PROBLEMS IN 02612 CONSTRAINED OPTIMIZATION

ASSIGNMENT 1

Jens	Østergaard	s173130
Mirza	Hasanbasic	s172987
Yevgen	Zainchkovskyy	s062870

Tuesday April 3, 2018

DTU Compute
Department of Applied Mathematics and Computer Science

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Problem 1

Quadratic optimization

Consider the problem

$$\begin{aligned}
\min_x \quad & f(x) = 3x_1^2 + 2x_1x_2 + x_1x3 + 2.5x_2^2 + 2x_2x_3 + 2x_3^2 - 8x_1 - 3x_2 - 3x_3 && \text{(1.1a)}\\
\text{s.t.} \quad & x_1 + x_3 = 3 && \text{(1.1b)}\\
& x_2 + x_3 = 0 && \text{(1.1c)}
\end{aligned}$$

in the form

$$\begin{aligned}
\min_x \quad & f(x) = \frac{1}{2}x'Hx + g'x && \text{(1.2a)}\\
\text{s.t.} \quad & A'x = b && \text{(1.2b)}
\end{aligned}$$

Question 1 What are H, g, A and b.

After rewriting (1.1) in the matrix form (1.2), we get the following values for H, g, A and b:

$$H = \begin{bmatrix} 6 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix} \quad g = \begin{bmatrix} -8 \\ -3 \\ -3 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

Question 2 Write the KKT optimality conditions.

The Lagrangian

$$\mathcal{L}(x, \lambda) = f(x) - \lambda'(A'x - b)$$

Then the KKT opbjective functions condtions:

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla_x f(x) - \nabla_x (A'x - b)\lambda = Hx - g - A\lambda = 0 \Leftrightarrow$$

$$\begin{bmatrix} 6 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix} x + \begin{bmatrix} -8 \\ -3 \\ -3 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \lambda = 0$$

KKT constaint conditions:

$$(A'x - b) = 0 \Leftrightarrow \quad x_1 + x_3 - 3 = 0 \quad x_2 + x_3 = 0$$

Finally we get the KKT matix

$$\begin{bmatrix} H & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \Leftrightarrow$$

$$\begin{bmatrix} 6 & 2 & 1 & 1 & 0 \\ 2 & 5 & 2 & 0 & 1 \\ 1 & 2 & 4 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = - \begin{bmatrix} -8 \\ -3 \\ -3 \\ 3 \\ 0 \end{bmatrix}$$

Question 3 Make a function `[x,lambda]=EqualityQPSolver(H,g,A,b)` for solution of equality constrained convex quadratic programs.

Hint: Consider and discuss which factorization to use when you factorize the KKT-matrix.

In order to solve the system, we need to factorize the KKT-matrix. While we could obtain the factors by using Gaussian elimination with partial pivoting, we also note the symmetry of the KKT-matrix. This lets us factor it with LDL-factorization much more effectively.

Listed below is the full Matlab function that solves equality constained convex quadratic programs.

```
function [x, lambda] = problem1_equality_qpsolver(H, g, A, b)
    % This function solves EQP's for Problem 1
    % by using LDL-factorization.
    [n,m] = size(A);
    KKT = [H, -A; -A', zeros(m)];
    d = -[g; b];
    [L, D, p] = ldl(KKT,'lower','vector');
    sol = L'\(D\(L\d(p)));
    x = sol(1:n);
    lambda = sol(n:end);
end
```

Question 4 Test your program on the above problem.

By simply defining H, g, A and b as given in **Question 1** and calling

```
[x, lambda] = problem1_equality_qpsolver(H, g, A, b)
```

we obtained following result (values for `x` and `lambda`):

```
x =
     2
    -1
     1

lambda =
    3.0000
   -2.0000
```

Question 5 Generate random convex quadratic programs (consider how this can be done) and test you program.

First, we are going to need a positive definite Hessian which we can get by multiplying a matrix of random numbers X with it's transpose and adding an identity matrix to increase the determinant (and therefore precision):

$$H = X^\top X + I$$

Second, a full row-rank matrix corresponding to equality constraints $A \in \mathbb{R}^{m\times n}$ with $m \leq n$:

$$A = \texttt{rand(n, n)} \quad \text{having} \quad \det(A) \neq 0$$

Finally, we are going to need a vector of x's and λ's (both $\in \mathbb{R}$) such that the right hand side of the KKT system can be calculated. The idea is then to try to solve the system backwards and seeing if we arrive at roughly the same x and λ.

The code below is used to generate KKT-systems and test our LDL-based EQP solver accoding to the description above. As a success metric, we calculate sum of squared error separately for both x and λ.

```
%% Randomly generated KKT systems
%  for the LDL-Based EQP solver
n = 5;
X = rand(n);
H = X'*X + eye(n);
A = rand(n);

x = rand(n,1);
lambda = rand(n,1);
```

```
KKT = [H, -A; -A', zeros(n)];
right_hand = KKT*[x; lambda];
g = -right_hand(1:n);
b = -right_hand(n+1:end);

[x_hat, lambda_hat] = problem1_equality_qpsolver(H, g, A, b);
ssex = sum((x - x_hat).^2)/n;
ssel = sum((lambda - lambda_hat).^2)/n;

[ssex; ssel]
```

After running the code several times it is learned that the error for both x and λ is in the neighbourhood of 10^{-27}. It is however also important to note that we observed a substential increase in error when $\det(H)$ was small.

Question 6 Write the sensitivity equations for the equality constrained convex QP.

Slides from Lecture 3 provides us with the parameter sensitivity equations:

$$\begin{aligned}
\begin{bmatrix}\nabla x(p) & \nabla\lambda(p)\end{bmatrix} &= -\begin{bmatrix}W_{xp} & -\nabla_p c(x,p)\end{bmatrix}\begin{bmatrix}W_{xx} & -\nabla_x c(x,p)\\ -\nabla_x c(x,p)^\top & 0\end{bmatrix}^{-1}\\
W_{xx} &= \nabla^2_{xx} f(x,p) - \sum_{i\in\mathcal{E}} \lambda_i \nabla^2_{xx} c_i(x,p)\\
W_{xp} &= \nabla^2_{xp} f(x,p) - \sum_{i\in\mathcal{E}} \lambda_i \nabla^2_{xp} c_i(x,p)
\end{aligned}$$

In our EQP case $p = [g, b]^\top$ with both $\nabla^2_{xx} c_i(x,p)$ and $\nabla^2_{xp} c_i(x,p)$ being 0 as constraints are only linear in p. We now proceed with calulation of all of the unknowns of the sensitivity equations:

$$\begin{aligned}
W_{xx} &= \nabla^2_{xx}(\frac{1}{2}x^\top Hx + g^\top x) = H\\
W_{xp} &= \nabla^2_{xp}(\frac{1}{2}x^\top Hx + g^\top x)\\
&= \nabla^2_p(g)\\
&= \begin{bmatrix}I & 0\end{bmatrix}^\top\\
\nabla_p c(x,p) &= \nabla_p(A^\top x - b) \quad \text{where} \quad b = \begin{bmatrix}0\\ I\end{bmatrix} p\\
&= -\begin{bmatrix}0\\ I\end{bmatrix}
\end{aligned}$$

And finally $\nabla_x c(x,p) = \nabla x(A^\top x - b) = A$, which all combines into the sensitivity expression for the type of EQP we have:

$$\begin{aligned}\begin{bmatrix}\nabla x(p) & \nabla \lambda(p)\end{bmatrix} &= -\begin{bmatrix}\begin{bmatrix}I\\0\end{bmatrix} & \begin{bmatrix}0\\I\end{bmatrix}\end{bmatrix}\begin{bmatrix}H & -A\\ -A^\top & 0\end{bmatrix}^{-1}\\ &= -\begin{bmatrix}H & -A\\ -A^\top & 0\end{bmatrix}^{-1}\end{aligned}$$

Interestingly, this is practically the inverse of KKT matrix we worked with in **Question 3** and because we already factorized it, we can save computation as it is easier to compute an inverse of a triangular matrix.

Question 7 Make a function that returns the sensitivities of the solution with respect to g and b. Test your program and discuss how you can verify that the sensitivities you compute are correct.

Using Matlab we wrote a simple function that returns the sensitivities of the solution when provided with H and A:

```
function [x_sens, l_sens] = problem1_sensitivities(H, A)
    % This function returns sensitivites for EQP's
    % as seen in Problem 1

    [n,m] = size(A);
    LikeKKT = -[H, -A; -A', zeros(m)];
    sol = inv(LikeKKT);
    x_sens = sol(:, 1:n);
    l_sens = sol(:, n+1:end);
end
```

Running the program on our problem, we get the following sensitivities:

$$\texttt{x\_sens} = \begin{bmatrix} -0.0769 & -0.0769 & 0.0769\\ -0.0769 & -0.0769 & 0.0769\\ 0.0769 & 0.0769 & -0.0769\\ 0.462 & -0.538 & 0.538\\ -0.385 & 0.615 & 0.385\end{bmatrix} \quad \texttt{l\_sens} = \begin{bmatrix} 0.462 & -0.385\\ -0.538 & 0.615\\ 0.538 & 0.385\\ 2.23 & -0.692\\ -0.692 & 3.08\end{bmatrix}$$

Which are read as follows: take the first entry -0.0769 (row 1, col 1) of the `x_sens`. This corresponds to a change in the solution x_1 when g_1 is incresed by 1.

As sensitivities are gradients, by changing g and b in increments of 1, we test by seeing if the solution is changing as expected. Here we try to change g from it's original value -8 to -7 and observe a change in the solution which alignes with the sensitivities found.

Recall **Question 4** where we found x_1 to be 2 with $g = -8$, now with the changed $g = -7$ the solution is 1.9231 which corresponds to a decrease of 0.0769 as we expected ($2 - 0.0769 = 1.9231$).

Question 8 Write the dual program of the equality constrained QP.

We define the constrained QP

$$\max_{x,\lambda} \quad L(x,\lambda) = \frac{1}{2}x'Hx + g'x - \lambda'(A'x - b) \tag{1.3}$$

with s.t. being

$$\nabla_x L(x,\lambda) = Hx + g - A\lambda = 0 \tag{1.4}$$

Write the dual program of the equality constrained QP. We wish to minimize the problem and then we multiply it with -1

$$\begin{aligned}
L(x,\lambda) &= -\left(\frac{1}{2}x'Hx + g'x - \lambda'(A'x - b)\right)\\
&= -\frac{1}{2}x'Hx - g'x + \lambda'A'x - \lambda'b\\
&= -\frac{1}{2}x'Hx + x'Hx - x'Hx - g'x + \lambda'A'x - \lambda'b\\
&= \frac{1}{2}x'Hx - (x'Hx - g'x + \lambda'A'x) - \lambda'b\\
&= \frac{1}{2}x'Hx - (Hx - g + \lambda A)'x - \lambda'b\\
&= \frac{1}{2}x'Hx - \lambda'b\\
&= \frac{1}{2}\begin{bmatrix} x \\ \lambda \end{bmatrix}'\begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ \lambda \end{bmatrix} + \begin{bmatrix} 0 \\ -b \end{bmatrix}'\begin{bmatrix} x \\ \lambda \end{bmatrix}
\end{aligned} \tag{1.5}$$

So our dual problem is

$$\begin{aligned}
\min_{x,\lambda} \quad & -L(x,\lambda) = -\frac{1}{2}\begin{bmatrix} x \\ \lambda \end{bmatrix}'\begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ \lambda \end{bmatrix} + \begin{bmatrix} 0 \\ -b \end{bmatrix}'\begin{bmatrix} x \\ \lambda \end{bmatrix}\\
\text{s.t.} \quad & \nabla_x L(x,\lambda) = \begin{bmatrix} H \\ -A \end{bmatrix}'\begin{bmatrix} x \\ \lambda \end{bmatrix} = -g\\
& \lambda_i \geq 0 i \in I
\end{aligned} \tag{1.6}$$

Question 9 What is the optimality conditions of the dual QP? How are they related to the primal QP? How are the variables in the primal and the dual problem related?

Now the Lagrainan can be constructed, where μ is the vector containing the Lagraian multipliers:

$$\mathcal{L}\left(\begin{bmatrix} x \\ \lambda \end{bmatrix}, \mu\right) = L - \mu(Hx - A\lambda + g) = L - \mu\left(\begin{bmatrix} H \\ -A \end{bmatrix}\begin{bmatrix} x \\ \lambda \end{bmatrix} + g\right)$$

And thus our KKT conditions become:

$$\nabla_{\begin{bmatrix} x \\ \lambda \end{bmatrix}} \mathcal{L}\left(\begin{bmatrix} x \\ \lambda \end{bmatrix}, \mu\right) = \begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ \lambda \end{bmatrix} + \begin{bmatrix} 0 \\ -b \end{bmatrix} + \begin{bmatrix} H \\ -A \end{bmatrix}\mu = 0 \tag{1.7}$$

$$\text{s.t.} \quad \begin{bmatrix} H \\ -A \end{bmatrix}\begin{bmatrix} x \\ \lambda \end{bmatrix} = -g \tag{1.8}$$

rewriting this into matrix notation:

$$\begin{bmatrix} H & 0 & -H \\ 0 & 0 & -A^T \\ -H & -A & 0 \end{bmatrix}\begin{bmatrix} x \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} 0 \\ -b \\ -g \end{bmatrix} \tag{1.9}$$

From the first row we gather that $Hx - H\mu = 0$ and since H is invertable due to it being simidefinite, the result is $x = s$. Therefor the KKT system can be simplified to the exact same problem as seen in the primal. This is also expected due to the Strong law of duality and further more that the dual variables to the dual problem are the solution to the primal.

Solving the dual problem can be done with the existing solver as the structure of the problem remains. Only slight post processing is needed to separate the values for x, λ and μ. It is expected that the solution time for the dual will be larger as the linear system now has grown in size.

Question 10 Make a function that solves the dual QP. Is there any advantages in solving the dual QP instead of the primal QP for the equality constrained convex quadratic program?

From the figure below, there doesn't seem to be any advantage, in using the dual instead of the primal. In fact, it seems, that the larger the matrices become, then it is better to use the primal method.

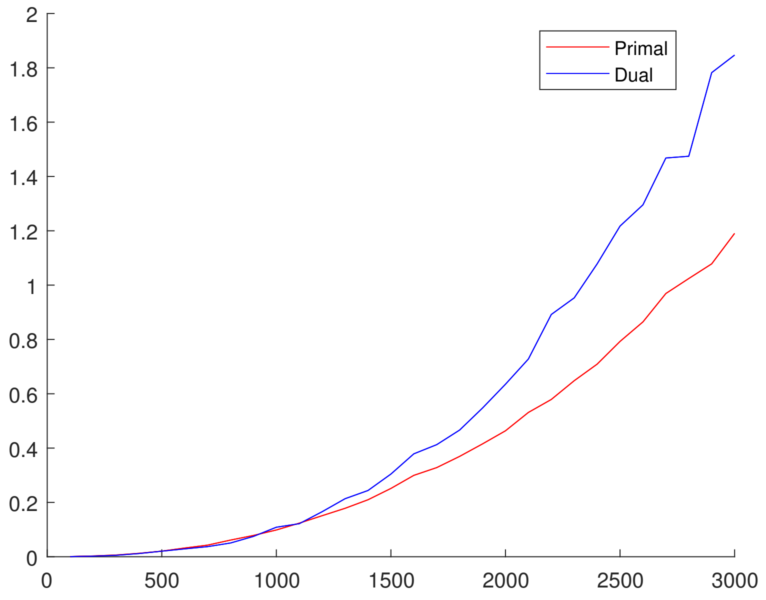


Figure 1.1: The dual runtime vs the primal

Problem 2

Equality Constrained Quadratic Optimization

This problems illustrates how solution of the equality constrained convex quadratic program scales with problem size and factorization method applied.

Consider the convex quadratic optimization problem

$$\min_{u} \quad \frac{1}{2}\sum_{i=1}^{n+1}(u_i - \bar{u})^2 \tag{2.1a}$$

$$\text{s.t.} \quad -u_1 + u_n = -d_0 \tag{2.1b}$$

$$u_i - u_{i+1} = 0 \qquad i = 1, 2, \ldots, n-2 \tag{2.1c}$$

$$u_{n-1} - u_n - u_{n+1} = 0 \tag{2.1d}$$

$\bar{u}$ and d_0 are parameters of the problem. The problem size can be adjusted selecting $n \geq 3$. Let $\bar{u} = 0.2$ and $d_0 = 1$. The constraints models a recycle system as depicted by the directed graph in figure below.

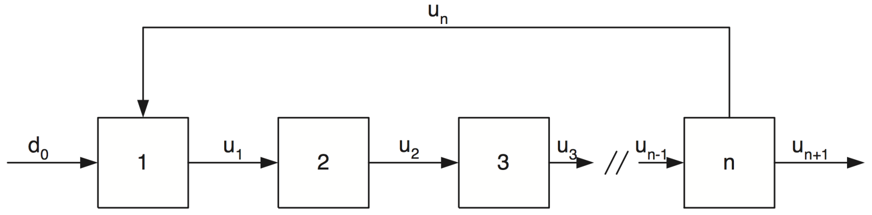


Figure 2.1: Directed graph representation of the constraints in (2.1). This graph represents a recycle system

Question 1 Express the problem in matrix form, i.e. in the form

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \phi = \frac{1}{2} x^\top H x + g^\top x \\ \text{s.t} \quad & A^\top x = b \end{aligned}$$

Let $n = 10$. What is x, H, g, A and b.

Expanding the objective function (2.1a) we get:

$$\begin{aligned} \phi &= \frac{1}{2}((u_1 - \bar{u})^2 + \cdots + (u_{11} - \bar{u})^2) \\ &= \frac{1}{2}(\underbrace{u_1^2 + \cdots + u_{11}^2}_{u^\top I u} \underbrace{-2\bar{u}u_1 - \cdots - 2\bar{u}u_{11}}_{-2\bar{u}[1]^{11\times 1}} \underbrace{+11\bar{u}^2}_{\text{constant}}) \\ H &= I \\ g &= -\bar{u}[1]^{11\times 1} \end{aligned}$$

Note that the $11\hat{u}^2$ constant can safely be ignored as it only affects the value of the objective function and does not change the solution.

For constraints, we have:

$$A = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \ddots & 1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 1 \\ 1 & 0 & 0 & \cdots & 0 & -1 \\ 0 & 0 & 0 & \cdots & 0 & -1 \end{bmatrix}^{11\times 10} \qquad b = \begin{bmatrix} -d_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^{12\times 1}$$

Question 2 What is the Lagrangian function and the first order optimality conditions for the problem? Explain why the optimality conditions are both necessary and sufficient for this problem.

The Lagrangian and first order optimality conditions are constructed according to the material in Lecture 3. Lagrangian:

$$\begin{aligned} L(u, \lambda) &= f(u) - \sum_{i \in \mathcal{E}} \lambda_i c_i(u) \\ &= \frac{1}{2} u^\top I u + g^\top u - \lambda^\top (A^\top u - b) \end{aligned}$$

First order optimality conditions with inequality constraints ignored as we do not have any of those:

$$\nabla_u L(u, \lambda) = Iu + g - A\lambda = 0$$
$$\nabla_\lambda L(u, \lambda) = -(A^\top u - b) = 0$$

Because our objective function ϕ is convex (H is positive definite) and equality constraints are linear, the first order optimality conditions are sufficient for this problem.

Question 3 Make a Matlab function that constructs H, g, A, and b as function of n, $\bar{u}$ and d_0.

The function `HgAb` below takes 3 inputs, and returns four values. The KKT is constructed by two lines of code, as seen in the next question. We have not made it into a function, but it is part of the 'solvers' as variables. Then we have constructed the needed solvers, which can be seen below.

```
function [H, g, A, b] = HgAb(n, ubar, d0)
        H = eye(n+1);
        g = -2 * ubar * ones(n+1,1);
        b = zeros(n, 1);
        b(1, 1) = -d0;
        A = [zeros(1, n-1); eye(n-1)];
        A = [A zeros(n, 1)] - eye(n);
        A = A';
        A = [A; zeros(1, n)];
        A(n, 1) = 1; A(n+1, n) = -1;
end
```

Question 4 Make a Matlab function that constructs the KKT-matrix as function of n, $\bar{u}$ and d_0.

After running `HgAb` from the question above, we construct KKT by simple concatenation:

```
KKT = [H, -A; -A', zeros(m)];
d = -[g; b];
```

Question 5 Make a Matlab function that solves (2.1) using an LU factorization.

When we have a square matrix, then it is possible to use LU factorization with partial pivoting. We can use this to solve a linear system If we assume that A is a square matrix and nonsingular, then we with L being the lower triangular matrix, U being the upper and P the permutation matrix, then we have

$$PA = LU$$

```
function [Sol] = LU_solver(H,g,A,b)
        % LU_SOLVER for problem 2
        KKT = [H, -A; -A', zeros(size(A,2))];
        d = -[g; b];
        [L,U,p] = lu(KKT,'vector');
        z = U\(L\d(p));
        Sol = z;
end
```

We have numerical instability, where if there are small rounding errors in the algorithm, then it will cause very large errors in the final solution.

Question 6 Make a Matlab function that solves (2.1) using an LDL factorization.

With A being symmetric positiv definit, then it can be written

$$A = LDL'$$

where L is the lower triangular matrix and D is diagonal. If it happens that A is indefinite, then we can end up with numerically unstable results, since L and D will end up with large values.

Since LDL is nummericaly unstable, we can use the symmetric indefinite factorization procedure. Then we can write

$$PAP' = LBL'$$

where P is the permutation matrix, L is the lower triangular matrix and B is a block diagonal matrix. It is possible to use LDL' instead of LBL', but as mentioned above we can end up with nummericaly unstable results.

```
function [Sol] = LDL_solver(H,g,A,b)
        % LDL_SOLVER for problem 2
        [n,m] = size(A);
        KKT = [H, -A; -A', zeros(m)];
        d = -[g; b];
        z = zeros(n+m,1);
        [L,D,p] = ldl(KKT,'lower','vector');
        z(p)=L'\(D\(L\d(p)));
        Sol = z;
end
```

Question 7 Make a Matlab function that solves (2.1) using the Null-Space procedure based on QR-factorizations.

We have to use Null-space procedure based on the QR-Factorization. Based on the slides from lecture 5, if we have a KKT in the form of

$$\begin{bmatrix} H & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \quad A \in \mathbb{R}^{n \times m}$$

then we can do QR-factorization on A, where we end up with

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R$$

with R_1 being a $n \times n$ upper triangular matrix, Q_1 being a $m \times n$ and Q_2 $m \times (m-n)$. It should be noted that Q_2 is our null space.

```
function [Sol] = NS_solver(H,g,A,b)
        % NS_SOLVER Note that from answer 2.1 H=I therefore
        [Q,Rbar] = qr(A);
        [n,m] = size(A);
        m1 = size(Rbar,2);
        Q1 = Q(:,1:m1);
        Q2 = Q(:,m1+1:n);
        R = Rbar(1:m1,1:m1);
        x_Y = R'\b;
        x_Z = (Q2'*Q2)\(-Q2'*(Q1*x_Y + g));
        x = Q1*x_Y + Q2*x_Z;
        l=R\Q1'*(x + g);
        Sol = [x;l];
end
```

Question 8 Make a Matlab function that solves (2.1) using the Range-Space procedure.

The method is usefull when H is diagonal or a block diagonal matrix. Assume that we have a KKT

$$\begin{bmatrix} H & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \quad A \in \mathbb{R}^{n \times m} \tag{2.2}$$

where H is positive definite.

From page 455 in NW, we see, that in equation 2.2, we can multiply the first equation by $A'H^{-1}$ and then subtract the second equation to obtain a linear system in the vector λ.

We have

$$Hx - A\lambda = -g \tag{2.3}$$

where we now multiply with $A'H^{-1}$

$$\begin{aligned} A'x - A'H^{-1}A\lambda &= -A'H^{-1}g \\ A'x &= A'H^{-1}A\lambda - A'H^{-1}g \\ x &= H^{-1}A\lambda - H^{-1}g \end{aligned} \tag{2.4}$$

Now we subtract the second equation to obtain a linear system in the vector λ.

We see that

$$A'x = b$$

substracting equation 2.4, where we take the middle from the one above, we get

$$\begin{aligned} A'x - (A'x - A'H^{-1}A\lambda) &= b - (-A'H^{-1}g) \\ A'H^{-1}A\lambda &= b + A'H^{-1}g \\ \lambda &= (A'H^{-1}A)^{-1}(b + A'H^{-1}g) \end{aligned} \tag{2.5}$$

It should be noted, that we first obtain λ and then x, from equation 2.3

```
function [sol] = RS_solver(H,g,A,b)
        % RS_SOLVER For problem 2
        v = g;
        H_A = A'*A;
        lambda = H_A\(b + A'*v);
        x = H*(A*lambda - g);
        sol = [x;lambda];
end
```

Question 9 Evaluate the performance of your QP-solvers based on LU, LDL, Null-Space, and Range-Space factorizations by plotting the cputime as function of problem size (say in the range $n = 10 - 1000$). Comment on the results.

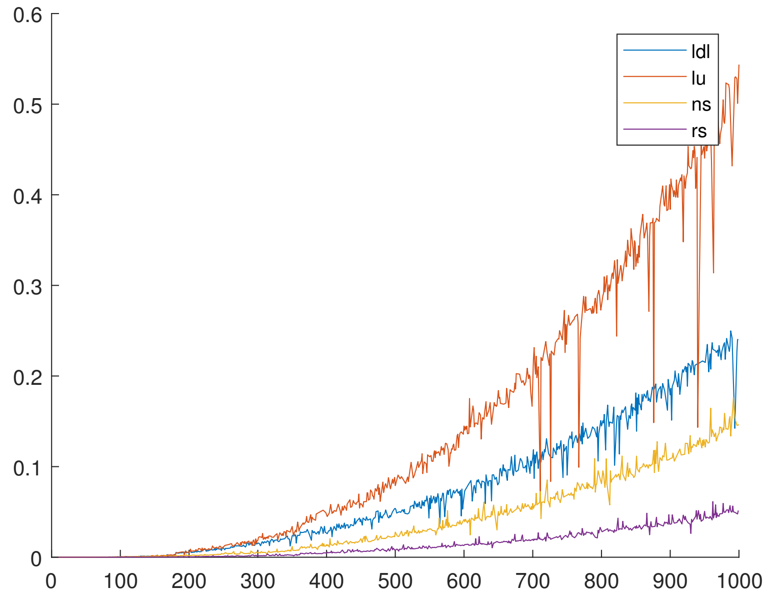


Figure 2.2: From top to bottom: LU; LDL; NS; RS

Question 10 Let $n = 100$ and plot the sparsity pattern of the KKT-matrix.

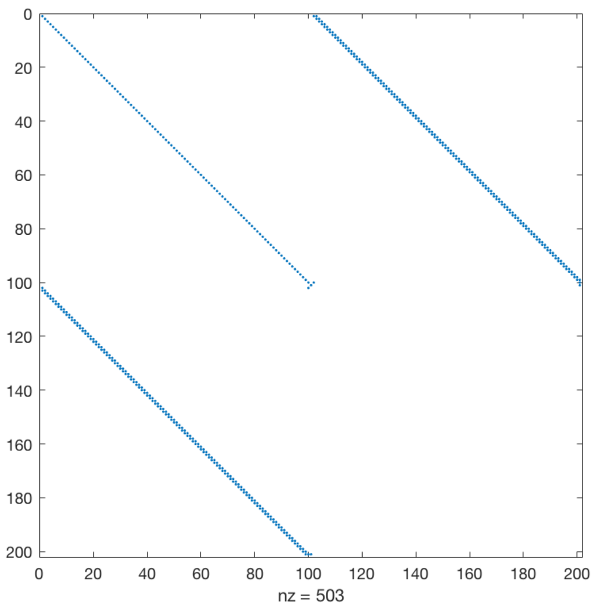


Figure 2.3: The blue lines are where there is density in the matrix.

Question 11 Make a function that treats the system as a sparse system (see sparse) using an LU-factorization. Evaluate the performance (cputime) of this solver as function of problem size ($n = 10 - 1000$). Comment on the results.

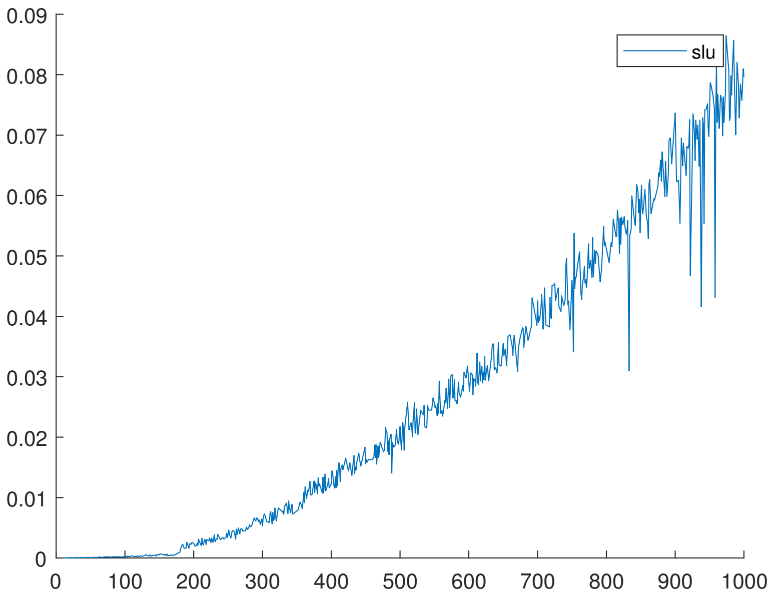


Figure 2.4

Question 12 Make a function that treats the system as a sparse system (see sparse) using an LDL-factorization. Evaluate the performance (cputime) of this solver as function of problem size (n=10-1000). Comment on the results.

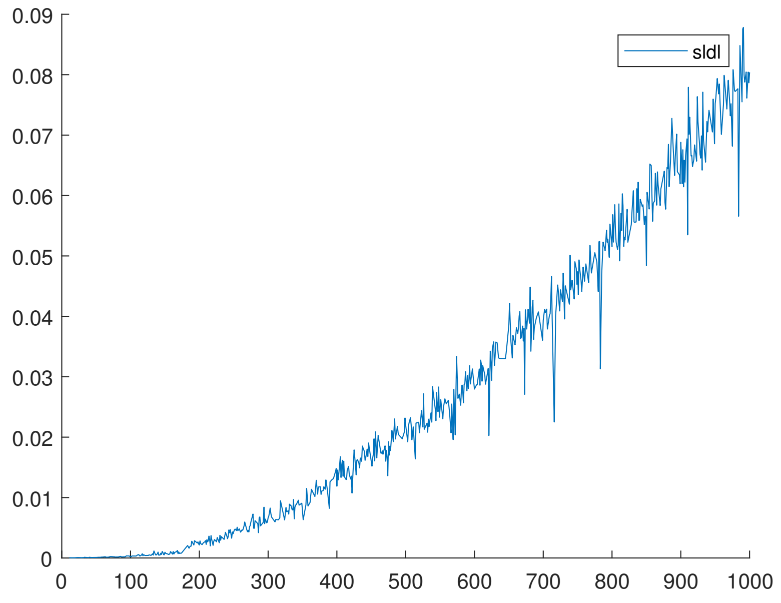


Figure 2.5

Question 13 Discuss the performance of the sparse solvers compared to the dense solvers.

As seen in the figure, there is not a big difference between the sparse and dense solvers when we have small matrices. But as the matrices grow larger, the sparse solvers are much faster than the dense. With matrices being 1000x1000, then we see that the sparse solvers are around 0.08 seconds in solving, while the dense LDL solver is around 0.2 seconds and the dense LU is the slowest with being so slow, that it takes 0.6 seconds to find a solution.

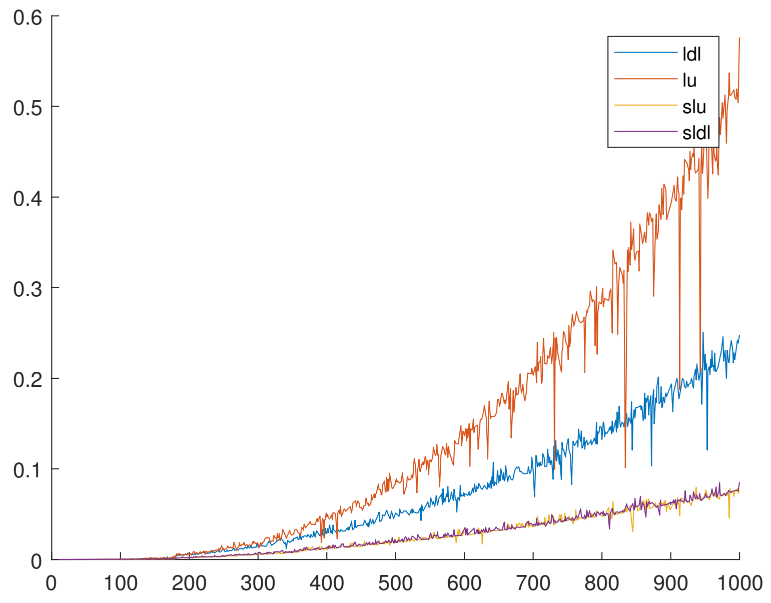


Figure 2.6: From top to bottom: LU; LDL; Sparse LDL; Sparse LU (Note that sparse LU and Sparse LDL are laying a top of eachother)

In the last figure, we have made it easier to compare all of the different methods compared to each other.

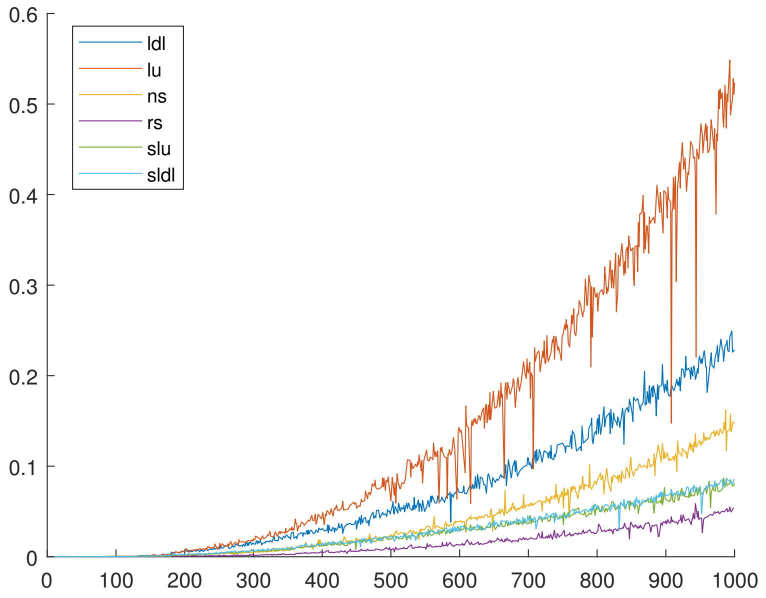


Figure 2.7: From top to bottom: LU; LDL, NS, Sparse LDL; Sparse LU (Note that sparse LU and Sparse LDL are laying a top of eachother); RS.

Problem 3

Inequality Constrained QP

Consider the QP in Example 16.4 (p.475) in Nocedal and Wright.

Question 1 Make a contour plot of the problem.

Below is the contour plot of the problem. Please refer to the file `task3.m` for the actual MatL´+09 ab code that generated the figure.

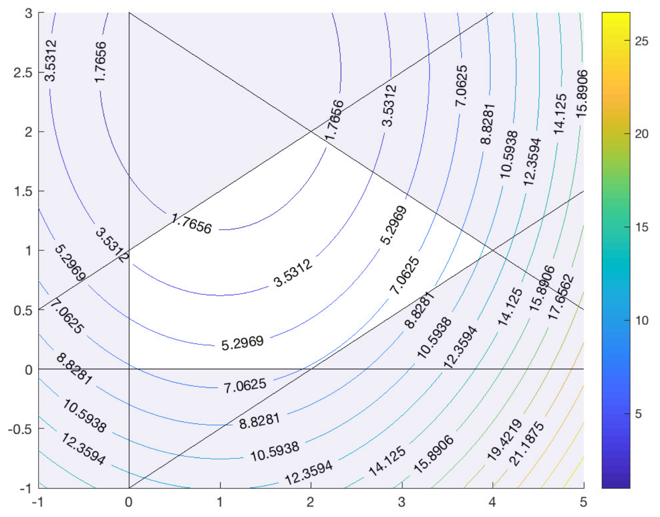


Figure 3.1: Contour plot of the problem

Question 2 Write the KKT-conditions for this problem.

The problem has the from:

$$\begin{aligned}\min_x \quad & q(x) = (x_1-1)^2 + (x_2-2.5)^2 \\ \text{s.t.} \quad & x_1 - 2x_2 + 2 \geq 0 \\ & -x_1 - 2x_2 + 6 \geq 0 \\ & -x_1 + 2x_2 + 2 \geq 0 \\ & x_1, x_2 \geq 0\end{aligned} \tag{3.1}$$

Rewriting the problem to

$$\begin{aligned}\min_x \quad & q(x) = \frac{1}{2}x^T Hx + g^T x + \gamma \\ \text{s.t.} \quad & Ax = b\end{aligned} \tag{3.2}$$

and thus

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad g = \begin{bmatrix} -2 \\ -5 \end{bmatrix} \quad \gamma = 0 \tag{3.3}$$

$$A = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 \\ -2 & -2 & 2 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \end{bmatrix} \tag{3.4}$$

Firstly the lagraian is defined, that is:

$$\mathcal{L}(x,\lambda) = \frac{1}{2}x^T Hx + g^T x + \lambda^T (A^T x - b), \tag{3.5}$$

where

$$\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{bmatrix} \tag{3.6}$$

Now the first order conditions can be found by differentiation of the Lagrangian function $\mathcal{L}$, and as all constraints in (3.1) are given as inequality constraints all the Lagrangian variables are $\lambda_i \geq 0$.

$$\nabla\mathcal{L}(x,\lambda) = Hx + g^T - A\lambda \tag{3.7}$$

and conditions

$$\begin{aligned} Ax - B &\geq 0 \\ \lambda(Ax - b) &= 0 \\ \lambda &\geq 0 \end{aligned} \tag{3.8}$$

The second order conditions are $h'\nabla_{xx}\mathcal{L}h \geq 0, \forall h \neq 0$ and $\nabla_x c_i(x)h = 0, \forall i \in A(x)$.

Question 3 Argue that the KKT-conditions are both necessary and sufficient optimality conditions.

In this case H is positive definite as H is a multiple of the identify matrix, and furthermore $\nabla_{xx}\lambda^T(A^Tx - b) = 0$, due to all constraints being linear. As H is positive definite then so is $\mathcal{L}$ and convexity of the problem is ensured and therefor any local minimizer is also a global. Therefore the KKT conditions are both necessary and sufficient to find a global minimizer.

Question 4 Make a function for solution of convex equality constrained QPs (see Problem 1 and Problem 2).

In part 2 there has been implemented various different was of solving an Equality constraint QP problem where the pattern of the structure of the problem is known, and therefor it was possible to implement solvers that utilized this. In this problem 3.1 it has been shown that again H has a equivalent pattern to the problems solved in part 2, yet as the matrix A is highly dense there is a space solver can not be used and therefor the dense Range space solver will be most applicable for the this problem. As the is not large choosing any of the other dense solvers would not penalize the runtime so much.

Question 7 Explain the active-set method for convex QPs listed on p. 472 in N&W

The concept of the Active Set algorithm is very close to the simplex algorithm used in the linear case, yet with small but significant changes. As the objective function and constraints are not ensured to be linear any more, the iterations from the simplex are not sufficient. In the QP active set method active inequality constraints are also chosen, yet in contrast to the simplex the iterations can move outside the feasible region.

Initially a basis feasible solution is found where all equality constraints are imposed and we lie within the feasible region. In many cases it would be setting all variables to zero, $x = 0$. Now the iterative process begins where equality constraints remain yet a subset of the inequality constraints are added as equality constraints. This set is called the active set and denoted W_k, k noting the iteration. Now the problem becomes, and is called the subproblem:

$$\begin{aligned} &\min_p \frac{1}{2}p^T H p + g_k^T p + \gamma \\ &s.t. \quad a_i^T p = 0 \quad i \in W_k \end{aligned} \tag{3.9}$$

the solution to this problem is called p_k and$g_k = Hx_k + g$. Now a step $\alpha_k \in [0, 1]$ size is chosen such that the change in the objective while retaining feasibility for all constraints not in the working set. if $\alpha_k < 1$ then a new working set is constructed and the blocking constraint, i.e. the one that is constraint α_k is added to the working set. This pattern is continued until no more constraints can be added to the working set and the subproblem will have solution $p_k = 0$. Now the Lagrangian multipliers are checked for the KKT conditions if all are feasible then a optimal solution has been found to the original problem. If on the other hand not all KKT conditions are satisfied the constraint corresponding to the multiplier is dropped from the subproblem and the iterative process is resumed.

Question 5 Apply a conceptual active set algorithm to the problem. Use the iteration sequence in Figure 16.3 of Nocedal and Wright. Plot the iterations sequence in your contour plot. For each iteration (guess of working set) you should list the working set, the solution, x, and the Lagrange multipliers, λ.

Initializing in $x = (2, 0)$, this point lies on the constraints 3 and 5, therefor the working set is set to $W_0 = \{3, 5\}$. From this point the subproblem $p_0 = 0$ and therefor the testing of multipliers is initialed. Here it is found that $\lambda_3 = -1$ and $\lambda_5 = -2$. This shows that there are two different directions where the objective function can be decreased. The λ are an expression of big a degree a one unit change on the right hand side will effect the objective function, in this case the constraint will be removed but the effect will be the same, it is to not that it may not be possible to actualize this change as other constrains can become active in the general case. Now as the direction of λ_5 will yield the largest change in objective constraint five is chosen to leave the active set and the iteration process is resumed. α_1 is found to be one as no blocking constraint is present, and therefor no constraints are added to the working set and the new location is $x_1 = (1, 0)$.

Now in iteration 2 the current working set remains the same $W_2 = W_1 = \{5\}$ but $p_2 = 0$ and therefor the recalculation of the $\lambda_i | i \in W_2$ is necessary. Here λ_5 is found to be -5 and therefor is dropped from the working set. Third iteration is started by solving the unconstrained problem, α_3 is found to be 0.6, and the blocking constraint is constraint 1. x is found to be $x_4 = (1, 1.5)$ and the working set is $W_4 = \{1\}$.

The final step length is found to be one as well and therefor no constraints are added to the working set. $x_5 = (1.4, 1.7)$ and $p_5 = 0$, therefor the multipliers are calculated and found to be $\lambda_1 = 0.8$ i.e. strictly positive and therefor the found $x_5 = x^*$ is the optimal solution. Figure 3.2 shows the iteration process in the contour plot and starting point is $x = (2, 0)^T$ as in the exsample.

3.1. Question 8 Use `linprog` to compute a feasible point to a QP. Apply and test this procedure to the problem in Example 16.4.

Applying the method noted as "phase 1" on p. 472 in Nocedal & Wright we have given $\tilde{x}$, a initial estimate of a good choose based on knowledge of the QP. The guess is not needed to be

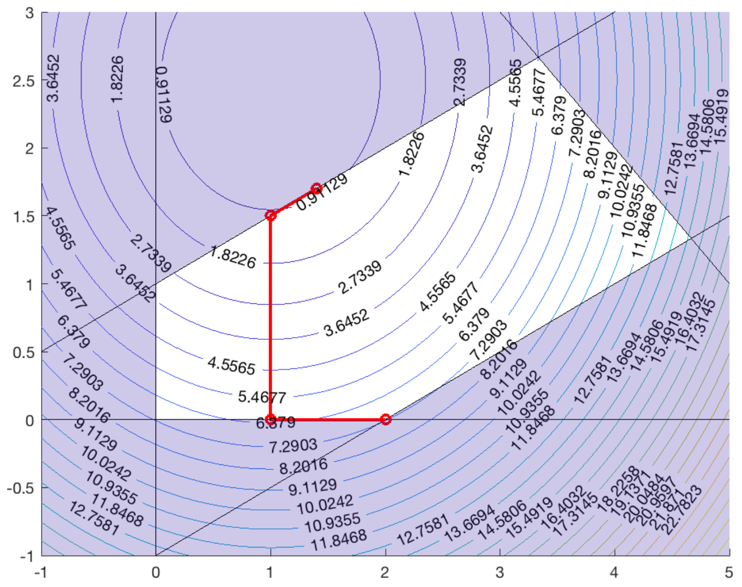


Figure 3.2: Plot over iterations

freable the following will correct any in feasibility. Given the $\tilde{x}$, the following problem is solved:

$$\begin{aligned} &\min_{x,z} e^T z \\ &s.t. a_I^T x + \gamma_i z_i = b_i \quad i \in \mathcal{E} \\ &\quad a_I^T x + \gamma_i z_i \geq b_i \quad i \in \mathcal{I} \\ &\quad x \in X \\ &\quad z \geq 0 \end{aligned} \tag{3.10}$$

where $e = (1, 1, ..., 1)^T, \gamma_i = -sign(a_i^T x_i - b_i), \forall i \in \epsilon, \gamma_i, i \in I$. In this case if the initial guess is feasible then the solution to the above problem will be $(\tilde{x}, 0)$.

Now as the initial feasible solution has been found then the algorithm as described above can be initialized. The stepping is calculated by $_{k+1} = x_k + \alpha_k p_k$ and α_k is calculated by,

$$\alpha_k = \min_{i \notin W_k, a_i^T p_k < 0} \{1, \frac{b_i - a_i^T x_k}{a_i^T p_k}\} \tag{3.11}$$

Now all sub calculations have been presented the formal algorithm can be formulated:

Algorithm 1 Active-Set Method for Convex QP

```
1: Compute a feasible starting point $x_0$, use 3.10;
2: $W_0 \leftarrow W_0 \subset A(x_0)$ Initialize $k = 0$
3: while True do
4:     Solve 3.9 to find $p_k$
5:     if $p_k = 0$ then
6:         if $\lambda_i \geq 0, \forall i \in W_k \cap I$ then
7:             then Optimal solution is $x* = x_k$
8:             break
9:         else
10:            $W_{k+1} = W_k \setminus arg\min\{\lambda_i, \forall i \in W_k \cap I\}$
11:            $x_{k+1} = x_k$
12:        end if
13:    else
14:        Compute $\alpha_k$ from 3.11
15:        $x_{k+1} = x_k + \alpha_k p_k$
16:        if $\alpha_k < 1$ then
17:            add one of the blocking constraints to $W_k$
18:            $W_{k+1} = W_k \cup i$
19:        else
20:            $W_{k+1} = W_k$
21:        end if
22:    end if
23: end while
```

Now inputting the initial guess $\tilde{x} = (1, 2.5)$ into phase 1 the solution to the unconstrained QP the initial starting point is $x = (2, 2)$. From this point the Active set W_0 is set to the constraints active in x_0 i.e. $W_0 = 1, 2$.

Question 9 Implement the algorithm in p. 472 and test it it for the problem in Example 16.4. Print information for every iteration of the algorithm (i.e. the point x_k, the working set W_k, etc) and list that in your report.

From here the subproblem is solved and found to be zero, such that the KKT conditions are calculated and found to be $\lambda_1 = 1.25$ and $\lambda_2 = -0.75$ and thus $i = 2$ will be removed from the working set W_0. Now $W_1 = W_0 \setminus 2 = \{1\}$. α is found to be 0.67 and $x_2 = (1.4, 1.7)^T$. Finally $P_2 = 0$ and all KKT conditions are upheld. using this method has now yielded in only two iterations compared to the previous 5. Figure 3.3 shows the iteration process using the 'Phase 1' method together with the initial guess from the unconstrained problem.

The implementation of the active set Method has been done in Python for this exercise and the code follows on the following page.

Table 3.1: Iterations carried out with active set method and $\tilde{x} = (1, 2.5)^T$

k	x$_k$	$\lambda_i, i \in W_k$	W_k
0	(2,2)		$\{\emptyset\}$
1	(2,2)	-0.75	$\{1\}$
2	(1.4,1.7)	0.8	$\{1\}$

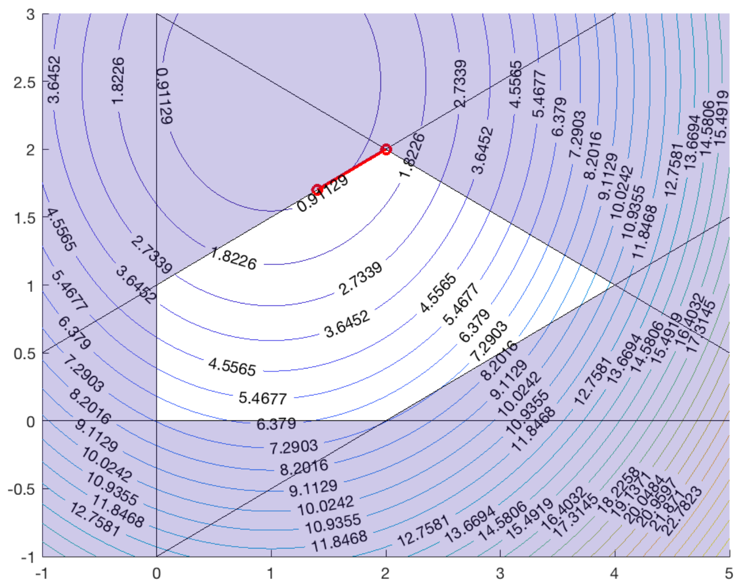


Figure 3.3: Plot over iterations using 'Phase 1' method

```
def inital(xtilde,A,b):
    z=-np.sign(A.T@xtilde-b.T[0])
    n,m = A.shape
    Ainint = np.concatenate([A.T, np.diag(z)],axis=1);
    f = np.concatenate([np.zeros((1,n)), np.ones((1,m))],axis =1);
    sol =scipy.optimize.linprog(f[0], A_ub=Ainint,b_ub=b);
    return np.array( [[i] for i in sol.x[:len(xtilde)]])

def ActiveSet(H,g0,A,b,xtilde,solver = 'rs', tol=1e-5):
    x=[];p=[];g=[];W = [[]];l = [];k = 0;
    x.append(inital(xtilde,A,b))
    p=[];g=[];W = [[]];l = [];k = 0;
    x = [inital(xtilde,A,b)]

    W.append([i for i in range(len(A.T)) if A.T[i]@[2,2]-b[i]==0])
```

```
while True:
    sol = EqualityQPSolver(H, H@x[k]+g0, A[:,W[k]],
     ↪  np.zeros((len(W[k]),1,)), N=3,method=solver)
    p.append(sol[:len(xtilde)])
    l.append(sol[len(xtilde):])
    if all(p[k] == 0):
        print(l[k])
        if all(l[k]>=-tol):
            break
        else:
            tempw = W[k].copy()
            x.append(x[k].copy())
            tempw.pop(np.argmin(l[k]))
            W.append(tempw)# = copy(W[k])
    else:
        alpha = min([(b[i]-A[:,i].T@x[k] )/(A[:,i].T@p[k]) if
         ↪  A[:,i].T@p[k]<0 else 1 for i in range(len(A)) if i not in
         ↪  W[k]])
        x.append(x[k] +alpha*p[k])
        if alpha<1:
            tempw = W[k].copy()
            tempw.append([i for i in range(len(A.T)) if
             ↪  A.T[i]@x[k]-b[i]==0][0])
            W.append(tempw)
        else:
            W.append(W[k])
    k+=1
    if k>10:
        break
return x, p,l,alpha,W
```

Question 10 Test your active set algorithm for the problem in Example 16.4 using (16.47) and (16.48) in N&W. Do the same using `quadprog`.

An alternative approach is the big M method. In this method the phase 1 is removed and a measure of feasibility is added in its place. A constant M sufficiently large is added to the problem together with a variable ν. This method utilized the l_∞ norm.

The problem now becomes

$$\begin{aligned}
&\min_{x,\nu} \frac{1}{2} x^T H x + g^T x + M\nu \\
&s.t. (a_i^T x - b_i) \le \nu \quad i \in \mathcal{E} \\
&\quad -(a_i^T x - b_i) \le \nu \quad i \in \mathcal{E} \\
&\quad b_i - a_i^T x \le \nu \quad i \in \mathcal{I} \\
&\quad 0 \le \nu
\end{aligned} \tag{3.12}$$

Now rewriting this into matrix form, and omitting equality constraints as non exist in this problem

$$\begin{aligned}
\min_{x,\nu} \quad & \frac{1}{2} \begin{bmatrix} x \\ \nu \end{bmatrix}^T \begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \nu \end{bmatrix} + \begin{bmatrix} g \\ M \end{bmatrix}^T \begin{bmatrix} x \\ \nu \end{bmatrix} \\
\text{s.t.} \quad & \begin{bmatrix} A & 0 \\ e^T & 1 \end{bmatrix} \begin{bmatrix} x \\ \nu \end{bmatrix} - \begin{bmatrix} b \\ 0 \end{bmatrix} \ge 0 \quad i \in \mathcal{I}
\end{aligned} \tag{3.13}$$

where $e = [1, 1, ..., 1]^T$. The problem increases with one dimension in this case but now we can start the problem with an infeasible x. Though choosing x with some prior knowledge to the problem simplifies the iterative process.

There also exists a variant of the Big-M method that instead penalizes on the l_1 norm and is as follows:

$$\begin{aligned}
&\min_{x,\nu} \frac{1}{2} x^T H x + g^T x + M e_{\mathcal{E}}^T (s + t) + M e_{\mathcal{I}}^T v \\
&\text{s.t.} a_i^T x - b_i + s_i - t_i = 0 \quad i \in \mathcal{E} \\
&\quad b_i - a_i^T x + v_i \ge 0 \quad i \in \mathcal{I} \\
&\quad s, t, v \ge 0
\end{aligned} \tag{3.14}$$

Where $e_{\mathcal{E}}^T = (1, 1, \ldots, 1)^T$ of length $|\mathcal{E}|$ and equivalently for $e_{\mathcal{I}}^T$.

This method removed the ν and in its place adds slack (s and v) and surplus variables (t) to the model. These variables absorb all the feasibility in the system much in the same way ν did in the previous example.

Now rewriting this into matrix form, and omitting equality constraints as none exist in this problem

$$\begin{aligned} &\min_{x,s,t,v} \frac{1}{2} \begin{bmatrix} x \\ \nu \end{bmatrix}^T \begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \nu \end{bmatrix} + \begin{bmatrix} g \\ Me_{\mathcal{I}}^T \end{bmatrix}^T \begin{bmatrix} x \\ v \end{bmatrix} \\ &s.t. \begin{bmatrix} A & 0 \\ I & I \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} - \begin{bmatrix} b \\ 0 \end{bmatrix} \geq 0 \quad i \in \mathcal{I} \end{aligned} \tag{3.15}$$

Test of Alternative methods In the test of the alternative methods the matrix H now becomes singular in both cases as there has been added a zero row and column. This con though be handled by the 'backlash' operator in Matlab. Noting that the previous was implemented in Python, to implement the Active set now with the big-M Matlab code was necessary, the code has been added to the appendix.

The iterations for both alternative methods find the same path and iterate from $x_0 = (0,0)^T$,$x_0 = (0,1)^T$ and to the solution, $x_3 = (1.4, 1.7)^T$. The solutions given are both very close the the solution by `quadprog` yet big-M method using the l_∞ norm gives a more similar solution to the one by `quadprod`. the Difference are highlighted by the following:

$$||x^*_{l_1} - x^*_{quadprog}||= 6.8212e - 09 \tag{3.16}$$

$$||x^*_{l_\infty} - x^*_{quadprog}||= 5.7865e - 13 \tag{3.17}$$

As we can see here they are both very similar but it seems that the l_∞ is just a bit better (closer to the `quadprog` solution). This could lead us to believe that `quadprog` also uses this method.

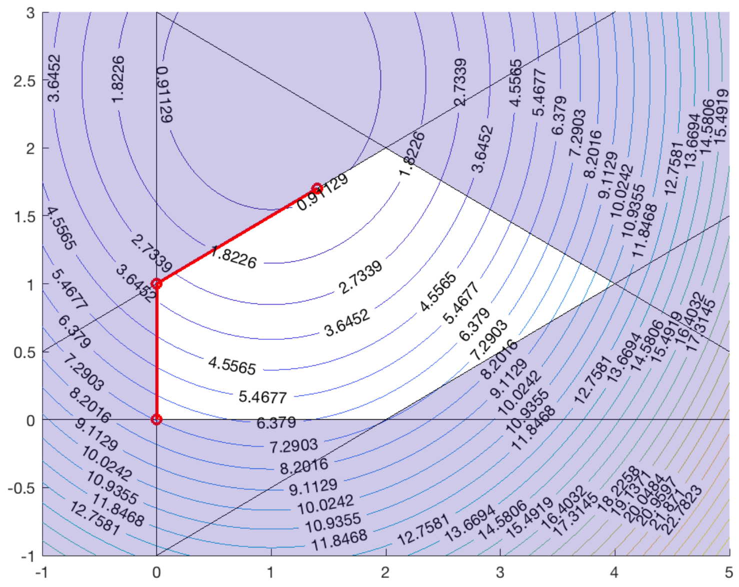


Figure 3.4: Plot over iterations using big-M method

This page is intentionally left blank.

Problem 4

Markowitz Portfolio

Question 1 For a given return, R, formulate Markowitz' Portfolio optimization problem as a quadratic program.

The Markowitz Portfolio problem is very well known and is the foundation for much of current portfolio theory. The problem solves the problem of incurring minimum risk at a given rate of return. This is also known as the minimum variance portfolio. The most basic formulations as follows,

$$\begin{aligned} &\min_x x^T \Sigma x \\ &s.t. Rx \geq \pi \\ &\quad 1^T x = 1 \\ &\quad x \geq 0 \end{aligned} \tag{4.1}$$

Where Σ is the variance-covariance matrix, R is the return, π is the minimum required rate of return and all wights in the portfolio must sum to one, and no shorting is permitted. Alternatively if shorting is permitted the final constraint is omitted.

Question 2 What is the minimal and maximal possible return in this financial market?

In the case of no shorting the minimum and maximum possible returns are 9.02 and 17.68, which corresponds to only investing in security four or five.

Under allowance of shorting the minimum remains the same but the maximum is unbounded, as this strategy will be achieved by shoring lower return securities and buying high.

Question 3 Use `quadprog` to find a portfolio with return, $R = 10.0$, and minimal risk. What is the optimal portfolio and what is the risk (variance)?

To achieve a return of 10 then the portfolio must be constructed with $x = (0, 0.2816, 0, 0.7184, 0.0)^T]$ and the risk (Variance) is the given objective or simply calculated by $x^T \Sigma x$ to be $\sigma = 2.0923$.

Note that a trader would never choose this portfolio as if a lower risk can be made by increasing the return (or replacing the wanted return such that $R \geq 10$) then the portfolio would be: $x = (0.088, 0.251, 0.282, 0.104, 0.275)^T$ with a return of 14.41 and variance 0.625.

Question 4 Compute the efficient frontier, i.e. the risk as function of the return. Plot the efficient frontier as well as the optimal portfolio as function of return.

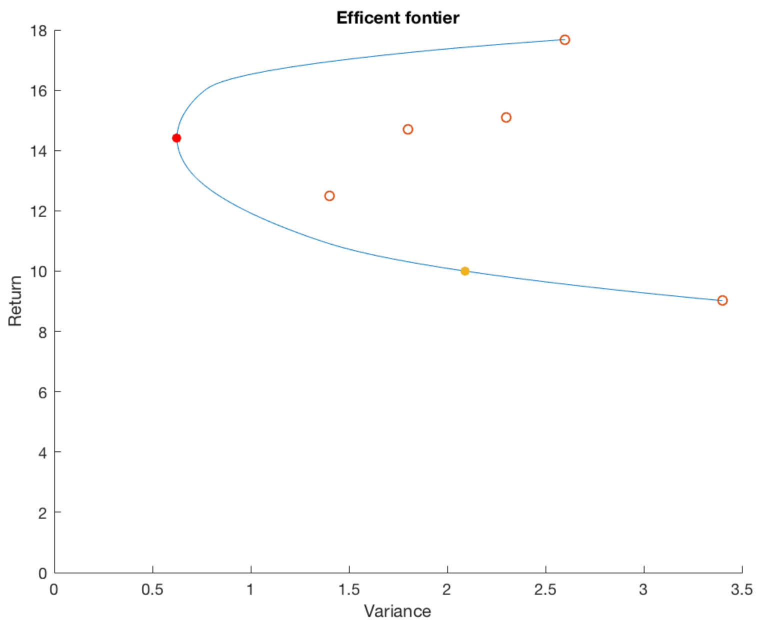


Figure 4.1: Efficient frontier of the portfolio

where the orange filled dot is the asked portfolio, the red is the minimum variance that is coincidental the same if $R \geq 10$ is imposed. finally all unfilled dots are the securities.

In the following we add a risk free security to the financial market. It has return $r_f = 2.0$.

Question 1 What is the new co-variance matrix and return vector.

Adding the risk free security to the problem now adds a row/column of zeros to the variance co-variance matrix and further appends the risk free return to the return matrix i.e.:

$$\Sigma_{+r_f} = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.4 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.8 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad R_{+r_f} = \begin{bmatrix} 15.10 \\ 12.50 \\ 14.70 \\ 9.02 \\ 17.68 \\ 0 \end{bmatrix} \tag{4.2}$$

Question 2-3 Compute the efficient frontier, plot it as well as the (return,risk) coordinates of all the securities. Comment on the effect of a risk free security. Plot the optimal portfolio as function of return.
What is the minimal risk and optimal portfolio giving a return of R = 15.00. Plot this point in your optimal portfolio as function of return as well as on the efficient frontier diagram.

The solutions found in exactly the same way as in previous part and has been found to be $x = (0.1655, 0.1365, 0.3115, 0.0266, 0.3352, 0.0247)$ and the risk of 0.6383.

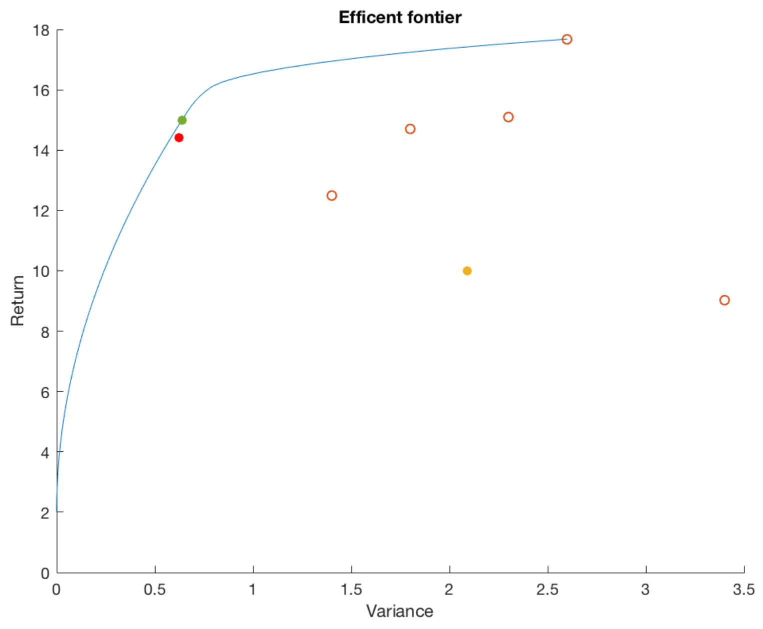


Figure 4.2: Efficient frontier of the portfolio.

Dots remain from previous plot, now green is equivalent to $R = 15$. where the orange filled dot is the asked portfolio, the red is the minimum variance that is coincidental the same if $R \geq 10$ is imposed. Finally all unfilled dots are the securities.

Problem 5

Interior-Point Algorithm for Convex Quadratic Programming

Question 1 Write an interior point algorithm on paper for solution of the convex quadratic program

$$\min_{x \in \mathbb{R}} \quad \phi = \frac{1}{2} x^\top H x + g^\top x \tag{5.1a}$$
$$\text{s.t.} \quad A^\top x = b \tag{5.1b}$$
$$C^\top x \geq d \tag{5.1c}$$

Question 2 Explain the Primal-Dual Interior Point Algorithm for convex QPs.

Based on the material learned in Lecture 6, and in particular slide 28[1] we present an overview of the interior-point algorithm and an explanation of it's workings effectively covering both **Question 1** (Algorithm 2) and **Question 2** of this problem.

First thing to note is that algorithm is iterative, and searches for the solution continuously until it converges. Convergence is defined as state where the gradient (residuals) is below a given small value ϵ. Naturally, a smaller ϵ results in a more precise solution.

The algorithm starts from a given starting location: x_0, y_0, z_0 and s_0 by computing the residuals. Then it iterates, and for each step it tries to come closer to the solution:

[1] of the `QuadraticOptimization - Interior-Point Algorithm.pdf`

Algorithm 2 Interior-point algorithm for solution of Convex QP

```
1: Given an input $H$, $g$, $A$, $b$, $C$, $d$ and starting point $x_0$, $y_0$, $z_0$, $s_0$
2: Compute the residuals $r_L, r_A, r_C, r_{sz}$ and duality gap $\mu$
3: while not converged do
4:     Compute the affine step direction $\Delta x^{\text{aff}}$, $\Delta z^{\text{aff}}$ and $\Delta s^{\text{aff}}$
5:     Compute the affine step size $\alpha^{\text{aff}}$
6:     Compute the affine duality gap $\mu^{\text{aff}}$
7:     Compute centering parameter $\sigma$
8:     Compute affine-centering-correction direction
9:     Compute the step size $\alpha * \eta$ and take the actual step by updating $x, y, z$ and $s$.
10:    Re-compute the residuals $r_L, r_A, r_C, r_{sz}$ and duality gap $\mu$.
11:    Check for convergence and stop if we did.
12: end while
```

For this algorithm, a starting location can be chosen arbitrary (NB: $s_0 > 0$, $z_0 > 0$) as it does not need to be in the feasible region. Additionally, as the problem is convex (H is positive semidefinite), we are guaranteed to not accidentally start in a local minima and miss a solution.

However, if one wishes to optimize this step, a heuristic for initial point is given on Slide 29, where a good value of s_0 and z_0 is computed by calculating an affine search direction.

Key idea of the algorithm are the slack variables introduced for the inequality constraints (s) and their relation to corresponding Lagrangian multipliers z. Simultaneously minimizing both and closing the duality gap brings the algorithm closer to the solution.

The actual direction is found with the Newton method, however as Newton will tend to move to the boundary where either s or z is zero, a correction is needed. This correction will guide the search back to the line $z = s$ (central path) which effectively will speed up the search process, as Newton will be taking larger steps. The correction is guided by the $\sigma \in [0; 1]$, where in case of a good affine duality gap, sigma is close to zero.

To summerize: an affine step is computed first. Then the step is corrected such that we stay on the interior path (hence the name of the algorithm), residuals are recomputed and convergence is checked.

Finally we note that computational beauty of this method lays in the fact that even though an expensive LDL-factorization is computed at every step, here we are able to reuse the computed factors when doing the actual step speeding up the computation considerably.

Question 3 Implement the Primal-Dual Interior-Point Algorithm for this convex quadratic program

Please see the actual implementation of the algorithm in the Appendix, section 6.4

Question 4 What is H, g, A, C, b, and d for the Markowitz Portfolio Optimization Problem with $R = 15$ and the presence of a risk-free security?

For the Markowitz Portfolio Optimization Problem with an expected return of $R = 15$, the CQP program values are as follows:

$$H = \begin{bmatrix} 2.3 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.4 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.8 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.4 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad g = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad A = \begin{bmatrix} 1.0 & 15.1 \\ 1.0 & 12.5 \\ 1.0 & 14.7 \\ 1.0 & 9.02 \\ 1.0 & 17.68 \\ 1.0 & 2.0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \end{bmatrix} \quad b = \begin{bmatrix} 1.0 \\ 15.0 \end{bmatrix} \quad d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

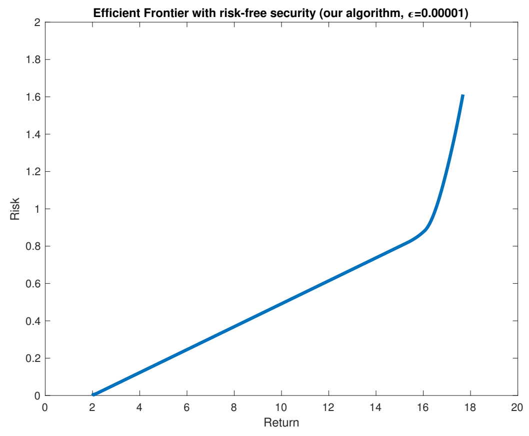
Question 5 Test this algorithm on the Markowitz Portfolio Optimization Problem, i.e. compute the efficient frontier and optimal portfolio for the situation with a risk-free security. Do your algorithm give the same solution as `quadprog`?

Following are the results for the optimal portfolio given a desired return of $R = 15$. Those are obtained by our algorithm (with $\epsilon = 0.001$) and Matlabs `quadprog`(with `optimoptions` = `'Algorithm','interior-point-convex'`):

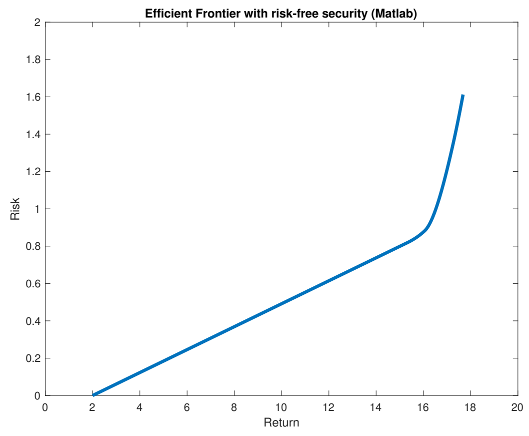
$$\texttt{x\_ours} = \begin{bmatrix} 0.1669 \\ 0.1336 \\ 0.3116 \\ 0.02572 \\ 0.3363 \\ 0.0259 \end{bmatrix} \quad \texttt{x\_matlab} = \begin{bmatrix} 0.1655 \\ 0.1365 \\ 0.3115 \\ 0.02663 \\ 0.3352 \\ 0.02468 \end{bmatrix}$$

As expected, by tweaking the ϵ value in our algorithm, we see that results are getting closer to the solution found by Matlab. With $\epsilon = 0.0001$ the results become the same to the fourth decimal point.

The efficient frontier calculated by the two algorithms is presented in figure 5.1 below:



(a) Our algorithm $\epsilon = 0.00001$

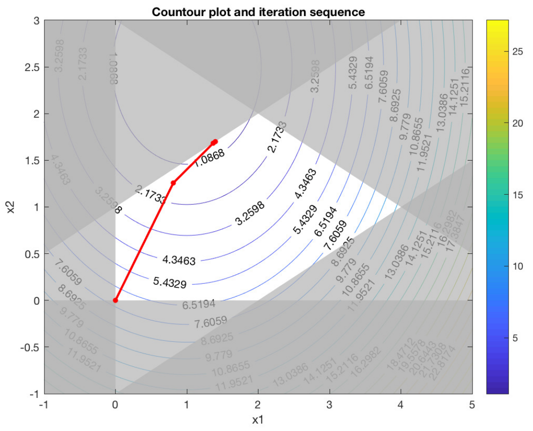


(b) Matlab built-in quadprog

Figure 5.1: The efficient frontier

Question 6 Apply the algorithm to the quadratic program in Problem 3 (i.e. the problem in Example 16.4). Plot the iteration sequence in the contour plot.

Following figure 5.2 shows the visualization of our algorithm applied to Problem 3 with two different initial starting points x_0. In both cases algorithm converged to a solution $x = [1.4, 1.7]^\top$ where the objective function have value 0.8.



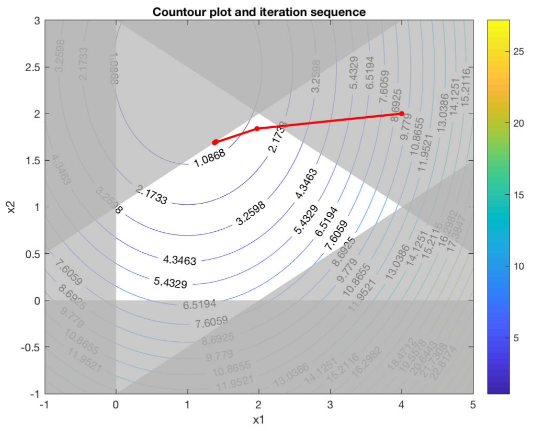


Figure 5.2: Problem 3 solution visualized

6

Appendix

Remarks to using python vs. Matlab In this report we chose to try to use Python as the main language for solving all programming parts, this was done as a challenge and an attempt to use other programs than MalLab. We where able to solve the first and most of the second question without the major hick-ups. Yet LDL whose not implemented in any current version of Scipy and therefor found beta version of an implementation. Furthermore, the usage of the "\" backlash operator in Matlab is comparably difficult to implement in Python and we ended up defining two functions that cover the most basic operations from the backslash operator, the forward and backward substitution. This was not enough in most cases and furthermore using sparse matrixes we where forces to do everything in Matlab.

6.1. Active set Algorithm (Python)

```
import scipy, numpy as np
import scipy.optimize
def EqualityQPSolver(H, g, A, b, N=5, method="inv"):
    KKT = np.concatenate([H, -A.T], axis=0)
    temp = np.concatenate([-A, np.zeros((A.T@A).shape)])
    KKT = np.concatenate([KKT, temp], axis=1)
    Z = [0, 0]
    d = -np.concatenate([g, b])
    if method.lower() == "inv":
        #try:
        sol = np.linalg.inv(KKT)@d
    if method.lower() == "pinv":
        #try:
        sol = np.linalg.pinv(KKT)@d
        #except Exception:
        #    print('Singular Matrix')
        #    return None, None
    elif method.lower() == "ldl":
        L, D, perm = ldl(KKT)
        #Linv = np.linalg.inv(L)
        sol = bS(L.T,(bS(D,fS(L,d[perm,:]))))#Forward sub/backward
        #sol = Linv.T@(np.linalg.inv(D)@Linv@d[perm, :])
    elif method.lower() == "lu":
        from scipy.linalg import lu
        p, L, U = lu(KKT)
        #sol = np.linalg.inv(U)@(np.linalg.inv(L)@(d.T@p).T)
        sol = bS(U,(fS(L,(d.T@p).T)))
    elif method.lower() == "rs":
        from scipy.linalg import cholesky
        L = cholesky(H, lower=True)
        Linv=np.linalg.inv(L)
        Hinv = Linv.T@Linv
        v = Hinv@g
        H_A = A.T@Hinv@A
        _lambda = np.linalg.inv(H_A)@(b + A.T@v)
        x = Hinv@(A@_lambda - g)
        sol = np.concatenate([x, _lambda], axis=0)
    elif method.lower() == "rs_eye":
        from scipy.linalg import cholesky
        L = cholesky(H, lower=True)
        v = g*1/eye
        H_A = A.T@A*1/eye
        _lambda = np.linalg.inv(H_A)@(b + A.T@v)*1/eye
        x = A@_lambda - g
        sol = np.concatenate([x, _lambda], axis=0)
    elif method.lower() == "ns_slow":
        Y = (np.linalg.pinv(A)@np.identity(len(A))).T
        Z = nullSpace(A)
        AYinv = np.linalg.inv(A.T@Y)
        x_Y = AYinv@b
        x_Z = np.linalg.inv(Z.T@H@Z)@(-Z.T@(H@Y@x_Y + g))
        x = Y@x_Y + Z@x_Z
        _lambda = AYinv@Y.T@(H@x + g)
        sol = np.concatenate([x, _lambda], axis=0)
    elif method.lower() == "ns":
        Q,Rbar = np.linalg.qr(A,mode='complete')
        m1 = len(Rbar)
        Q1=Q[:,0:m1-1]
        Q2=Q[:,m1-1:]
        R = Rbar[:m1-1][:m1-1]
        x_Y = fS(R.T,b)
        x_Z = bS((Q2.T@H@Q2),(-Q2.T@(H@Q1@x_Y+g)))
        x = Q1@x_Y+Q2@x_Z
        _lambda = bS(R,Q1.T@(H@x+g))
        sol = np.concatenate([x, _lambda], axis=0)
    return np.round(sol,N)


def nullSpace(A, atol=1e-13, rtol=0):
    A = np.atleast_2d(A.T)
    u, s, vh = scipy.linalg.svd(A)
    tol = max(atol, rtol * s[0])
    nnz = (s >= tol).sum()
    ns = vh[nnz:].conj().T
    return ns / ns.max()


def HgAb(n, ubar, d0):
    H = np.identity(n + 1)
    g = -2 * ubar * np.ones((n+1, 1))
    b = np.zeros((n, 1))
    b[0, 0] = -d0
    A = np.concatenate([np.zeros((1, n-1)),np.identity(n-1)], axis=0)
    A = np.concatenate([A,np.zeros((n, 1))], axis=1) - np.identity(n)
    A = np.concatenate([A,np.zeros((n, 1))], axis=1)
    A[0, n-1],A[n-1,n]=1,-1
    A = A.T
    return H, g, A, b


def tester(solver, rangeN):
    ubar = random.randint(-100, 100)
    d0 = random.randint(-100, 100)
    N = random.randint(*rangeN)
    H, g, A, b = HgAb(N, ubar, d0)
    # print("H={}, g={}, A={} and B={}".format(H,g,A,b))
    start = time.time()
    EqualityQPSolver(H, g, A, b, method=solver)
    # getattr(func)(H,g,A,b,solver)
    end = time.time()
    runtime = end - start
    return runtime, N

def fS(L,b):
    N= len(b)
    x = np.array([[0.0] for i in range(N) ])
    x[0] = b[0]/L[0][0]
    for i in range(1,N,1):
        x[i][0]=(b[i]-x[0:i].T@L[i][0:i])/L[i][i] if L[i][i]!=0 else 0
    return x

def bS(U,b):
    N= len(b)
    x = np.array([[0.0] for i in range(N) ])
    for i in range(N-1,-1,-1):
        x[i] = ( b[i] - U[i,:]@x)/U[i][i]
    return x

def inital(xtilde,A,b):
    z=-np.sign(A.T@xtilde-b.T[0])
    n,m = A.shape
    Ainint = np.concatenate([A.T, np.diag(z)],axis=1);
    f = np.concatenate([np.zeros((1,n)), np.ones((1,m))],axis =1);
    sol =scipy.optimize.linprog(f[0], A_ub=Ainint,b_ub=b);
    return np.array( [[i] for i in sol.x[:len(xtilde)]])

def ActiveSet(H,g0,A,b,xtilde,solver = 'rs', tol=1e-5):
    x=[];p=[];g=[];W = [[]];l = [];k = 0;
    x.append(inital(xtilde,A,b))
    p=[];g=[];W = [[]];l = [];k = 0;
    x = [inital(xtilde,A,b)]

    W.append([i for i in range(len(A.T)) if A.T[i]@[2,2]-b[i]==0])
    while True:
        sol = EqualityQPSolver(H, H@x[k]+g0, A[:,W[k]],
        ↪  np.zeros((len(W[k]),1,)), N=3,method=solver)
        p.append(sol[:len(xtilde)])
        l.append(sol[len(xtilde):])
        if all(p[k] == 0):
            print(l[k])
            if all(l[k]>=-tol):
                break
            else:
                tempw = W[k].copy()
                x.append(x[k].copy())
                tempw.pop(np.argmin(l[k]))
                W.append(tempw)# = copy(W[k])
        else:
            alpha = min([(b[i]-A[:,i].T@x[k] )/(A[:,i].T@p[k]) if
            ↪  A[:,i].T@p[k]<0 else 1 for i in range(len(A)) if i not in
            ↪  W[k]])
            x.append(x[k] +alpha*p[k])
            if alpha<1:
                tempw = W[k].copy()
                tempw.append([i for i in range(len(A.T)) if
                 ↪  A.T[i]@x[k]-b[i]==0][0])
                W.append(tempw)
            else:
                W.append(W[k])
        k+=1
        if k>10:
            break
    return x, p,l,alpha,W
```

6.2. myldl (Python)

```
from __future__ import division, print_function, absolute_import

from warnings import warn

import numpy as np
from numpy import (atleast_2d, ComplexWarning, arange, zeros_like, imag, diag,
                   iscomplexobj, tril, triu, argsort, empty_like)

from scipy.linalg.decomp import _asarray_validated
# from lapack import get_lapack_funcs, _compute_lwork
from scipy.linalg.lapack import get_lapack_funcs, _compute_lwork

__all__ = ['ldl']


def ldl(A, lower=True, hermitian=True, overwrite_a=False, check_finite=True):
    """ Computes the LDLt or Bunch-Kaufman factorization of a symmetric/
    hermitian matrix.

    This function returns a block diagonal matrix D consisting blocks of size
    at most 2x2 and also a possibly permuted unit lower triangular matrix
    ``L`` such that the factorization ``A = L D L^H`` or ``A = L D L^T``
    holds. If ``lower`` is False then (again possibly permuted) upper
    triangular matrices are returned as outer factors.

    The permutation array can be used to triangularize the outer factors
    simply by a row shuffle, i.e., ``lu[perm, :]`` is an upper/lower
    triangular matrix. This is also equivalent to multiplication with a
    permutation matrix ``P.dot(lu)`` where ``P`` is a column-permuted
    identity matrix ``I[:, perm]``.

    Depending on the value of the boolean ``lower``, only upper or lower
    triangular part of the input array is referenced. Hence a triangular
```

```
matrix on entry would give the same result as if the full matrix is
supplied.

Parameters
----------
a : array_like
    Square input array
lower : bool, optional
    This switches between the lower and upper triangular outer factors of
    the factorization. Lower triangular (``lower=True``) is the default.
hermitian : bool, optional
    For complex-valued arrays, this defines whether ``a = a.conj().T`` or
    ``a = a.T`` is assumed. For real-valued arrays, this switch has no
    effect.
overwrite_a : bool, optional
    Allow overwriting data in ``a`` (may enhance performance). The default
    is False.
check_finite : bool, optional
    Whether to check that the input matrices contain only finite numbers.
    Disabling may give a performance gain, but may result in problems
    (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns
-------
lu : ndarray
    The (possibly) permuted upper/lower triangular outer factor of the
    factorization.
d : ndarray
    The block diagonal multiplier of the factorization.
perm : ndarray
    The row-permutation index array that brings lu into triangular form.

Raises
------
ValueError
    If input array is not square.
ComplexWarning
    If a complex-valued array with nonzero imaginary parts on the
    diagonal is given and hermitian is set to True.

Examples
--------
Given an upper triangular array `a` that represents the full symmetric
array with its entries, obtain `l`, 'd' and the permutation vector `perm`:
```

```
>>> import numpy as np
>>> from scipy.linalg import ldl
>>> a = np.array([[2, -1, 3], [0, 2, 0], [0, 0, 1]])
>>> lu, d, perm = ldl(a, lower=0) # Use the upper part
>>> lu
array([[ 0. ,  0. ,  1. ],
       [ 0. ,  1. , -0.5],
       [ 1. ,  1. ,  1.5]])
>>> d
array([[-5. ,  0. ,  0. ],
       [ 0. ,  1.5,  0. ],
       [ 0. ,  0. ,  2. ]])
>>> perm
array([2, 1, 0])
>>> lu[perm, :]
array([[ 1. ,  1. ,  1.5],
       [ 0. ,  1. , -0.5],
       [ 0. ,  0. ,  1. ]])
>>> lu.dot(d).dot(lu.T)
array([[ 2., -1.,  3.],
       [-1.,  2.,  0.],
       [ 3.,  0.,  1.]])

Notes
-----
This function uses ``?SYTRF`` routines for symmetric matrices and
``?HETRF`` routines for Hermitian matrices from LAPACK. See [1]_ for
the algorithm details.

Depending on the ``lower`` keyword value, only lower or upper triangular
part of the input array is referenced. Moreover, this keyword also defines
the structure of the outer factors of the factorization.

.. versionadded:: 1.1.0

See also
--------
cholesky, lu

References
----------
.. [1] J.R. Bunch, L. Kaufman, Some stable methods for calculating
   inertia and solving symmetric linear systems, Math. Comput. Vol.31,
   1977. DOI: 10.2307/2005787
```

```
    """
    a = atleast_2d(_asarray_validated(A, check_finite=check_finite))

    if a.shape[0] != a.shape[1]:
        raise ValueError('The input array "a" should be square.')
    # Return empty arrays for empty square input
    if a.size == 0:
        return empty_like(a), empty_like(a), np.array([], dtype=int)

    n = a.shape[0]
    r_or_c = complex if iscomplexobj(a) else float

    # Get the LAPACK routine
    if r_or_c is complex and hermitian:
        s, sl = 'hetrf', 'hetrf_lwork'
        if np.any(imag(diag(a))):
            warn('scipy.linalg.ldl():\nThe imaginary parts of the diagonal'
                 'are ignored. Use "hermitian=False" for factorization of'
                 'complex symmetric arrays.', ComplexWarning, stacklevel=2)
    else:
        s, sl = 'sytrf', 'sytrf_lwork'

    solver, solver_lwork = get_lapack_funcs((s, sl), (a,))
    lwork = _compute_lwork(solver_lwork, n, lower=lower)
    ldu, piv, info = solver(a, lwork=lwork, lower=lower,
                            overwrite_a=overwrite_a)
    if info < 0:
        raise ValueError('{} exited with the internal error "illegal value '
                         'in argument number {}". See LAPACK documentation '
                         'for the error codes.'.format(s.upper(), -info))

    swap_arr, pivot_arr = _ldl_sanitize_ipiv(piv, lower=lower)
    d, lu = _ldl_get_d_and_l(ldu, pivot_arr, lower=lower, hermitian=hermitian)
    lu, perm = _ldl_construct_tri_factor(lu, swap_arr, pivot_arr, lower=lower)

    return lu, d, perm


def _ldl_sanitize_ipiv(a, lower=True):
    """
    This helper function takes the rather strangely encoded permutation array
    returned by the LAPACK routines ?(HE/SY)TRF and converts it into
    regularized permutation and diagonal pivot size format.

    Since FORTRAN uses 1-indexing and LAPACK uses different start points for
```

```
upper and lower formats there are certain offsets in the indices used
below.

Let's assume a result where the matrix is 6x6 and there are two 2x2
and two 1x1 blocks reported by the routine. To ease the coding efforts,
we still populate a 6-sized array and fill zeros as the following ::

    pivots = [2, 0, 2, 0, 1, 1]

This denotes a diagonal matrix of the form ::

    [x x        ]
    [x x        ]
    [    x x    ]
    [    x x    ]
    [        x  ]
    [          x]

In other words, we write 2 when the 2x2 block is first encountered and
automatically write 0 to the next entry and skip the next spin of the
loop. Thus, a separate counter or array appends to keep track of block
sizes are avoided. If needed, zeros can be filtered out later without
losing the block structure.

Parameters
----------
a : ndarray
    The permutation array ipiv returned by LAPACK
lower : bool, optional
    The switch to select whether upper or lower triangle is chosen in
    the LAPACK call.

Returns
-------
swap_ : ndarray
    The array that defines the row/column swap operations. For example,
    if row two is swapped with row four, the result is [0, 3, 2, 3].
pivots : ndarray
    The array that defines the block diagonal structure as given above.

"""
n = a.size
swap_ = arange(n)
pivots = zeros_like(swap_, dtype=int)
skip_2x2 = False
```

```
    # Some upper/lower dependent offset values
    # range (s)tart, r(e)nd, r(i)ncrement
    x, y, rs, re, ri = (1, 0, 0, n, 1) if lower else (-1, -1, n-1, -1, -1)

    for ind in range(rs, re, ri):
        # If previous spin belonged already to a 2x2 block
        if skip_2x2:
            skip_2x2 = False
            continue

        cur_val = a[ind]
        # do we have a 1x1 block or not?
        if cur_val > 0:
            if cur_val != ind+1:
                # Index value != array value --> permutation required
                swap_[ind] = swap_[cur_val-1]
            pivots[ind] = 1
        # Not.
        elif cur_val < 0 and cur_val == a[ind+x]:
            # first neg entry of 2x2 block identifier
            if -cur_val != ind+2:
                # Index value != array value --> permutation required
                swap_[ind+x] = swap_[-cur_val-1]
            pivots[ind+y] = 2
            skip_2x2 = True
        else:  # Doesn't make sense, give up
            raise ValueError('While parsing the permutation array '
                             'in "scipy.linalg.ldl", invalid entries '
                             'found. The array syntax is invalid.')
    return swap_, pivots


def _ldl_get_d_and_l(ldu, pivs, lower=True, hermitian=True):
    """
    Helper function to extract the diagonal and triangular matrices for
    LDL.T factorization.

    Parameters
    ----------
    ldu : ndarray
        The compact output returned by the LAPACK routing
    pivs : ndarray
        The sanitized array of {0, 1, 2} denoting the sizes of the pivots. For
        every 2 there is a succeeding 0.
    lower : bool, optional
        If set to False, upper triangular part is considered.
    hermitian : bool, optional
        If set to False a symmetric complex array is assumed.

    Returns
    -------
    d : ndarray
        The block diagonal matrix.
    lu : ndarray
        The upper/lower triangular matrix
    """
    is_c = iscomplexobj(ldu)
    d = diag(diag(ldu))
    n = d.shape[0]
    blk_i = 0  # block index

    # row/column offsets for selecting sub-, super-diagonal
    x, y = (1, 0) if lower else (0, 1)

    lu = tril(ldu, -1) if lower else triu(ldu, 1)
    diag_inds = arange(n)
    lu[diag_inds, diag_inds] = 1

    for blk in pivs[pivs != 0]:
        # increment the block index and check for 2s
        # if 2 then copy the off diagonals depending on uplo
        inc = blk_i + blk

        if blk == 2:
            d[blk_i+x, blk_i+y] = ldu[blk_i+x, blk_i+y]
            # If Hermitian matrix is factorized, the cross-offdiagonal element
            # should be conjugated.
            if is_c and hermitian:
                d[blk_i+y, blk_i+x] = ldu[blk_i+x, blk_i+y].conj()
            else:
                d[blk_i+y, blk_i+x] = ldu[blk_i+x, blk_i+y]

            lu[blk_i+x, blk_i+y] = 0.
        blk_i = inc

    return d, lu


def _ldl_construct_tri_factor(lu, swap_vec, pivs, lower=True):
```

```
"""
Helper function to construct explicit outer factors of LDL factorization.

If lower is True the permuted factors are multiplied as L(1)*L(2)*...*L(k).
Otherwise, the permuted factors are multiplied as L(k)*...*L(2)*L(1). See
LAPACK documentation for more details.

Parameters
----------
lu : ndarray
    The triangular array that is extracted from LAPACK routine call with
    ones on the diagonals.
swap_vec : ndarray
    The array that defines the row swapping indices. If k'th entry is m
    then rows k,m are swapped. Notice that m'th entry is not necessarily
    k to avoid undoing the swapping.
pivs : ndarray
    The array that defines the block diagonal structure returned by
    _ldl_sanitize_ipiv().
lower : bool, optional
    The boolean to switch between lower and upper triangular structure.

Returns
-------
lu : ndarray
    The square outer factor which satisfies the L * D * L.T = A
perm : ndarray
    The permutation vector that brings the lu to the triangular form

Notes
-----
Note that the original argument "lu" is overwritten.

"""
n = lu.shape[0]
perm = arange(n)
# Setup the reading order of the permutation matrix for upper/lower
rs, re, ri = (n-1, -1, -1) if lower else (0, n, 1)

for ind in range(rs, re, ri):
    s_ind = swap_vec[ind]
    if s_ind != ind:
        # Column start and end positions
        col_s = ind if lower else 0
        col_e = n if lower else ind+1
```

```
            # If we stumble upon a 2x2 block include both cols in the perm.
            if pivs[ind] == (0 if lower else 2):
                col_s += -1 if lower else 0
                col_e += 0 if lower else 1
            lu[[s_ind, ind], col_s:col_e] = lu[[ind, s_ind], col_s:col_e]
            perm[[s_ind, ind]] = perm[[ind, s_ind]]

    return lu, argsort(perm)
```

6.3. Active set Algorithm (MatLab)

```
function [ alpha , blocker ] = AlphaCal(p,x_old,A,b,w_index)
    stepset = A' * p < 0;
    stepset(w_index,:) = 0;
    alphaset = (b(stepset,:)- A'(stepset,:)*x_old)./(A'(stepset,:)*p);
    [ alpha , blocker ] = min([alphaset;1]);
end

function [Sol] = LDL_solver(H,g,A,b)
    %   LDL_SOLVER for the active set method
    [n,m] = size(A);
    if n==0
        p = - H\g;
        lambda = nan(m,1);
        Sol = [p;l];
    else
        KKT = [H, -A; -A', zeros(m)];
        d = -[g; b];
        z = zeros(n+m,1);
        [L,D,p] = ldl(KKT,'lower','vector');
        z(p)=L'\(D\(L\d(p)));
        Sol = z;
    end
end

function [x_star ,out] = ActiveSet(H,r,A,b,xtilde)

K=0;

m = size(A',1); % Number of constraints
n = size(A',2); % Number of variables ... p in R^n

k = 1;
not_converged = 1;
error = 1e-10;
```

```
maxIter = 100;
% Set the initial working set, this is done a bit differently than in
% pyhton
w = abs(A_t*xtilde - b) < error;
w_index = find(w);
w_index
x_old = xtilde;
    while (not_converged && k < maxIter)
        g_k = H * x_old + r;
        [ p, lambda ] = LDL_solver(H,g_k,A,A_t,w_index,n,m);
        p
        lambda
        if all(abs(p)<error)

            if all(lambda(w_index)>= 0)
               not_converged = 0;
               x_star = x_old;
            else
               [~,i_min_lambda] = min(lambda(w_index));
               x_new = x_old;
               out.x(:,k) = x_new;
               w_index(i_min_lambda,:) = [];
               w_index
            end

        else
            [ alpha , blockers ] = AlphaCal(p,x_old,A,b,w_index);
            x_new = x_old + alpha * p;
            x_new
            if alpha < 1
                w_index(blockers(1),:) = blockers(1);
            end
            w_index
            k+1
        end
        x_old = x_new;
        out.x(:,k) = x_old;
        k = k + 1;

    end
    iter = k-1;
end
```

6.4. Primal-Dual interior-point algorithm

```
function [x, history] = ipcqp(H,g,A,b,C,d,x,y,z,s)
    max_iterations = 1e3;
    epsilon = 1e-3; % 0.001 tolerance
    eta = 0.995;    % dampening factor

    history = x;  % iteration history that we can plot later

    mh = size(H,1); % dimensionality of x
    ma = size(A,2); % number of equality constraints
    mc = size(C,2); % number of inequality constraints

    % Residuals (gradients in point)
    rL  = H*x + g - A*y - C*z;
    rA  = b - A'*x;
    rC  = s + d - C'*x;
    rsz = s.*z; % inequality slack complementary

    % Duality gap measure
    mu = (s'*z)/mc;

    is_converged = false;
    iteration = 0;

    % Lecture 6, slide 27
    while(~is_converged && iteration <= max_iterations)
        % Affine direction
        KKT = [(H+C*diag(z./s)*C'), -A;
                -A',        zeros(ma)];
        [L, D, p] = ldl(KKT, 'lower', 'vector');

        rLbar = rL - C * diag(z./s) * (rC - rsz./z);
        rhs = -[rLbar; rA];
        Delta = zeros(mh+ma,1);
        Delta(p) = L'\(D\(L\rhs(p)));
        DeltaX = Delta(1:mh);
        DeltaZ = -diag(z./s) * C' * DeltaX + diag(z./s)*(rC - rsz./z);
        DeltaS = -rsz./z - diag(s./z) * DeltaZ;

        % Finding the largest valid alpha_affine
        DeltaUnion = [DeltaZ; DeltaS];
        AllAlphas = (-[z;s]./DeltaUnion);
        alphaAff = min([1;AllAlphas(DeltaUnion<0)]);

        % Computing the affine duality gap
        muAff = (z + alphaAff*DeltaZ)'*(s+alphaAff*DeltaS)/mc;
        % Centering parameter
        sigma = (muAff/mu)^3;

        % Affine-Centering-Correction Direction
        rszbar = rsz + DeltaS.*DeltaZ - sigma*mu;

        % recompute rLbar with rszbar and solve KKT
        % reusing the factorization from above but
        % with an updated right-hand side.
        rLbar = rL - C * diag(z./s) * (rC - rszbar./z);
        rhs = -[rLbar; rA];
        Delta(p)  = L'\(D\(L\rhs(p)));
        DeltaX = Delta(1:mh);
        DeltaY = Delta(mh+(1:ma));
        DeltaZ = -diag(z./s) * C' * DeltaX + diag(z./s)*(rC - rszbar./z);
        DeltaS = -rszbar./z - diag(s./z) * DeltaZ;

        % Finding the largest valid alpha and apply eta-dampening
        DeltaUnion = [DeltaZ; DeltaS];
        AllAlphas = (-[z;s]./DeltaUnion);
        alpha = min([1;AllAlphas(DeltaUnion<0)])*eta;

        % Finally take the step
        x = x + alpha*DeltaX;
        y = y + alpha*DeltaY;
        z = z + alpha*DeltaZ;
        s = s + alpha*DeltaS;

        % Re-compute the residuals and duality measure
        rL  = H*x + g - A*y - C*z;
        rA  = b - A'*x;
        rC  = s + d - C'*x;
        rsz = s.*z;
        mu = (s'*z)/mc;

        % did we converge?
        factors = [norm(rL,1), norm(rA,1), norm(rC,1), abs(mu)];
        is_converged = length(factors(factors < epsilon)) == 4;

        % update history and iteration counter
        history = [history, x];
        iteration = iteration + 1;
    end
end
```