27Interior point methods for different $\mu$ with $\epsilon = 10^{-8}$ and with gradient-based backtracking-figure.caption.6

# Convex optimization and application for machine learning

## Encadrants :

-Alexandre d'Aspremont

## 1 Introduction

This report concerns the third homework related to the course convex optimization and application to machine learning. It deals with the application of a particular *interior points* method : the barrier method. The interior point methods are recent methods coined to deals with the minimisation of objective functions subject to convex inequality constraints. Indeed, affine constraints are the only one convex, and it is well known how to take them into account in the minimization algorithm ( i.e Newton's method for affine equality constraint). A barrier method consists in removing the inequality constraints with the help of a barrier function. This function has to diverge when x is approaching the unfeasible domain $\mathcal{D} = \{x | f_i(x) \leq 0 \quad \forall \quad i \in \{1, \cdots m\}\}$. Such a procedure can be applied to problems of the form :

$$\begin{array}{ll} \min\limits_{x} & f_0(x) \\ \text{s.t.} & Ax = b \\ & f_i(x) \leq 0 \quad \forall \text{i} \in \{1, \cdots m\} \end{array} \tag{1}$$

where $f_0$ and $(f_i)_i$ are convex. The barrier method establish then the following non-equivalent form :

$$\begin{array}{ll} \min\limits_{x} & tf_0(x) + \phi(x) \\ \text{s.t.} & Ax = b \end{array} \tag{2}$$

where $\phi$ is the barrier playing the role of a penalisation. As stated sooner, one have to impose $\phi$ the following property : $\phi(x) = +\infty$ if $x \notin \mathcal{D}$. The barrier method only introduces two new parameters: $t$ and $\mu$. $t$ will control how close we wish to be from the ideal solution. Indeed, if we increase $t$, we observe that the original objective function $f_0$ becomes the priority to minimize since we may have for a certain t big enough $tf_0 \gg \phi$. However, it will hold only in $\mathcal{D}$, yet imposing the inequality constraints. $\mu$ represents the step length we are taking every step to increase our previous calculated solution i.e at step $(n+1)$, $t^{(n+1)} = \mu t^n$. These two parameters are key, and will be crucial for this homework.

The hardest part in the method is not the running of the algorithm itself (a succession of centering step via Newton's method), but the initialisation of its parameters. There is not only one initial condition, but two : $x^{(0)}$ and $t^{(0)}$). The initial point $x^{(0)}$ has to be strictly feasible i.e $f_i(x^{(0)}) < 0 \quad \forall \quad i \in \{1, \cdots, m\}$, which is not always possible. The setting of initialisation is known as phase I, and it requires its own treatment like for example another barrier method.

The main advantage of this method is that even though the problems are not strictly equivalent, it can give pretty good approximations of the real solution $p*$, with a controlled computational error : $f_0(x^*(t)) - p* \leq \frac{m}{t}$, at least in the convex case. Also, the computational cost is even controlled in certain case. It can indeed to be shown that $\#steps \approx \dfrac{\log(m/\epsilon t^{(0)})}{\log(\mu)}$ where $\epsilon$ is the desired accuracy. , as well as the number of Newton's method iteration in each centering step ($\approx 40$ iterations), which is a very important property. I will not detail further the functioning of the barrier method.

In this report, we will apply it to the LASSO (Least Absolute Shrinkage Operator and Selection Operator) objective, which is a classic least-norm square objective granted with a regularization term, in order to impose a certain regularity to the solution. Indeed, the $l_1$ norm term will tend to have the effect to produce sparsity in $w$.

# 2 Formulation of the problem

The LASSO optimization problem is formulated this way :

$$\min_{w \in \mathbf{R}^d} \quad \tfrac{1}{2}||Xw - y||_2^2 + \lambda ||w||_1 \tag{LASSO}$$

where $X \in \mathcal{M}_{n \times d}(\mathbf{R})$, $w \in \mathbf{R}^d$, $y \in \mathbf{R}^n$, $\lambda > 0$. $||\cdot||$ is the $l_1$ norm, defined as $||w||_1 = \sum\limits_{i=1}^{d} |w_i|$. Let's first notice this problem is convex. The domain is trivially convex and so is the objective function, as the sum of two convex functions. Moreover, we can assure a solution exists since the objective function is coercive and it is lower semi-continuous. It is trivial to see that $||Xw - y||_2 \longrightarrow +\infty$ and $||w||_1 \longrightarrow +\infty$ when $||w||_2 \longrightarrow +\infty$ This form is not appropriate to solve via the barrier method, so we will compute its dual and put it under the general form of Quadratic Program (QP) :

$$\begin{array}{ll} \min\limits_{v \in \mathbf{R}^n} & \tfrac{1}{2} v^T Q v + p^T v + r \\ \text{s.t.} & Av = b \\ & Gv \preceq h \end{array} \tag{QP}$$

To do that, let's add an artificial variable to our problem (LASSO) by setting $z = Xw - y$. We then have two variables, and we can express it with this form :

$$\begin{array}{ll} \min\limits_{(z,w)\in\mathbf{R}^n\times\mathbf{R}^d} & \frac{1}{2}||z||_2^2 + \lambda||w||_1 \\ \text{s.t.} & z = Xw - y \end{array} \tag{LASSO'}$$

We introduce the lagrangian $\mathcal{L}(z,w,v) = \frac{1}{2}||z||_2^2 + \lambda||w||_1 + v^T(z - Xw + y)$ with $v \in \mathbf{R}^n$ the lagrange multiplier. By taking the $\inf\limits_{z,w} \sup\limits_{v\in\mathbf{R}^n}$, one find back the original problem (LASSO'). To find the dual problem, we exchange the sup and inf, and we first have to compute :

$$\inf_{(z,w)\in\mathbf{R}^{n\times d}} \mathcal{L} = (z,w,v) = \inf_{(z,w)\in\mathbf{R}^{n\times d}} \frac{1}{2}||z||_2^2 + \lambda||w||_1 + v^T(z - Xw + y)$$

We can split the variables $z$ and $w$, so we have next : $\mathcal{L}(z,w,v) = \frac{1}{2}||z||_2^2 + \lambda||w||_1 + v^T(z - Xw + y) + \inf\limits_{z\in\mathbf{R}^n} v^T y + \frac{1}{2}||z||_2^2 + v^T z + \inf\limits_{w\in\mathbf{R}^d} \lambda||w||_1 - v^T Xw$.

- $\inf\limits_{z\in\mathbf{R}^n} v^T z + \frac{1}{2}||z||_2^2$ is calculated easily by setting its gradient to 0. We have immediately $z^* = -v$ and so $\inf\limits_{z\in\mathbf{R}^n} \frac{1}{2}||z||_2^2 + v^T z = -\frac{||v||_2^2}{2}$
- $\inf\limits_{w\in\mathbf{R}^d} \lambda||w||_1 - v^T Xw = -\lambda \sup\limits_{w\in\mathbf{R}^d} \frac{v^T Xw}{\lambda} - ||w||_1 = -f^*(\frac{X^T v}{\lambda})$ where f is the $l_1$ norm function and $f^*$ its Legendre-Fenchel conjugate. It is easy to see that

  $$f^*(y) = \begin{cases} 0 \text{ if } ||y||_\infty \leq 1 \\ +\infty \text{ otherwise} \end{cases}$$

  A condition here is appearing : $\frac{X^T v}{\lambda} \in \text{dom}(g) = \{y \in \mathbf{R}^n |\ ||y||_\infty \leq 1\}$ where $g$ is obviously $g(v) = \inf\limits_{(z,w)\in\mathbf{R}^{n\times d}} \mathcal{L}(z,w,v)$.

Eventually, we have $\inf\limits_{(z,w)\in\mathbf{R}^{n\times d}} \mathcal{L}(z,w,v) = v^T y - f^*(X^T w) - \frac{||v||_2^2}{2} = v^T y - \frac{||v||_2^2}{2}$ if $v \in \text{dom}(g)$

We can finally present the dual problem :

$$\begin{array}{ll} \max\limits_{v\in\mathbf{R}^n} & v^T y - \frac{||v||_2^2}{2} \\ \text{s.t.} & X^T v \leq \lambda e \\ & -X^T v \leq \lambda e \end{array}$$

where $e_d = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathbf{R}^d$. By setting $Q = I_n$, $p = -y$, $F = \begin{pmatrix} X^T \\ -X^T \end{pmatrix}$, $h = \lambda e_{2d}$, $A = 0$ and $b = 0$, we find back the (QP) formulation, except for the minus sign in front of the inf. We implement the barrier method with the log barrier

$$\phi(x) = -\sum_{i=1}^{2d} \log(\lambda - f_i^T x)$$

, with $f_i \in \mathbf{R}^n$ are the row of the F. Therefore, the point is to solve the problem:

$$\min_{v \in \mathbf{R}^n} \quad \frac{||v||_2^2}{2} - v^T y + \phi(v) \qquad \text{(Interior Point)}$$

To solve this minimization problem, I will use the classical Newton's method, described in the next section. We can one more time state that the problem is convex, there is no equality/inequality constraint, the feasible set is (trivially) convex, and the objective function is convex as the sum of two convex function ($Q$ is definite positive). The existence is also assured as the objective function is coercive and lower semi-continuous.

# 3 Newton's method

The point of Newton's method is to resolve the following equation $F(x) = 0$, where $F : \mathbf{R}^n \longrightarrow \mathbf{R}^m$. The Newton's method is part of a more general class of method : the gradient descent methods. First order conditions for (global in this case) minimum $x^*$ is the well known Peano-Kantorovitch condition

$$\nabla f_t(x^*) \in (T_{x^*}X)^+$$

or rather $\nabla f_t(x^*) = 0$ in this unconstrained case. Therefore, we are going to apply Newton's method on $F = \nabla f$. So, starting from a certain point $x^{(n)}$, we want to compute a new point $x^{(n+1)}$ verifying the property : $f(x^{(n+1)}) \leq f(x^{(n)})$. The Newton's method propose the following point $x^{(n+1)} = x^{(n)} - \alpha^{(n)}(\nabla^2 f(x))^{-1}\nabla f(x)$, more precisely the descent direction $\Delta x = -(\nabla^2 f(x))^{-1}\nabla f(x)$. This method has very good convergence properties. Indeed, it is theoretically proved that the method has a q-quadratic rate of convergence, at least locally. Concerning the step size $\alpha^{(k)}$, it takes a different value at every step. Intuitively, it cannot be too big, in order to not jump on too many minimum, risking thus to miss the global one or creating a oscillatory behavior. And not too small either, one do not want to remain trapped within a local minimum. There is no local minimum in this case because of convexity, but if it is too slow, the desired precision will require too many iterations. In this work, I will use a backtracking line search technique known as Armijo's rule. Other rules exist such as Goldstein rule, Wolfe rule etc... It consists in choosing a coefficient $\alpha_k^{(n)}$ verifying

$$f(x_k + \alpha^{(k)}\Delta x_k) \leq f(x^{(k)}) - \omega_1 \alpha^{(k)} \nabla f(x)^T (\nabla^2 f(x))^{-1} \nabla f(x)$$

with $\omega_1 \in ]0, 1[$. To find it, we proceed iteratively and start with $\alpha_0^{(k)} = 1$. At step i, if $\alpha_i^{(k)}$ doesn't verify this equation, we try with $\alpha_{i+1}^{(k)} = \tau \alpha_i^{(k)}$, with $\tau \in ]0, 1/2[$. If it is not yet satisfied, we repeat the procedure. In my code, I have chosen the following parameters : $\omega = 10^{-4}$ and $\tau = 0.2$. I used another trick and add a counter, to prevent $\alpha$ from becoming too small, as I discussed above. Goldstein's rule is supposed to prevent that, but I didn't have enough time to implement it. Therefore, the maximum of $k$ is $k_{max} = 6$.
An interesting point would be to say we can also apply the backtracking linear search on $\nabla f$ and not $f$. Indeed, setting $\nabla f$ to 0 could also be seen as minimizing the criteria $||\nabla f||_2^2$. Replacing up there $f$ by $||\nabla f||_2^2$, and using the useful property of $\nabla^2 f \Delta x = -\nabla f$, the backtracking line search takes the nice form

$$||\nabla f(x_k + \alpha^{(k)}\Delta x_k)||_2^2 \leq ||\nabla f(x_k)||_2^2 (1 - \omega_1 \alpha^{(k)})$$

Let's now root all this to our (QP) problem. We set here $f_t(x) = t(\frac{||x||_2^2}{2} - x^T y) + \phi(x) = t(\frac{1}{2}x^T Q x + p^T x) + \phi(x)$. We here have immediately :

$$\nabla f_t(x) = t(Qx + p) + \sum_{i=1}^{2d} \frac{f_i}{\lambda - f_i^T x}$$

and

$$\nabla^2 f_t(x) = tQ + \sum_{i=1}^{2d} \frac{f_i f_i^T}{(f_i^T x - \lambda)^2}$$

As advised by the guideline, $\mu$ took the values $\{2, 15, 50, 100\}$. I have tried the two backtracking strategies and both gives results. However, the gradient approach seems far superior, and do not need any kind of counter. Eventually, even though the backtracking strategy can appear as the simplest step in the algorithm, it is actually the more subtle one, because a fine tuning of the parameter can be necessary.

To stop Newton's descent, we'll use the following criteria :

$$\lambda^2(x) = \Delta x \nabla^2 f(x) \Delta x \leq 2\epsilon$$

where $\epsilon$ is an arbitrary constant. It tells us how much one can descent towards this direction. It is also a local estimator of how close we are from the minimum :

$$f(x) - p^* = \frac{1}{2}\lambda^2(x)$$

It was not necessary to implement a phase I method with another barrier method. Indeed, there is no equality constraint, and $v_O = O_n$ is an obvious strictly feasible point. We will start from this point.

# 4 Results

First thing I noticed was that the gradient-based backtracking strategy was more efficient and allowed much smaller $\epsilon$. Therefore, I will show the results for both approach.

Without any contest possible, the second approach for backtracking is much more efficient, and is also much faster (less newton's method iteration to compute).

We observe also that in the classic case, the higher $\mu$ is not the better. On the contrary to the gradient-based approach, the higher $\mu$ performs the better, with around 10 iterations.

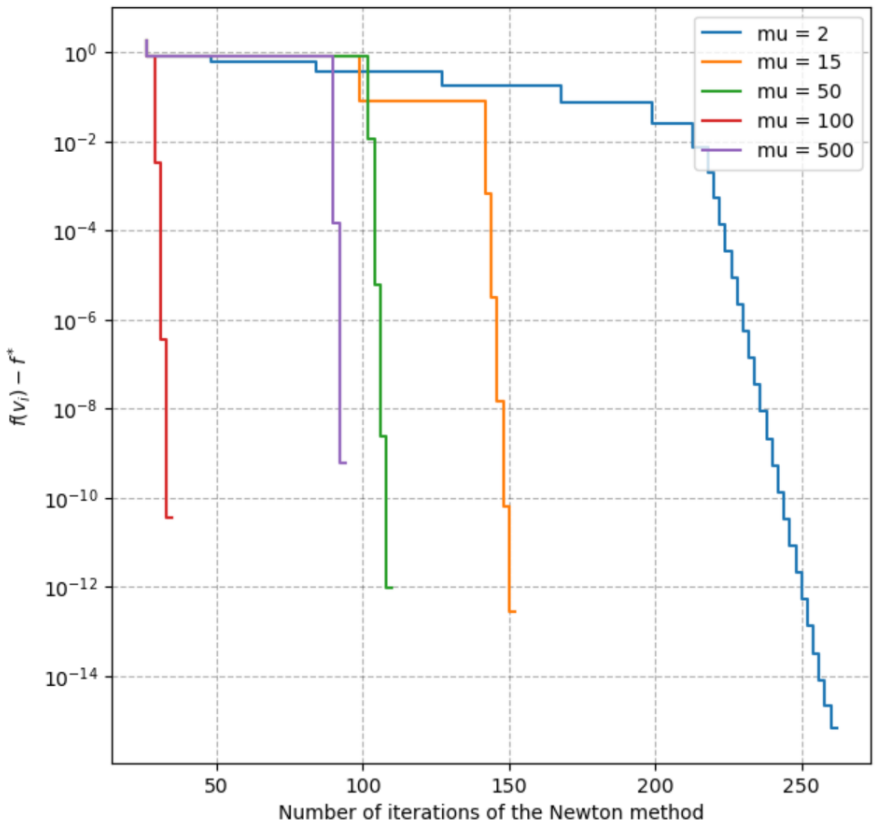



Figure 1: Interior point methods for different $\mu$ with $\epsilon = 10^{-8}$ and with classic backtracking

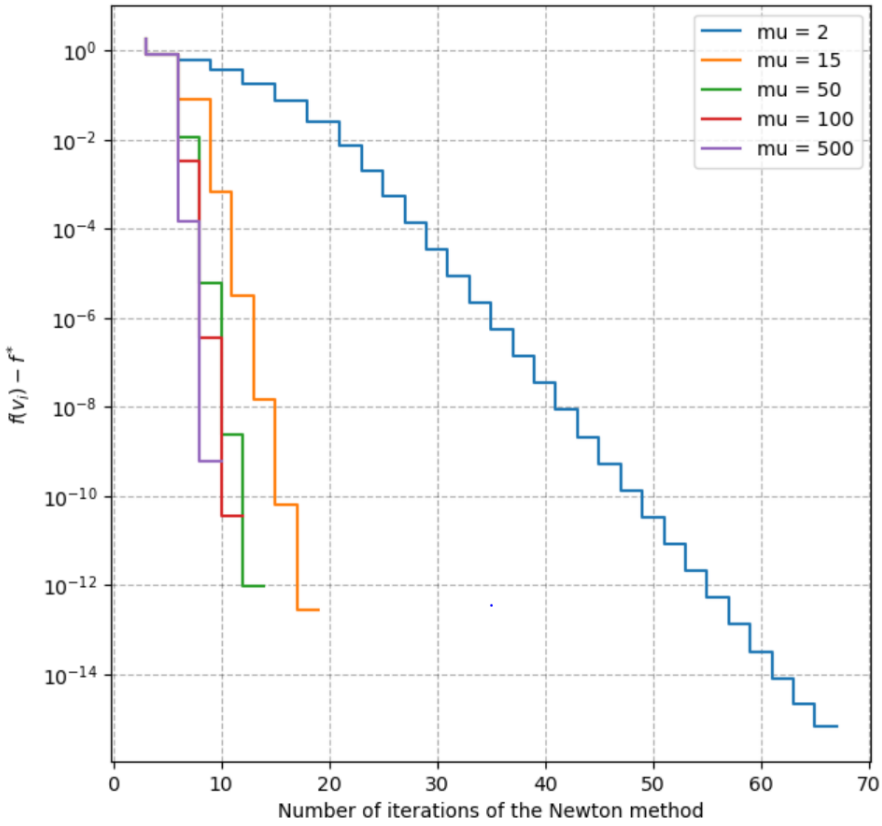



Figure 2: Interior point methods for different $\mu$ with $\epsilon = 10^{-8}$ and with gradient-based backtracking

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

## Introduction of the paramaters for the optimization problem

In [ ]:
```
#parameters of the problem
n = 10
d = 100
lambdad = 10

np.random.seed(43)
X = np.random.rand(n,d)
np.random.seed(12)
y = np.random.rand(n)
Q = np.eye(n)
e = np.array([1 for i in range(2*d)])
p = np.dot(y,-1)
F = np.concatenate((np.transpose(X), - np.transpose(X)), axis = 0)
b = np.dot(lambdad,e)
eps = 10e-8
t = 1
Mu = [2,15,50,100, 500]
```

We want to apply Newton's method on the following function :

$$f_0(x) = \frac{1}{2}x^TQx + p^Tx + \phi(x)$$

where

$$\phi(x) = -\sum_{i=1}^{2d}\log(\lambda - f_i^Tx)$$

We'll deal here in this case with no equality constraint, which simplify a bit the resolution. We have :

$$\nabla f_0(x) = Qx + p + \sum_{i=1}^{2d}\frac{f_i}{\lambda - f_i^Tx}$$

and

$$\nabla^2 f_0(x) = Q + \sum_{i=1}^{2d}\frac{f_if_i^T}{(f_i^Tx - \lambda)^2}$$

Computation of the hessian $\nabla^2 f_t$

In [ ]:
```
def hess_f_t(Q,p,A,b,x,t): #evaluate the hessian of f at x
    """
    return the hessian of the objective function at x

    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
    Returns:
        hessian of f_0(x) : (array), sized [n,n]
    """
    hess = Q*t
    for i in range(2*d):
        f_i = A[i] #ième ligne
        mat = np.array([[f_i[i]*f_i[j] for i in range(n)] for j in range(n)])
        hess += mat/(np.dot(f_i,x) - b[i])**2
    return hess
```

Computation of the jacobian $\nabla f_t$

In [ ]:
```
def grad_f_t(Q,p,A,b,x,t):
    """
    return the gradient of the objective function at x

    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
```

```
        Q: (array), sized [n,n]
        p: (array), sized [n,]
    Returns:
        f_0(x) : (array), sized [n,]
    """
    grad = t*(np.dot(Q,x) + p)
    for i in range(2*d):
        f_i = A[i]
        grad += f_i/(b[i] - np.dot(f_i,x))
    return grad
```

Computation of the log-barrier function $\phi$

In [ ]:
```
def check_feasible(A,b,x):
    for i in range(2*d):
        f_i = A[i]
        if (b[i] - np.dot(f_i,x) < 0):
            return False
    return True
```

In [ ]:
```
def log_barrier(A,b,x):
    """
    computation of the log_barrier function at x, return infinity if out of the domain
    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d, ]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
    Returns:
        phi(x) : (int)
    """
    phi = 0
    check = check_feasible(A,b,x)
    if check == False:
        return math.inf
    for i in range(2*d):
        f_i = A[i]
        phi -= np.log(b[i] - np.dot(f_i,x))
    return phi
```

Computation of the objective function $f_t$

In [ ]:
```
def f_t(Q,p,A,b,x,t):
    """
    return the objective function evaluated at x
    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
    Returns:
        f_0(x) : (int)
    """
    f = t*(1/2*np.dot(x,np.dot(Q,x)) + np.dot(p,x))
    phi = log_barrier(A,b,x)
    return f + int(phi)
```

In [ ]:
```
def f_0(Q,p,x):
    """
    return the objective function evaluated at x
    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
    Returns:
        f_0(x) : (int)
    """
    f = (1/2*np.dot(x,np.dot(Q,x)) + np.dot(p,x))
    return f
```

## Classic backtracking

Implementation of the backtracking, with $\omega_1 = 10^{-4}$ and $\tau = 0.2$.

In [ ]:
```
def backtracking(Q,p,A,b,x,t,dir):
    """
    return the right step validating Armijo's rule.
```

```
    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
        dir: (array) the descent direction at iteration k, sized [n,]

    Returns:
        alpha : (int) the step for gradient descent according to armijo's rule
    """
    k = 0
    w = 10e-4
    tau = 0.5
    alpha = 1

    grad_f = grad_f_t(Q,p,A,b,x,t)
    x_new = x + alpha*dir
    f = f_t(Q,p,A,b,x,t)
    diff = f_t(Q,p,A,b,x_new,t) - (f + w*alpha*np.dot(dir,grad_f))
    while (diff > 0 and k < 4):
        k += 1
        alpha *= tau
        x_new = x + alpha*dir
        diff = f_t(Q,p,A,b,x_new,t) - (f + w*alpha*np.dot(dir,grad_f))
      #  print('dir = ', dir, 'x = ', x, 'alpha = ', alpha ,'diff = ', diff )
       # print('diff = ', diff )
    return alpha
```

## gradient-based backtracking

In [ ]:
```
def backtracking_with_gradient(Q,p,A,b,x,t,dir):
    """
    return the right step validating Armijo's rule.

    Args :
        x: (array) the point at which we evaluate the function, sized [n,]
        t: (int) parameter of the barrier method
        b: (array) rhs of the linear inequality, sized [2d,]
        A: (array) lhs of the linear inequality, sized [2d,n]
        Q: (array), sized [n,n]
        p: (array), sized [n,]
        dir: (array) the descent direction at iteration k, sized [n,]

    Returns:
        alpha : (int) the step for gradient descent according to armijo's rule
    """
    k = 0
    w = 10e-4
    tau = 0.05
    alpha = 1

    grad_f = grad_f_t(Q,p,A,b,x,t)
    x_new = x + alpha*dir

    grad_f_new = grad_f_t(Q,p,A,b,x_new,t)
    diff = np.linalg.norm(grad_f_new)**2 - np.linalg.norm(grad_f)**2*(1 - w*alpha)
    while (diff > 0):
        k += 1
        alpha *= tau
        x_new = x + alpha*dir
        grad_f_new = grad_f_t(Q,p,A,b,x_new,t) #With new alpha
        diff = np.linalg.norm(grad_f_new)**2 - np.linalg.norm(grad_f)**(1 - w*alpha)
       # print('dir = ', dir, 'x = ', x, 'alpha = ', alpha ,'diff = ', diff )
    return alpha
```

Check whether the solution position is feasible i.e if

$$Fx \leq b$$

or more precisely if $\forall i \in \{1, 2d\}$

$$f_i^T x \leq \lambda$$

where $f_i$ is the $i^{th}$ row of F and F is defined as

$$F = \begin{pmatrix} X^T \\ -X^T \end{pmatrix}$$

In [ ]:
```
def test_symetry(M):
    for i in range(n):
        for j in range(n):
```

```
            if M[i,j] != M[j,i]:
                return False
    return True

def is_pos_def(x):
    return np.all(np.linalg.eigvals(x) > 0)

x = np.array([np.random.rand() for i in range(10)])
hess = hess_f_t(Q,p,F,b,x,t)
is_pos_def(hess)
```

Out[ ]:
```
True
```

## Centering step

This is the crucial step where we solve the minimization problem via Newton's method for a given t

In [ ]:
```
def centering_step(Q,p,A,b,t,v0,eps):
    """
    It computes the Newton method to solve the problem \grad f_t(x) = 0

    Args :
        Q: (array) positive definite matrix, sized [n,n]
        p: (array), sized [n,]
        F: (array), matrix of the inequality constraint Fx <= h and appearing in the barrier function, sized [d
        t: (int) parameter of the barrier method
        v0: (array) initialisation point, in the strictly feasible domain, sized [n,]
        eps: (int) accepted error/target precision

    Returns:
        The list of iterates x until convergence at precision epsilon
    """
    V = [v0]
    x = v0
    grad_f = grad_f_t(Q,p,A,b,x,t)
    hess_f = hess_f_t(Q,p,A,b,x,t)
    dir = - np.linalg.solve(hess_f, grad_f)
    criteria = np.dot(hess_f, dir)
    criteria = np.dot(dir,criteria)
    n_iter = 0

    while (criteria > 2*eps): #criteria to stop
        #alpha = backtracking_with_gradient(Q,p,A,b,x,t, dir)
        alpha = backtracking(Q,p,A,b,x,t, dir)

        f_t_p = f_t(Q,p,A,b,x,t)
        x = x + alpha*dir

        f_t_n = f_t(Q,p,A,b,x,t)

        #computation of derivatives
        grad_f = grad_f_t(Q,p,A,b,x,t)
        hess_f = hess_f_t(Q,p,A,b,x,t)

        #computation of criteria
        criteria = np.dot(hess_f, dir)
        criteria = np.dot(dir,criteria)

        #new direction
        dir = - np.linalg.solve(hess_f, grad_f)
        V += [x]

        n_iter += 1 #every passage in the loop is a Newton iteration
    return V, n_iter
```

## Barrier method

We solve here the centering problem to draw a central path in the strictly feasible domain $\mathcal{D} = \{x \in \mathbf{R}^n \,|\, Fx < \lambda e_{2d}\}$. An obvious point here is $v_0 = O_n$, since $\lambda > 0$

In [ ]:
```
def barr_method(Q,p,A,b,v0,eps) :
    X = [[] for i in range(5)]
    N_iter = [[] for i in range(len(Mu))]
    for i in range(len(Mu)):
        t = 1
        x = v0
        while (2*d/t > eps):
            V, n_iter = centering_step(Q,p,A,b,t,x,eps) #n_iter is the number of column of V
            t *= Mu[i]
            x = V[-1] #Principle of the central path : we start from the last solution computed previously
            N_iter[i] += [n_iter]
            X[i] += [x] #sequence of iteration for this t*mu
    return X, N_iter
v0 = np.zeros((n))
```

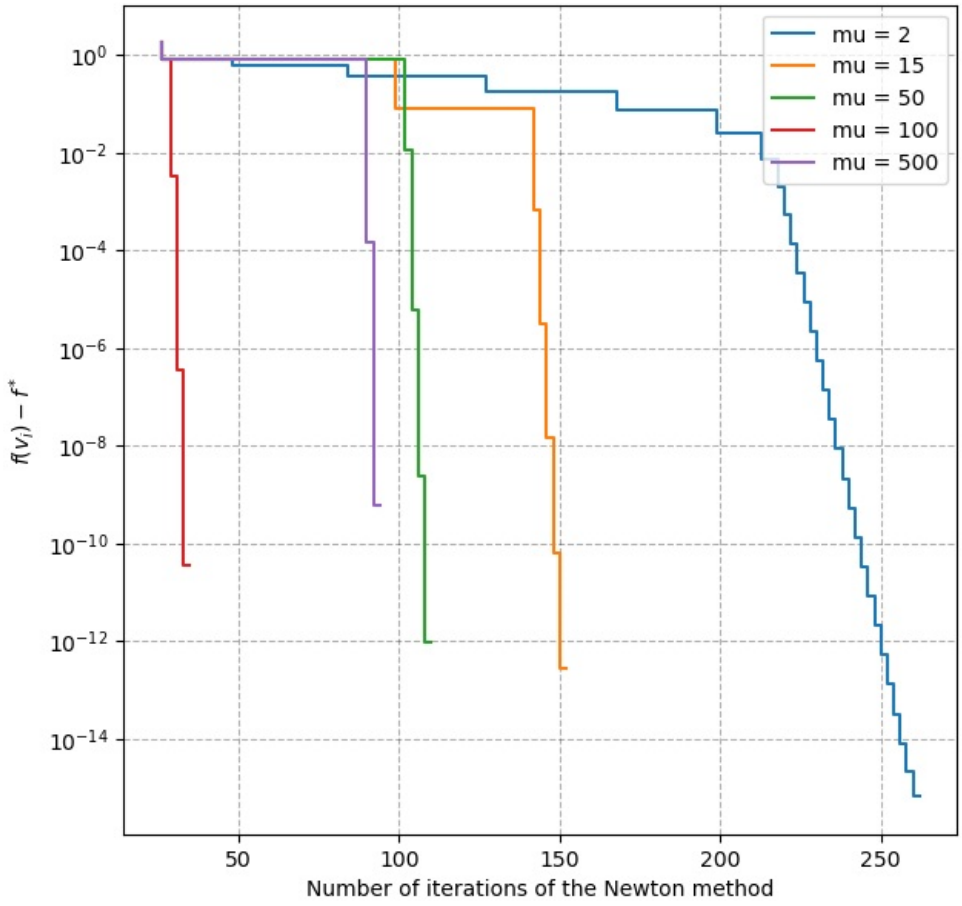
```
X, N_iter = barr_method(Q,p,F,b,v0,eps)
```

In [ ]:
```
def sum(L): #Sum with the left neighbour
    N = np.zeros(len(L))
    N[0] = L[0]
    for i in range(1,len(L)):
        N[i] = N[i - 1] + L[i]
    return N
```

## Resultat

### For classical backtracking and $\epsilon = 10^{-8}$

In [ ]:
```
for i in range(len(Mu)):
    plt.figure(num=1,figsize=(7,7))
    n_iter = N_iter[i]
    Newton_iter = sum(n_iter) #abscisse
    V = X[i] #give me the sequence for this mu
    f = [f_0(Q,p,V[j]) for j in range(len(V))] #I evaluate at every sequence the objective function
    gap = [f_0(Q,p,[0 for i in range(n)]) - f[-1]] + [p - f[-1] for p in f[:-1]] #This is the gap at every iter
    plt.step(Newton_iter, gap, label='mu = '+str(Mu[i]))
    plt.figure(1)
    plt.legend(loc = 'upper right')
    plt.semilogy()
    plt.xlabel('Number of iterations of the Newton method')
    plt.ylabel('$f(v_i)-f^*$')
    plt.rc('grid', linestyle="--", color='black',alpha = 0.3)
    plt.grid(True)
```




### With gradient-based backtracking and $\epsilon = 10^{-8}$

In [ ]:
```
for i in range(len(Mu)):
    plt.figure(num=1,figsize=(7,7))
    n_iter = N_iter[i]
    Newton_iter = sum(n_iter) #abscisse
    V = X[i] #give me the sequence for this mu
    f = [f_0(Q,p,V[j]) for j in range(len(V))] #I evaluate at every sequence the objective function
    gap = [f_0(Q,p,[0 for i in range(n)]) - f[-1]] + [p - f[-1] for p in f[:-1]] #This is the gap at every iter
    plt.step(Newton_iter, gap, label='mu = '+str(Mu[i]))
    plt.figure(1)
    plt.legend(loc = 'upper right')
    plt.semilogy()
    plt.xlabel('Number of iterations of the Newton method')
    plt.ylabel('$f(v_i)-f^*$')
```

```
    plt.rc('grid', linestyle="--", color='black',alpha = 0.3)
    plt.grid(True)
```