Convex Optimization - Homework 3

Student : Théo Di Piazza

Date : November 2022

Course : Convex Optimization

Teacher : Alexandre d'Aspremont

Question 1

Let's consider the LASSO problem :

$$\min_w \frac{1}{2}||Xw - y||_2^2 + \lambda||w||_1$$

Derive the dual problem of LASSO and format it as a general Quadratic Problem as follows :

$$\min_v v^TQv + p^Tv$$
$$s.t.\ \ Av \le b$$

Answer:

To start, it's important to remark that the objective function of the LASSO problem is similar to the objective function of the Exercice 2 of the Homework 2.

Hence, we can wewrite the LASSO problem such as :

$$\min_w \frac{1}{2}||z||_2^2 + \lambda||w||_1$$
$$s.t.\ \ z = Xw - y$$

Now, the Lagrangian can be defined such as :

$$L(w, z, \mu) = \frac{1}{2}||z||_2^2 + \lambda||w||_1 + \mu^T(z + y - Xw)$$

with $\mu \in \mathbb{R}^n$.

Then, it comes that :

$$\begin{aligned} g(\mu) &= \inf_{w,y} \frac{1}{2}||z||_2^2 + \lambda||w||_1 + \mu^T(z + y - Xw) \\ &= \inf_y\ [\frac{1}{2}||z||_2^2 + \mu^Tz] + \inf_w\ [\lambda||w||_1 - \mu^TXw] + \mu^Ty \\ &= \inf_y\ [\frac{1}{2}||z||_2^2 + \mu^Tz] - \lambda \sup_w\ [(\frac{1}{\lambda}X^T\mu)^Tw - ||w||_1] + \mu^Ty \end{aligned}$$

Since $\frac{1}{2}||z||_2^2 + \mu^Tz$ is convex and 1-differentiable, we can show that :

$$\nabla_z(\frac{1}{2}||z||_2^2 + \mu^Tz) = z + \mu$$

Hence, z^* solution of $z^* + \mu = 0$ if $z^* = -\mu$

Moreover, we can use result of Exercice 2 of Homework 2 to show that :
[Math Processing Error]

So finally, the dual of the LASSO problem can be written such as :

$$\max_\mu\ \ -\frac{1}{2}||\mu||_2^2 + \mu^Ty$$
$$s.t.\ ||\frac{1}{\lambda}X^T\mu||_\infty > 1$$

Where the constraint can be rewritten such as :

$$||\frac{1}{\lambda}X^T\mu||_\infty > 1$$

$$\iff \forall i \in \{1,..,n\} : (\frac{1}{\lambda}X^T\mu)_i \le 1 \text{ and } (\frac{1}{\lambda}X^T\mu)_i \ge -1$$

[Math Processing Error] where $1_{2d} \in \mathbb{R}^{2d}$ such as $\forall i \in \{1,..,2d\} : (1_{2d})_i = 1$

So finally, the dual of the LASSO problem can be written such as :

$$\min_\mu\ \frac{1}{2}\mu^T\mu - \mu^Ty$$

[Math Processing Error]

which corresponds to the quadratic form.

Question 2

In the following parts, the objective is to implement the **Barrier method** to solve (QP).

Inside the **Barrier method**, it's needed to implement the **Newton method** to solve the **centering step**.

Then, before implementing these methods, it's important to define the function which will be minimized in the Newton method and its gradient and its Hessian.

With notations of the course (5_BarrierMethod, Slides 13), it can be defined :

$$g_t(v) = t(v^tQv + p^Tv) - \sum_{i=1}^{2d} log(b_i - a_i^Tv)$$

where A =
[Math Processing Error]

with $\forall i \in \{1,..,2d\}, a_i \in \mathbb{R}^n$

Hence, it comes that :

$$\nabla_v g_t(v) = 2tQv + tp + \sum_{i=1}^{2d} a_i(b_i - a_i^Tv)^{-1}$$

And also, it comes that :

$$\nabla_v^2 g_t(v) = 2tQ + \sum_{i=1}^{2d} \frac{a_ia_i^T}{(b_i - a_i^Tv)^2}$$

Now, to start, let's implement the function **centering_step**.

In [11]:

```
# Import libraries
import warnings
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# Hide warnings
warnings.filterwarnings('ignore')
warnings.simplefilter('ignore')

np.random.seed(2012)
```

Below are some functions to make coding easier.

In [12]:

```
# Compute value of g at v
def value_g(Q, p, A, b, t, v):
  '''
    Value function of g at v with parameters of the problems.
    Inputs:
      Q, p, A, b (np.arrays): parameters of the problems.
      v (np.array): where the function is evaluated.
      t (int): parameter of the method.
    Returns:
    value (np.array): value of the function g at v.
  '''

  value = t*(v.T[np.newaxis] @ Q @ v + p.T[np.newaxis] @ v) - np.sum(np.log(b - A @ v))

  return value[0]

# Compute value of dual (QP) at v
def value_dual(Q, p, v):
  '''
    Value of dual (QP) at v.
    Inputs:
      Q, p (np.arrays): parameters of the problem.
      v (np.array): where the dual is evaluated.
    Returns:
      the value associated.
  '''

  return v.T[np.newaxis] @ Q @ v + p.T[np.newaxis] @ v

# Compute Gradiant of g at v
def gradiant_g(Q, p, A, b, t, v):
  '''
    From given Q, p, A, b, v, t of the problem, returns the gradiant of g at v.
    Inputs:
      Q, p, A, b, v (np.arrays): parameters of the dual defined in Question 1.
      t: barrier method parameter.
    Returns:
      gradiant_g (np.array): gradiant of g.
  '''

  gradiant_g = 2*t*(Q @ v) + t*p + np.sum( A * np.reciprocal(b - A @ v)[:, np.newaxis], axis=0)

  return gradiant_g

# Compute Hessian of g at v
def hessian_g(Q, A, b, t, v):
  '''
    From given Q, A, b, v, t of the problem, returns the hessian of g at v.
    Inputs:
      Q, p, A, b, v (np.arrays): parameters of the dual defined in Question 1.
      t: barrier method parameter.
    Returns:
      hessian_g (np.array): hessian of g.
  '''

  # Calculation of the sum apart to facilitate the code
  sum_term = pow(b - A @ v, 2)
  sum = A[0][np.newaxis].T @ A[0][np.newaxis] / sum_term[0]
  # For each row of A, compute the a @ a.T and sum
  for i in range(1, 2*d):
    sum += A[i][np.newaxis].T @ A[i][np.newaxis] / sum_term[i]

  hessian_g = 2*t*Q + sum

  return hessian_g
```

Below is the implementation of line search.

In [13]:

```
def line_search(v, dv, alpha=.5, beta=.9):
  '''
    Implements backtracking line search for the Newton method.
    Inputs:
      v, dv (np.arrays): points to evaluate the function, with delta.
      alpha, beta (floats): parameters of the backtracking line search.
    Returns:
      t (float): step to use.
    '''

  # Initialize the stopping criterium, t
  stopping_criterium = False
  t = 1

  while((not stopping_criterium) and (t>1e-6)):

    objective_v = value_g(Q, p, A, b, t, v) # Value of g at v
    objective_dv = value_g(Q, p, A, b, t, v+t*dv) # Value of g at v+dv
    gradient_v = gradiant_g(Q, p, A, b, t, v) # Value of the gradiant of g at v

    # Compute stopping criterium
    criterion = objective_v + alpha*t*(gradient_v.T @ dv)
    stopping_criterium = (objective_dv < criterion)


    # Update t
    t = beta*t

  return t
```

Below is the implementation of centering step method.

In [14]:

```
def centering_step(Q, p, A, b, t, v0, eps, max_iter=500):
  '''
    Implements the Newton method to solve the centering step.
    Inputs:
      Q, p, A, b (np.arrays): parameters of the dual defined in Question 1.
      t: barrier method parameter.
      v0: initial variable.
      eps: precision
    Returns:
      v (np.array): sequence of variable iterates.
      nb_iter (int): number of iterations.
  '''

  nb_iter = 0 # number of iteration

  # Initialize the stopping criterium, v and v_n
  stopping_criterium = False
  v = v0.copy() # Current v in the loop
  v_n = [v0] # Sequence of all v

  i = 0 # Count iteration
  while((not stopping_criterium) & (i<max_iter)):

    i +=1

    # Compute the Newton step
    grad_g, hess_g = gradiant_g(Q, p, A, b, t, v), hessian_g(Q, A, b, t, v)
    delta_v = np.linalg.pinv(hess_g) @ grad_g
    # Update t, then v
    step = line_search(v, delta_v)
    v = v-step*delta_v
    v_n.append(v)
    # Compute the decrement
    lambda2 = grad_g.T @ delta_v

    # Update stopping_criterium
    stopping_criterium = (lambda2/2 <= eps)

    nb_iter += 1 # incremente


  return v_n, nb_iter
```

Below is the implementation of the Barrier method.

In [15]:

```
def barr_method(Q, p, A, b, v0, eps, mu=2, t=1, max_iter=500):
  '''
    Implements the Barrier Method.
    Inputs:
      Q, p, A, b (np.arrays): parameters of the dual defined in Question 1.
      v0: initial variable.
      eps: precision
    Returns:
      v (np.array): sequence of variable iterates.
      nb_iter (int): number of iterations.
  '''

  nb_iter = 0

  # Initialize the stopping criterium, v and v_n
  stopping_criterium = False
  v = v0.copy() # Current v in the loop
  v_n = [] # Sequence of all v

  m = A.shape[0]

  i = 0 # Count iteration
  while((not stopping_criterium) & (i<max_iter)):

    i += 1
    v_n.append(v)

    # Centering step, and update v
    v_all, nb_iter_centerStep = centering_step(Q, p, A, b, t, v_n[-1], eps)
    v = v_all[-1]

    # Update stopping_criterium
    stopping_criterium = (m/t < eps)

    # Increase t
    t = mu*t

    nb_iter += nb_iter_centerStep

  return v_n, nb_iter
```

Question 3

Now that **centering_step** and **barr_method** are implemented, let's test these functions on randomly generated matrices X and observations y with $\lambda = 10$.

In [16]:

```
# np.random.seed(2022)

# Randomly initialize the problem
def get_problem(n=4, d=3, lmbda=10):
  '''
    Initializes and compute parameters of the problem.
    Inputs:
      n, d : dimensions of matrixes.
      mu, t: barrier method parameter
      eps: precision criterium
    Returns:
      X, y, Q, p, A, b, v0, eps
  '''
  # Initialize parameters
  v0 = np.zeros(n)
  eps = 0.01


  # Initialize matrixes
  X = np.random.rand(n, d)
  y = np.random.rand(n)

  # If needed, ake regression if you easily want to compare
  w = np.random.rand(d)
  y = X @ w

  # Compute Q, p, A, b
  Q = np.eye(n)/2
  p = -y.copy()
  A = np.concatenate([X.T, -X.T])
  b = lmbda*np.ones(2*d)

  return X, y, Q, p, A, b, v0, w
```

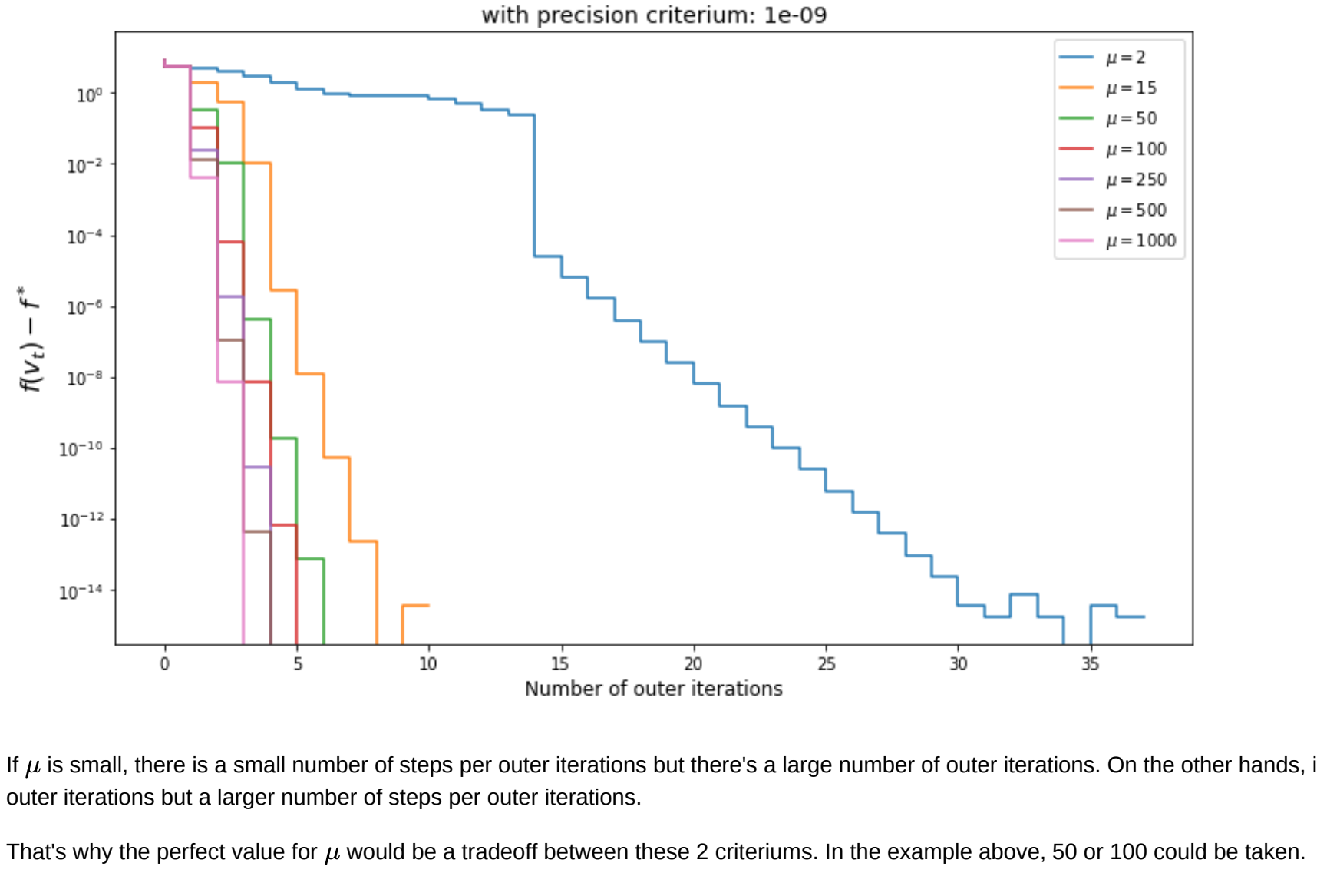
In the next cell, the calculation of the sequences for v is done for different values of μ : 2, 15, 50, 100, 250, 500 and 1000.

Then for each of the obtained sequences, the associated dual value is calculated.

Once all sequences are obtained for the different μ values tested, let's represent the gap $f(v) - f^*$ versus the number of iterations.

In [17]:

```
#np.random.seed(2022) # Fix seed

# Get parameters of the problem for the
n, d = 50, 50
X, y, Q, p, A, b, v0, w = get_problem(n, d)
eps = 1e-9

mu_to_try = [2, 15, 50, 100, 250, 500, 1000] # Different values to try for mu

# Plot GAP vs ITERATIONS for each mu
nb_iterations_list, objective = [], []

plt.figure(figsize=(12,7))

# For each value of mu to try
for mu in mu_to_try :

  # Get sequences v and number of iterations - Barrier Method
  v_list , num_iter = barr_method(Q, p, A, b, v0, eps=eps, mu=mu, t=1)
  nb_iterations_list.append(num_iter)
  gap_list =[]

  # Find the 'best' sequence to compute the gap (not necessary)
  dv_min = np.infty
  best_v = v_list[0]
  for v in v_list:
    if(value_dual(Q, p, v) < dv_min):
      dv_min = value_dual(Q, p, v)
      best_v = v

  # For each sequence of all sequences
  for v in v_list:

    # Compute gap between the current sequence and the 'best one'
    current_gap = value_dual(Q, p, v) - value_dual(Q, p, best_v)
    gap_list.append(current_gap[0])

  # Plot the gap VS iteration
  nb_iter = range(len(gap_list))
  plt.step(nb_iter , gap_list)

# Figure parameters
plt.semilogy();
plt.xlabel('Number of outer iterations', fontsize=12); plt.ylabel('$f(v_t) - f^*$', fontsize=16);
plt.title(f'Gap $f(v_t) - f^*$ VERSUS $\\mu$\nwith precision criterium: {eps}', fontsize=14);
plt.legend(['$\mu = $' + str(x) for x in mu_to_try], loc='best');
```



If μ is small, there is a small number of steps per outer iterations but there's a large number of outer iterations. On the other hands, if μ is large, there's less outer iterations but a larger number of steps per outer iterations.

That's why the perfect value for μ would be a tradeoff between these 2 criteriums. In the example above, 50 or 100 could be taken.

Finally, it is possible to study what would be the impact of μ on w.

Indeed, thanks to the KKT conditions applicable to the problem (LASSO), it comes that : $w = X^{-1}(y - v^*)$.

Thus, a dictionary is created to calculate the distance between the estimates of w and the value of w.

In [18]:

```
w_list = []

# For each value of mu, compute an estimation of w
for current_mu in mu_to_try:

  # Get v from Barrier Method for each mu to try
  v_list, _ = barr_method(Q, p, A, b, v0, eps=eps, mu=current_mu, t=1)
  # Estimate w
  w_current = np.dot(np.linalg.pinv(X), y - v_list[-1])
  w_current = w_current / np.linalg.norm(w_current)
  w_list.append(w_current)

distances = []
w /= np.linalg.norm(w)

# For each estimation of w
for w_current in w_list:

  distances.append(np.linalg.norm(w_current - w))

# Display the result
print(f'Distances between estimations of w and w: \n')
dict_result = dict(zip(mu_to_try, distances))
dict_result
```

```
Distances between estimations of w and w:
```

Out[18]:

```
{2: 1.7887106331804603,
 15: 1.7887104010557133,
 50: 1.7887107317243158,
 100: 1.7887107115662584,
 250: 1.7887107184970916,
 500: 1.7887107438532472,
 1000: 1.7887108203138113}
```

Finally, a dictionary is created to calculate the distance between each estimate of w.

In [19]:

```
w_list = []

# For each value of mu, compute an estimation of w
for current_mu in mu_to_try:

  # Get v from Barrier Method for each mu to try
  v_list, _ = barr_method(Q, p, A, b, v0, eps=eps, mu=current_mu, t=1)
  # Estimate w
  w_current = np.dot(np.linalg.pinv(X), y - v_list[-1])
  w_current = w_current / np.linalg.norm(w_current)
  w_list.append(w_current)

# Compute distances between estimations of w
distances_w = {} # dict{mu1}{mu2}: distance between estimation of w with mu=mu1 and mu=mu2

# For each values of (mu1, mu2)
for i in range(len(mu_to_try)):
  distances_w[str(mu_to_try[i])] = {}
  for j in range(len(mu_to_try)):
    # Compute distance
    distances_w[str(mu_to_try[i])][str(mu_to_try[j])] = round(np.linalg.norm(w_list[i] - w_list[j]), 8)

# Display the result
print(f'Distances between each estimation of w: \n{distances_w}')
```

```
Distances between each estimation of w:
{'2': {'2': 0.0, '15': 1.479e-05, '50': 5.9e-06, '100': 5.35e-06, '250': 7.21e-06, '500': 4.52e-06, '1000': 6.26e-0
6}, '15': {'2': 1.479e-05, '15': 0.0, '50': 1.094e-05, '100': 1.128e-05, '250': 9.91e-06, '500': 1.791e-05, '1000':
1.04e-05}, '50': {'2': 5.9e-06, '15': 1.094e-05, '50': 0.0, '100': 1.51e-06, '250': 1.67e-06, '500': 8.12e-06, '100
0': 1.18e-06}, '100': {'2': 5.35e-06, '15': 1.128e-05, '50': 1.51e-06, '100': 0.0, '250': 2.25e-06, '500': 7.6e-06,
'1000': 1.26e-06}, '250': {'2': 7.21e-06, '15': 9.91e-06, '50': 1.67e-06, '100': 2.25e-06, '250': 0.0, '500': 9.51e-0
6, '1000': 1.17e-06}, '500': {'2': 4.52e-06, '15': 1.791e-05, '50': 8.12e-06, '100': 7.6e-06, '250': 9.51e-06, '500':
0.0, '1000': 8.62e-06}, '1000': {'2': 6.26e-06, '15': 1.04e-05, '50': 1.18e-06, '100': 1.26e-06, '250': 1.17e-06, '50
0': 8.62e-06, '1000': 0.0}}
```

End of HW3.

Thank you for reading !

Théo Di Piazza.

In [21]:

```
# Connect to google colab to save the script as HTML as the end of the execution
from google.colab import drive
drive.mount('/content/drive')

# Command to save the file as HTML
! jupyter nbconvert --to html /content/drive/MyDrive/ENS_MVA/Convex_Optimization/CVX_HW3_DIPIAZZA_Theo
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_rem
ount=True).
[NbConvertApp] Converting notebook /content/drive/MyDrive/ENS_MVA/Convex_Optimization/CVX_HW3_DIPIAZZA_Theo to html
[NbConvertApp] Writing 355440 bytes to /content/drive/MyDrive/ENS_MVA/Convex_Optimization/CVX_HW3_DIPIAZZA_The.html
```