MVA Convex optimization 2024 - Homework 3

RABEH Samar

Utility functions

In [ ]:
```
import numpy as np


def phi(A, b, v):
    return -np.sum(np.log(b-A@v))


def grad_phi(A, b, v):
    return np.sum(A.T / (b-A@v), axis=1)

def check_strict_feasible(A, b, v):
    return np.all(A@v-b < 0)


def f_(Q, p, v):
    return v.T@Q@v+p.T@v


def grad_f(Q, p, v):
    return 2*(Q@v)+p


def hessian_f(Q, p, v):
    return 2*Q

def hessian_phi(A, b, v):
    return A.T@(np.diag(1./(A@v-b)))**2@A


def log_barr_f(Q, p, A, b, t, v):
    if not check_strict_feasible(A, b, v):
        return float("NaN")
    return t*f_(Q, p, v) - phi(A, b, v)


def grad_barr_f(Q, p, A, b, t, v):
    return t*grad_f(Q, p, v) + grad_phi(A, b, v)


def hess_barr_f(Q, p, A, b, t, v):
    return t*hessian_f(Q, p, v) + hessian_phi(A, b, v)
```

Question 2.1.

In [ ]:
```
def centering_step(Q, p, A, b, t, v0, eps, alpha, beta):
    v_seq = [v0]
```

```
    while True:
        grad = grad_barr_f(Q, p, A, b, t, v0)
        hess = hess_barr_f(Q, p, A, b, t, v0)
        step = - np.linalg.inv(hess) @ grad
        decrement_2 = grad.T @ np.linalg.inv(hess) @ grad
        if(decrement_2/2 <=eps):
            break;
        s = backtrack_line_search(v0, step, alpha, beta)
        v0 = v0+s*step
        v_seq.append(v0)
    return np.array(v_seq)


f0 = lambda v: f_(Q, p, v)
f = lambda v: log_barr_f(Q, p, A, b, t, v)
g = lambda v: grad_barr_f(Q, p, A, b, t, v)
def backtrack_line_search(x, step, alpha, beta):
    t=1
    while(f(x+t*step)>=f(x)+alpha*t*g(x).T@step):
        t = beta*t
    return t
```

Question 2.2.

In [ ]:
```
def barr_method(Q, p, A, b, v0, eps, mu, alpha, beta,t):
    m = A.shape[0]
    v_seq = [v0]
    precision = [m/t]
    f_values = []
    Newton_iter = []
    centering_path = [v0]

    while True:
        f_values.append(f0(v0))
        v_seq_newton = centering_step(Q, p, A, b, t, v0, eps, alpha, beta
        v0 = v_seq_newton[-1]
        v_seq.append(v0)
        precision.append(m/t)
        Newton_iter.append(len(v_seq))
        centering_path += [v0]*len(v_seq_newton)
        if (m/t < eps):
            best_f = f0(v0)
            break
        t = mu*t
    return v_seq, precision, f_values, best_f, Newton_iter, centering_pat
```

Question 3.

3.1. Data generation

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")

# Problem parameters
```

```
n = 50
d = 100
Lambda = 10

X = np.random.random_sample((n, d))
w0 = np.random.random_sample(d)
y = X.dot(w0) + np.random.normal(n)

# Dual parameters
Q = 1/2*np.eye(n)
p = y
A = np.vstack((X.T, -X.T))
b = np.array([Lambda]*2*d)

# Feasible starting point
v = np.zeros(n)
```

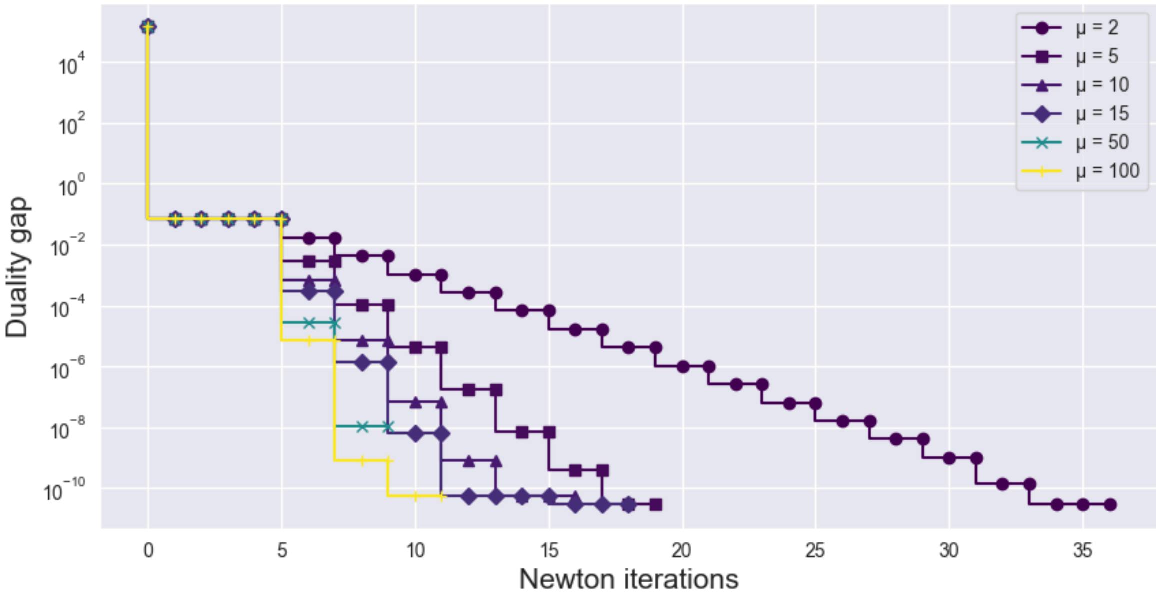
3.2. Parameters

In [ ]:
```
# Parameters of the algorithms
eps = 1e-6
alpha = 0.01
beta = 0.5
t = 1
# Values for test
mus = [2, 5, 10, 15, 50, 100]
```

3.3. Duality gap over iterations

In [21]:
```
import matplotlib.cm as cm


plt.figure(figsize=(10, 5))

labels = []
w_best = []
markers = ['o', 's', '^', 'D', 'x', '+']  # Different markers for each pl
cmap = cm.viridis  # Choose the colormap (viridis)

# Create a normalize object to map mu values to colormap
norm = plt.Normalize(vmin=min(mus), vmax=max(mus))

# Loop over different values of mu
for i, mu in enumerate(mus):
    v_seq, _, f_values, best_f, Newton_iter, centering = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta, t=1)

    # Compute the duality gap and delta
    duality_gap = np.abs(f_values - np.array(best_f))
    delta = np.array(list(map(f0, centering))) - best_f
    delta = delta[delta > 0]

    # Get a color from the viridis colormap
    color = cmap(norm(mu))

    # Plot the duality gap with a different marker and color for each mu
    plt.step(x=np.arange(0, len(delta)), y=list(delta), label="μ = {}".fo
```

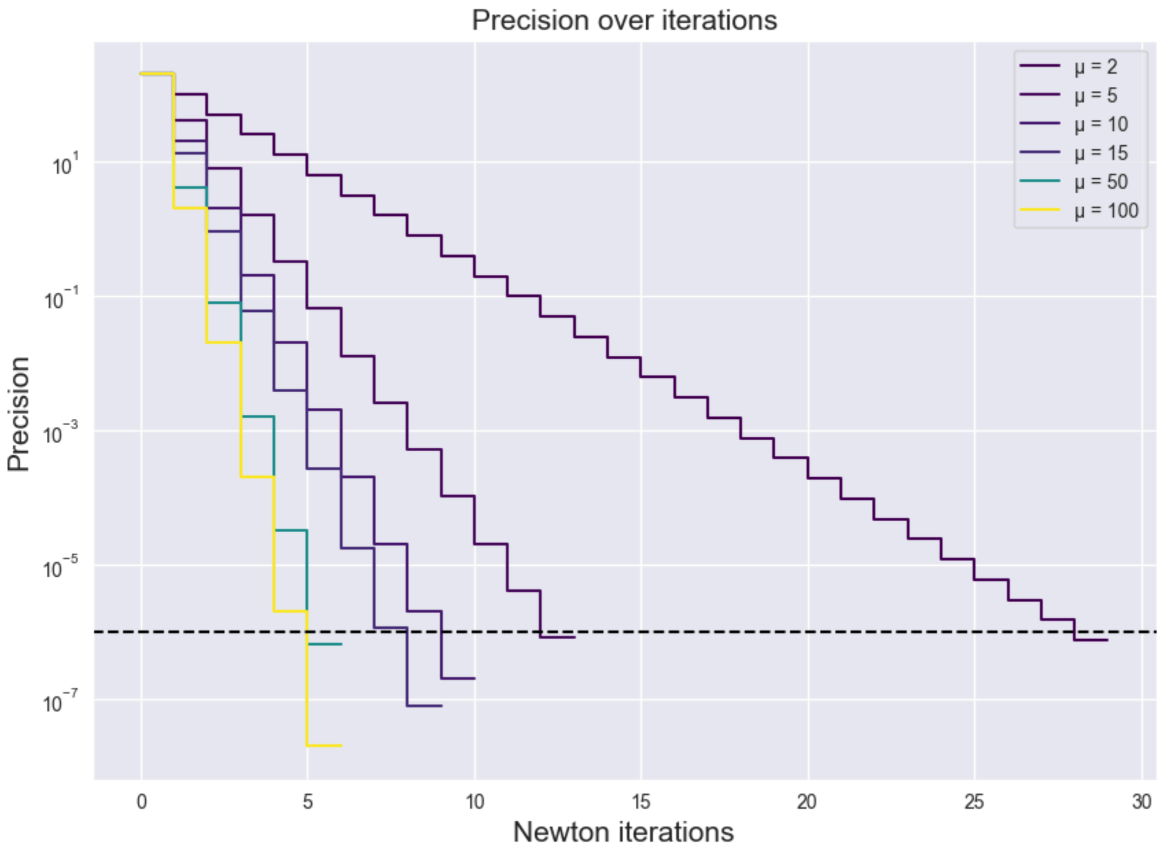
```
                marker=markers[i % len(markers)], linestyle='-', markersize=

        w_best.append(np.dot(np.linalg.pinv(X), v_seq[-1] + y))

# Set the scale for the y-axis to logarithmic
plt.yscale('log')

# Add legend, labels, and title
plt.legend()
plt.xlabel("Newton iterations", fontsize=15)
plt.ylabel("Duality gap", fontsize=15)

# Display the plot
plt.show()
```



3.4. Precision over iterations

In [22]:
```
import matplotlib.cm as cm

plt.figure(figsize=(10, 7))

labels = []
cmap = cm.viridis  # Choose the colormap (viridis)
norm = plt.Normalize(vmin=min(mus), vmax=max(mus))  # Normalize for color

# Loop over different values of mu
for i, mu in enumerate(mus):
    precision = barr_method(Q, p, A, b, v, eps, mu, alpha, beta, t=1)[1]

    # Get a color from the viridis colormap
    color = cmap(norm(mu))

    # Plot the precision with a unique color for each mu
    plt.step(range(len(precision)), precision, label="μ = {}".format(mu),

    labels.append(" $\mu$ = {}".format(mu))

# Set the scale for the y-axis to logarithmic
plt.yscale('log')
# Add legend, labels, and title
```

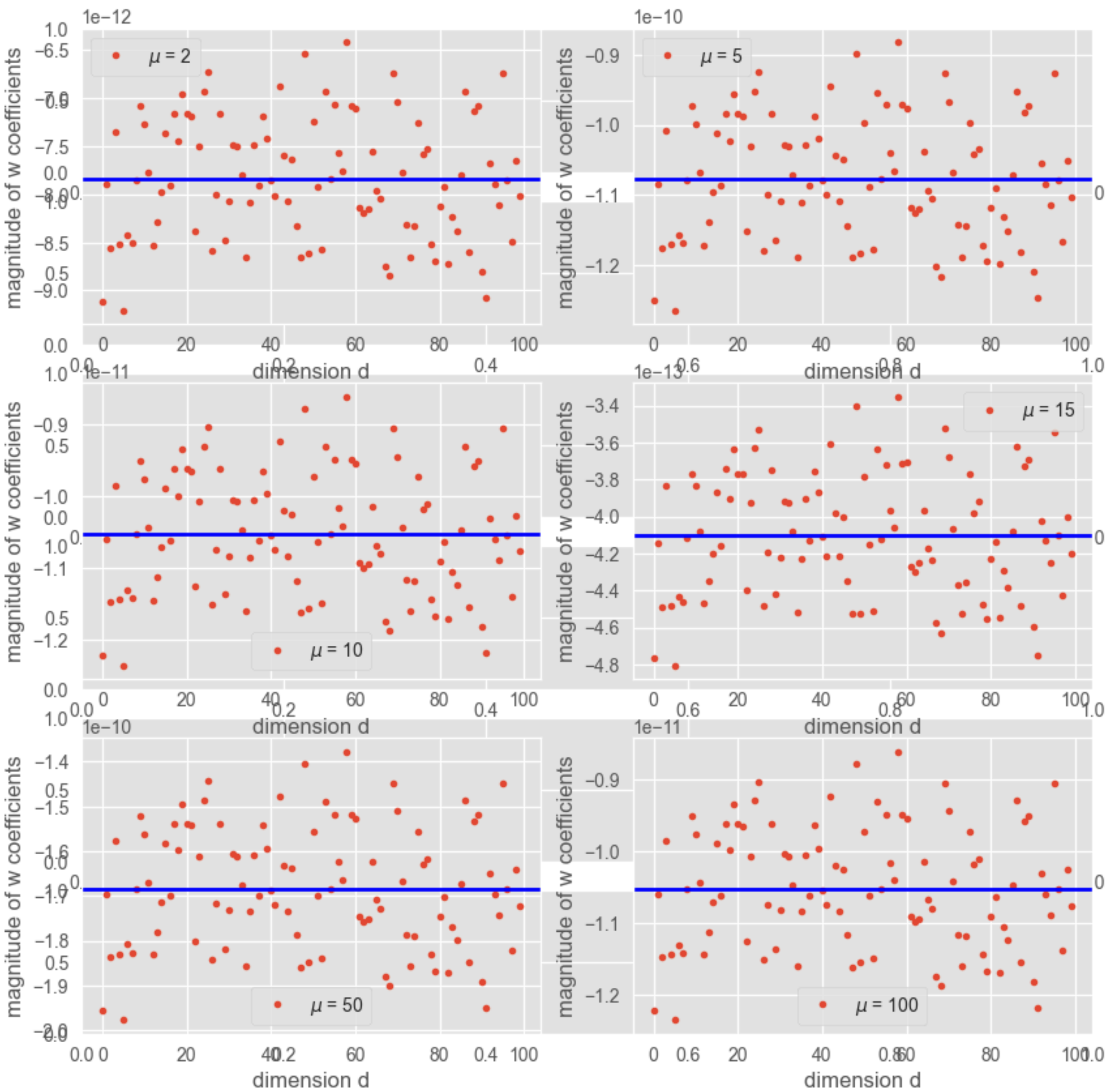
```
plt.legend()
plt.xlabel("Newton iterations", fontsize=15)
plt.ylabel("Precision", fontsize=15)
plt.title('Precision over iterations', fontsize=15)
# Add horizontal line for the epsilon value
plt.axhline(y=eps, c='black', linestyle='--')
# Display the plot
plt.show()
```



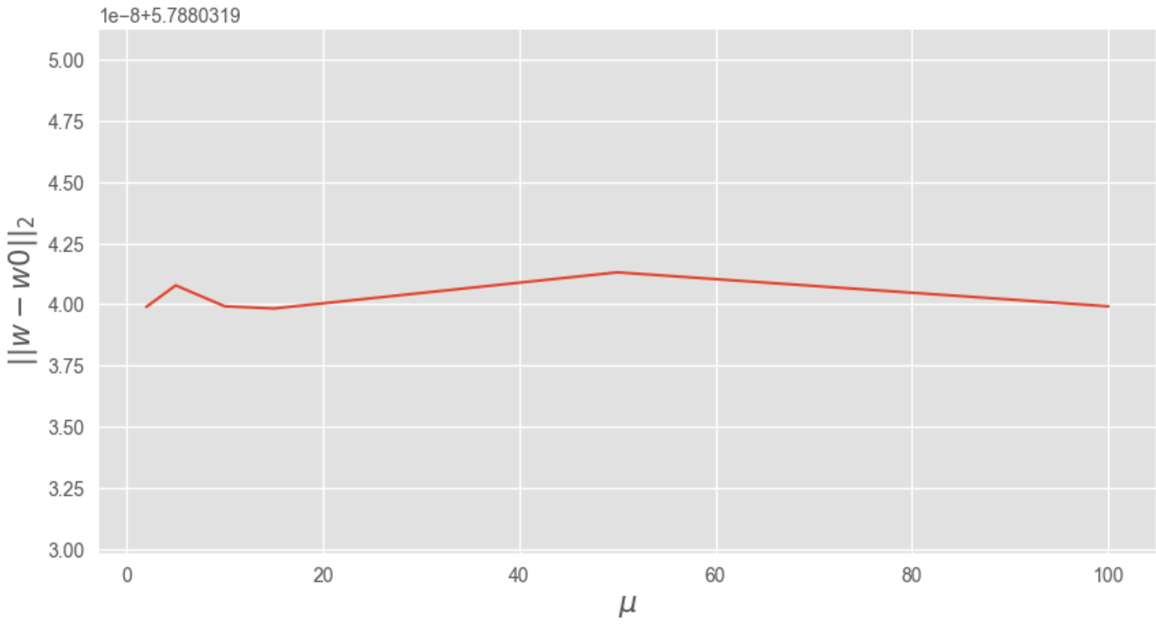
3.5. Impact of μ on w

In [28]:
```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import math

# Suppress warnings
warnings.simplefilter("ignore")

# Create subplots with the correct number of rows and columns
plt.subplots(len(mus), figsize=(10, 10))

# Loop over mus and create subplots for each
for i in range(len(mus)):
    ax = plt.subplot(math.ceil(len(mus) / 2), 2, i + 1)  # Ensure number
    #plt.suptitle('Influence of $\mu$ on magnitude of $w$')
    ax.plot(w_best[i], '.')
    plt.axhline(np.mean(w_best[i]), linestyle='-', color='blue', linewidt
    ax.legend(["$\mu$ = {}".format(mus[i])])
    ax.set_xlabel("dimension d")
    ax.set_ylabel("magnitude of w coefficients")
```

```python
# Display the plot
plt.show()
```



In [30]:
```python
plt.figure(figsize=(10, 5))

differences = []
for mu in mus:
    v_seq  = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta,t=1)[0]
    differences.append(np.linalg.norm(np.dot(np.linalg.pinv(X),v_seq[-1]

plt.plot(mus,differences)
plt.ylabel("$||w-w0||_2$",  fontsize=15)
plt.xlabel("$\mu$", fontsize=15)

plt.ylim((min(differences)-1e-8,max(differences)+1e-8));
```



3.6. Number of Newton iterations

In [29]:

```
plt.figure(figsize=(10, 5))
mu_s = [2,5,10,20,30,40,50,65,80,100,120,135,160,175,200]
iter_newton = []
for mu in mu_s:
    Newton_iter  = barr_method(
        Q, p, A, b, v, eps, mu, alpha, beta,t=1)[-1]
    iter_newton.append(sum(Newton_iter))

plt.plot(mu_s,iter_newton,marker='o')
plt.ylabel("Newton iterations",  fontsize=15)
plt.xlabel("$\mu$", fontsize=15)
```

Out[29]:

```
Text(0.5, 0, '$\\mu$')
```