UNIVERSIDAD POLITÉCNICA DE CATALUNYA

Path-Following Primal-Dual Interior Point Algorithm

Ignasi Mañé Bosch and Joaquim Girbau Xalabarder

Professor: Jordi Castro

Contents

1 Path Following Algorithm

The goal of this assignment is to develop an interior point algorithm to solve a linear programming problem of the form:

$$\begin{aligned} \min_{x} \quad & c'x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

$$x, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$$

The first order necessary KKT conditions of this linear programming problem that can be defined as:

$$\begin{aligned} & A'\lambda + s = c \\ & Ax = b \\ & x_i s_i = 0 \qquad i = 1, ..., n \\ & (x, s) \geq 0 \end{aligned}$$

$$x, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}, \lambda \in \mathbb{R}^m, s \in \mathbb{R}^n$$

Since a linear programming problem is a convex problem, that is, it has a convex optimization function and convex feasible region, the global optimal solution can be obtained solving the KKT system of equations. Notice, though, that the KKT system is non linear due to term $x_i s_i$. This system of equations could, therefore, be solved applying the Newton's method. However, Newton's direction of movement can be improved introducing the idea of the central path.

Path following algorithms try to make movements following the central path, which is defined as the set of points that result form the solution of the modified $KKT - \tau$ system of equations for every value of $\tau \in \mathbb{R}^+$.

$$\begin{aligned} & A'\lambda + s = c \\ & Ax = b \\ & x_i s_i = \tau \qquad i = 1, ..., n \\ & (x, s) \geq 0 \end{aligned}$$

$$x, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}, \lambda \in \mathbb{R}^m, s \in \mathbb{R}^n$$

We will implement an algorithm that approximately follows the previously defined central path iteratively until value τ is close to 0 and therefore the system reduces to the original KKT conditions using two types of directions of movement: The Newton's and the modified Mehrotra's directions.

1.1 Newton's direction

Primal dual path following algorithm using Newton's direction of movement is defined in Figure 1. At each iteration variable $\tau_k = \mu_k \sigma_k$ is decreased by a reduction parameter $\sigma \in (0, 1)$ and then one iteration to solve the resulting $KKT - \mu$ system of equations is performed. Then the new value of μ_k is computed as the average $(x^k)'s/n$.

Algorithm *primal-dual path-following IPM*
Initial point (x^0, λ^0, s^0), $\sigma = 0.1$, $k = 0$
while (x^k, λ^k, s^k) is not solution **do**
$\mu_k = (x^k)^T s^k / n$

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S^k & 0 & X^k \end{bmatrix} \begin{bmatrix} \Delta_x^k \\ \Delta_\lambda^k \\ \Delta_s^k \end{bmatrix} = \begin{bmatrix} -r_c \\ -r_b \\ -X^k S^k + \sigma_k \mu_k e \end{bmatrix}$$

Compute α_p, α_d
$x^{k+1} = x^k + \alpha_p \Delta_x^k$
$(\lambda^{k+1}, s^{k+1}) = (\lambda^k, s^k) + \alpha_d (\Delta_\lambda^k, \Delta_s^k)$
$k := k + 1$
end_while
Return: $(x^*, \lambda^*, s^*) = (x^k, \lambda^k, s^k)$
End_algorithm

Figure 1: Primal-Dual Path following Algorithm

1.2 Mehrotra's direction

Primal dual path following algorithm using Newton's direction of movement is defined in Figure 2. This method works similarly to the previous explained version. However, in this algorithm the direction of movement used to solve the $KKT - \tau$ system is calculated using a second order Tylor approximation. The resulting system of equations, though, is non linear, so, to be able to easily obtain an approximate solution Mehrotra's heuristic is applied.

Algorithm *Mehrotra predictor-corrector*

Starting point (x^0, λ^0, s^0), $k = 0$

while (x^k, λ^k, s^k) is not solution **do**

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S^k & 0 & X^k \end{bmatrix} \begin{bmatrix} \Delta_x^{aff} \\ \Delta_\lambda^{aff} \\ \Delta_s^{aff} \end{bmatrix} = \begin{bmatrix} -r_c \\ -r_b \\ -X^k S^k e \end{bmatrix}$$

Compute α_p^{aff}, α_d^{aff}

$\mu^{aff} = (x^k + \alpha_p^{aff} \Delta_x^{aff})^T (s^k + \alpha_d^{aff} \Delta_s^{aff})/n$

$\mu = (x^k)^T s^k / n$

$\sigma = \left(\mu^{aff}/\mu\right)^3$

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S^k & 0 & X^k \end{bmatrix} \begin{bmatrix} \Delta_x^{cc} \\ \Delta_\lambda^{cc} \\ \Delta_s^{cc} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sigma\mu e - \Delta_X^{aff} \Delta_S^{aff} e \end{bmatrix}$$

$(\Delta_x, \Delta_\lambda, \Delta_s) = (\Delta_x^{aff} + \Delta_x^{cc}, \Delta_\lambda^{aff} + \Delta_\lambda^{cc}, \Delta_s^{aff} + \Delta_s^{cc})$

Compute α_p, α_d

$x^{k+1} = x^k + \alpha_p \Delta_x$

$(\lambda^{k+1}, s^{k+1}) = (\lambda^k, s^k) + \alpha_d (\Delta_\lambda, \Delta_s)$

$k := k + 1$

end_while

Return: $(x^*, \lambda^*, s^*) = (x^k, \lambda^k, s^k)$

End_algorithm

Figure 2: Primal Dual Path Following with Mehrotra's direction

1.3 Normal Equations

The matrix structure of all the system of equations of the previous algorithms is identical. In theory classes, three approaches to deal with this type of system of equations were presented: The direct approach, the augmented system of equations and the normal equations, which will be used in this implementation and are defined in Figure 3.

$$\begin{aligned} (A\Theta A^T)\Delta_\lambda &= -r_b + A\Theta(-r_c + X^{-1} r_{xs}) &&= -r_b + A(-XS^{-1} r_c + S^{-1} r_{xs}) \\ \Delta_s &= -r_c - A^T \Delta_\lambda && [\text{ from } A^T \Delta_\lambda + \Delta_s = -r_c] \\ \Delta_x &= -S^{-1}(r_{xs} + X\Delta_s) && [\text{ from } S\Delta_x + X\Delta_s = -r_{xs}] \end{aligned}$$

Figure 3: Normal System of Equations

Where $\Theta = XS^{-1}$, being X and S two diagonal matrices with vectors x and s in their diagonal respectively, and right hand side vector $b = (r_b, r_c, r_{xs})$.

The normal equations appear after performing simple Gaussian elimination to the original linear system. Notice that the normal equations can be applied to any of the three system of equations appearing in Figures 1 and 2 as long as vector b is defined accordingly.

1.4 Initial Point Heuristic

The initial point is an important practical issue in interior point methods. Depending on the initial seed from which the algorithm starts to iterate, the convergence might not be guaranteed. Ideally, the value (x^0, λ^0, s^0) should be an interior point a as close as possible

to the center path.

Although not being stated in the instructions of the assignment, to provide our algorithm with a good starting point we have implemented in the Matlab function StartPoint the Heuristic described in [1, page 410]. This heuristic will allow us to compute a reasonably good starting point also satisfying $x^0 >$ and $s^0 > 0$. It is defined as follows:

First we find the minimum norm vectors $\tilde{x}$ and $(\lambda s, \tilde{s})$ which satisfy the primal and dual constraints $Ax = b$ and $A^{'}\lambda + s = c$ respectively, solving the following convex quadratic optimization problems:

$$\min_{x} \tfrac{1}{2}x^T x \ \text{ subject to } Ax = b,$$
$$\min_{(\lambda,s)} \tfrac{1}{2}s^T s \ \text{ subject to } A^T\lambda + s = c.$$

The result of the previous optimization problems might not be a positive vector, violating the necessary initial conditions $x^0 >$ and $s^0 > 0$. Therefore, the following operations are defined so that the resulting vectors have non negative components.

$$\delta_x = \max(-(3/2)\min_i \tilde{x}_i, 0), \quad \delta_s = \max(-(3/2)\min_i \tilde{s}_i, 0),$$

$$\hat{x} = \tilde{x} + \delta_x e, \quad \hat{s} = \tilde{s} + \delta_s e,$$

Finally, to ensure that vectors $\tilde{x}$ and $(\lambda s, \tilde{s})$ not too close to 0 nor too dissimilar the following transformations are applied.

$$\hat{\delta}_x = \frac{1}{2}\frac{\hat{x}^T\hat{s}}{e^T\hat{s}}, \quad \hat{\delta}_s = \frac{1}{2}\frac{\hat{x}^T\hat{s}}{e^T\hat{x}}$$

$$x^0 = \hat{x} + \hat{\delta}_x e, \ \ \lambda^0 = \tilde{\lambda}, \ \ s^0 = \hat{s} + \hat{\delta}_s e.$$

Thus, obtaining an initial valid solution that will be used to initialize the implemented algorithm.

1.5 Cholesky-Infinity Factorization Method

The primal dual path following algorithm involves the solution of linear system of equations iteratively (see normal equations). Actually, these operations are the most computationally expensive procedures done during the execution process. Therefore, to increase the performance of the algorithm and since matrix $A\Theta A'$ is positive semi definite, it is worth to use a cholesky factorization approach. However, this decomposition is only defined for positive definite matrices, and would fail when applied to a matrix with at least one

eigenvalue equal to 0. To overcome this problem, two approaches were considered, implemented, and tested in Matlab.

In the first case, a perturbation matrix defined as ϵI, where the identity matrix $I \in \mathbb{R}^{m \times m}$ and ϵ is close to zero, was added to matrix $A\Theta A'$, so that matrix $A\Theta A' + \epsilon I$ is no longer positive semi definite but positive definite. This solution has been applied in Matlab functions Newton and Mehrotra.

The second applied method consisted on implementing the modified version of the Cholesky factorization algorithm defined in Figure 4. This modified version of the factorization, which has been implemented in Matlab function Cholesky, substitutes any 0 pivot by a big enough value such as 10^{68}. This solution has been applied in Matlab functions NewtonChol and MehrotraChol.

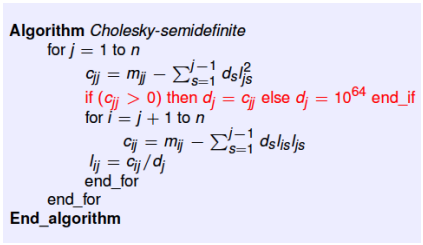

Figure 4: Cholesky Factorization for semidefinite matices

2 Matlab Implementation

The algorithms have been implemented in Matlab, a numerical computing environment developed by MathWorks. This program offers a friendly and high performance environment for matrix manipulation and algorithm implementation.

2.1 Function Cholesky

This function implements the modified version of the Cholesky factorization for positive semi definite matrices described in Figure 3. This function is called in functions Matlab functions MewtonChol and MehrotraChol to apply the factorization to matrix $A\Theta A'$.

Input Arguments

- **A**: a Matrix object containing the matrix to be factorized.

Output Values

- **L**: Lower diagonal matrix of the cholesky factorization.
- **D**: Diagonal matrix of the cholesky factorization with zero pivots replaced by 10^{68}.

```
function [D,L] = Cholesky(A)

[m,n] = size(A);
L = sparse(diag(ones(m,1)));
```

```
D = sparse(zeros(m,1));
zero = 1e-16;

for j = 1:m
     c = A(j,j)-D'*(L(j,:)'.*L(j,:)');
     if (c>zero)
        D(j)=c;
     else
        D(j)=10^68;
     end

     for i = (j+1):m
         L(i,j)=(A(i,j)-D'*(L(i,:)'.*L(j,:)'))/D(j);
     end
end

end
```

2.2 Function StartPoint

This matlab function applies the heuristic described in Section 1.4 to get an initial point from which to start the primal dual path following algorithm implemented in Matlab functions Newton, NewtonChol, Mehrotra, and MehrotraChol. The function calls fmincon to solve the convex optimization problem.

Input Arguments

- **c**: a Matrix object containing vector c
- **A**: a Matrix object containing matrix A
- **b**: a Matrix object containing vector b
- **w**: a weight to change the initial point of the optimization algorithm

Output Values

- **X**: the initial point x^0
- **L**: the initial point λ^0
- **S**: the initial point s^0

```
function [X,L,S] = StartPoint(A, b, c, w)

if ~exist('w','var')
  w =10;
end

[m,n]= size(A);
options.MaxFunEvals = 30000;

%P1
%--------------------
Aeq = A;
beq = b;
c = c;
```

```
A = [];
b = [];
lb=[];
ub=[];
nonlcon = [];

x0 = ones(n,1)*w;
fun = @(x)x'*x;
sol1 = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);

%P2
%--------------------
Aeq=[Aeq' eye(n)];
beq=c;
e2 = [zeros(m,1)
    ones(n,1)];

options.MaxFunEvals = 30000;

fun = @(x)(e2.*x)'*(e2.*x);
x0 = ones(m+n,1)*w;
sol2 = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options);

%Correctors
%--------------------

deltax = max(-(3/2)*min(sol1),0);
deltas = max(-(3/2)*min(sol2(m+1:m+n)),0);

xstart = sol1 + deltax*ones(n,1);
sstart = sol2(m+1:m+n) + deltas*ones(n,1);
lambdastart = sol2(1:m);

deltax = (1/2)*((xstart'* sstart)/(ones(n,1)'*sstart));
deltas = (1/2)*((xstart'* sstart)/(ones(n,1)'*xstart));

xstart = xstart + deltax*ones(n,1);
sstart = sstart + deltas*ones(n,1);

X=xstart;
L=lambdastart;
S=sstart;
end
```

2.3 Functions NewtonChol and Newton

We include here the code of NewtonChol function, Newton code can be found on the appendix.

These functions apply the primal dual path following algorithm using the Newton's direction. NewtonChol function uses Cholesky function, which performs the Cholesky factorization for positive semi definite matrices, to factorize matrix $A\Theta A'$, whereas Newton

uses the native matlab version of Cholesky to factorize the perturbed version of $A\Theta A'$, $A\Theta A' + \epsilon I$.

Input Arguments

- **c**: a Matrix object containing vector c
- **A**: a Matrix object containing matrix A
- **b**: a Matrix object containing vector b
- **x**: a vector containing the initial point x^0 from which start the algorithm.
- **lambda**: a vector containing the initial point $\lambda^{`0}$ from which start the algorithm.
- **s**: a vector containing the initial point s^0 from which start the algorithm.
- **gap**: a numeric value containing the optimality gap at which the algorithm must stop.
- **weight**: a numeric value to change the initial point: $x_0 = weight * e$, where e is a vector of ones, if not provided.

Output Values

- **X**: a vector containing the optimal solution.
- **Z**: a numeric variable containing the value of the objective function at the optimal solution.
- **K**: The total number of iterations needed to reach the optimal solution.

```
function [X,Z,K] = NewtonChol(A, b, c, x, lambda, s, gap, weight)

    %parameters
    if isempty(gap)
        gap = 1.0e-6; %optimality gap
    end
    if isempty(weight)
        weight = 20; %weight initial point
    end

    k=0; %iteration number
    rho= 0.99;
    eps = 1e-4;
    zero = 1e-16;
    sigma = 0.1;

    %dimensions of inputs
    [m,n]= size(A);

    %starting point (might be infeasible)
    if isempty(x)
        x = ones(n,1)*weight;
    end
    if isempty(s)
        s = ones(n,1)*weight;
    end
    if isempty(lambda)
        lambda = ones(m,1)*weight;
    end
```

```
%matrix definitions
e = ones(n,1);
is = 1./s;
xs = x.*s;
rc = A'*lambda + s - c;
rb = A*x - b;
mu = x'*s/n;
rxs = x.*s - sigma*mu*e;
I = eye(m);

%Main loop while not solution
%----------------------------
while (((norm(rc)/(1+norm(c)))> gap || (norm(rb)/(1+norm(b)))> gap
            || (mu > gap)) & k < 150)
    %compute the affine direction

        %matrix Theta
        Theta=sparse(diag(x.*is));
        AThA=sparse(A*Theta*A');
        P = colamd(AThA);
        [D,L] = Cholesky(AThA(P,P));
        righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

        %direction affine
        dlam(P) = (diag(D)*L')\(L\righ_hand(P));
        dlam = dlam(:);
        ds = -rc - A'*dlam;
        dx = -is.*rxs - x.*is.*ds;

        %alphas p and d
        alphas = -x./dx;
        alpha_p = min(1, rho*min(alphas(alphas>=zero)));

        alphas = -s./ds;
        alpha_d = min(1, rho*min(alphas(alphas>=zero)));

        %new point x
        x = x + alpha_p*dx;
        s = s + alpha_d*ds;
        lambda = lambda + alpha_d*dlam;

        %update iteration number
        k = k +1;

        %update values
        mu = x'*s/n;
        rc = A'*lambda + s - c;
        rb = A*x - b;
        rxs = x.*s - sigma*mu*e;
        is = 1./s;

%End main loop if condition satisfied
%------------------------------------
```

```
    end

    Z = c'*x;
    X = x;
    K = k;

end
```

2.4 Functions MehrotraChol and Mehrotra

We include here the code of MehrotraChol function, Mehrotra code can be found on the appendix.

These functions apply the primal dual path following algorithm using the Mehrotra's direction. MehrotraChol function uses Cholesky function, which performs the Cholesky factorization for positive semi definite matrices, to factorize matrix $A\Theta A'$, whereas Mehrotra uses the native matlab version of Cholesky to factorize the perturbed version of $A\Theta A'$, $A\Theta A' + \epsilon I$.

Input Arguments

- **c**: a Matrix object containing vector c
- **A**: a Matrix object containing matrix A
- **b**: a Matrix object containing vector b
- **x**: a vector containing the initial point x^0 from which start the algorithm.
- **lambda**: a vector containing the initial point $\lambda^{`0}$ from which start the algorithm.
- **s**: a vector containing the initial point s^0 from which start the algorithm.
- **gap**: a numeric value containing the optimality gap at which the algorithm must stop.
- **weight**: a numeric value to change the initial point: $x_0 = weight * e$, where e is a vector of ones, if not provided.

Output Values

- **X**: a vector containing the optimal solution.
- **Z**: a numeric variable containing the value of the objective function at the optimal solution.
- **K**: The total number of iterations needed to reach the optimal solution.

```
function [X,Z,K] = MehrotraChol(A, b, c, x, lambda, s, gap, weight)

    %parameters
    if isempty(gap)
        gap = 1.0e-6; %optimality gap
    end
    if isempty(weight)
        weight = 20; %weight initial point
    end

    k=0; %iteration number
    rho= 0.99;
    zero = 1e-68;
```

```
%dimensions of inputs
[m,n]= size(A);

%starting point (might be infeasible)
if isempty(x)
    x = ones(n,1)*weight;
end
if isempty(s)
    s = ones(n,1)*weight;
end
if isempty(lambda)
    lambda = ones(m,1)*weight;
end

%matrix definitions
e = ones(n,1);
is = 1./s;
xs = x.*s;
rc = A'*lambda + s - c;
rb = A*x - b;
rxs = x.*s;
I = eye(m);
mu = x'*s/n;

%Main loop while not solution
%----------------------------
while (((norm(rc)/(1+norm(c)))> gap || (norm(rb)/(1+norm(b)))> gap
            || (mu > gap)) & k < 150)
    %compute the affine direction

        %matrix Theta
        Theta=sparse(diag(x.*is));
        AThA=sparse(A*Theta*A');
        P = colamd(AThA);
        [D,L] = Cholesky(AThA(P,P));
        righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

        %direction affine, %hem d'aplicar choleski
        daflam(P) = (diag(D)*L')\(L\righ_hand(P));
        daflam = daflam(:);
        dafs = -rc - A'*daflam;
        dafx = -is.*rxs - x.*is.*dafs;

        affdir = [dafx
                  daflam
                  dafs];

        %alphas p and d affine (repassar) logica de alphas
        alphas = -x./dafx;
        alpha_p_aff = min([alphas(alphas>=zero) ;1]);

        alphas = -s./dafs;
```

```
        alpha_d_aff = min([alphas(alphas >= zero);1]);

        %mu affine, mu and sigma
        mu_aff = (x + alpha_p_aff*dafx)'*(s + alpha_d_aff*dafs)/n;
        sigma = (mu_aff/mu)^3;

        %center corrector direction falta cholesky
        rb = zeros(m,1);
        rc = zeros(n,1);
        rxs = - sigma*mu*e + dafx.*dafs;
        righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

        dcclam(P) = (diag(D)*L')\(L\righ_hand(P));
        dcclam = dcclam(:);
        dccs = -rc - A'*dcclam;
        dccx = -is.*rxs - x.*is.*dccs;

        ccdir = [dccx
                 dcclam
                 dccs];

        %direction
        direction = ccdir + affdir;

        %alphas p and d
        alphas = -x./direction(1:n);
        alpha_p = min(1, rho*min(alphas(alphas>=zero)));

        alphas = -s./direction(n+m+1:2*n+m);
        alpha_d = min(1, rho*min(alphas(alphas>=zero)));

        %new point x
        x = x + alpha_p*direction(1:n);
        s = s + alpha_d*direction(n+m+1:2*n+m);
        lambda = lambda + alpha_d*direction(n+1:n+m);

        %update iteration number
        k = k +1;

        %update values
        mu = x'*s/n;
        rc = A'*lambda + s - c;
        rb = A*x - b;
        rxs = x.*s;
        is = 1./s;

%End main loop is condition satisfied
%-------------------------------------
end

Z = c'*x;
X = x;
K = k;
```

```
end
```

3 Testing the Algorithm

In order to test the code we attempted the same problems that we solved in the previous assignment using the primal affine scaling algorithm implemented in R plus one extra problem. That is, problems 596, 597, 598 and 614 from the Netlib list of linear problems. All the information related to them can be found on this link.

Function StartPoint, which apply the heuristic presented in Section 1.4, was used to get an initial feasible solution in each problem. The execution of StartPoint worked well in all the problems except problem 614, in which the optimization problem did not converge to the global optimal solution because the maximum number of allowed function evaluations was reached. However, although calculated form a suboptimal solution of the optimization problem of the heuristic, the initial seed served well its propose in practice, so it was no necessary to increase the limit.

Each problem 596, 597, 598 and 614 was solved four times using one of the functions Newton.mat, NewtonChol, Mehrotra and MehrotraChol.

- **Newton.mat**: Problem solved applying Newton's direction and the Matlab's version of Cholesky along with the perturbed matrix $A\Theta A' + \epsilon I$
- **NewtonChol.mat**: Problem solved applying Newton's direction and the hand made implementation of the Cholesky factorization version for positive semi definite matrices (Cholesky.mat).
- **Mehrotra**: Problem solved applying Mehrotra's direction and the Matlab's version of Cholesky along with the perturbed matrix $A\Theta A' + \epsilon I$
- **MehrotraChol**: Problem solved applying Mehrotra's direction and the hand made implementation of the Cholesky factorization version for positive semi definite matrices (Cholesky.mat)

3.1 Body Script and procedure

For each problem, we load the matrix A, the right hand side vector b, and the coefficient cost vector c from the problem 596. Then, the function StarPoint is called to find an reasonably good starting point using the described heuristic in Section 1.4. Finally each problem is solved using each of the functions Newton, NewtonChol, Mehrotra, and MehrotraChol to obtain the number of iterations, the value of the objective function and the optimal solution.

```
%-------------------------------------%
%Problem 596
%-------------------------------------

load('596_lp_adlittle.mat')
```

```
A = Problem.A;
b = Problem.b;
c = Problem.aux.c;
gap=1.0e-6;

%initial point problem
[x,lambda,s] = StartPoint(A,b,c);

%Newton
%--------------------------
    %native chol
    [x596n,z596n,k596n] = Newton(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x596nc,z596nc,k596nc] = NewtonChol(A, b, c, x, lambda, s,gap,[]);

%Mehrotra
%--------------------------
    %native chol
    [x596m,z596m,k596m] = Mehrotra(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x596mc,z596mc,k596mc] = MehrotraChol(A, b, c, x, lambda, s,[],[]);

%-------------------------------------%
%PROBLEM 597
%-------------------------------------%

load('597_lp_afiro.mat')

A = Problem.A;
b = Problem.b;
c = Problem.aux.c;
gap=1.0e-6;

[x,lambda,s] = StartPoint(A,b,c);

%Newton
%--------------------------
    %native chol
    [x597n,z597n,k597n] = Newton(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x597nc,z597nc,k597nc] = NewtonChol(A, b, c, x, lambda, s,gap,[]);

%Mehrotra
%--------------------------
    %native chol
    [x597m,z597m,k597m] = Mehrotra(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x597mc,z597mc,k597mc] = MehrotraChol(A, b, c, x, lambda, s,gap,[]);
```

```
%-------------------------------------%
%PROBLEM 598
%-------------------------------------%

load('598_lp_agg.mat')

A = Problem.A;
b = Problem.b;
c = Problem.aux.c;
gap=1.0e-6;

%initial point problem
[x,lambda,s] = StartPoint(A,b,c);

%Newton
%--------------------------
    %native chol
    [x598n,z598n,k598n]  = Newton(A, b, c, x,lambda,s,gap,[]);

    %own cholesky-inf funtion
    [x598nc,z598nc,k598nc]  = NewtonChol(A, b, c, x,lambda,s,gap,[]);

%Mehrotra
%--------------------------
    %native chol
    [x598m,z598m,k598m] = Mehrotra(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x598mc,z598mc,k598mc] = MehrotraChol(A, b, c, x, lambda, s,gap,[]);

%-------------------------------------%
%PROBLEM 614
%-------------------------------------%

load('614_lp_czprob.mat')

A = Problem.A;
b = Problem.b;
c = Problem.aux.c;
gap=1.0e-6;

%initial point problem
[x,lambda,s] = StartPoint(A,b,c);

%Newton
%--------------------------
    %native chol
    [x614n,z614n,k614n] = Newton(A, b, c, x, lambda, s ,gap,[]);

    %own cholesky-inf funtion
    [x614nc,z614nc,k614nc] = NewtonChol(A, b, c, x, lambda, s ,gap,[]);

%Mehrotra
```

```
%--------------------------
    %native chol
    [x614m,z614m,k614m] = Mehrotra(A, b, c, x, lambda, s,gap,[]);

    %own cholesky-inf funtion
    [x614mc,z614mc,k614mc] = MehrotraChol(A, b, c, x, lambda, s,gap,[]);
```

4 Results

In this section we compare the results obtained in terms of the number of iterations and the value of the objective function for each of the attempted problems. First, the solutions obtained through the previous explained path following algorithms will be summarized and, second, the best result form all of them will be compared to the those obtained using the primal affine scaling algorithm of the previous assignment and some standard software.

Table 1 summarizes the outputs of each function applied to each problem. Column Newt. includes the results obtained with Newton function, column Newt. Chol the results obtained with NewtonChol function, column Mehr. the outputs of Mehrotra function, and column Mehr. Chol. those obtained with MehrotraChol function.

As expected, Newton's direction is less efficient than Mehrotra's. It is worth mention that, since they failed to reach the optimal solution in problem 598 due to definiteness problems in the Cholesky factorization step, the functions using the native version of Cholesky to factorize the perturbed matrix $A\Theta A' + \epsilon I$ seem less robust than those implementing the hand made version of Cholesky factorization for positive semi definite matrices.

	Dim.	Newt.		Newt. Chol		Mehr.		Mehr. Chol	
Pr.	$m \times n$	Iter	Value	Iter	Value	Iter	Value	Iter	Value
596	56×138	19	2.25e5	19	2.25e5	12	2.25e5	12	2.25e5
597	27×51	13	-464.75	13	-464.75	8	-464.75	8	-464.75
598	488×615	error	error	45	-35.99e6	error	error	35	-35.99e6
614	$929 \times 3,562$	65	2.18e6	65	2.18e6	41	2.18e6	39	2.18e6

Table 1: Summary of the Results Obtained with path following alghorithms

Taking the best results form Table 1 and comparing them in Table 2 to the results obtained through a standard software solver such as MINOS and our implementation of the primal affine scaling in R of the last assignment, it is immediate to see that the primal dual path following algorithm is more efficient that any of the other methods previously used to solve problems 596, 597, and 598.

	Dim.	R affineScaling		Minos 5.51		IPM Newt.		IPM Mehr.	
Pr.	$m \times n$	Iter	Opt. Value	Iter	Value	Iter	Value	Iter	Value
596	56×138	33	2.25e5	138	2.25e5	19	2.25e5	12	2.25e5
597	27×51	17	-464.75	10	-464.75	13	-464.7531	8	-464.75
598	488×615	error	error	108	-35.99e6	45	-35.99e6	35	-35.99e6

Table 2: Previous assignment results comparaison

5 References

[**1**] Nocedal, J., Wright, S. J. (2006). Numerical optimization. New York, NY: Springer. ISBN: 978-0-387-30303-1

[**2**] Castro, J. Class Slides

A Function Newton

```
function [X,Z,K] = Newton(A, b, c, x, lambda, s, gap, weight)

    %parameters
    if isempty(gap)
        gap = 1.0e-6; %optimality gap
    end
    if isempty(weight)
        weight = 20; %weight initial point
    end

    k=0; %iteration number
    rho= 0.99;
    eps = 1e-4;
    zero = 1e-16;
    sigma = 0.1;

    %dimensions of inputs
    [m,n]= size(A);

    %starting point (might be infeasible)
    if isempty(x)
        x = ones(n,1)*weight;
    end
    if isempty(s)
        s = ones(n,1)*weight;
    end
    if isempty(lambda)
        lambda = ones(m,1)*weight;
    end

    %matrix definitions
    e = ones(n,1);
    is = 1./s;
    xs = x.*s;
    rc = A'*lambda + s - c;
    rb = A*x - b;
    mu = x'*s/n;
    rxs = x.*s - sigma*mu*e;
    I = eye(m);


    %Main loop while not solution
    %----------------------------
    while (((norm(rc)/(1+norm(c)))> gap || (norm(rb)/(1+norm(b)))> gap
                || (mu > gap)) & k < 500)
        %compute the affine direction

            %matrix Theta
            Theta=sparse(diag(x.*is));
            AThA=sparse(A*Theta*A' + eps*I);
            P = colamd(AThA);
```

```
            AThA_Chol = chol(AThA(P,P));
            righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

            %direction affine, %hem d'aplicar choleski
            dlam(P) = AThA_Chol\(AThA_Chol'\righ_hand(P));
            dlam = dlam(:);
            ds = -rc - A'*dlam;
            dx = -is.*rxs - x.*is.*ds;

            %alphas p and d
            alphas = -x./dx;
            alpha_p = min(1, rho*min(alphas(alphas>=zero)));

            alphas = -s./ds;
            alpha_d = min(1, rho*min(alphas(alphas>=zero)));

            %new point x
            x = x + alpha_p*dx;
            s = s + alpha_d*ds;
            lambda = lambda + alpha_d*dlam;

            %update iteration number
            k = k +1;

            %update values
            mu = x'*s/n;
            rc = A'*lambda + s - c;
            rb = A*x - b;
            rxs = x.*s - sigma*mu*e;
            is = 1./s;
    %End main loop is condition satisfied
    %-------------------------------------
    end

    Z = c'*x;
    X = x;
    K = k;

end
```

B Function Mehrotra

```
function [X,Z,K] = Mehrotra(A, b, c, x, lambda, s, gap, weight)

    %parameters
    if isempty(gap)
        gap = 1.0e-6; %optimality gap
    end
    if isempty(weight)
        weight = 20; %weight initial point
    end

    k=0; %iteration number
```

```
rho= 0.99;
eps = 1e-4;
zero = 1e-16;

%dimensions of inputs
[m,n]= size(A);

%starting point (might be infeasible)
if isempty(x)
    x = ones(n,1)*weight;
end
if isempty(s)
    s = ones(n,1)*weight;
end
if isempty(lambda)
    lambda = ones(m,1)*weight;
end

%matrix definitions
e = ones(n,1);
is = 1./s;
xs = x.*s;
rc = A'*lambda + s - c;
rb = A*x - b;
rxs = x.*s;
I = eye(m);
mu = x'*s/n;

%Main loop while not solution
%----------------------------
while (((norm(rc)/(1+norm(c)))> gap || (norm(rb)/(1+norm(b)))> gap
            || (mu > gap)) & k < 500)
    %compute the affine direction

        %matrix Theta
        Theta=sparse(diag(x.*is));
        AThA=sparse(A*Theta*A' + eps*I);
        P = colamd(AThA);
        AThA_Chol = chol(AThA(P,P));
        righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

        %direction affine, %hem d'aplicar choleski
        daflam(P) = AThA_Chol\(AThA_Chol'\righ_hand(P));
        daflam = daflam(:);
        dafs = -rc - A'*daflam;
        dafx = -is.*rxs - x.*is.*dafs;

        affdir = [dafx
                  daflam
                  dafs];

        %alphas p and d affine (repassar) logica de alphas
        alphas = -x./dafx;
```

```
        alpha_p_aff = min([alphas(alphas>=zero) ;1]);

        alphas = -s./dafs;
        alpha_d_aff = min([alphas(alphas >= zero);1]);

        %mu affine, mu and sigma
        mu_aff = (x + alpha_p_aff*dafx)'*(s + alpha_d_aff*dafs)/n;
        sigma = (mu_aff/mu)^3;

        %center corrector direction falta cholesky
        rb = zeros(m,1);
        rc = zeros(n,1);
        rxs = - sigma*mu*e + dafx.*dafs;
        righ_hand = -rb + A*(-x.*is.*rc + is.*rxs);

        dcclam(P) = AThA_Chol\(AThA_Chol'\righ_hand(P));
        dcclam = dcclam(:);
        dccs = -rc - A'*dcclam;
        dccx = -is.*rxs - x.*is.*dccs;

        ccdir = [dccx
                 dcclam
                 dccs];

        %direction
        direction = ccdir + affdir;

        %alphas p and d
        alphas = -x./direction(1:n);
        alpha_p = min(1, rho*min(alphas(alphas>=zero)));

        alphas = -s./direction(n+m+1:2*n+m);
        alpha_d = min(1, rho*min(alphas(alphas>=zero)));

        %new point x
        x = x + alpha_p*direction(1:n);
        s = s + alpha_d*direction(n+m+1:2*n+m);
        lambda = lambda + alpha_d*direction(n+1:n+m);

        %update iteration number
        k = k +1;

        %update values
        mu = x'*s/n;
        rc = A'*lambda + s - c;
        rb = A*x - b;
        rxs = x.*s;
        is = 1./s;
%End main loop is condition satisfied
%-------------------------------------
end

Z = c'*x;
```

```
    X = x;
    K = k;

end
```