Convex Optimization
Homework #06

Submitted By: Rifat Bin Rashid

RUID: 237000174

Problem -1:

Quadratic objective function:

$$f(x) = \frac{1}{2}\left(x_1^2 + \gamma x_2^2\right)$$

initial point: $x^{(0)} = \begin{bmatrix} \gamma \\ 1 \end{bmatrix}$

Now, $D_{x_1} f(x) = x_1$

$D_{x_2} f(x) = \gamma x_2$

so, gradient at $x^{(k)} = \begin{bmatrix} x_1^{(k)} \\ \gamma x_2^{(k)} \end{bmatrix}$

So,

$$x^{(k)} - t \nabla f(x^{(k)}) = \begin{bmatrix} (1-t) x_1^{(k)} \\ (1-\gamma t) x_2^{(k)} \end{bmatrix}$$

$$= \left(\frac{\gamma - 1}{\gamma + 1}\right)^k \begin{bmatrix} (1-t)\gamma \\ (1-\gamma t)(-1)^k \end{bmatrix}$$

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

This is minimized by $t = \dfrac{2}{1+\gamma}$

So,

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

$$= \begin{bmatrix} (1-t)\,x_1^{(k)} \\ (1-\gamma t)\,\gamma\, x_2^{(k)} \end{bmatrix}$$

$$= \left(\frac{\gamma-1}{\gamma+1}\right)\begin{bmatrix} x_1^{(k)} \\ -x_2^{(k)} \end{bmatrix}$$

$$= \left(\frac{\gamma-1}{\gamma+1}\right)^{k+1}\begin{bmatrix} \gamma \\ (-1)^k \end{bmatrix}$$

Problem 2

Given, $f: \mathbb{R}^n \to \mathbb{R}$ is twice differentiable & strongly convex

$$\nabla^2 f(x) = D + vu^T$$

$D \in \mathbb{R}^{n \times n}$ diagonal, $u, v \in \mathbb{R}^n$

Now, Newton's Method update Rule:

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

Given $\nabla^2 f(x) = D + Vu^T$

So, $\left[\nabla^2 f(x)\right]^{-1} = (D + UV^T)^{-1}$ [SMW method]

$$= D^{-1} - \frac{D^{-1} UV^T D^{-1}}{1 + V^T D^{-1} U}$$

So cost in computing $(D + UVT)^{-1} \nabla f(x^{(k)})$ will be:

(i) Computing D^{-1} ; D is $\mathbb{R}^{n\times n}$ Diagonal, so inverting it takes $O(n)$

(ii) Comuting $D^{-1}\nabla f(x^{(k)})$: diagonal matrix-vector multiplicates, takes $O(n)$

(iii) Computing $v^T D^{-1} u$: dot product of two n vectors, takes $O(n)$

(iv) Computing $D^{-1}u$: elementwise vector operation, takes $O(n)$

(v) Computing $v^T D^{-1}$: takes $O(n)$

(vi) Combine terms using the formula: $O(n)$
(scaler-vector multiplication & vector inner/outer products)

So, each iteration of Newton's method using SMW formula will take $\boxed{O(n)}$ time problem 3:

Given function:

$$f(x) = (c^T x)^4 + \sum_{i=1}^{n} w_i \exp(x_i) \quad , \quad w \geq 0$$

$\nabla f(x)$:

$$\nabla\left((c^T x)^4\right) = 4(c^T x)^3 c$$

$$\nabla\left(\sum_{i=1}^{n} w_i \exp(x_i)\right) = \begin{bmatrix} w_1 \exp(x_1) \\ \vdots \\ w_n \exp(x_n) \end{bmatrix}$$

$$\nabla f(x) = 4(c^T x)^3 c + \begin{bmatrix} w_1 \exp(x_1) \\ \vdots \\ w_n \exp(x_n) \end{bmatrix}$$

so computing $\nabla f(x)$ will require dot product $c^T x$, scaler multiplication & element wise operations

$\downarrow$

$O(n)$ operations.

$\underline{\nabla^2 f(x):}$ $\nabla^2\left((c^T x)^4\right) = 12\,(c^T x)^2\, c c^T$

This is a rank-1 matrix, scaled by $12(c^T x)^2$

$$\nabla^2\left(\sum_{i=1}^{n} w_i \exp(x_i)\right) = \begin{bmatrix} w_1 \exp(x_1) & & 0 \\ & \ddots & \\ 0 & & w_n \exp(x_n) \end{bmatrix}$$

This is a diagonal matrix

$\underline{\text{So,}}$ $\nabla^2 f(x)$ will be $D + UV^T$ form

with: $\begin{cases} D = \mathrm{Diag}\left(w_1 \exp(x_1), \ldots, w_n \exp(x_n)\right) \\ U = 12(c^T x)^2 c \\ V = c \end{cases}$

From previous problem-2, $\nabla^2 f(x)$ can be done in $O(n)$ operations using SMW formula.

So; (a) $\boxed{\text{True.}}$

Newton's method has quadratic convergence, while gradient descent has linear convergence, so Newton's method will require fewer iterations.

(b) $\boxed{\text{False}}$

Each gradient descent iterations will cost $O(n)$ operations.

Each Newton's method will also cost $O(n)$ operations because: Computing $\nabla f(x)$ is $O(n)$

Computing $\nabla^2 f(x)$ is $O(n)$

Computing $[\nabla^2 f(x)]^{-1} \nabla f(x)$ is $O(n)$ using SMW formula.

(c) $\boxed{\text{False.}}$

Since Newton's method has a better convergence but same cost per iteration, it is unambiguously better.

Problem 4:

I have solved the problem in these steps :

Step 1: Defining the objective function f(x)

Here, the given objective function was defined:

$$f\,(x1,\,x2) = e^{x1\,+3x2-0.1} + e^{x1-3x2-0.1} + e^{-x1\,-0.1}$$

```
# Step 1: Defining the objective function f(x)
def f(x):
    return np.exp(x[0] + 3*x[1] - 0.1) + np.exp(x[0] - 3*x[1] - 0.1) +
np.exp(-x[0] - 0.1)
```

Step 2: Defining the gradient ∇ f(x)

Here, the gradient of the objective function with respect to two variables was defined.

$$\nabla f(x) = [\ \partial f/\partial x1,\ \partial f/\partial x2\ ]$$

Computed as:

$$\partial f/\partial x1 = \exp(x1 + 3x2 - 0.1) + \exp(x1 - 3x2 - 0.1) - \exp(-x1 - 0.1)$$
$$\partial f/\partial x2 = 3 * \exp(x1 + 3x2 - 0.1) - 3 * \exp(x1 - 3x2 - 0.1)$$

So:

$$\nabla f(x) = [\ \exp(x1 + 3x2 - 0.1) + \exp(x1 - 3x2 - 0.1) - \exp(-x1 - 0.1),\ 3\exp(x1 + 3x2 - 0.1) - 3\exp(x1 - 3x2 - 0.1)\ ]$$

```
# Step 2: Defining the gradient ∇f(x)
def grad_f(x):
    df_dx1 = np.exp(x[0] + 3*x[1] - 0.1) + np.exp(x[0] - 3*x[1] - 0.1)
- np.exp(-x[0] - 0.1)
```

```
    df_dx2 = 3*np.exp(x[0] + 3*x[1] - 0.1) - 3*np.exp(x[0] - 3*x[1] -
0.1)
    return np.array([df_dx1, df_dx2])
```

Step 3: Defining the Hessian ∇²f(x)

$$\nabla^2 f(x) = [[\ h11, h12\ ],\ [\ h12, h22\ ]]$$

Where:

e1 = exp(x1 + 3x2 - 0.1)
e2 = exp(x1 - 3x2 - 0.1)
e3 = exp(-x1 - 0.1)

Then:

h11 = e1 + e2 + e3
h12 = 3e1 - 3e2
h22 = 9e1 + 9e2

```
# Step 3: Defining the Hessian ∇²f(x)
def hess_f(x):
    e1 = np.exp(x[0] + 3*x[1] - 0.1)
    e2 = np.exp(x[0] - 3*x[1] - 0.1)
    e3 = np.exp(-x[0] - 0.1)
    h11 = e1 + e2 + e3
    h12 = 3*e1 - 3*e2
    h22 = 9*e1 + 9*e2
    return np.array([[h11, h12], [h12, h22]])
```

Step 4: Backtracking line search (Armijo rule)

In this step, backtracking line search with given parameters ($\alpha = 0.1$ and $\beta = 0.7$.) was implemented. It finds a step size t such that:

$$f(x + t * d) <= f(x) + \alpha * t * \nabla f(x)^T d$$

```
# Step 4: Backtracking line search (Armijo rule)
def backtracking(x, d, alpha=0.1, beta=0.7):
```

```
    t = 1
    fx = f(x)
    grad_fx = grad_f(x)
    while f(x + t*d) > fx + alpha * t * grad_fx.dot(d):
        t *= beta
    return t
```

Step 5: ℓ1-norm steepest descent
Here, ℓ1-norm steepest descent chooses the direction along the coordinate with the largest gradient.

```
# Step 5: ℓ1-norm steepest descent
def sd_l1(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        i = np.argmax(np.abs(g))
        d = -np.sign(g[i]) * np.eye(2)[i]
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)
```

Step 6: ℓ∞-norm steepest descent
ℓ∞-norm steepest descent moves in the direction of the sign of the gradient scaled by its 1-norm.

```
# Step 6: ℓ∞-norm steepest descent
def sd_linf(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
```

```
    for _ in range(max_iter):
        g = grad_f(x)
        d = -np.linalg.norm(g, 1) * np.sign(g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)
```

Step 7: Quadratic norm (P-norm) steepest descent

In this step, P-norm descent moves in direction $-P^{-1} \nabla f(x)$, where P is a positive definite matrix.

```
# Step 7: Quadratic norm (P-norm) steepest descent
def sd_quad(x0, P, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        d = -np.linalg.solve(P, g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)
```

Step 8: Euclidean gradient descent

Here, Euclidean gradient descent moves in direction $-\nabla f(x)$.

```
# Step 8: Euclidean gradient descent
def gd_euclidean(x0, max_iter=100):
```

```
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        d = -g
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)
```

Step 9: Newton's method using Hessian

In this step, Newton's method uses $-[\nabla^2 f(x)]^{-1} \nabla f(x)$ for rapid convergence.

```
# Step 9: Newton's method using Hessian
def newton_method(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        H = hess_f(x)
        d = -np.linalg.solve(H, g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)
```

Step 10-16: Execution and plot

Here, all the methods were executed and plotted accordingly.

Full code for my implementation :

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

# Step 1: Defining the objective function f(x)
def f(x):
    return np.exp(x[0] + 3*x[1] - 0.1) + np.exp(x[0] - 3*x[1] - 0.1) + np.exp(-x[0] - 0.1)

# Step 2: Defining the gradient ∇f(x)
def grad_f(x):
    df_dx1 = np.exp(x[0] + 3*x[1] - 0.1) + np.exp(x[0] - 3*x[1] - 0.1) - np.exp(-x[0] - 0.1)
    df_dx2 = 3*np.exp(x[0] + 3*x[1] - 0.1) - 3*np.exp(x[0] - 3*x[1] - 0.1)
    return np.array([df_dx1, df_dx2])

# Step 3: Defining the Hessian ∇²f(x)
def hess_f(x):
    e1 = np.exp(x[0] + 3*x[1] - 0.1)
    e2 = np.exp(x[0] - 3*x[1] - 0.1)
    e3 = np.exp(-x[0] - 0.1)
    h11 = e1 + e2 + e3
    h12 = 3*e1 - 3*e2
    h22 = 9*e1 + 9*e2
    return np.array([[h11, h12], [h12, h22]])

# Step 4: Backtracking line search (Armijo rule)
def backtracking(x, d, alpha=0.1, beta=0.7):
    t = 1
    fx = f(x)
```

```python
    grad_fx = grad_f(x)
    while f(x + t*d) > fx + alpha * t * grad_fx.dot(d):
        t *= beta
    return t

# Step 5: ℓ1-norm steepest descent
def sd_l1(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        i = np.argmax(np.abs(g))
        d = -np.sign(g[i]) * np.eye(2)[i]
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)

# Step 6: ℓ∞-norm steepest descent
def sd_linf(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        d = -np.linalg.norm(g, 1) * np.sign(g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
```

```
    return np.array(history)

# Step 7: Quadratic norm (P-norm) steepest descent
def sd_quad(x0, P, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        d = -np.linalg.solve(P, g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)

# Step 8: Euclidean gradient descent
def gd_euclidean(x0, max_iter=100):
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        d = -g
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)

# Step 9: Newton’s method using Hessian
def newton_method(x0, max_iter=100):
```

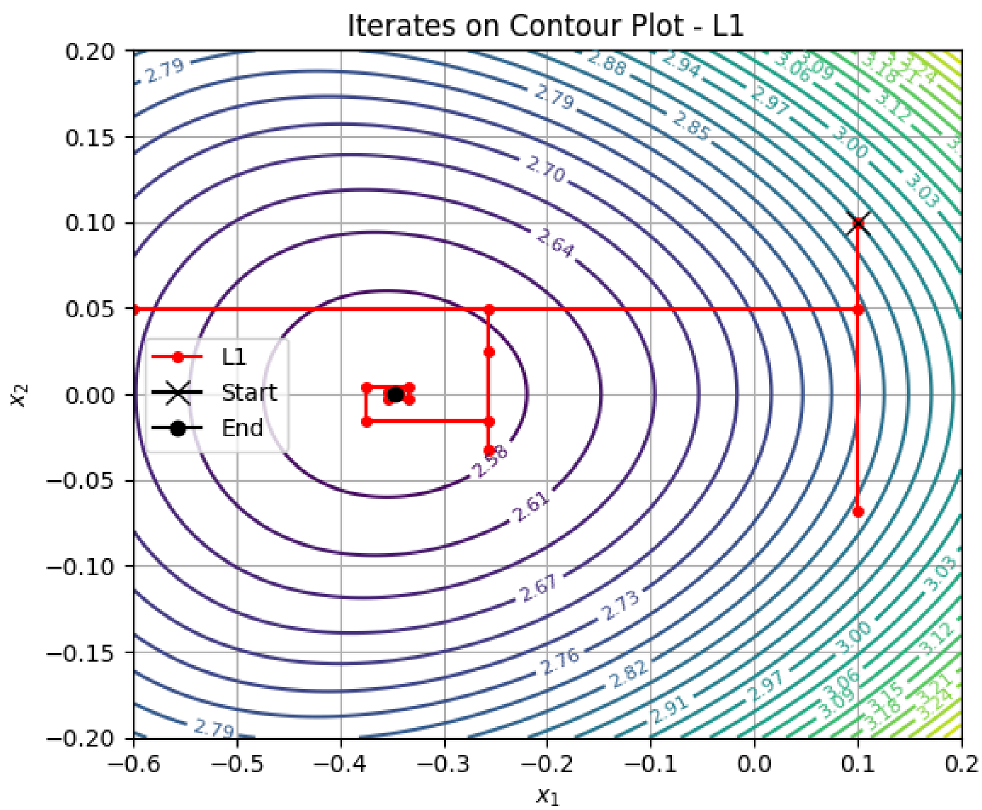
```python
    x = x0.copy()
    history = [x.copy()]
    for _ in range(max_iter):
        g = grad_f(x)
        H = hess_f(x)
        d = -np.linalg.solve(H, g)
        t = backtracking(x, d)
        x = x + t * d
        history.append(x.copy())
        if np.linalg.norm(g) < 1e-6:
            break
    return np.array(history)

# Step 10: Execute all methods and store trajectories
x0 = np.array([0.1, 0.1])
paths = {
    'L1': sd_l1(x0),
    'Linf': sd_linf(x0),
    'P1': sd_quad(x0, np.array([[2, 0], [0, 8]])),
    'P2': sd_quad(x0, np.array([[8, 0], [0, 2]])),
    'Euclidean': gd_euclidean(x0),
    'Newton': newton_method(x0)
}

# Step 11: Determine optimal function value p_star
p_star = min(f(path[-1]) for path in paths.values())

# Step 12: Define grid for contour plotting
x1 = np.linspace(-0.6, 0.2, 400)
x2 = np.linspace(-0.2, 0.2, 400)
X1, X2 = np.meshgrid(x1, x2)
Z = f(np.array([X1, X2]))
```

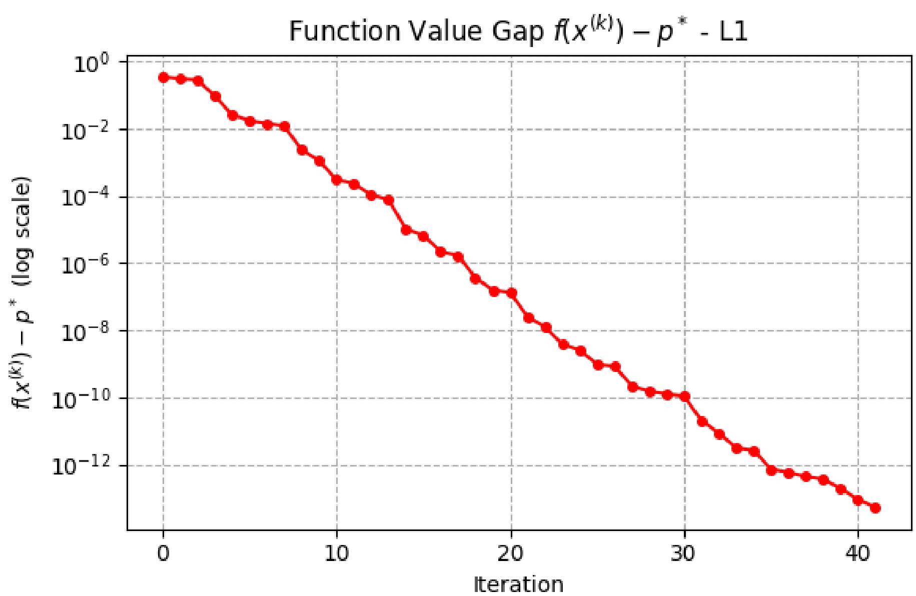
```python
colors = ['r', 'b', 'g', 'm', 'orange', 'cyan']

# Step 13: Plot contour with iterates for each method
for i, (label, path) in enumerate(paths.items()):
    plt.figure(figsize=(6, 5))
    CS = plt.contour(X1, X2, Z, levels=30, cmap='viridis')
    plt.clabel(CS, inline=1, fontsize=8)
    plt.plot(path[:, 0], path[:, 1], marker='o', markersize=4,
linestyle='-', linewidth=1.5, color=colors[i], label=label)
    plt.plot(path[0, 0], path[0, 1], marker='x', color='black',
markersize=10, label='Start')
    plt.plot(path[-1, 0], path[-1, 1], marker='o', color='black',
markersize=6, label='End')
    plt.title(f'Iterates on Contour Plot - {label}')
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Step 14: Plot function value gap for each method (semi-log)
for i, (label, path) in enumerate(paths.items()):
    gaps = [f(xk) - p_star for xk in path]
    plt.figure(figsize=(6, 4))
    plt.semilogy(range(len(gaps)), gaps, marker='o',
markersize=4, color=colors[i])
    plt.title(f'Function Value Gap $f(x^{{(k)}}) - p^*$ -
{label}')
    plt.xlabel('Iteration')
    plt.ylabel('$f(x^{(k)}) - p^*$ (log scale)')
    plt.grid(True, which='both', ls='--')
```

```
    plt.tight_layout()
    plt.show()

# Step 15: Plot function value gap for all methods combined
(semi-log)
plt.figure(figsize=(10, 6))
for i, (label, path) in enumerate(paths.items()):
    gaps = [f(xk) - p_star for xk in path]
    plt.semilogy(range(len(gaps)), gaps, marker='o',
markersize=3, linewidth=1.5, label=label, color=colors[i])
plt.title('Function Value Gap $f(x^{(k)}) - p^*$ vs Iteration
(All Methods)')
plt.xlabel('Iteration Number')
plt.ylabel('$f(x^{(k)}) - p^*$ (log scale)')
plt.legend()
plt.grid(True, which="both", ls="--", linewidth=0.5)
plt.tight_layout()
plt.show()

# Step 16: Print summary of iterations and final values
print("Method Summary:")
print(f"{'Method':<10} | {'Iterations':<10} | {'Final
f(x)':<12}")
print("-" * 36)
for label, path in paths.items():
    iterations = len(path) - 1
    final_fx = f(path[-1])
    print(f"{label:<10} | {iterations:<10} | {final_fx:<12.6f}")
```

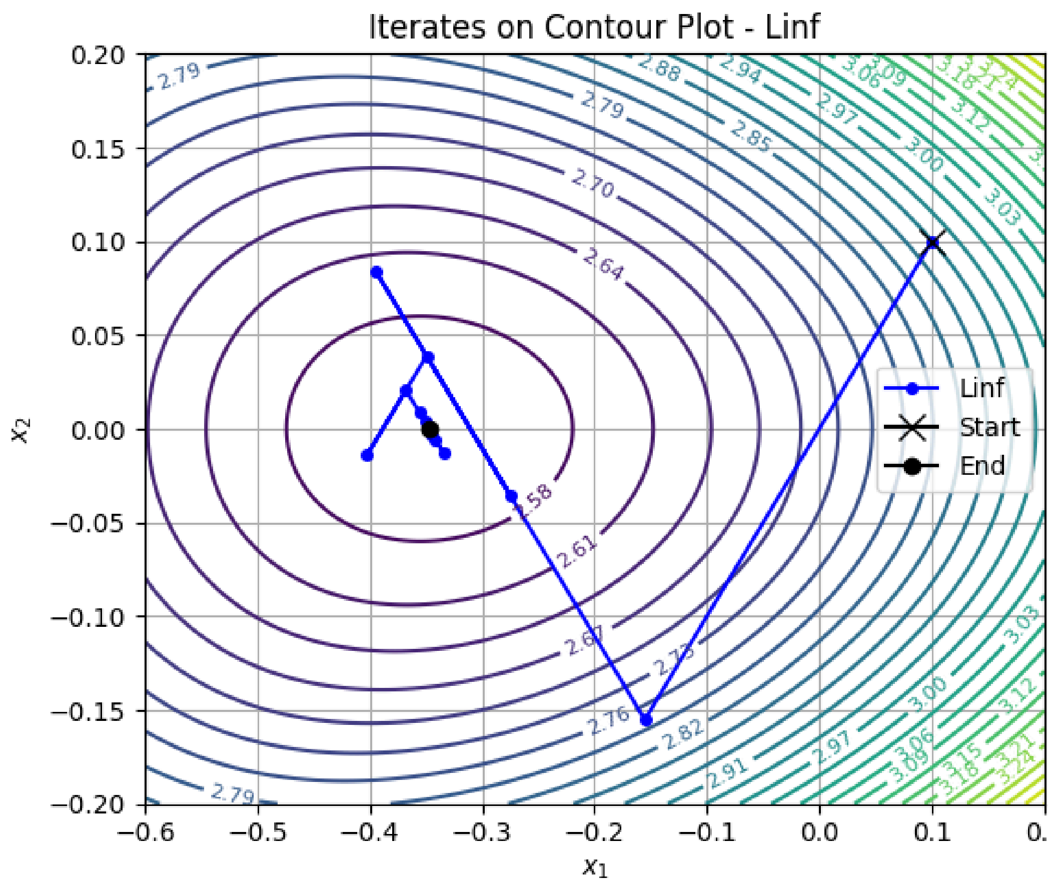
Results : *(All methods were justified using same initial point* $x_0 = (0.1,0.1)$ *and tolerance = 1e-6)*

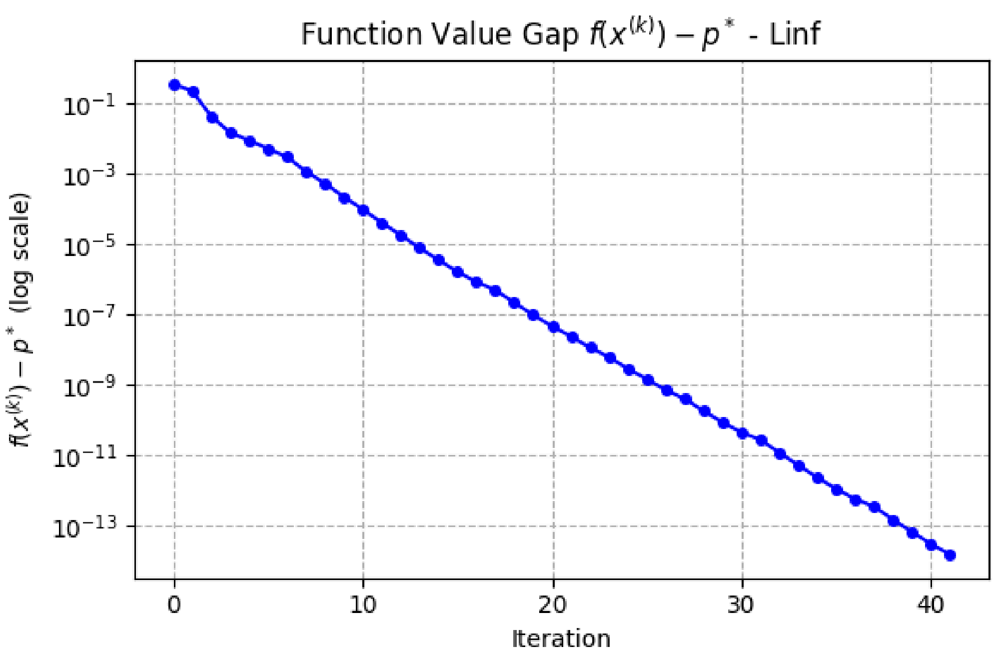
Method 1 : Steepest descent in the ℓ1 norm



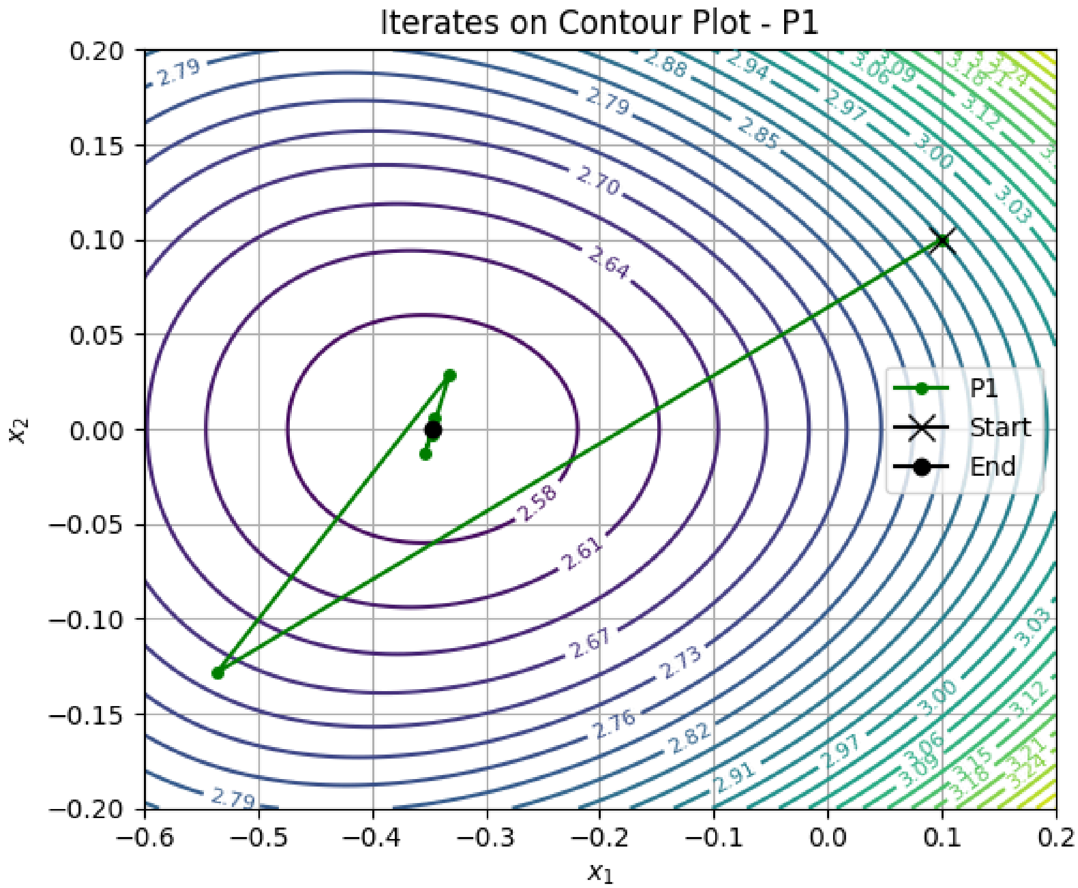


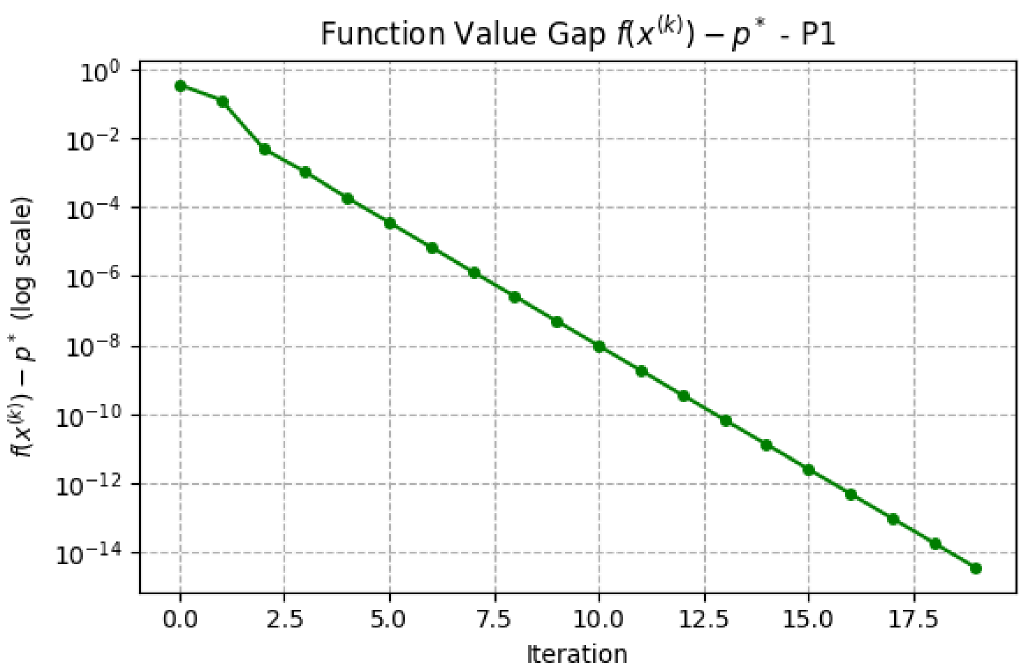
Method 2 : Steepest descent in the ℓ∞ norm



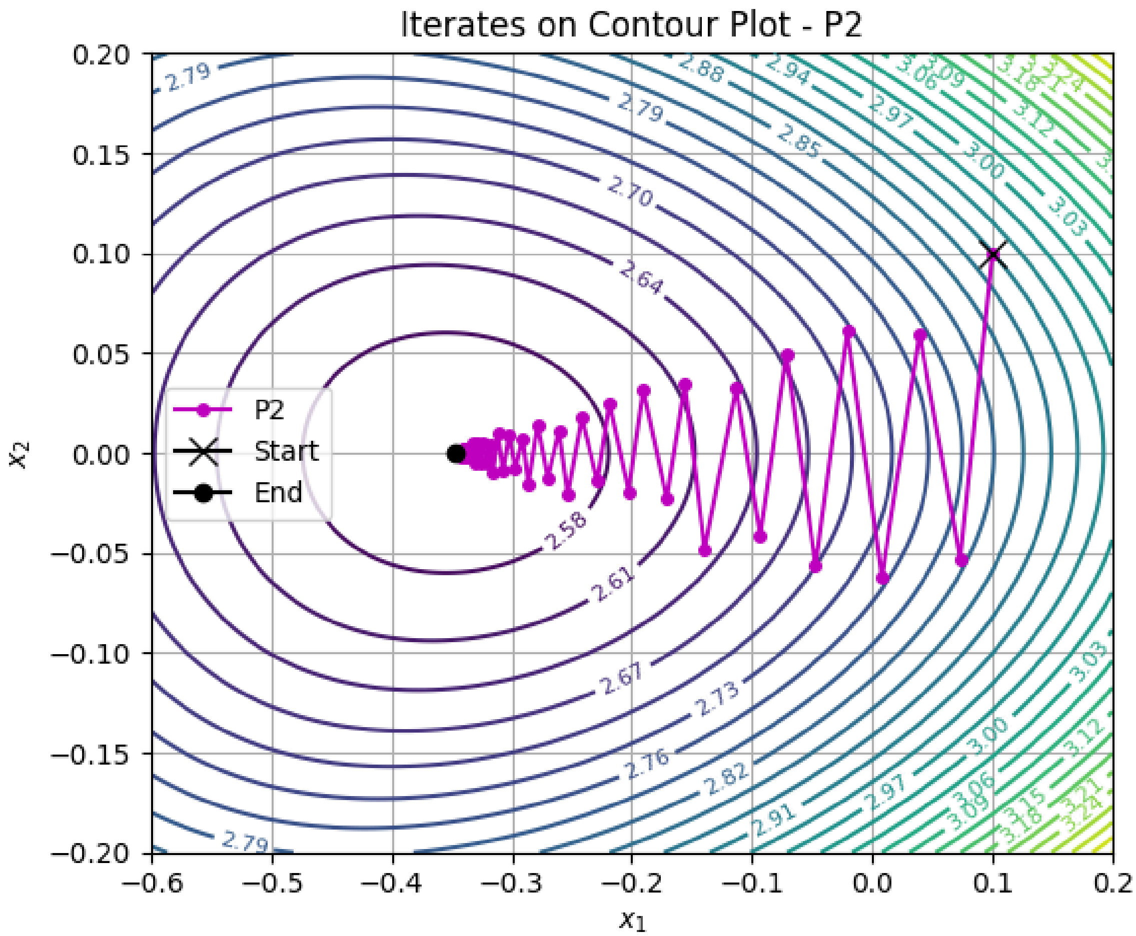


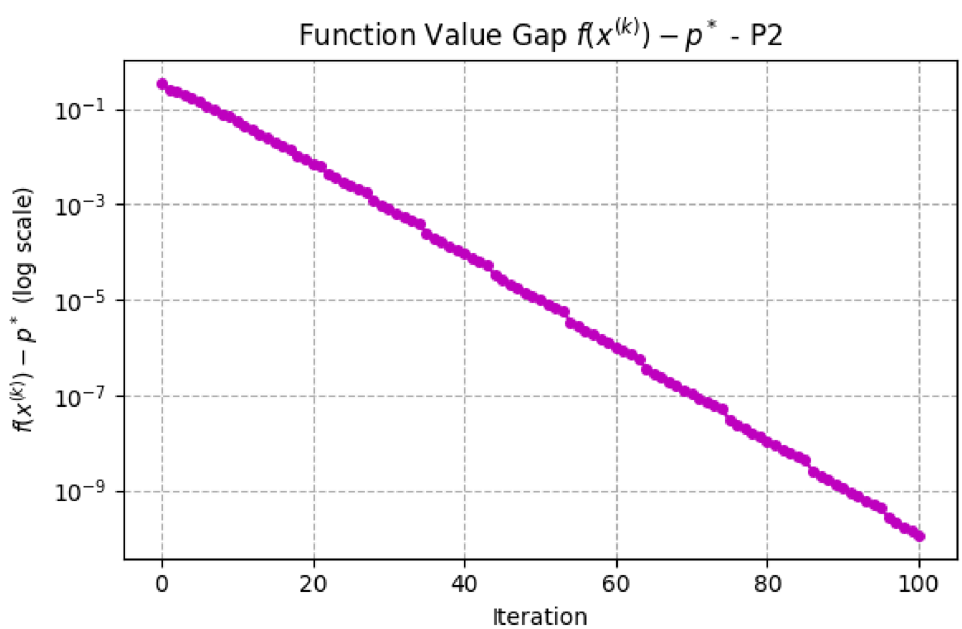
Method 3(A) : Steepest descent in a quadratic P -norm(P1)



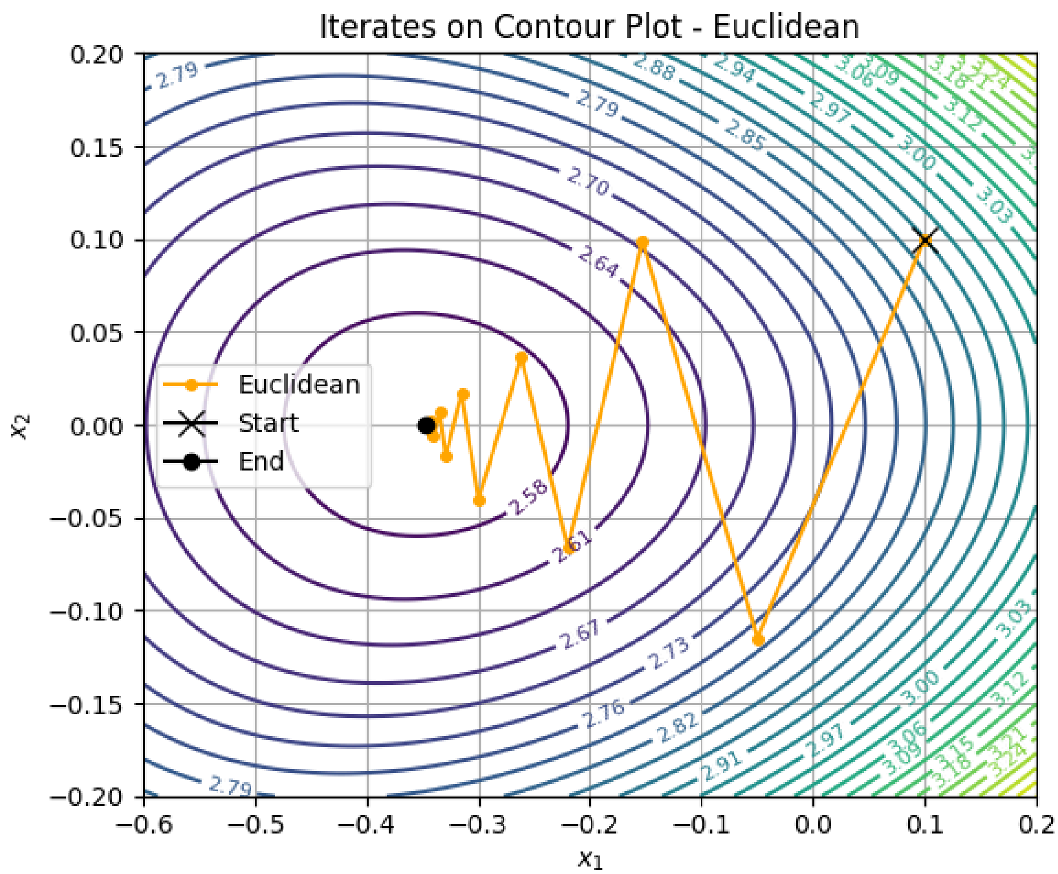


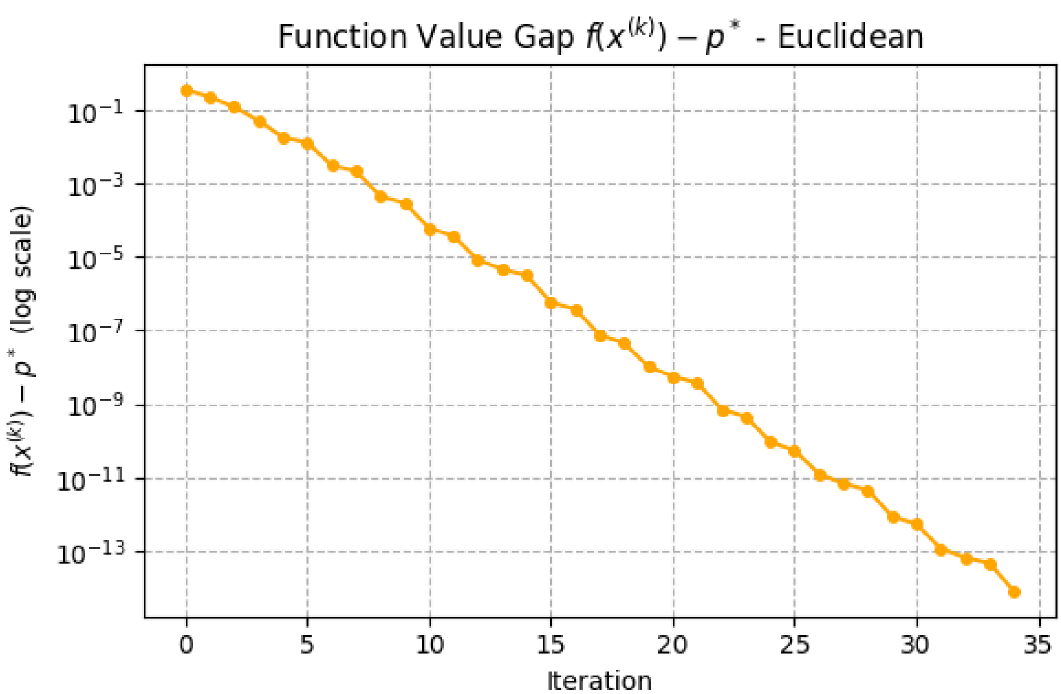
Method 3(B) : Steepest descent in a quadratic P -norm(P2)



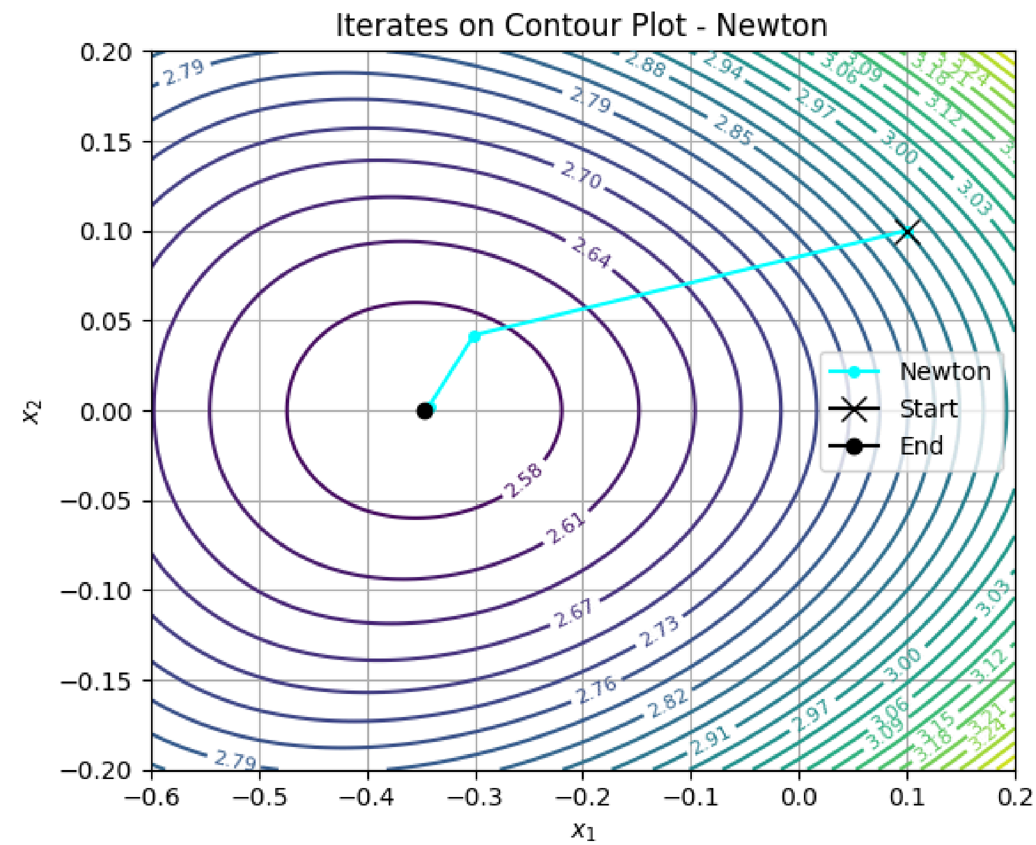


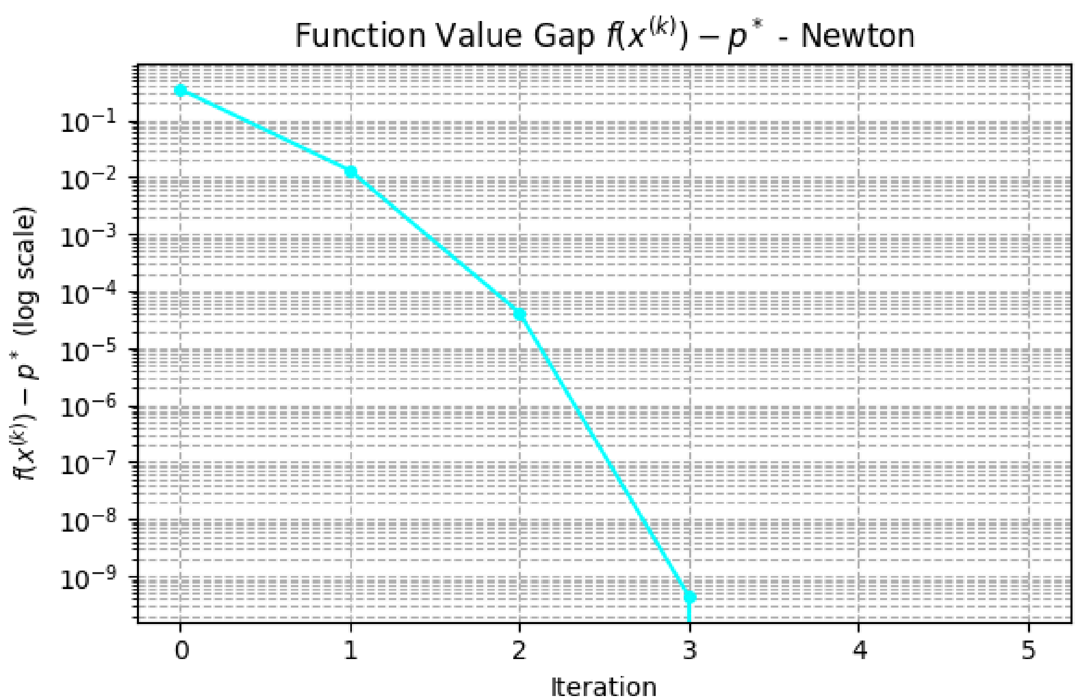
Method 4 :Gradient descent using the standard Euclidean norm.



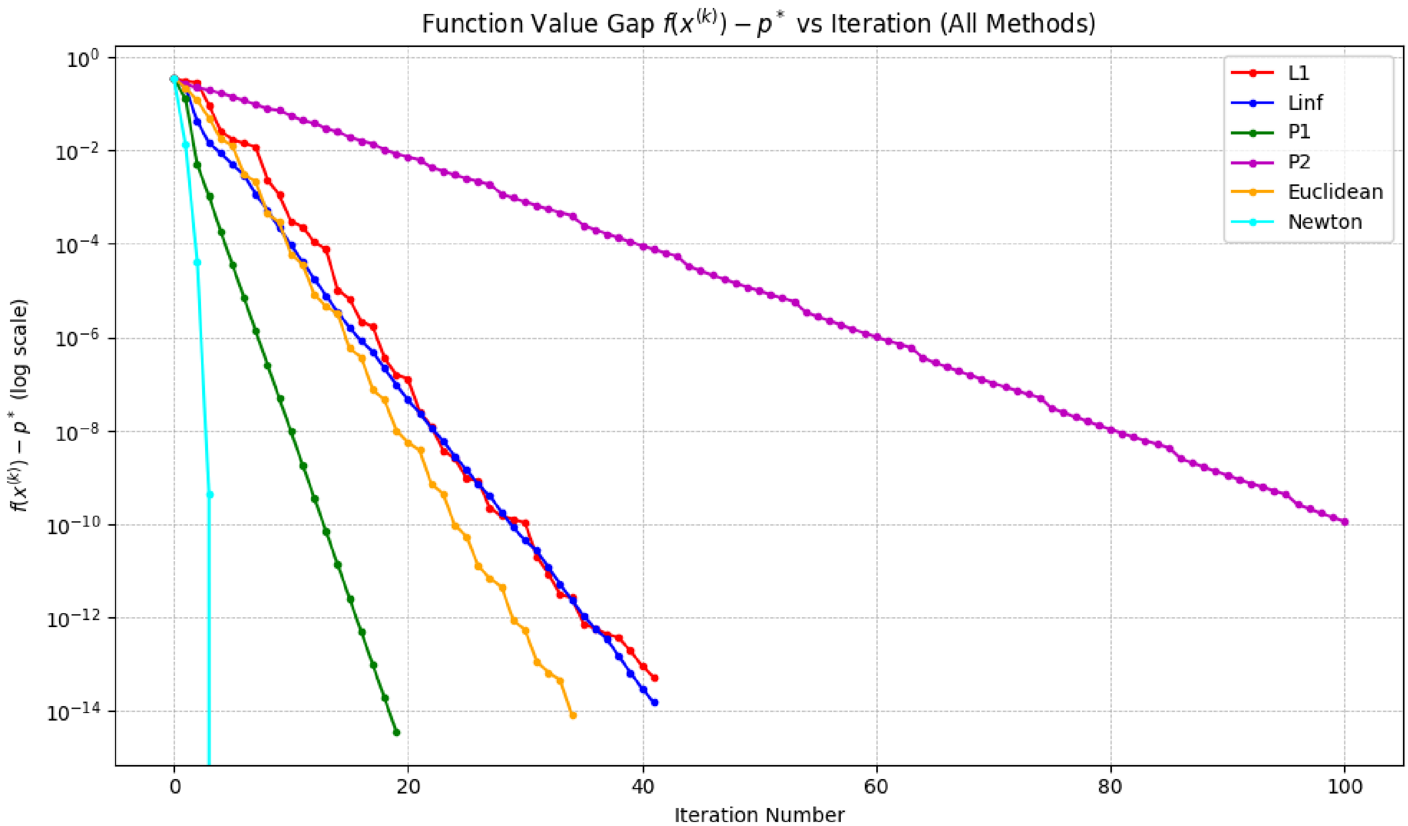


Method 5 :Newton's method, using the exact Hessian of the function.





All Combined :



Method Summary: *(with initial point (0.1, 0.1)*

Method	#Iterations	Final f(x)
L1	41	2.559267
Linf	41	2.559267
P1	19	2.559267
P2	100	2.559267
Euclidean	34	2.559267
Newton	5	2.559267

Comments :

1. Best performing steepest descent variant: Here, steepest descent in a quadratic P-norm, with P1 matrix, performed best among the steepest descent variants.

Justification: This happens because the P1 matrix aligns well with the function geometry.

Here, the given objective function was defined:

$$f(x1, x2) = e^{x1 + 3x2 - 0.1} + e^{x1 - 3x2 - 0.1} + e^{-x1 - 0.1}$$

So, the Function has curvated steeper along x2 (3x2) than that of x1. The diagonal matrix P1 scales more aggressively along x2, which leads to faster convergence. (19 iterations)

2. Worst performing steepest descent variant: Here, steepest descent in a quadratic P-norm, with P2 matrix, performed worst among the steepest descent variants.

Justification: This happens because the P2 matrix aligns badly with the function geometry.

Just opposite as that of P1, P2 focus more on the x1 axis, which leads to small steps in critical x2 directions and finally slower convergence. (100 iterations)

3. Comparison of the performance of norm-based steepest descent methods with classical gradient descent and Newton's method:

Here, gradient descent performed better than (34 iterations) $\ell 1$ and $\ell\infty$ steepest descent (41 iterations each) but worse than P-norm variant with P1 matrix. This is because gradient descent follows the gradient direction but lacks curvature adaptation.

On the contrary, Newton's method using the exact Hessian of the function outperforms all of the other methods, converging in only 5 iterations. This happens because Newton's method has the privilege of using second-order information (Hessian) to adapt to the local curvature, achieving quadratic convergence near the optimum. The exact Hessian resolves the anisotropy of the level sets perfectly, resulting in a direct path to the minimum.

Conclusion :

1. **Best steepest descent:** P1-norm (adapts best to function geometry).
2. **Worst steepest descent:** P2-norm (adapts poorly to function geometry).
3. **Newton's method outperform all when Hessian computation is feasible.**