracking = X_backtracking.shape[1]
        f_vals_backtracking = np.zeros(num_iters_backtracking)
        grad_norms_backtracking = np.zeros(num_iters_backtracking)

        for k in range(num_iters_backtracking):
            f_vals_backtracking[k] = f(X_backtracking[:, k])
```

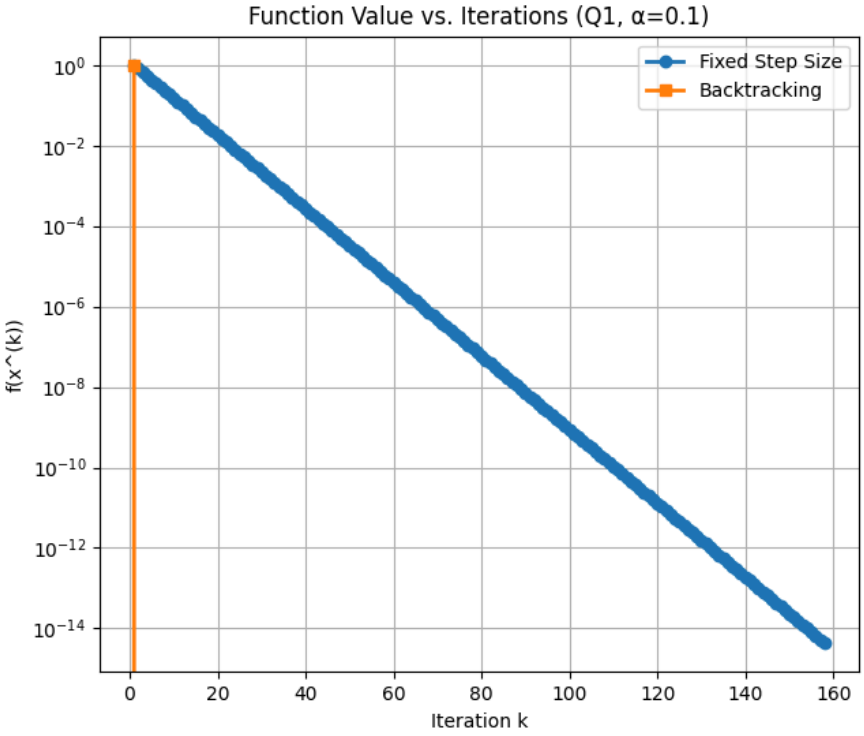
```
            grad_norms_backtracking[k] =
np.linalg.norm(grad_func(X_backtracking[:, k]), 2)

        # (a) Contour Plot with Iterates
        plt.figure(figure_number)
        figure_number += 1
        X1, X2 = np.meshgrid(np.linspace(-1.5, 1.5, 30), np.linspace(-1.5, 1.5,
30))
        F_vals = np.array([[f(np.array([x1, x2])) for x1, x2 in
zip(np.ravel(X1), np.ravel(X2))]]).reshape(X1.shape)
        plt.contour(X1, X2, F_vals, 20)
        plt.plot(X_fixed[0, :], X_fixed[1, :], '-o', linewidth=2, markersize=5,
label='Fixed Step Size')
        plt.plot(X_backtracking[0, :], X_backtracking[1, :], '-s', linewidth=2,
markersize=5, label='Backtracking')
        plt.title(f'Contour of f(x) with Iterates ({Q_name}, α={alpha})')
        plt.xlabel('x_1')
        plt.ylabel('x_2')
        plt.legend()
        plt.grid(True)
        plt.show()

        # (b) Function Value vs. Iterations (Semi-log Scale)
        plt.figure(figure_number)
        figure_number += 1
        plt.semilogy(range(1, num_iters_fixed + 1), f_vals_fixed, '-o',
linewidth=2, label='Fixed Step Size')
        plt.semilogy(range(1, num_iters_backtracking + 1), f_vals_backtracking,
'-s', linewidth=2, label='Backtracking')
        plt.title(f'Function Value vs. Iterations ({Q_name}, α={alpha})')
        plt.xlabel('Iteration k')
        plt.ylabel('f(x^(k))')
        plt.legend()
        plt.grid(True)
        plt.show()

        # (c) Gradient Norm vs. Iterations
        plt.figure(figure_number)
```

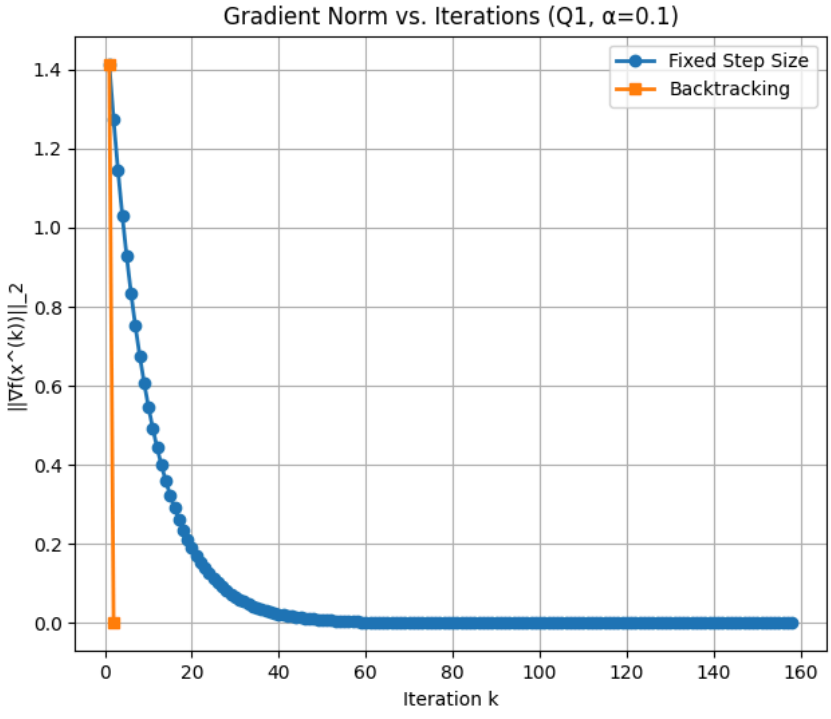
```
        figure_number += 1
        plt.plot(range(1, num_iters_fixed + 1), grad_norms_fixed, '-o',
linewidth=2, label='Fixed Step Size')
        plt.plot(range(1, num_iters_backtracking + 1), grad_norms_backtracking,
'-s', linewidth=2, label='Backtracking')
        plt.title(f'Gradient Norm vs. Iterations ({Q_name}, α={alpha})')
        plt.xlabel('Iteration k')
        plt.ylabel('||∇f(x^(k))||_2')
        plt.legend()
        plt.grid(True)
        plt.show()
```

## Labeled output results for each case:

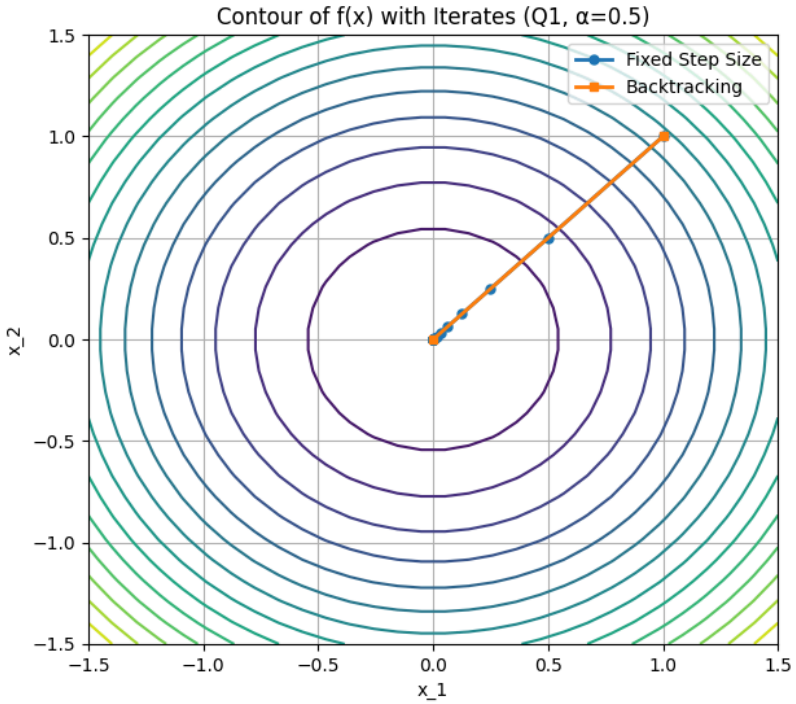
**Case 1:** Q = [1 0 , 0 1], α = 0.1

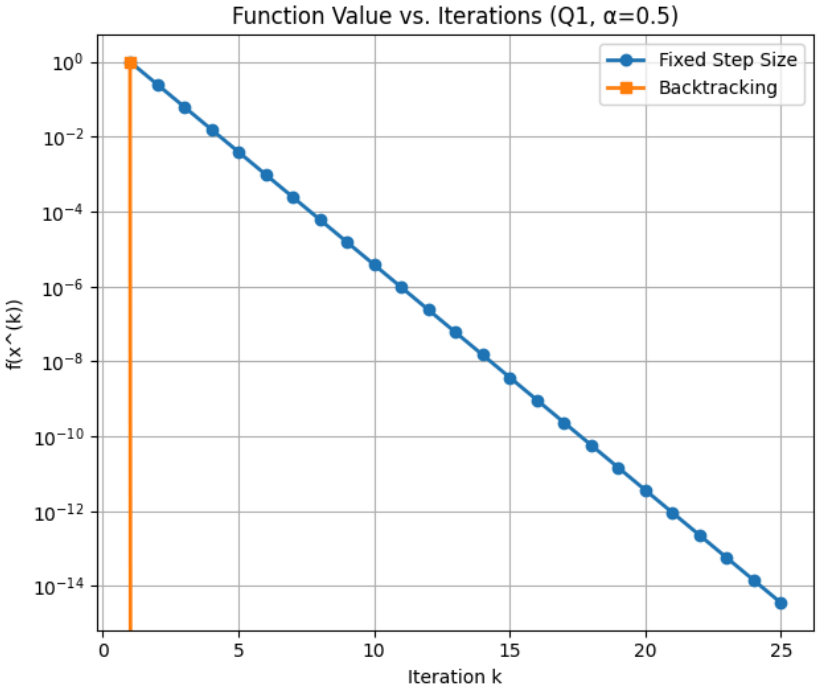


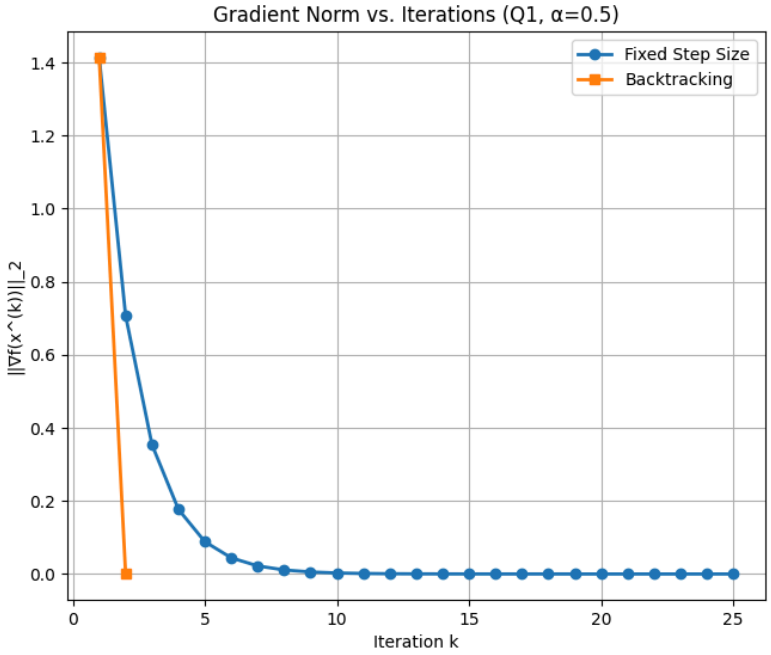

Function Value vs. Iterations (Q1, α=0.1)
Fixed Step Size
Backtracking
f(x^(k))
Iteration k


Gradient Norm vs. Iterations (Q1, α=0.1)
Fixed Step Size
Backtracking
||∇f(x^(k))||_2
Iteration k

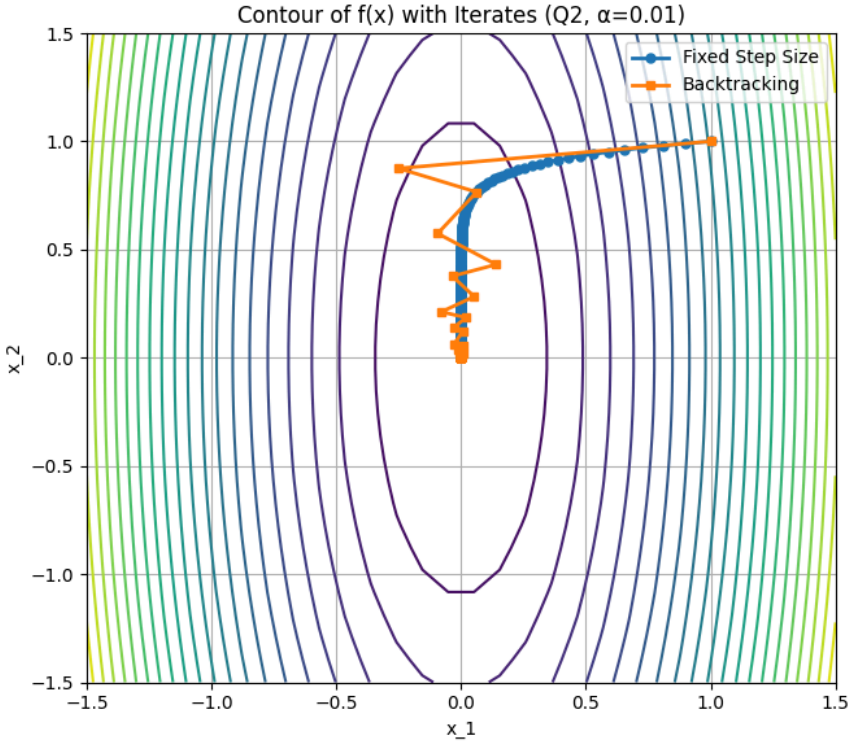
**Case 2 :** Q = [1 0 , 0 1], α = 0.5

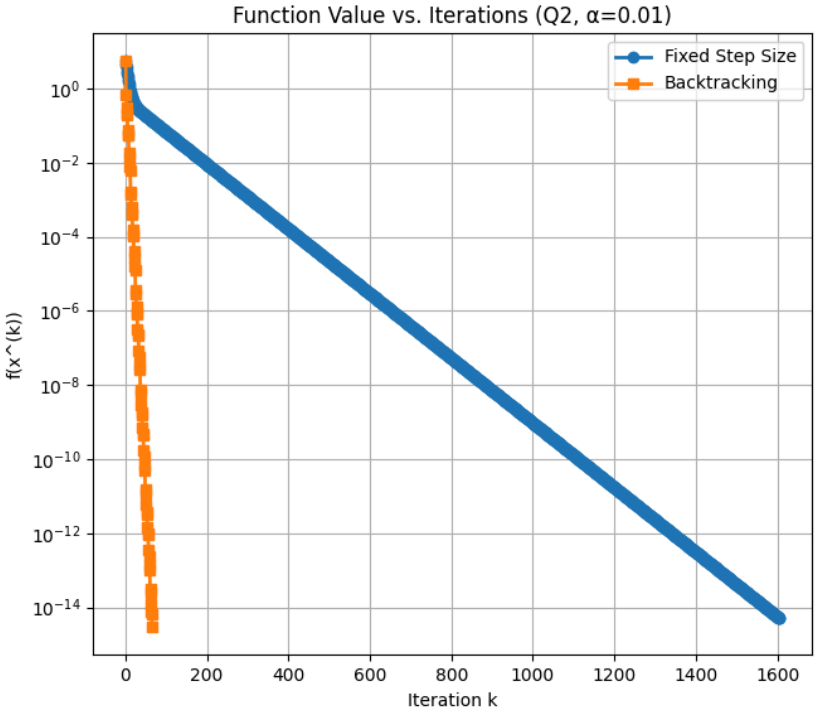


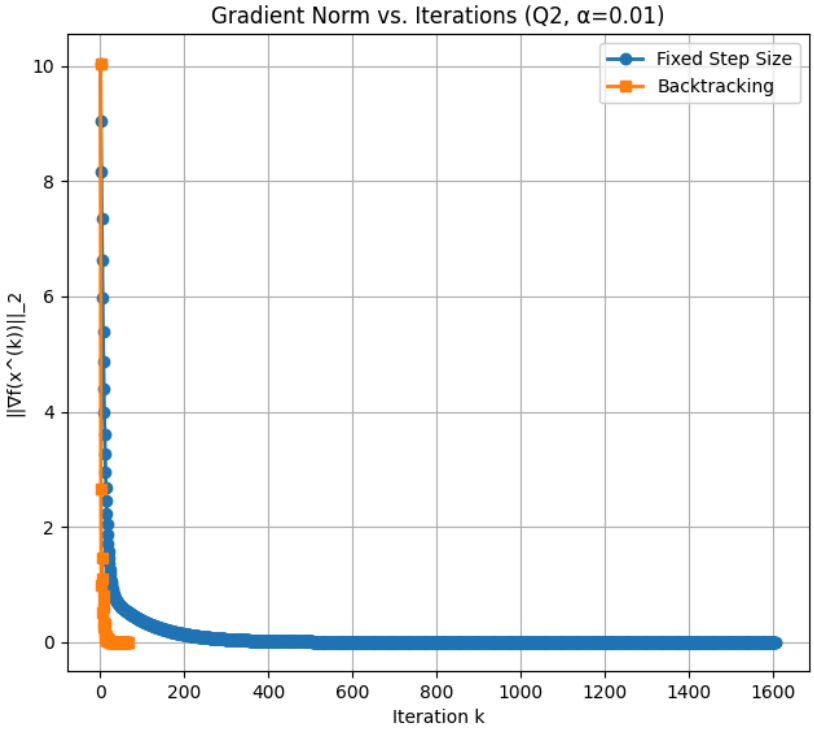





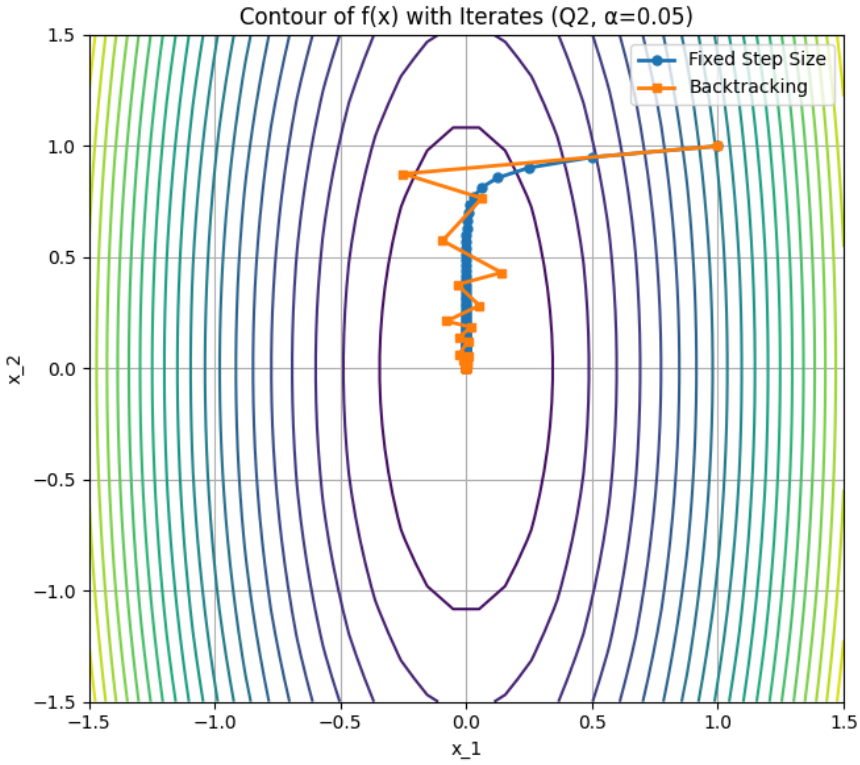

**Case 3:** Q = [10 0, 0 1], α = 0.01

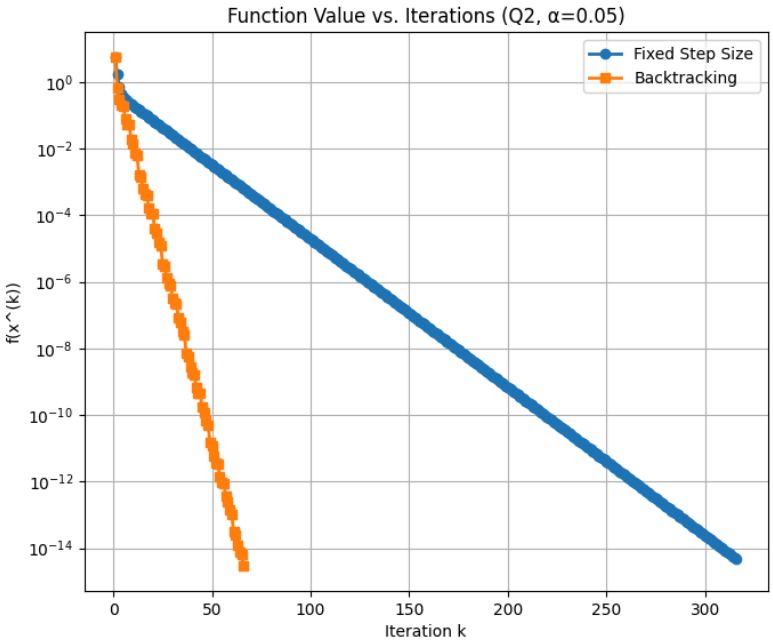


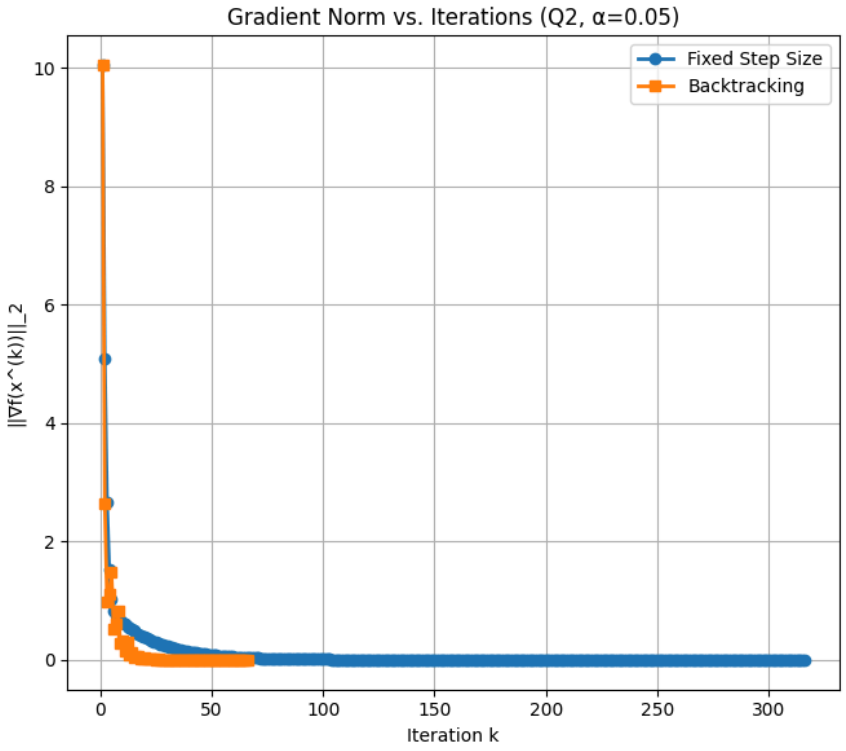

Function Value vs. Iterations (Q2, α=0.01)
Fixed Step Size
Backtracking
f(x^(k))
Iteration k


Gradient Norm vs. Iterations (Q2, α=0.01)
Fixed Step Size
Backtracking
||∇f(x^(k))||_2
Iteration k

**Case 4 :** Q = [10 0, 0 1], $\alpha = 0.05$






**Comparison of the convergence behavior of fixed step size and variable step size:**

From my results and the plots: (with initialization at [1,1], tolerance= $10^{-7}$, max_iterations= 2000)

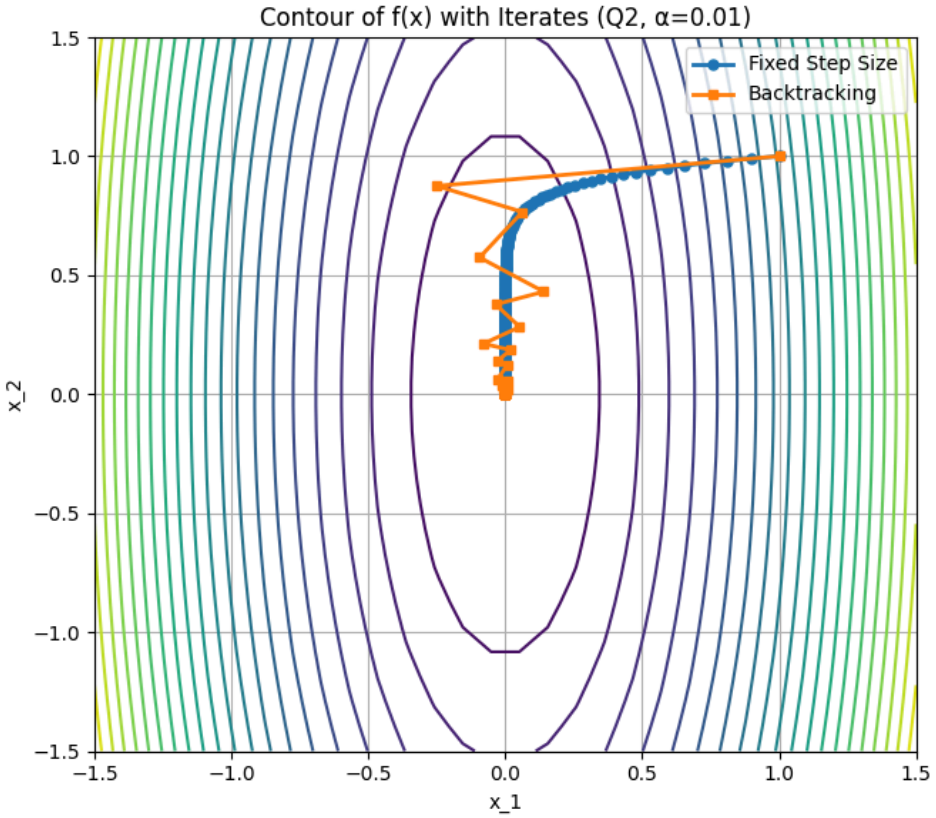
**No of iterations needed to converge in all the cases:**

Case: Q1, Step size (α): 0.1, Iterations (Fixed): 158
Case: Q1, Backtracking, Iterations: 2
Case: Q1, Step size (α): 0.5, Iterations (Fixed): 25
Case: Q1, Backtracking, Iterations: 2

Case: Q2, Step size (α): 0.01, Iterations (Fixed): 1605
Case: Q2, Backtracking, Iterations: 66
Case: Q2, Step size (α): 0.05, Iterations (Fixed): 316
Case: Q2, Backtracking, Iterations: 66

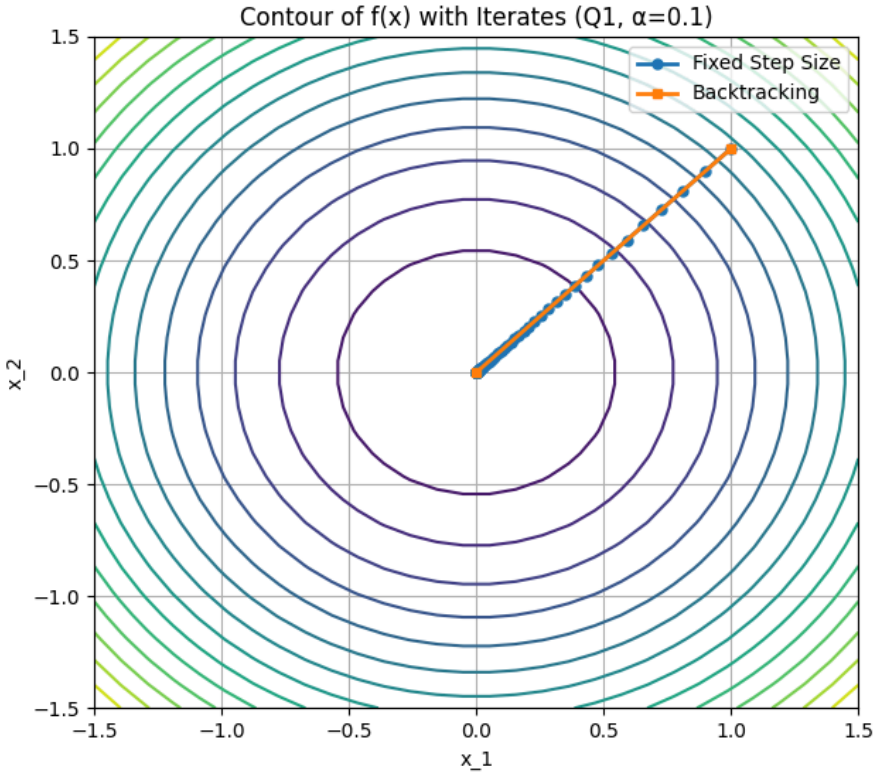
So, clearly the **gradient descent algorithm with variable step size (backtracking) converges faster.**

First comparison point is that, fixed step size approach uses the same step size throughout the entire iterations, whereas backtracking uses variable step sizes on the fly based on the local curvature of the function, utilizing the gradient information more effectively. So, backtracking takes more "**direct path towards the minima**" compared to fixed step size approach. This argument is evident from this case below:




Secondly, the main reason behind **backtracking being faster** is that it updates step size adaptively, which means it dynamically adjusts the step size 't' in each iteration to ensure sufficient decrease in the objective function. So effectively, this allows the **backtracking** approach to take l**arger steps when possible** and **smaller steps when necessary**, leading to faster convergence.

This argument is clearly evident from this case below:




Here, both the fixed and variable step size approach takes a linear path towards the minima. But the fixed step size approach has to take the predefined manual step size α =0.5, even though we have the opportunity to take much larger steps which is cleverly utilized by backtracking. So for Q1 matrix, the fixed step size approach takes 158 iterations (with α =0.5) , whereas backtracking took only 2 iterations to converge.