02612

Constrained Optimization

Assignment 1

Phillip Brinck Vetter (s144097)
Ali Saleem (s154675)
Raja Shan Zaker Mahmood (s144102)

April 3, 2018
Technical University of Denmark

Contents

Introduction **1**

Problem 1 - Quadratic Optimization **1**

Problem 2 - Equality Constrained Quadratic Optimization **9**

Problem 3 - Inequality Constrained Quadratic Programming **18**

Problem 4 - Markowitz Portfolio Optimization **29**

Problem 5 - Interior-Point Algorithm for Convex Quadratic Programming **38**

References **55**

Introduction

This assignment presents answers to Assignment 1 in the course. The general topic of the assignment is Quadratic Programs. Information are gathered from course slides or the book [Nocedal and Wright, 2006] in both cases it will be evident from citations. The structure of this assignment is a bit different than what we usually do. A lot of the code is included inside the text, instead of in an appendix. This is because we find it easier to read a rapport with this much implementation if the source code is provided close to the results. We hope that this does not cause any inconvenience. We think continuously flicking back and forth between text and appendix would become tedious.

Problem 1 - Quadratic Optimization

We consider the convex quadratic optimization problem given by

$$\begin{aligned}
&\min_{\text{x}} && f(x) = 3x_1^2 + 2.5x_2^2 + 2x_3^2 + 2x_1x_2 + x_1x_3 + 2x_2x_3 - 8x_1 - 3x_2 - 3x_3 \\
&s.t && x_1 + x_3 = 3 \\
& && x_2 + x_3 = 0
\end{aligned} \tag{1}$$

1.

The problem can be rewritten into matrix form by introducing the quantities

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

From which we can see that the problem can be formulated as

$$\begin{aligned}
&\min_{\text{x}} && f(x) = \frac{1}{2}x^T Hx + g^T x \\
&s.t && A^T x = b
\end{aligned} \tag{3}$$

2.

The KKT optimality conditions for the equality constrained optimization problem are given by the following two statements. Firstly the LICQ condition must be fulfilled. That is to say the columns of A must be linearly independent. It is trivial to see that A has this property in this case by inspection of (2). Secondly, and perhaps more importantly, the gradient of the Lagrangian associated with the system (4) must be zero. The Lagrangian is given by

$$\begin{aligned}
\mathcal{L}(x, \lambda) &= \frac{1}{2}x^T Hx + g^T x - \lambda^T(A^T x - b) \\
&= \frac{1}{2}x^T Hx + g^T x - \lambda^T A^T x + \lambda^T b \\
&= \frac{1}{2}x^T Hx + g^T x - (A\lambda)^T x + \lambda^T b
\end{aligned} \tag{4}$$

The expanded forms are provided because they are useful when applying the gradient. Now define the gradient of the Lagrangian by

$$F(x,\lambda) = \nabla \mathcal{L}(x,\lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ \nabla_\lambda \mathcal{L}(x,\lambda) \end{bmatrix} = 0 \tag{5}$$

The element terms can be expressed analytically by the following

$$\begin{aligned} \nabla_x \mathcal{L}(x,\lambda) &= Hx + g - A\lambda \\ \nabla_\lambda \mathcal{L}(x,\lambda) &= -A^T x + b \end{aligned}$$

Insertion into (5) then yields the second KKT condition which takes the form of a linear system

$$\begin{bmatrix} Hx + g - A\lambda \\ -A^T x + b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Leftrightarrow \underline{\underline{\begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix}}} \tag{6}$$

3.

The following function was developed to solve the programs derived from the KKT conditions shown in (6).

```matlab
function [x,lambda] = EqualityQPSolver(H,g,A,b)
%Solves an equality constrained quadratic program.

%We require the number of constraints to generate a zero matrix
n_constraints = size(A,2);
N = zeros(n_constraints);

%No. of Dimensions
dim = size(H,1);

%The system is given by
LHS = [H, -A ; -A', N];
RHS = -[g ; b];

%Solution. Should be factorized using LDL since LHS is symmetric
% and indefinite in the general case. Especially useful for many iterates
% of different right-hand sides.
[L,D,p] = ldl(LHS,'lower','vector');
sol(p) = L' \ ( D \ ( L \ RHS(p) ) );

%Exctracting x and lambda from solution-vector
x = sol(1:dim);
lambda = sol(dim+1:end);
end
```

As a comment within the function already describes the appropriate factorization used here should be that of LDL. This approach is suitable for symmetric indefinite matrices as the KKT reveals to be as discussed in *Chapter 16.2* [Nocedal and Wright, 2006].

4.

The solution obtained by solving the program defined by the quantities in (2) using the just-presented EqualityQPSolver is found to be

$$\begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 1 \\ 3 \\ -2 \end{bmatrix} \tag{7}$$

The solution is in correspondence with what is obtained through simply substitution of $x_1 = 3 - x_3$ and $x_2 = -x_3$. The resulting one-dimensional function then becomes $f(x_3) = 6.5x_3^2 - 13x_3 + 3$ which has its minimum at $x_3 = 1$. Back-substitution is seen to yields the same solution as the one found by the program.

5.

In order to test the implementation and whether the method provides the correct solution an algorithm to construct pseudo-random convex QPs has been implemented. The process is carried out in three steps. First the elements of H, A and the solution are randomly sampled from a discrete uniform distribution of integers $I \in [1, 2, ..., 20]$. The remaining unknowns g and b can then be computed from the KKT conditions. In order to obtain a positive definite hessian a while loop checking the sign of the eigenvalues is implemented in the function *randomQP* seen below.

```matlab
function [H,g,A,b,x] = randomQP(n,m)

%n is the number of variables
%m is the number of equalities

% 1. Generate random H, A and solution vector (X,L)
rand_H = randi(20,n,n);
H = rand_H*rand_H';
while(min(eig(H)) <= 0)
    H = H + 0.1*eye(n);
end
A = randi(20,n,m);
X = randi(20,n,1);
L = randi(20,m,1);

%Construct g and b
g = -(H*X-A*L);
b = (A'*X);

%Concatenate solution
x = [X;L];
end
```

Using the function one can test whether the randomly generated solution to the QP match with the one found using the QP solver developed. This was found to be the case for a large number of tests.

6.

The sensitivity equations for the problem can be derived using the *Implicit Function Theorem* presented in Lecture 3. The theorem states that for the function $F(x, y(x))$ it holds that

$$\nabla y(x) = -\nabla_x F(x, y(x)) \left[\nabla_y F(x, y(x))\right]^{-1} \tag{8}$$

Consider again the gradient of the Lagrangian now as a function of the parameter inputs denoted by $F(x(p), \lambda(p), p)$. Applying (8) one finds that the sensitivities for both $x(p)$ and $\lambda(p)$ are given by

$$\begin{aligned}\nabla x(p) &= -\nabla_p F(x(p), \lambda(p), p) \left[\nabla_x F(x(p), \lambda(p), p)\right]^{-1} \\ \nabla \lambda(p) &= -\nabla_p F(x(p), \lambda(p), p) \left[\nabla_\lambda F(x(p), \lambda(p), p)\right]^{-1}\end{aligned} \tag{9}$$

The individual components of (9) will now be derived in order to evaluate the expression for the sensitivities. The general form of the Lagrangian and its gradient is given by

$$\mathcal{L}(x(p), \lambda(p), p) = f(x(p), p) - \lambda(p)^T c(x(p), p) \qquad \begin{aligned}\nabla_x \mathcal{L} &= \nabla_x f(x(p), p) - \nabla_x c(x(p), p)\lambda \\ \nabla_\lambda \mathcal{L} &= -c(x(p), p)\end{aligned}$$

The usually defined function $F(x(p), \lambda(p), p)$ then takes the form

$$F(x(p), \lambda(p), p) = \begin{bmatrix} \nabla_x f(x(p), p) - \nabla_x c(x(p), p)\lambda \\ -c(x(p), p) \end{bmatrix} = \begin{bmatrix} F_1(x(p), \lambda(p), p) \\ F_2(x(p), \lambda(p), p) \end{bmatrix} \tag{10}$$

The evaluation of the gradient of F with respect to x and λ can be expressed by operation of the appropriate gradient on (10)

$$\begin{aligned}\nabla_x F(x(p), \lambda(p), p) &= \begin{bmatrix} \nabla_x F_1(x(p), \lambda(p), p) & \nabla_x F_2(x(p), \lambda(p), p) \end{bmatrix} \\ &= \begin{bmatrix} \nabla_x^2 f(x(p), p) & -\nabla_x c(x(p), p) \end{bmatrix} \\ &= \begin{bmatrix} H & -A \end{bmatrix}\end{aligned} \tag{11}$$

$$\begin{aligned}\nabla_\lambda F(x(p), \lambda(p), p) &= \begin{bmatrix} \nabla_\lambda F_1(x(p), \lambda(p), p) & \nabla_\lambda F_2(x(p), \lambda(p), p) \end{bmatrix} \\ &= \begin{bmatrix} -(\nabla_x c(x(p), p))^T & 0 \end{bmatrix} \\ &= \begin{bmatrix} -A^T & 0 \end{bmatrix}\end{aligned} \tag{12}$$

Finally the first term of (9) which is the gradient of F with respect to p can be expressed by

$$\begin{aligned}\nabla_p F(x(p), \lambda(p), p) &= \begin{bmatrix} \nabla_p F_1(x(p), \lambda(p), p) & \nabla_p F_2(x(p), \lambda(p), p) \end{bmatrix} \\ &= \begin{bmatrix} \nabla_p(\nabla_x f(x(p), p)) - \nabla_p(\nabla_x c(x(p), p)\lambda) & -\nabla_p c(x(p), p) \end{bmatrix} \\ &= \begin{bmatrix} \nabla_p g & \nabla_p b \end{bmatrix} \\ &= I\end{aligned} \tag{13}$$

Where the expressions have been simplified under the assumption that $\nabla_p(-A^T x + b) = \nabla_p b$ and $\nabla_p(Hx + g) = \nabla_p g$ despite the implicit dependence of p on both $x(p)$ and $\lambda(p)$. The expressions obtained in (11), (12) and (13) can then be inserted into a joined version of (9) to retrieve the sensitivity vector-components

$$\begin{bmatrix} \nabla x(p) \\ \nabla \lambda(p) \end{bmatrix} = -I \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix}^{-1} = \underline{\underline{- \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix}^{-1}}} \tag{14}$$

Thus it is found that the sensitivities are actually minus the inverse system matrix yielded by the KKT condition as shown previously in (6). The sensitivities can be used to approximate a solution for a different choice of p close to a previously known solution. This is shown in the following section.

7.

A function for computing the sensitivities for the problem is presented below. The function simply retrieves the system matrix from the KKT conditions and finds its inverse. For large systems its computationally advantageous to use the backslash operator with the identity-matrix.

```matlab
function S = compSensConvex(H,g,A,b)
%Computes the Sensitivities for the constrained equility
%convex quadratic program.

n_constraints = size(A,2);
N = zeros(n_constraints);

K = [H, -A ; -A', N];

%The sensitivities are given by
s = size(K,1);
S = -K\eye(s);
end
```

As mentioned briefly previously the sensitivities can in general be used to approximate the minimizer of a unknown p close to some unknown solution with p_0. This is achieved by a first order Taylor expansion of the solution (and similarly for $\lambda(p)$.

$$x(p) \approx x(p_0) + \nabla x(p_0)^T (p - p_0) \tag{15}$$

It is however evident from (14) that any higher order terms vanish in the case of a constrained quadratic program so the approximation becomes exact. It follows that the minimizer of any choice of p can be found without solving the system that arises from the KKT conditions but instead by a simple matrix-vector product as given by(15) given that the solution is known for any one p_0. The script below illustrates how this can be achieved. The addition of a random vector to p_0 using the *randi*-function in MATLAB allows for various possible p's. The script continued to show correspondence between the solution found by actually showing the KKT system, and the *approximation* approach using the sensitivities, confirming the discussed theory.

```
%% Sensitivities using Taylor Expansion
%Define the system
H = [6 2 1; 2 5 2; 1 2 4];
A = [1 0; 0 1; 1 1];

%1. Choose p0
p0 = [-8 ; -3 ; -3 ; 3 ; 0];
g0 = p0(1:3);
b0 = p0(4:5);

%2 Choose p0 + epsilon
p = p0 + randi(20,5,1);
g = p(1:3);
b = p(4:5);

%Calculate solution to p0 and p
[x0,lambda0] = EqualityQPSolver(H,g0,A,b0);
[x,lambda] = EqualityQPSolver(H,g,A,b);

%Sensitivities in p0:
dx = compSensConvex(H,g0,A,b0);

%Calculate p solution with taylor expansion
X = [x0;lambda0] + dx'*(p-p0);

%Print comparsion solution
[[x0;lambda0],[x;lambda],X]
```

8.

The dual of (3) can be expressed as

$$\begin{aligned} \min_{x,\mu} \quad & f(x,\mu) = \frac{1}{2}x^T Hx - b^T\mu \\ s.t \quad & Hx + g - Au = 0 \end{aligned} \tag{16}$$

9.

Now applying the usual method of finding the Lagrangian and its associated gradients the following is obtained

$$\nabla\mathcal{L}(x,\mu,\lambda) = \nabla\left[\frac{1}{2}x^T Hx - b^T u - \lambda^T(Hx + g - Au)\right] \tag{17}$$

Recall that the gradient this time consists of three contributions, namely that of x,μ and λ thus

$$\nabla\mathcal{L}(x,\mu,\lambda) = \begin{bmatrix}\nabla_x\mathcal{L}\\ \nabla_\mu\mathcal{L}\\ \nabla_\lambda\mathcal{L}\end{bmatrix} = \begin{bmatrix} Hx - H\lambda \\ -b + A^T\lambda \\ -(Hx + g - Au)\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix} \tag{18}$$

which is equivalent to the system of equations given by

$$\begin{bmatrix} H & 0 & -H \\ 0 & 0 & A^T \\ -H & A & 0 \end{bmatrix}\begin{bmatrix}x\\ \mu\\ \lambda\end{bmatrix} = \begin{bmatrix}0\\ b\\ g\end{bmatrix} \tag{19}$$

The system is seen to be slightly larger than the equivalent primal problem (obviously since additional Lagrange multipliers have been added). The system matrix is seen to also be symmetric for the dual since $H = H^T$.

10.

The script below shows how the dual of (3) was solved. The result showed correspondence with the solution including three additional multipliers. The solution was

$$\begin{bmatrix}x\\ \mu\\ \lambda\end{bmatrix} = \begin{bmatrix}x_1\\x_2\\x_3\\ \mu_1\\ \mu_2\\ \lambda_1\\ \lambda_2\\ \lambda_3\end{bmatrix} = \begin{bmatrix}2\\-1\\1\\3\\-2\\2\\-1\\1\end{bmatrix} \tag{20}$$

```matlab
%% 8. Solving the Dual Problem

%Specify system matrices
H = [6 2 1; 2 5 2; 1 2 4];
g = -[8 3 3]';
A = [1 0; 0 1; 1 1];
b = [3 0]';

%Construct system
LHS = [H zeros(3,2) -H ; zeros(2,3) zeros(2,2) A' ; -H A zeros(3,3)];
RHS = [zeros(3,1);b;g];

%Solution
sol = LHS \ RHS;
```

```

%We see that the solution 1:5 yields the same as the primal problem
sol(1:5)

%The remaining 6:8 are new lagrange multipliers.
sol(6:8);
```

Advantages of solving the dual is that at times it may be easier to solve computationally [Nocedal and Wright, 2
Notice that in the case where the primal has a higher number of constraints than its number of variables the dual problem reverses this, which can make things simpler to solve. The dual can also be helpful in sensitivity analysis since where a change in the right hand side of the constraint-vector can cause the primal problem to become infeasible, while this merely changes the objective function of the dual, which then remains feasible.

Problem 2 - Equality Constrained Quadratic Optimization

For this part, we are looking at a convex optimization problem, modelling a recycling system.

1.

We want to express the problem in matrix form. In order to do this, we can rewrite the objective function on a more familiar form.

$$f(u) = \frac{1}{2}\sum_{i=1}^{n+1}(u_i - \bar{u})^2 = \frac{1}{2}\sum_{i=1}^{n+1}\left(u_i^2 + \bar{u}^2 - 2u\bar{u}\right) = \frac{1}{2}\left(\sum_{i=1}^{n+1}u_i^2 + \sum_{i=1}^{n+1}\bar{u}^2 - \sum_{i=1}^{n+1}2u\bar{u}\right) \tag{21}$$

By the use of inner product notation, this can be rewritten.

$$f(u) = \frac{1}{2}\left(u^T u + \bar{u}^T\bar{u} - 2u^T\bar{u}\right) \tag{22}$$

As $\bar{u}$ is a parameter of the problem, it will not affect the minimum value of u and so it can be removed from the expression. We can thus state the objective function.

$$f(u) = \frac{1}{2}u^T u - u^T\bar{u} \tag{23}$$

It is evident from the above expression that the hessian of this objective function simply is the identity matrix (multiplied with 2) and g is a vector consisting of $\bar{u}$. To express the constraints in matrix form, A and b will attain the following form:

$$b = \begin{bmatrix} -d_0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad A = \begin{bmatrix} -1 & 1 & 0 & 0 & \dots \\ 0 & -1 & 1 & 0 & \dots \\ \vdots & & \ddots & \ddots & \\ 0 & 0 & \dots & -1 & 1 \\ 1 & 0 & 0 & \dots & -1 \\ 0 & 0 & 0 & \dots & -1 \end{bmatrix}$$

Choosing $n = 10$ the attained form of H, g, A and b is

$$H = \begin{bmatrix} 2&0&0&0&0&0&0&0&0&0&0\\ 0&2&0&0&0&0&0&0&0&0&0\\ 0&0&2&0&0&0&0&0&0&0&0\\ 0&0&0&2&0&0&0&0&0&0&0\\ 0&0&0&0&2&0&0&0&0&0&0\\ 0&0&0&0&0&2&0&0&0&0&0\\ 0&0&0&0&0&0&2&0&0&0&0\\ 0&0&0&0&0&0&0&2&0&0&0\\ 0&0&0&0&0&0&0&0&2&0&0\\ 0&0&0&0&0&0&0&0&0&2&0\\ 0&0&0&0&0&0&0&0&0&0&2 \end{bmatrix} \qquad g = \begin{bmatrix} 0.2\\0.2\\0.2\\0.2\\0.2\\0.2\\0.2\\0.2\\0.2\\0.2 \end{bmatrix} \tag{24}$$

$$A = \begin{bmatrix} -1&1&0&0&0&0&0&0&0&0\\ 0&-1&1&0&0&0&0&0&0&0\\ 0&0&-1&1&0&0&0&0&0&0\\ 0&0&0&-1&1&0&0&0&0&0\\ 0&0&0&0&-1&1&0&0&0&0\\ 0&0&0&0&0&-1&1&0&0&0\\ 0&0&0&0&0&0&-1&1&0&0\\ 0&0&0&0&0&0&0&-1&1&0\\ 0&0&0&0&0&0&0&0&-1&1\\ 1&0&0&0&0&0&0&0&0&-1\\ 0&0&0&0&0&0&0&0&0&-1 \end{bmatrix} \qquad b = \begin{bmatrix} -1\\0\\0\\0\\0\\0\\0\\0\\0\\0 \end{bmatrix} \tag{25}$$

This allows us to write the problem in the following form:

$$\begin{aligned} &\min_{x\in \mathrm{R}^n} && f(x) = \frac{1}{2}x^T H x + g^T x \\ &s.t. && A^T x = b \end{aligned} \tag{26}$$

2.

The Lagrangian function is defined as: $\mathcal{L}(x,\lambda) = f(x) - \sum_{i=1}^{n} \lambda_i c_i(x)$ where $f(x)$ is the quadratic function in (22) and $c_i(x)$ is the ith constraint. This can be written out in the matrix notation as

$$\mathcal{L}(x,\lambda) = \frac{1}{2}u^T u - \lambda^T c(x) \tag{27}$$

The 1st order optimality conditions is then given as

$$F(x,\lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ c(x) \end{bmatrix} = \begin{bmatrix} \nabla f(x) - \nabla c(x)\lambda \\ c(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The gradient of a quadratic function on the form $\frac{1}{2}x^T H x + g^T x + \rho$ is simply $Hx + g$. The constraints $c(x)$ can be rewritten to $A^T x - b$. Since the constraints are linear, the gradient simply becomes the A^T-matrix multiplied with lambda:

$$F(x,\lambda) = \begin{bmatrix} Hx + g - A\lambda \\ A^T x - b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Moving g and b over to the right-hand-side of the equation yields

$$F(x,\lambda) = \begin{bmatrix} Hx - A\lambda \\ A^T x \end{bmatrix} = \begin{bmatrix} -g \\ b \end{bmatrix}$$

This matrix can be rewritten by taking λ and x outside the matrix:

$$F(x,\lambda) = \begin{bmatrix} H & -A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -g \\ b \end{bmatrix}$$

Lastly, by multiplication with minus one on the second row of the system of linear equations the left hand side becomes the desired symmetric matrix

$$F(x,\lambda) = \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = -\begin{bmatrix} g \\ b \end{bmatrix} \tag{28}$$

Thus the KKT-matrix has been obtained. Solving this linear system of equations will yield a minimum because of the convex property of the program. For this reason the first order optimality conditions are both necessary and sufficient conditions for this problem.

3.

```matlab
function [H,g,A,b] = recycleConstruct(n,ubar,d0)

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

for i = 1:n-1
    A(i,i:i+1) = [-1,1];
end

A(n,1) = 1;
A(n:n+1,n) = [-1;-1];

end
```

This function takes n, $\bar{u}$ and d_0 as input and outputs H, g, A and b.

4.

```matlab
function [KKT,rhs] = recycleKKT(n,ubar,d0)

[H,g,A,b] = recycleConstruct(n,ubar,d0);

KKT = [H, -A; -A',zeros(n,n)];
rhs = -[g;b];
end
```

This function constructs the KKT-matrix and the right-hand-side as function of n, $\bar{u}$ and d_0. This is done by calling the former function *recycleConstruct* which returns the necessary components to construct the KKT-matrix and right-hand-side.

5.

```matlab
function [x,lambda] = recycleSolverLU(n,ubar,d0)

[KKT,rhs] = recycleKKT(n,ubar,d0);



[L,U,p] = lu(KKT,'vector');

sol(p) = U\(L\rhs(p));

x = sol(1:n+1);
lambda = sol(n+2:end);
end
```

This function outputs the solution and the Lagrange multipliers of the constrained optimization problem seen in (26). It calls the function described in the previous exercises which constructs the KKT-matrix and right-hand-side. The system is then solved using a LU-factorization. The found solution and Lagrange multipliers when the parameters are $n = 10$, $d_0 = 1$ and $\bar{u} = 0.2$ using this method is given by

u	λ
0.2	-3.6
0.2	-3.4
0.2	-3.2
0.2	-3.0
0.2	-2.8
0.2	-2.6
0.2	-2.4
0.2	-2.2
0.2	-2.0
-0.8	-1.8
1	-

Table 1: Solution to the convex program (28) and the associated Lagrange multipliers

6.

```
function [x,lambda] = recycleSolverLDL(n,ubar,d0)


[KKT,rhs] = recycleKKT(n,ubar,d0);

[L,D,p] = ldl(KKT,'lower','vector');  %Use LDL factorizatin because Matrix is symmetric a
sol(p) = L'\( D \(L\rhs(p)));


x = sol(1:n+1);
lambda = sol(n+2:end);

end
```

This function solves the same program but by using LDL-factorization. The solution and Lagrange multipliers are the same as the one seen on Table 1 as expected.

7.

```
function [x,lambda] = recycleNullSpace(n,ubar,d0)

[H,g,A,b] = recycleConstruct(n,ubar,d0);
[Q,Rbar] = qr(A);
m1 = size(Rbar,2);
Q1 = Q(:,1:m1);
Q2 = Q(:,m1+1:end);
R = Rbar(1:m1,1:m1);

x_y = (R')\b;

x_z = ((Q2')*H*Q2)\( (-Q2') *(H*Q1*x_y+g));

x  = Q1*x_y + Q2*x_z;

lambda = R\(Q1'*(H*x+g));

end
```

This function solves the same program but with the Null-Space method described from the lecture slides during *Lecture 5* of the course. The output is again the solution and the associated Lagrange multipliers at the solution. They remain the same as the ones seen in Table 1.

8.

```
function [x,lambda] = recycleRangeSpace(n,ubar,d0)

[H,g,A,b] = recycleConstruct(n,ubar,d0);
KKT = recycleKKT(n,ubar,d0);

L = chol(H,'lower');

v = (L*L')\g;

H_a = A'*inv(H)*A;
L_a = chol(H_a,'lower');
lambda = (L_a*L_a')\(b+A'*v);
x = H\(A*lambda-g);

end
```

This function solves the same program but uses the Range-Space procedure described on the lecture slides during *Lecture 5* as well. The output of the function is the found solution and the Lagrange multipliers at the solution. The found solution is in correspondence with all other found solutions, thus equivalent to the one listed in Table 1.

9.

In order to estimate the performance of each method as a function of the size of the problem a performance test was carried out. This can be seen in Figure 1.

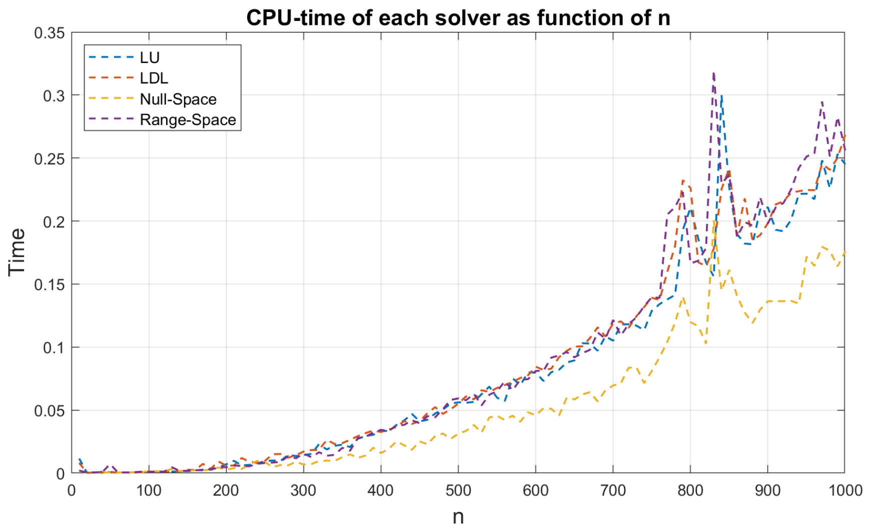


Figure 1: CPU-time as a function of n for each of the solvers. Time is measured in seconds

It is evident that the solver using the Null-Space method outperforms all the other solvers at every iteration step. The other solvers seems to be grouped relatively close to one another in terms of performance and are thus not significantly different.

10.

For a system of size $n = 100$ the sparsity of the KKT-matrix is investigated using the MATLAB *spy* command. The result can be seen on Figure 2.

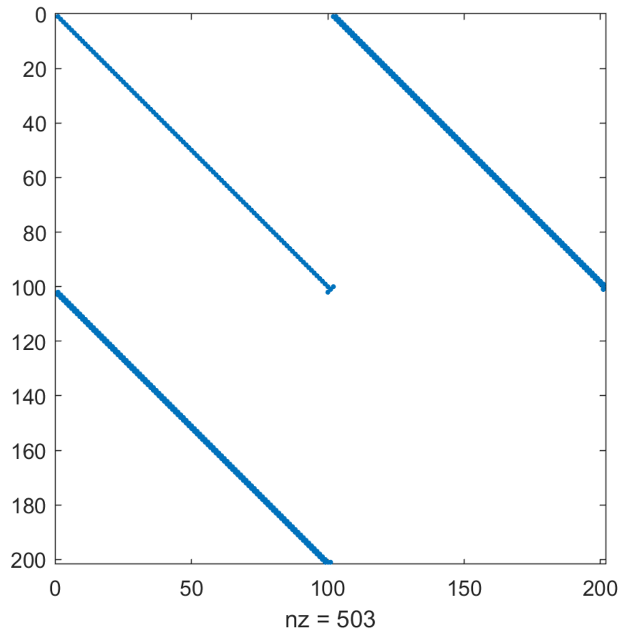


Figure 2: Sparsity of the KKT-matrix

The KKT-matrix is seen to be very sparse as only 503 out of 40401 elements are nonzero.

11.

We modify the previous LU-solver so it treats the KKT-matrix as a sparse system. The function looks like this:

```matlab
function [x,lambda,P] = recycleSparseLUSolver(n,ubar,d0)

[KKT,rhs] = (recycleKKT(n,ubar,d0));

sp_KKT = sparse(KKT);

[L,U,P] = lu(sp_KKT);

sol = U\(L\(P*rhs));

x = sol(1:n+1);
lambda = sol(n+2:end);
```

```

end
```

12.

We also modified the LDL solver to treat the KKT-matrix as a sparse system. The function looks like this:

```
function [x,lambda,P] = recycleSparseLDLSolver(n,ubar,d0)

[KKT,rhs] = (recycleKKT(n,ubar,d0));

sp_KKT = sparse(KKT);

[L,D,P,Q] = ldl(sp_KKT,'lower','vector');

sol(P) = L'\(D\(L\rhs(P)));

x = sol(1:n+1);
lambda = sol(n+2:end);

end
```

13.

We made the same performance test on the sparse LU and LDL solvers as we did with the dense-solvers. This benchmark can be seen on Figure 3

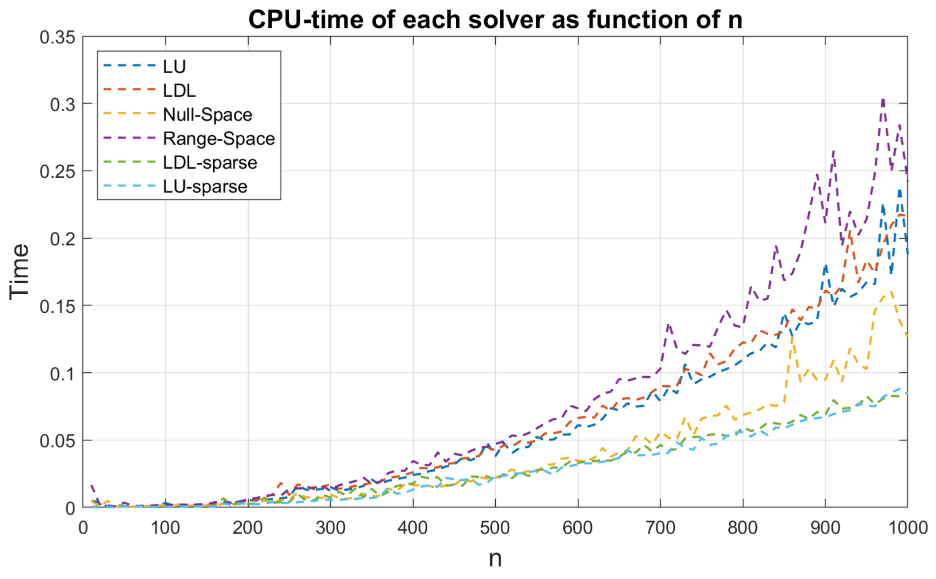
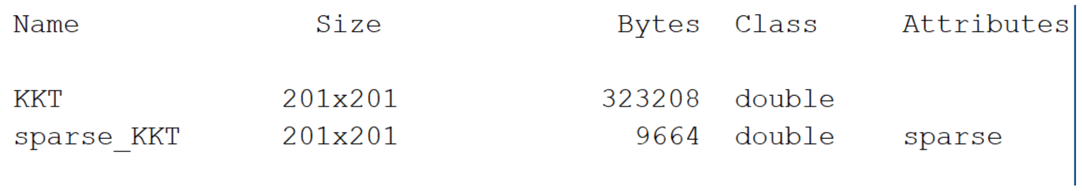


Figure 3: CPU benchmark test of all solvers. Time is measured in seconds

It is clear that the sparse-solvers outperforms all the dense-solvers. This result is expected because of the high sparsity of the KKT-matrix as just shown. It is crucial to take advantage of this property in order to optimize a solving program. Further advantages of using a sparse method is the drastically lowered amount of memory that needs to be stored. A comparison between how much memory a dense KKT-matrix and a sparse KKT-matrix for $n = 100$ was made and this can be seen in Figure 4 below.



```
Name                Size                Bytes  Class     Attributes

KKT               201x201              323208  double
sparse_KKT        201x201                9664  double    sparse
```

Figure 4: Memory comparison between a dense and a sparse KKT-matrix

The sparse matrix is seen to only require 3% of the memory that an equivalent dense system requires. This is an important factor to consider when solving much larger systems where only a limited amount of memory is readibly accessible.

Problem 3 - Inequality Constrained Quadratic Programming

1.

We consider the convex quadratic optimization problem given by

$$\begin{aligned}
\min_{\mathrm{x}} \quad & f(x) = (x_1 - 1)^2 + (x_2 - 2.5)^2 \\
s.t \quad & c_1(x) = x_1 - 2x_2 + 2 \geq 0 \\
& c_2(x) = -x_1 - 2x_2 + 6 \geq 0 \\
& c_3(x) = -x_1 + 2x_2 + 2 \geq 0 \\
& c_4(x) = x_1 \geq 0 \\
& c_5(x) = x_2 \geq 0
\end{aligned} \tag{29}$$

In order to obtain a better overview of the problem a contour plot is drawn. This is shown in Figure 5. The grey area indicates the infeasible region in which the inequality constraints are not fulfilled. The feasible region is thus confined to the irregular pentagon in the lower left corner.

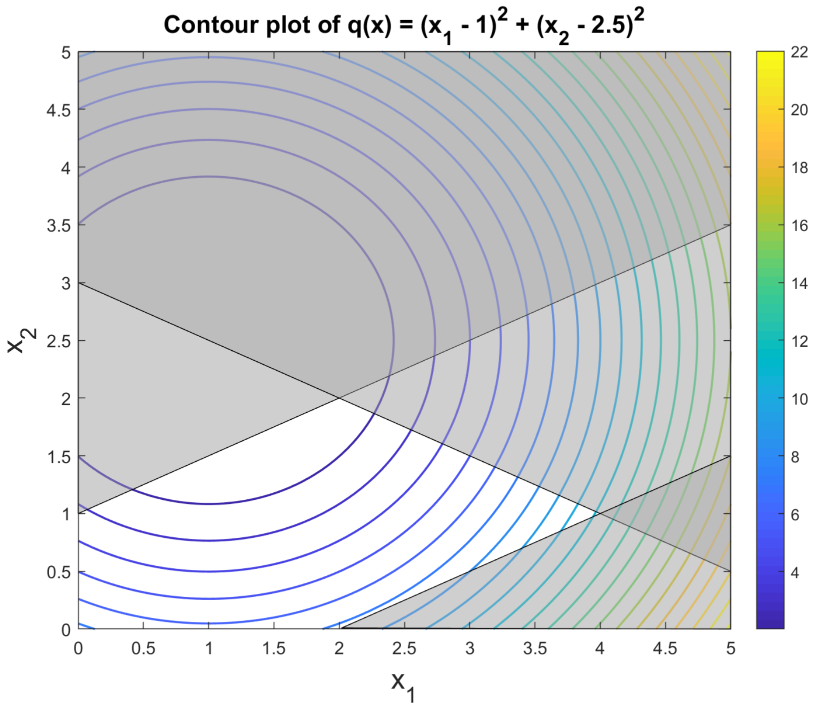


Figure 5: Contour plot of the objective function f and its associated constraints as given in (29). This function is taken from *Example 16.4* [Nocedal and Wright, 2006]

2.

The first-order conditions are given by the equations (16.37) [Nocedal and Wright, 2006]. They state that the Lagrangian with respect to x must be zero as usual and the constraints must be fulfilled. Additionally they state that all Lagrange multipliers λ_i for the inequality constraints of the problem that lie in the active set $\mathcal{A}$ must be non-negative while at the same time all other multipliers must be zero. The mathematical notation for this is covered by

$$\begin{aligned} \nabla_x L &= 0 & & \\ c_i(x) &\geq 0 & & i \in \mathcal{I} \\ \lambda_i &\geq 0 & & i \in \mathcal{I} \\ \lambda_i &= 0 & & i \in \mathcal{I}/\mathcal{A} \end{aligned} \tag{30}$$

The problem (29) can be rewritten into matrix-form such that it takes the usual shape of a quadratic program. The details of this procedure has been demonstrated a number of times in this report already. The objective function is first expanded to obtain the quadratic terms

$$f(x) = (x_1 - 1)^2 + (x_2 - 2.5)^2 = x_1^2 + 1 - 2x_1 + x_2^2 + 6.25 - 5x_2 = \underline{\underline{x_1^2 + x_2^2 - 2x_1 - 5x_2 + 7.25}} \tag{31}$$

Thus one finds that by introducing the following five quantities

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \qquad g = \begin{bmatrix} -2 \\ -5 \end{bmatrix} \qquad \rho = 7.25 \qquad A = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 \\ -2 & -2 & 2 & 0 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \end{bmatrix}$$

the system can be expressed in the usual way of a quadratic program. The Lagrangian can then be defined by

$$\mathcal{L}(\mathbf{x}, \lambda) = \frac{1}{2} x^T H x + g^T x + \rho - \lambda^T \left(A^T x - b \right) = (x_1 - 1)^2 + (x_2 - 2.5)^2 - \sum_{i=1}^{5} \lambda_i c_i(x) \tag{32}$$

where c_i is the i$^{\text{th}}$ inequality constraint. The matrix form of the KKT condition is then derived as shown in (6) using the gradient of the Lagrangian given by

$$F(\mathbf{x}, \lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(\mathbf{x}, \lambda) \\ \nabla_\lambda \mathcal{L}(\mathbf{x}, \lambda) \end{bmatrix} = \begin{bmatrix} \nabla f(x) - \nabla c(x) \lambda \\ -c(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{33}$$

to obtain

$$\begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{34}$$

As opposed to previously it is not necessary to impose the linear independence requirement for the active set as described by the following quote *...Hence, in the optimality conditions for quadratic programming given above, we need not assume that the active constraints are linearly independent at the solution* [Nocedal and Wright, 2006].

3.

It follows from *Theorem 16.4* [Nocedal and Wright, 2006] that if the conditions described in (30) are fulfilled then the point in question is indeed the global minimizer of the problem. The practicalities are carried out by considering a subset of the problem. This will be evident later.

4.

A QP solver has already been implemented and is shown in Section 1.3.

5.

The conceptual part of the Active-Set algorithm will be explained using the initial starting point shown in *Figure 16.3* in [Nocedal and Wright, 2006]. This starting point is $x_0 = (2,0)^T$. The calculations carried out are based on the derived formulae from [Nocedal and Wright, 2006]. The system (35) is solved to obtain the step direction p

$$\begin{aligned} \min_{p} \quad & \frac{1}{2}p^T H p + g_k^T p & \\ s.t \quad & A^T p = 0 & i \in \mathcal{W}_k \end{aligned} \tag{35}$$

while the step size is given by α which can be computed from

$$\alpha_k = \min\left(1, \min_{i \notin \mathcal{W}_k, A^T p < 0}\left(\frac{b - A^T x_k}{A^T p_k}\right)\right) \tag{36}$$

Note that while the notation does not explicit say so the matrices b and A are reduced according to the working set and according to negative entries in the case of α. The procedure begins by inspection of the constraints. Notice that the constraints associated with $i = \{3,5\}$ are active for the chosen initial point since

$$c_3(2,0) = 2 - 2 \cdot 2 + 2 = 0 \qquad c_5(2,0) = 0 \tag{37}$$

The working set is chosen to be this set: $\mathcal{W}_1 = \{3,5\}$. Now the system (35) is solved for p: First the affine-related vector g_1 is computed by

$$g_1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} -2 \\ -5 \end{bmatrix} = \begin{bmatrix} 2 \\ -5 \end{bmatrix} \tag{38}$$

The system is then solved using only constraints $c_3(x), c_5(x)$ which is equivalent to removing the columns of A corresponding to index $j = \{1,2,4\}$ and the same rows of b. The solution is then found to be

$$p_1 = \begin{bmatrix} p^1 \\ p^2 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -2 \\ -1 \end{bmatrix} \tag{39}$$

The step direction is found to be zero, thus the Lagrange multipliers are inspected to clarify whether the point is the global minimizer as given by *Theorem 16.4* [Nocedal and Wright, 2006]. Since both multipliers are negative this is not the case. While either one in theory could be removed the most negative one is chosen since this results in the largest decrease in the function value as evident from the following statement: *...This choice is motivated by the sensitivity analysis given in Chapter 12, which shows that the rate of decrease in the objective function when one constraint is removed is proportional to the magnitude of the Lagrange multiplier for that constraint* [Nocedal and Wright, 2006]. Thus the constraint $c_3(x)$ is removed from the working set. The next iterate then has the working set $W_2 = \{5\}$ and the solution-iterate $x_2 = x_1 = (2, 0)$. The system (35) is solved again using g_1 and the solution is found to be

$$p_2 = \begin{bmatrix} -1 \\ 0 \\ 5 \end{bmatrix} \tag{40}$$

The step size must then be computed using (36). Note that the computation is carried out using the indices outside of the working set. Firstly it is decided which elements satisfy $A^T p_2 < 0$. Consider the elements not in the working set $\mathcal{W}_2$ and extract these to obtain

$$A_{\notin \mathcal{W}_2} p_2 = \begin{bmatrix} 1 & -1 & -1 & 1 \\ -2 & -2 & 2 & 0 \end{bmatrix}^T \begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}^T \tag{41}$$

It is evident that only the indices $i = \{1, 4\}$ fulfill this property so only the associated constraints should be used in computing α_2. The term of interest is found to be

$$\frac{b_{1,4} - A_{1,4}^T x_2}{A_{1,4}^T p_2} = \left(\begin{bmatrix} -2 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ -2 & 0 \end{bmatrix}^T \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right) \Big/ \left(\begin{bmatrix} 1 & 1 \\ -2 & 0 \end{bmatrix}^T \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 6 \\ 6 \end{bmatrix} \tag{42}$$

It is then clear from (36) that $\alpha_2 = 1$ and the next iteration $x_3 = x_2 + \alpha_2 p_2 = (2, 0) + (-1, 0) = (1, 0)$. The working set should not be updated since only $c_5(1, 0) = 0$ and thus $\mathcal{W}_3 = \mathcal{W}_2 = \{5\}$. The procedure continues in six iterations before it stops. The stop occurs when $p_6 = 0$ and all multipliers λ_i are positive. The table below (43) shows the iterations and associated data during the process of solving the inequality constrained QP.

k	1	2	3	4	5	6
x_k	$(2, 0)$	$(2, 0)$	$(1, 0)$	$(1, 0)$	$(1, 1.5)$	$(1.4, 1.7)$
$\mathcal{W}_k$	$\{5, 3\}$	$\{5\}$	$\{5\}$	$\varnothing$	$\{1\}$	$\{1\}$
p_k	$(0, 0)$	$(-1, 0)$	$(0, 0)$	$(0, 2.5)$	$(0.4, 0.2)$	$(0, 0)$
λ_k	$(-2, -1)$	-5	-5	$\varnothing$	0.8	0.8

(43)

6.

The Lagrange multipliers are negative at the first iteration which indicates due to sensitivity analysis [Nocedal and Wright, 2006] that the point x_1 can not be the minimum for the function f decreases along a direction where the two active constraints are relaxed. On the second iteration the third constraint is relaxed by moving along the fifth constraint. The multiplier remain negative at the third iteration which again indicates that the point can not be a global minimizer of the problem. On the fourth iteration the solution moves in a direction perpendicular to the constraint to arrive exactly at the first constraint. The solution is obtained on the fifth iteration where the multipliers are positive. Indicating at the last step that the function decreases in all other directions. Notice the change from negative to empty to positive through the six iterations.

7.

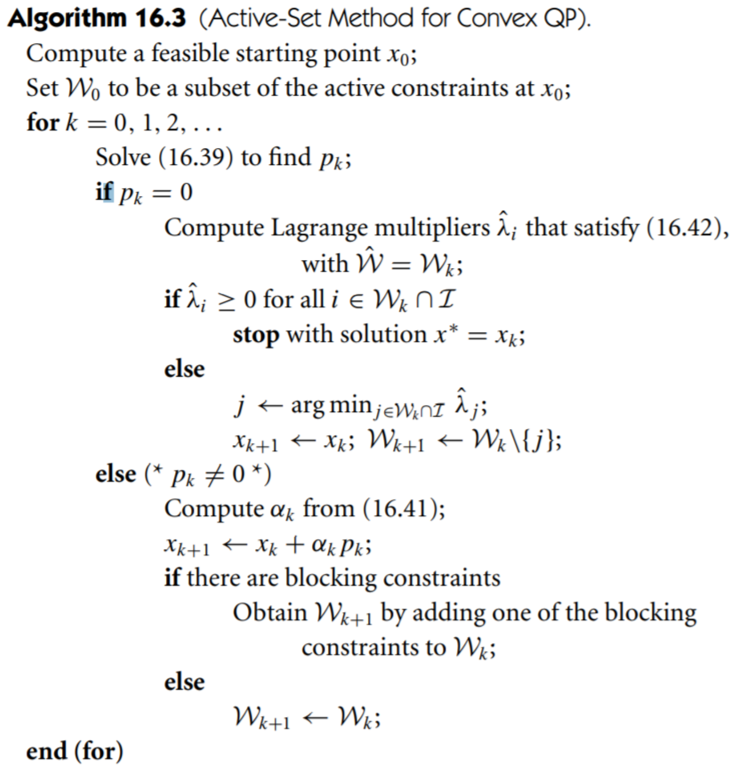
The Active-Set Method for convex QPs attempts to find a minimum of quadratic programs on the form

$$\begin{aligned} &\min_x && q(x) = \frac{1}{2}x^T G x + c^T x \\ &s.t && a_i^T x = b_i, \quad i \in \mathcal{E} \\ & && a_i^T x \geq b_i, \quad i \in \mathcal{I} \end{aligned}$$

where G is the hessian of the quadratic function and c is the first-order components. In general, the quadratic programs solved here will be convex quadratic programs. This means that the hessian G is a symmetric and positive semidefinite matrix. To solve these kind of inequality problems we can use active-set algorithms. In the simplest terms, these methods solves programs on the given form by defining *active* and inactive constraints. This means that if an inequality constraint $a_i^T x - b_i = 0$ for $i \in \mathcal{I}$ it is active and inactive if $a_i^T x - b_i > 0$. The active set $\mathcal{A}(x)$ is then defined to be the set of all active constraints at x. The active set is in general not know, but if it were known, we could simply just apply the techniques we've learned to solve equality-constrained QP's to the following problem:

$$\min_x q(x) = \frac{1}{2}x^T G x + x^T c \quad \text{subject to} \quad a_i^T x = b_i, \quad i \in \mathcal{A}(x^*) \tag{44}$$

where $\mathcal{A}(x^*)$ is the active set at the optimal point. The active set algorithm is based on the following template



Algorithm 16.3 (Active-Set Method for Convex QP).

```
Compute a feasible starting point $x_0$;
Set $\mathcal{W}_0$ to be a subset of the active constraints at $x_0$;
for $k = 0, 1, 2, \ldots$
    Solve (16.39) to find $p_k$;
    if $p_k = 0$
        Compute Lagrange multipliers $\hat{\lambda}_i$ that satisfy (16.42),
                with $\hat{\mathcal{W}} = \mathcal{W}_k$;
        if $\hat{\lambda}_i \geq 0$ for all $i \in \mathcal{W}_k \cap \mathcal{I}$
            stop with solution $x^* = x_k$;
        else
            $j \leftarrow \arg\min_{j \in \mathcal{W}_k \cap \mathcal{I}} \hat{\lambda}_j$;
            $x_{k+1} \leftarrow x_k$; $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k \backslash \{j\}$;
    else (* $p_k \neq 0$ *)
        Compute $\alpha_k$ from (16.41);
        $x_{k+1} \leftarrow x_k + \alpha_k p_k$;
        if there are blocking constraints
            Obtain $\mathcal{W}_{k+1}$ by adding one of the blocking
                constraints to $\mathcal{W}_k$;
        else
            $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k$;
end (for)
```

Figure 6: Active-set algorithm for convex QP[Nocedal and Wright, 2006]

The algorithm is a *primal* active-set method. It finds a step for each iterate for a sub-problem to the original QP, which only includes some of the original inequalities and all of the original equalities. These are all imposed as equalities in this sub-problem. This subset of inequalities are called the *working set* and is denoted W_k. The algorithm starts by computing a feasible starting point. This is carried out by solving a linear program. This is necessary as it is unexpected that a randomly generated point would be inside the feasible region. After the initial feasible point has been generated you define a new subset W_0 to be a subset of the active constraints at first (zeroth) iteration x_0. The iteration then starts by solving the following QP to compute a step direction p:

$$\begin{array}{ll} \min & \frac{1}{2}p^T G p + g_k^T p \\ s.t & A^T p = 0, \quad i \in W_k \end{array}$$

where $p = x - x_k$ and $g_k = Gx_k + c$ where G is the hessian of the QP we're wishing to solve and c is the first-order component of the problem. Once the step direction p_k has been found the Lagrange multipliers are obtained using the usual property of the gradient of the Lagrangian

$$A\tilde{\lambda} = g = G\tilde{x} + c \tag{45}$$

where $\tilde{W_k}$ is the current active working set. If all current Lagrange multipliers $\tilde{\lambda}$ are larger or equal to zero, we have found the minimizer to the problem as the first order optimality conditions are satisfied. However, if some of the Lagrange multipliers are negative, we remove the constraint that has most negative multiplier and start over again with the new and smaller working subset. If $p_k \neq 0$ then you compute α_k as defined in (16.41) in [Nocedal and Wright, 2006]:

$$\alpha_k = \min\left(1, \quad \min_{i \notin W_k, a_i^T p_k < 0} \frac{b_i - a_i^T x_k}{a_i^T p_k}\right) \tag{46}$$

$\alpha_k \in [0, 1]$ which is known as the step-length parameter, and tells you how far you should go along the step-direction p_k. If $\alpha_k < 1$, then that means the step direction p_k is blocked by a constraint not in the working set. Thus we can obtain the new working set W_{k+1} by adding the blocking constraint. The algorithm repeats this process until the optimal solution is found.

8.

```
function x_feas = compute_feasiblePhaseOne(A,b,Aeq,beq)

[varIn,n_cons] = size(A);
[~,n_consEq] = size(Aeq);
x_0 = rand(varIn,1);

if(~isempty(Aeq)) %Checks if there is any equality constraints
gamma_eq = -sign(beq - Aeq'*x_0);
Aeq = [Aeq',gamma_eq*eye(n_consEq)];
end

A = [A',eye(n_cons)];
```

```
f = [ones(n_cons,1);ones(varIn,1)];

optimopt = optimoptions('linprog','display','off');
lb = [zeros(1,n_cons), -inf(1,varIn)]; %Only the z's that are bounded to be nonnegative
x_feas = linprog(f,-A,-b,-Aeq,-beq,lb,[],optimopt);
x_feas = x_feas(n_cons+1:end);

end
```

This function computes a feasible point to a QP by using a "Phase 1" approach from linear programming. Essentially, we're solving the following linear program that is described in [Nocedal and Wright, 2006] on (p.473):

$$\min_{(x,z)} \; e^T z \tag{47}$$

$$\text{subject to} \quad a_i^T x + \gamma_i z_i = b_i, \quad i \in \mathcal{E}, \tag{48}$$

$$a_i^T x + \gamma_i z_i = b_i, \quad i \in \mathcal{I}, \tag{49}$$

$$z \geq 0, \tag{50}$$

where $e = \left[1, 1, \ldots, 1\right]^T$, $\gamma_i = -\text{sign}(a_i^T \tilde{x} - b_i)$ for $i \in \mathcal{E}$, $\gamma_i = 1$ for $i \in \mathcal{I}$ and $\tilde{x}$ is a starting point, which does not necessarily have to be a feasible point. The generated initial point for this example is $x = (0, 0)$, which indeed is a feasible point.

9.

```
function [x_opt,info] = activeSetQP(H,g,A,b,Aeq,beq)

kmax = 1000;
[varIn,n_cons] = size(A);
x_0 = compute_feasiblePhaseOne(A,b,Aeq,beq); %Compute feasible point

if(isempty(x_0))
    error('The algorithm was not able to find a feasible starting point');
end

W_0 = [];    %Pick random start workingset
workset(1,:) = zeros(1,n_cons);
workset(1,W_0) = 1;

%Define all variables
x(:,1) = x_0;
tol_step = 1e-9;
tol_lambda = 1e-9;
k = 1;
lambda_active = nan(n_cons,1);
```

```
while(k < kmax)
    work_idx = find(workset(k,:));
    inactive_set = find(~workset(k,:));

    %Solve subproblem to find step-direction
    Aeq_sub = [Aeq, A(:,work_idx)]';
    beq_sub = zeros(size(Aeq_sub,1),1);
    g_k = H*x(:,k) + g;
    [p(:,k),lam] = EqualityQPSolverActiveSet(H,g_k,Aeq_sub,beq_sub);
    lambda_active(work_idx,k) = lam;  %Update the lagrange multipliers
    lambda_active(inactive_set,k) = NaN;
    if(norm(p(:,k),'inf') <= tol_step)

      %Find the lagrange multipliers
      lambda_active(work_idx,k) = A(:,work_idx)\(H*x(:,k) + g);
      alpha(k) = NaN;
      %Check for optimum
      if(norm(lambda_active(~isnan(lambda_active(:,k)),k),'inf') >= tol_lambda)
          x_opt = x(:,k);
          break;
      else
        %Find idx of lowest lagrange multiplier
        [~,j] = min(lambda_active(:,k));
        x(:,k+1) = x(:,k);

        %Remove the constraint with most negative lambda
        workset(k,j) = 0;
        workset(k+1,:) = workset(k,:);
        lambda_active(j,k) = NaN;
      end
    else
        inactive_idx = find(~workset(k,:)); %Get idx of inactive constraints

        %Calculate 2nd argument
        min_arg2 = (b(inactive_idx) - A(:,inactive_idx)'*x(:,k) ) ./ ...
        (A(:,inactive_idx)'*p(:,k));

        %Finds the idx that fulfills a_i^T*p_k < 0
        valid_min_arg2 = find( (A(:,inactive_idx)'*p(:,k)) < 0);
        alpha(k) = min(1, min(min_arg2(valid_min_arg2)));
        x(:,k+1) = x(:,k) + p(:,k)*alpha(k);
        if(alpha(k) < 1)
          min_arg2(min_arg2 <0) = NaN;
          [~,blockIdx] = min(min_arg2);
          workset(k+1,blockIdx) = 1;
        else
```

```
            workset(k+1,:) = workset(k,:);
          end
          lambda_active(:,k+1) = lambda_active(:,k);


      end
      k = k +1;
end

info.alpha = alpha;
info.NumIter = k;
info.steps = p;
info.Workingset = workset;
info.lambda = lambda_active;
info.x = x;

end
```

The function that is shown is the code of the Active-Set method for inequality QPs. The function outputs the found minimizer and also returns a structure which contains information of each iteration such as the working set W_k, x_k, α_k etc. For example 16.4, we got the following values with the initial working-set set to be $\{\emptyset\}$ and $x_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$:

Iterations	Steps p_k	α_k	W_k	x_k	λ_k
1	$[1, 2.5]^T$	0.5	$\{\emptyset\}$	$[0, 0]^T$	$\{\emptyset\}$
2	$[0, 1.25]^T$	0	$\{4\}$	$[0.5, 1.25]^T$	$\{\lambda_4 = -1\}$
3	$[0.9, 0.45]^T$	1	$\{1\}$	$[0.5, 1.25]^T$	$\{\lambda_1 = 0.8\}$
4	$[0, 0]^T$	-	$\{1\}$	$[1.4, 1.7]^T$	$\{\lambda_1 = 0.8\}$

Table 2: Table over different information of the active-set algorithm

We confirm that the found solution $x^* = [1.4, 1.7]^T$ corresponds with the solution that is listed in the book at the bottom on page 475 in [Nocedal and Wright, 2006]

10.

```
function x_feas = compute_feasibleBigM(H,g,A,b,Aeq,beq)

[varIn,n_cons] = size(A);
[~,n_consEq] = size(Aeq);
M = 2;
eta_tol = 1e-12;
optimopt = optimoptions('quadprog','display','off');



```

```
posA = [Aeq',-ones(n_consEq,1)];
posBeq = beq;

negA = -[Aeq',ones(n_consEq,1)];
negBeq = -beq;

inEqA = -[A',ones(n_cons,1)];
inEqb = -b;

A = [posA;negA;inEqA];
b = [posBeq;negBeq;inEqb];

g(end+1) = M;
H(end+1,end+1) = 0;

lb = [-inf(1,varIn), 0];
x_feas = quadprog(H,g,A,b,[],[],lb,[],[],optimopt);

%Solves the problem again with a bigger value of M if eta is larger than tolerance
while(x_feas(end) >= eta_tol)
    M = M^2;
    g(end) = M;
    x_feas = quadprog(H,g,A,b,[],[],lb,[],[],optimopt);
end

end
```

This function computes the inital feasible point x_0 by solving the quadratic program listed on (16.47) in [Nocedal and Wright, 2006] instead of . This is essentially a penalty method known as the "big M" method. This is an alternative approach compared to the "Phase 1" method. The strength that lies in this method, is that the solution of (16.47) will also be an solution to the original problem.

```
function x_feas = compute_feasibleBigMV2(H,g,A,b,Aeq,beq)

M = 10^5;
[varIn,n_cons] = size(A);
[~,n_consEq] = size(Aeq);

Aeq = [Aeq',ones(n_consEq,1),-ones(n_consEq,1)];
A = [A',eye(n_cons)];
lb =  [-inf(1,varIn), zeros(1,2*n_consEq),zeros(1,n_cons)];

%Define the new Hessian and first-order-component vector
%with extra variables
H(end+1:(end+n_cons+n_consEq),end+1:(end+n_cons+n_consEq)) = 0;
g(end+1:(end+n_cons+n_consEq)) = M;
```

```

opt = optimoptions('quadprog','display','off');

x_feas = quadprog(H,g,-A,-b,Aeq,beq,lb,[],[],opt);
x_feas = x_feas(1:end-n_cons);

end
```

This second function is a second variant of the function from before. It generates an initial point x_0 for an active-set solver by solving the quadratic program (16.48) that can be found on page 474 in [Nocedal and Wright, 2006]. We will now compute an initial point with both of these functions and use them in our own active-set algorithm, and reuse these initial estimates and solve the problem using MATLABs own active-set algorithm in quadprog. The 2 found initial estimates $\tilde{x}$ are:

$x_0 = [1.4, 1.7]^T$	$x_0 = [1.4, 1.7]^T$

Table 3: The 2 initial points that have been estimated with the 2 variants with "big M" methods

We see that the 2 generated initial points are actually the solution to the original problem. This is a property that these 2 penalty methods are capable of as described in [Nocedal and Wright, 2006] on page 473. On Table 4 you can see how each of the active-set solvers performed for each of the algorithms:

info/algorithms	**quadprog**	**Active-set "big M"**	**Active-set "big M" v2**
x^*	$[1.4, 1.7]^T$	$[1.4, 1.7]^T$	$[1.4, 1.7]^T$
$\mathcal{A}(x^*)$	$c_1(x)$	$c_1(x)$	$c_1(x)$
$q(x^*)$	0.8	0.8	0.8
Num. of Iterations	1	2	2

Table 4: Comparison of the performance of the solvers with different methods of calculating the initial estimate

We see that we get the same results, except for the number of iterations. This is probably due to the initial guess of the working set we've decided at the start of our own Active-set algorithm. By setting the initial working-set $W_0 = \{1\}$, our algorithm would only take 1 iteration.

Problem 4 - Markowitz Portfolio Optimization

We consider a financial market with 5 securities as given in the table (51) below:

Security	Covariance					Return
1	2.30	0.93	0.62	0.74	−0.23	15.10
2	0.93	1.40	0.22	0.56	0.26	12.50
3	0.62	0.22	1.80	0.78	−0.27	14.70
4	0.74	0.56	0.78	3.40	−0.56	9.02
5	−0.23	0.26	−0.27	−0.56	2.60	17.68

(51)

1.

To formulate this problem as a quadratic program given R, we have a slight variation on the approach presented in chapter 16 of the textbook. As the return is a constant, it enters the QP as a constraint, while the objective function concerns only the covariance matrix. The QP is concerned with finding a vector $x \in \mathcal{R}^n$, in this case with $n = 5$, for which $x^T \mu = R$, where μ is the vector of expected returns. The covariance matrix $H \in \mathcal{R}^{n \times n}$ is the hessian of the problem. e is a column vector of ones with the same length as the vector x. I is the identity matrix with rank n.

$$\min_x \frac{1}{2} x^T H x \tag{52}$$
$$s.t. \tag{53}$$
$$\mu^T x = R \tag{54}$$
$$e^T x = 1 \tag{55}$$
$$Ix > 0 \tag{56}$$
$$\tag{57}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \tag{58}$$

$$\mu = \begin{bmatrix} 15.10 \\ 12.50 \\ 14.70 \\ 9.02 \\ 17.68 \end{bmatrix} \tag{59}$$

$$H = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 \end{bmatrix} \tag{60}$$

To conform with the notation used for the standard examples in the course, we can gather the 2 equality constraints into one, in the following way.

$$A = \begin{bmatrix} \mu^T \\ e^T \end{bmatrix} \tag{61}$$

The b matrix is thus altered to form.

$$b = \begin{bmatrix} R \\ 1 \end{bmatrix} \tag{62}$$

With these quantities we can then state, using that the non-negativity requirement can be stated as an inequality constraint, with C being the identity matrix with the same dimension as the number of assets and a a column vector of the same length as the number of assets.

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{63}$$

$$d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{64}$$

Thus the Quadratic Program can be simply stated as

$$\min_x \frac{1}{2} x^T H x \tag{65}$$

$$s.t. \tag{66}$$

$$Ax = b \tag{67}$$

$$Cx \geq d \tag{68}$$

2.

Due to the constraints on the QP, we are assuming that we are investing all available funds. As such, the smallest possible return would entail investing only in the security with the smallest return. This corresponds to the case where $x_4 = 1$ and all other components in the x vector are equal to zero. This would give a return of 9.02. Similarly the largest return would be found by investing all funds in the security with the largest return, that is, having $x_5 = 1$ and all other components 0. This would give a return of 17.68.

3.

We use the MATALB function *quadprog* to solve the QP defined in section 4.1.

```
%% 4.3 one solution

H = [2.30, 0.93, 0.62, 0.74, -0.23; 0.93, 1.40, 0.22, 0.56, 0.26;...
    0.62, 0.22, 1.80, 0.78, -0.27; 0.74, 0.56, 0.78, 3.40, -0.56;...
    -0.23, 0.26, -0.27, -0.56, 2.60];

mu = [15.10;12.50;14.70;9.02;17.68];
e = ones(5,1);

A = [mu';e'];
b = [10;1];

C = -eye(5);
d = zeros(size(C,1),1);


[x,var] = quadprog(H,[],C,d,A,b,[],[]);

figure(1)
title('Portfolio allocation using quadprog')
bar(x)
xticks([1 2 3 4 5])
xticklabels({'x_1', 'x_2', 'x_3', 'x_4', 'x_5'})
```

The resulting portfolio has a risk $\sigma^2 = 1.046$ and the optimal portfolia is illustrated in Figure 7.

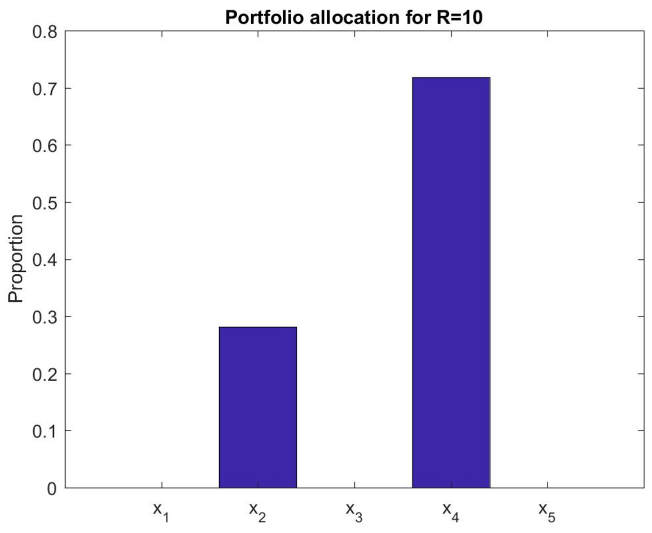


Figure 7: Portfolio allocation for a given return of 10, using the *quadprog* function in MATLAB.

4.

The efficient frontier is the set of portfolios that offer the highest level of return for a given level of risk. As we have seen, the risk is quantified by the quadratic form $x^T Hx$, while x, the optimal portfolio depends on the expected return. It follows that we can find an analytical expression for the risk in terms of the expected return by solving the QP for x. By using the KKT matrix we can obtain an analytical expression for the optimal portfolio x.

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix} \tag{69}$$

$$\rightarrow \tag{70}$$

$$\begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix}^{-1} \begin{bmatrix} -c \\ b \end{bmatrix} \tag{71}$$

In the textbook [Nocedal and Wright, 2006], an expression for the inverse is shown.

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix}^{-1} = \begin{bmatrix} C & E \\ E^T & F \end{bmatrix} \tag{72}$$

We are only concerned with the component E, the reason being that it suffices to find an analytical expression for x, as we have no linear component in the objective function for the problem ($c = 0$).

$$\begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} C & E \\ E^T & F \end{bmatrix} \begin{bmatrix} 0 \\ b \end{bmatrix} \tag{73}$$

$$\rightarrow \tag{74}$$

$$x = Eb \tag{75}$$

E is given by the following, stated in [Nocedal and Wright, 2006].

$$E = H^{-1}A^T(AH^{-1}A^T)^{-1} \tag{76}$$

All together, we can express the risk denoted by σ^2 as a function of R in the following way:

$$\sigma^2(R) = \frac{1}{2}(ER)^T H(ER) \tag{77}$$

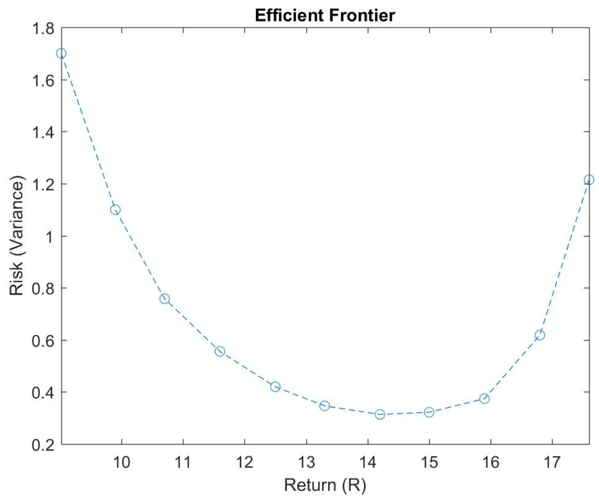
The optimal portfolio as a function of the return follows directly from the above considerations.

$$x(R) = ER \tag{78}$$

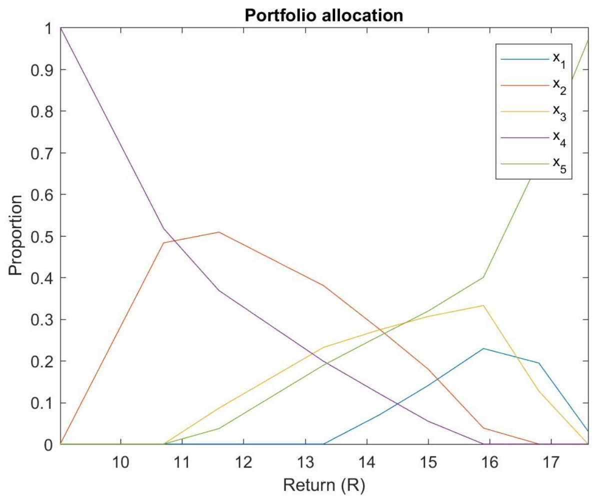
The same result can be arrived at by using a method such as the Schur-Complement or Null-Space methods. The constraint of no short selling introduces an inequality constraint, however, and so we can no longer use these methods. Instead, we can continue to use quadprog, for a range of returns R. Quadprog allows the implementation of the lower bound, either by specifying it directly or by defining an inequality constraint.

```
%% 4.4 Effective frontier

H = [2.30, 0.93, 0.62, 0.74, -0.23; 0.93, 1.40, 0.22, 0.56, 0.26;...
    0.62, 0.22, 1.80, 0.78, -0.27; 0.74, 0.56, 0.78, 3.40, -0.56;...
    -0.23, 0.26, -0.27, -0.56, 2.60];

mu = [15.10;12.50;14.70;9.02;17.68];
e = ones(5,1);

A = [mu';e'];

C = -eye(5);
d = zeros(size(C,1),1);

x_eff = zeros(5,11);
var_eff = zeros(1,11);
r = [9.02, 9.9,10.7,11.6,12.5,13.3,14.2, 15.0,15.9,16.8,17.6];

for i = 1:length(r)
    b=[r(i),1];
    [x_eff(:,i),var_eff(1,i)]=quadprog(H,[],C,d,A,b,[],[]);
end

figure(10)
plot(r,var_eff,'o--')
title('Efficient Frontier')
xlabel('Return (R)')
ylabel('Risk (Variance)')
```

```
xlim([min(r),max(r)])

plot(r,x_eff,'-')
title('Portfolio allocation')
xlabel('Return (R)')
ylabel('Proportion')
legend('x_1','x_2','x_3','x_4','x_5')
xlim([min(r),max(r)])
```



(a)



(b)

Figure 8: (a) Efficient frontier, computed by quadprog for a range of return. The efficient part is the part for which the return is higest for a given risk. This means that the efficient frontier is defined as the part of the graph to the right of the minimum of the parabola. (b) Optimal portfolio allocation as a function of return.

In general, we do not compute an initial point. This corresponds to taking a step, prior to running the algorithm. This may improve performance. On the small problems considered however, convergence has been so fast as to not make it that important.

5.

The risk free security corresponds to an asset in our financial market, which has no variance and no correlation with respect to the other objects in the market. The new return vector contains a new expected return, r_f, which comes in at the place of the corresponding security in the vector x.

$$\mu = \begin{bmatrix} 15.10 \\ 12.50 \\ 14.70 \\ 9.02 \\ 17.68 \\ 2.0 \end{bmatrix} \tag{79}$$

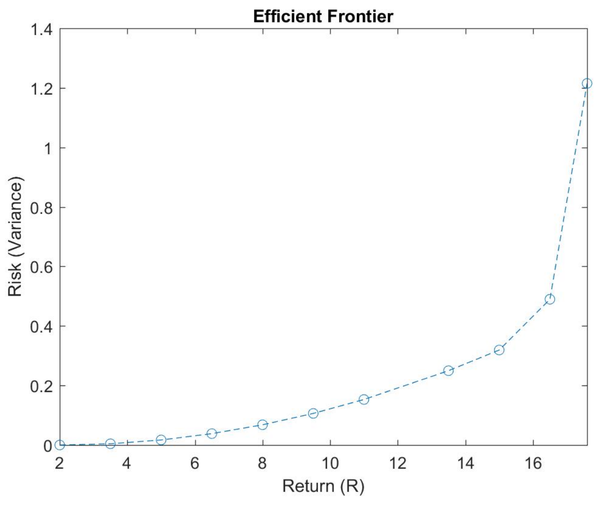
The new covariance matrix is singular, making sure that x_6 has no variance and no covariance with the remaining assets.

$$H = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{80}$$

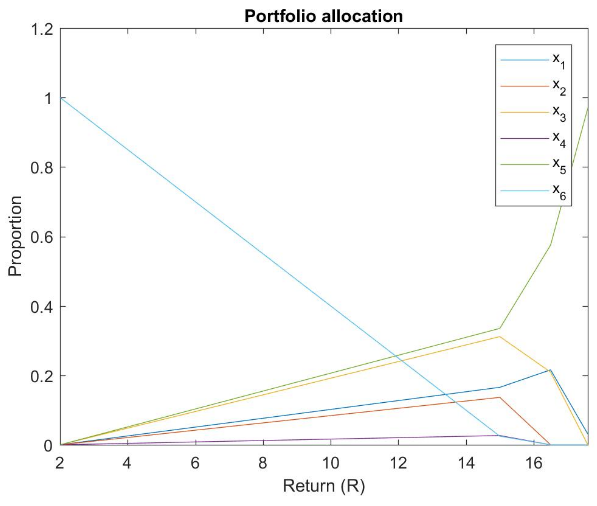
6.

```
%% 4.6 risk free

H = [2.30, 0.93, 0.62, 0.74, -0.23, 0; 0.93, 1.40, 0.22, 0.56, 0.26, 0;...
    0.62, 0.22, 1.80, 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];
mu = [15.10;12.50;14.70;9.02;17.68;2.0];

e = ones(6,1);

A = [mu';e'];



C = -eye(6);
d = zeros(size(C,1),1);

r = [2, 3.5,5,6.5,8.0,9.5,11.0, 13.5,15.0,16.5,17.6];

x_eff = zeros(6,length(r));
var_eff = zeros(1,length(r));
for i = 1:length(r)
    b=[r(i),1];
    [x_eff(:,i),var_eff(1,i)]=quadprog(H,[],C,d,A,b,[],[]);
end


```

```
figure(100)
plot(r,var_eff,'o--')
title('Efficient Frontier')
xlabel('Return (R)')
ylabel('Risk (Variance)')
xlim([min(r),max(r)])

figure(200)
plot(r,x_eff,'-')
title('Portfolio allocation')
xlabel('Return (R)')
ylabel('Proportion')
legend('x_1','x_2','x_3','x_4','x_5','x_6')
xlim([min(r),max(r)])
```



(a)



(b)

Figure 9: (a) The efficient frontier as a function of return, in the presence of a risk-free security. The risk-free security changes the shape of the efficient frontier, as it is no longer a parabola. The whole line is hence, efficient. (b) Optimal portfolio allocation as a function of return.

If we plot the original efficient frontier without the risk free security, together with the new frontier, with the risk-free asset, as well as the return-risk coordinate of each asset, we get the result seen in figure 10. The addition of the risk-free security makes it possible to achieve a return with a much lower risk than in the market without that particular asset.

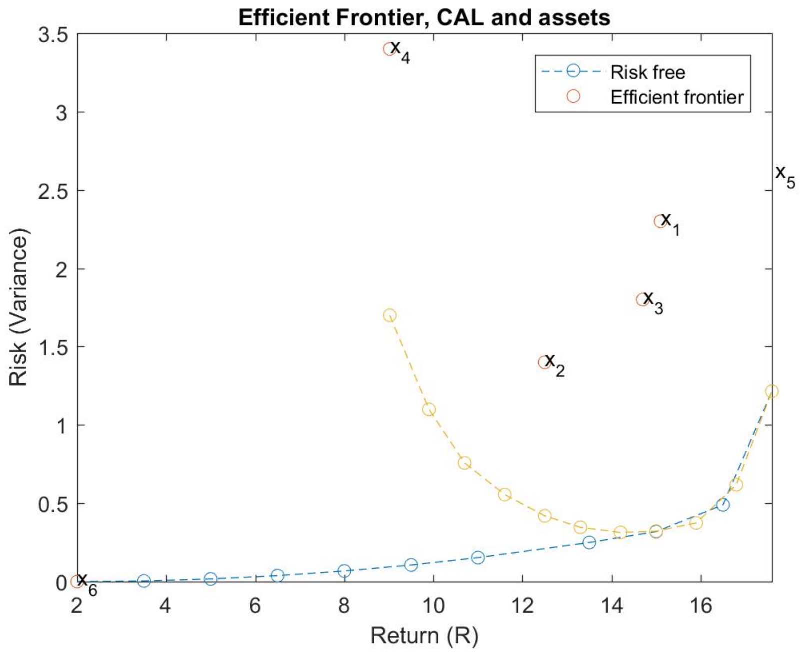


Figure 10: The return-risk coordinate of each of the assets as well as the efficient frontier in the presence of a risk free security and without it.

7.

From the previous subsection, we see that the efficient frontier for the two markets considered, are tangent at the point with return $R = 15.0$.

Problem 5 - Interior-Point Algorithm for Convex Quadratic Programming

We are interested in solving the following, general, quadratic program. We have n_A equality constraints and n_C inequality constraints.

$$\min_{x \in \mathcal{R}^n} \phi = \frac{1}{2} x^T H x + g^T x \tag{81}$$

$$s.t. \tag{82}$$

$$A^T x = b \tag{83}$$

$$C^T x \geq d \tag{84}$$

The Matrix H is a symmetric $n \times n$ matrix, A is a $n \times n_A$ matrix of equality constraints and C is a $n \times n_C$ matrix signifying the inequality constraints. b is $n_A \times 1$ and d is $n_C \times 1$. We begin the method by stating the optimality conditions and for this, we need to form the Lagrangian function.

1

We assume that what is meant by paper, is a pseudocode implementation of an interior point method for the problem. An interior point method uses strict feasibility in the iterations. When it comes to the practical implementation and the explanation of the primal-dual framework for the method, we will be using a predictor-corrector method as suggested in the textbook [Nocedal and Wright, 2006] We introduce vectors of Lagrange multipliers y, a $n_A \times 1$ vector tied to the equality constraints and z, a $n_C \times 1$ vector tied to the inequality constraints.

$$L(x, y, z) = \frac{1}{2} x^T H x - y^T (A^T x - b) - z^T (C^T x - d) \tag{85}$$

We introduce the slack vector $s \geq 0$, of the same dimension as z so that $s = C^T x - d$. This allows us to transform the inequality constraint to an equality constraint. The conditions on the gradient of the Lagrangian, as well as the complementary condition can then be stated. We form the gradient of the Lagrangian.

$$\nabla_{xyz} L(x, y, z) = \begin{bmatrix} Hx + g - Ay - Cz \\ -(A^T x - b) \\ -(C^T x - d) + s \end{bmatrix} = \begin{bmatrix} r_L \\ r_A \\ r_C \end{bmatrix} \tag{86}$$

We call the components of the gradient vector the residuals (as they are supposed to be equal to 0). rL, rA and rC respectively. That $s_i z_i = 0$ for $i = 1, 2, ..., n_C$ can be stated succintly as $SZe = 0$, where S and Z are diagonal matrices with the elements of s and z in the diagonal. e is a column vector of the same dimension. We call this residual r_{SZ}. So to sum up, we have the following optimality conditions.

$$F(x,y,z,s) = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{87}$$

$$(s, z \geq 0) \tag{88}$$

The roots of the function F are the solutions to the optimality conditions. We can thus use Newtons method to obtain these roots by iteratiion. We need the Jacobian of the Lagrangian gradient vector in equation 86.

$$J(x,y,z,s) = \begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \tag{89}$$

Newtons method gives us a linear system.

$$J(x,y,z,s) \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = -F(x,y,z,s) \tag{90}$$

The solution to this system defines in which direction we want the method to go at each iteration. We use α for the step length and so, for one step in the iterative model.

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \\ s_{k+1} \end{bmatrix} \leftarrow \begin{bmatrix} x_k \\ y_k \\ z_k \\ s_k \end{bmatrix} + \alpha \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} \tag{91}$$

This is just a version of Newtons method, but the want to take the non-negativity constraints into use. These iterates should follow the central path. We introduce a perturbation to the function F, defined as.

$$F(x_\tau, y_\tau, z_\tau, s_\tau) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \tau e \end{bmatrix} \tag{92}$$

The central path is the curve on which $\tau > 0$. The idea is to reduce τ at each step along the iterative progress of this path. As τ gets smaller the perturbed problem converges towards the optimality conditions. With the perturbation, the Newton step can again be used, but in order to control step size, we introduce a centering parameter $\sigma \in [0, 1]$ as well as the duality measure μ.

$$\mu = \frac{s^T z}{n_C} \tag{93}$$

This measure quantifies how much the inner product of s and z changes in each iteration. As we want to reduce it to converge towards a solution to the optimality conditions, a small decrease would entail a need for greater centering and vice versa. This is encapsulated in writing the following.

$$\tau = \sigma\mu \tag{94}$$

We call $\sigma = 0$ an affine step and $\sigma = 1$ a centering step. This completes the specification of the interior point method. We can state it more succintly by using pseudocode. Note that we have not provided a heuristic for choosing σ. We also need stopping condiitions for the algorithm, but these are assumed to be provided in the algorithm as tolerances and a cap on the number of iterations.

Algorithm 1 Interior Point Algorithm for QP

```
Input: $x_0, y_0, z_0, s_0$ with $(z_0, s_0 > 0)$
k
Maxit
**while** $tol$ & $k < maxit$ **do**
    $\mu_k = \frac{s_k^T z_k}{n_C}$
    $\sigma \in [0,1]$
    $r_L = Hx_k + g - Ay_k - Cz_k$
    $r_A = -A^T x_k + b$
    $r_C = -C^T x_k + d + s_k$
    $r_S Z = S_k Z_k e$
    solve
    $\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S_k & Z_k \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta y_k \\ \Delta z_k \\ \Delta s_k \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} + \sigma_k \mu_k e \end{bmatrix}$
    Choose step size$\alpha_k \in [0, \alpha_k^{max}]$
    $\begin{bmatrix} z_k \\ s_k \end{bmatrix} + \alpha_k^{max} \begin{bmatrix} \Delta z \\ \Delta s \end{bmatrix} \geq 0$
    Update iterations
    $\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \\ s_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ z_k \\ s_k \end{bmatrix} + \alpha_k \begin{bmatrix} \Delta x_k \\ \Delta y_k \\ \Delta z_k \\ \Delta s_k \end{bmatrix}$
**end while**
```

2

Now to actually implement a variation of the interior point algorithm, we need a heuristic for choosing step size and σ. This entails a so-called predictor-corrector algorithm to be applied. A step in the algorithm for $\sigma = 0$, we call an affine step. We use the superscript "aff" to denote an affine step. After such a step, we need to use a step size that preserves non-negativity, while still making some headway. That is, a a^{aff} so that.

$$z + \alpha^{aff} \Delta z^{aff} \geq 0 \tag{95}$$

$$s + \alpha^{aff} \Delta s^{aff} \geq 0 \tag{96}$$

(97)

The choice here depends on the sign of $\Delta z, \Delta s$. The same calculation is used for both for simplicity. If $\Delta z \geq 0$, we can choose α^{aff} to be 1. If on the other hand, the step is negative, in order to preserve the inequality.

$$z + \alpha^{aff} \Delta z = 0 \tag{98}$$

$$\rightarrow \tag{99}$$

$$\alpha^{aff} = \frac{-z}{\Delta z} \tag{100}$$

The duality measure for this affine step.

$$\mu^{aff} = \frac{(z + \alpha^{aff} \Delta z^{aff})^T (s + \alpha^{aff} \Delta s^{aff})}{n_C} \tag{101}$$

We now have a complementarity measure and a predicted complementarity measure. If the affine measure is much smaller than the usual measure, the affine step is good as we see a large reduction. We can use this to define σ, as a tradeoff between centering the step and reducing the complementarity measure.

$$\sigma = \left(\frac{\mu^{aff}}{\mu}\right)^3 \tag{102}$$

The affine step introduces a linearization error. The corrector step is introduced to compensate. This error can be found by expanding the numerator of equation 101, resulting in $\Delta S \Delta Z e$. Gathering the centering and corrector step into one, we get a linear system for the directions.

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} - \Delta S \Delta Z e + \sigma \mu e \end{bmatrix} \tag{103}$$

The left hand side is the same as for the predictor step.

3.

For the implementation of the predictor-corrector method, it may be prudent to introduce some matrix operations as well as a practical way of choosing the step size. In the lectures, we were presented with a method of factoring the LHS represented by the jacobian, to make the calculations more manageable. We don't think there is much to be gained in understanding by reproducing the derivations here, as they can by found in the course material, week 6. The full algorithm is called Algorithm 2 and is given here as pseudocode and as implemented in a matlab function.

Algorithm 2 Interior Point, Predictor-Corrector Algorithm for QP

```
Input: $x_0, y_0, z_0, s_0$ with $(z_0, s_0 > 0)$
k
Maxit
$r_L = Hx + g - Ay - Cz$
$r_A = -A^Tx + b$
$r_C = -C^Tx + d + s$
$r_SZ = SZe$

while $tol$ & $k < maxit$ do
    $\mu_k = \frac{s^Tz}{n_C}$

    $\bar{H} = H + C(S^{-1}Z)C^T$

    Factor $\begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix}$

    Affine Step

    $\bar{r}_L = r_L - C(S^{-1}Z)r_C - Z^{-1}r_{SZ}$

    Solve for step, using LDL

    $\begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
    $\Delta z^{aff} = -(S^{-1}Z)C^T\Delta x^{aff} + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
    $\Delta s^{aff} = -Z^{-1}r_{SZ} - Z^{-1}S\Delta z^{aff}$

    $\lambda = \begin{bmatrix} z \\ s \end{bmatrix}$
    $\alpha_{aff} = \min_i(1, \min\left(\frac{-\lambda_i}{\Delta\lambda_i^{aff}}\right))$

    $\mu_{aff} = (z + \alpha_{aff}\Delta z_{aff})^T(s + \alpha_{aff}\Delta s_{aff})/n_C$

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

    Centering – Correcting Step

    $\bar{r}_{SZ} = r_{SZ} + \Delta S_{aff}\Delta Z_{aff}e - \sigma\mu e$
    $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{SZ})$
    Solve for step, using same LDL as before

    $\begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
    $\alpha = \min_i(1, \min\left(\frac{-\lambda_i}{\Delta\lambda_i}\right))$

    Update steps and residuals. $\bar{\alpha} = \eta\alpha, x = x + \bar{\alpha}\Delta x, y = y + \bar{\alpha}\Delta y, z = z + \bar{\alpha}\Delta z, s = s + \bar{\alpha}\Delta s$
end while
```

```
function [x_sol,y_sol,z_sol,s_sol,k] = QPippd10(H,g,C,d,A,b,x,y,z,s)
% QPIPPD   Primal-Dual Interior-Point QP Solver
%
%          min  (1/2)x'Hx+g'x
%           x
%          s.t. A x  = b
%                C x >= d
%
% Syntax: [x_s,y_s,z_s,s_s,k] = QPIPPD10(H,g,C,d,A,b,x,y,z,S)
%
% Called "10" as this is the 10'th variation
% we have made.

eta = 0.995;

%residuals. Dimensions are not all used,
% useful for checking

[mA,nA]=size(A);
[mC,nC]=size(C);
e = ones(nC,1);
Z = diag(z);
S = diag(s);


rL = H*x+g-A*y-C*z;
rA = -A'*x+b;
rC = -C'*x+s+d;
rSZ = S*Z*e;
mu = z'*s/nC;

% iteration stopping criteria
k = 0;
maxit = 100;
tolL = 1.0e-9;
tolA = 1.0e-9;
tolC = 1.0e-9;
tolmu = 1.0e-9;



while (k<=maxit && norm(rL)>=tolL && norm(rA)>=tolA && norm(rC)>=tolC ...
        && abs(mu)>=tolmu)
    % factorization of the lhs using LDL
    H_bar = H + C*(S\Z)*C';

    lhs = [H_bar, -A;-A',zeros(size(A,2))];
    [L,D,P] = ldl(lhs,'lower','vector');

    % affine direction

    rL_bar = rL-C*(S\Z)*(rC-Z\rSZ);
    rhs = -[rL_bar;rA];
    dxy_a(P,:) = L'\(D\(L\(rhs(P,:))));

    dx_a = dxy_a(1:length(x));
    dy_a = dxy_a(length(x)+1:length(x)+length(y));

    dz_a = -(S\Z)*C'*dx_a+(S\Z)*(rC-Z\rSZ);
    ds_a = -(Z\rSZ)-(Z\(S*dz_a));

    % compute largest alpha so we preserve complementarity

    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end


    % affine duality gap

    mu_a = ((z+alpha_a*dz_a)'*(s+alpha_a*ds_a))/nC;

    % centering parameter (conventional choice)

    sigma = (mu_a/mu)^3;

    % corrector, using same factorization of lhs

    rSZ_bar = rSZ + diag(ds_a)*diag(dz_a)*e - sigma*mu*e;
    rL_bar = rL-C*(S\Z)*(rC-Z\rSZ_bar);
    rhs = -[rL_bar;rA];
    dxy(P,:) = (L'\(D\(L\(rhs(P,:)))));
    dx = dxy(1:length(x));
    dy = dxy(length(x)+1:length(x)+length(y));

    dz = -(S\Z)*C'*dx+(S\Z)*(rC-Z\rSZ_bar);
    ds = -Z\rSZ_bar-Z\S*dz;


    % alpha in the corrector step

    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    % update solutions

    x = x + eta*alpha*dx;
    y = y + eta*alpha*dy;
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;

    Z = diag(z);
    S = diag(s);
    k = k + 1;

    rL = H*x + g - A*y- C*z;
    rA = -A'*x+b;
    rC = -C'*x + s + d;
    rSZ = S*Z*e;
    mu =z'*s/nC;

end

x_sol = x;
y_sol = y;
z_sol = z;
s_sol = s;
```

4.

This question was answered back in section 4. We think that the reason it is asked again is that we don't we don't have to specify the lower bound as an inequality constraint in the problem when solving with quadprog, but we do here. In his case we have.

$$H = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{104}$$

$$g = 0 \tag{105}$$

$$A = \begin{bmatrix} 15.10 & 1 \\ 12.50 & 1 \\ 14.70 & 1 \\ 9.02 & 1 \\ 17.68 & 1 \\ 2.0 & 1 \end{bmatrix} \tag{106}$$

$$b = \begin{bmatrix} 15 \\ 1 \end{bmatrix} \tag{107}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{108}$$

$$d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{109}$$

5.

Applied to the Markowitz Portfolio problem with a risk-free security, the results look indistinguishable from the results using quadprog.

```
%% 5 Effective frontier using interior point, risk free

H = [2.30, 0.93, 0.62, 0.74, -0.23, 0; 0.93, 1.40, 0.22, 0.56, 0.26, 0;...
    0.62, 0.22, 1.80, 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];
mu = [15.10;12.50;14.70;9.02;17.68;2.0];

e1 = ones(6,1);
A = [mu,e1];
g=0;
%b = [10;1];

x = ones(size(H,1),1);
y = ones(size(A,2),1);
z = ones(size(x));

C = eye(length(x));
d = zeros(length(x),1);
S = 2.*ones(size(C,1),1);

x_eff = zeros(6,11);
y_eff = zeros(2,11);
z_eff = zeros(6,11);
s_eff = zeros(6,11);
r = [2, 3.5,5,6.5,8.0,9.5,11.0, 13.5,15.0,16.5,17.6];
var_eff = zeros(1,11);


for i=1:length(r)
    b = [r(i);1];
    [x_eff(:,i),y_eff(:,i),z_eff(:,i),s_eff(:,i),iter]=QPippd10(H,g,C,d,A,b,x,y,z,S);
    var_eff(1,i)=x_eff(:,i)'*H*x_eff(:,i);
end


figure(10)
plot(r,var_eff,'o--')
title('Efficient Frontier, interior point')
xlabel('Return (R)')
ylabel('Risk (Variance)')
xlim([min(r),max(r)])
figure(20)
plot(r,x_eff,'-')
title('Portfolio allocation, interior point')
xlabel('Return (R)')
ylabel('Proportion')
```

```
legend('x_1','x_2','x_3','x_4','x_5','x_6')
xlim([min(r),max(r)])
```

The results are shown in figure 11. Sometimes we call the market with a risk-free security the risk-free market. This is a misnomer off course, as the other assets still have risk.

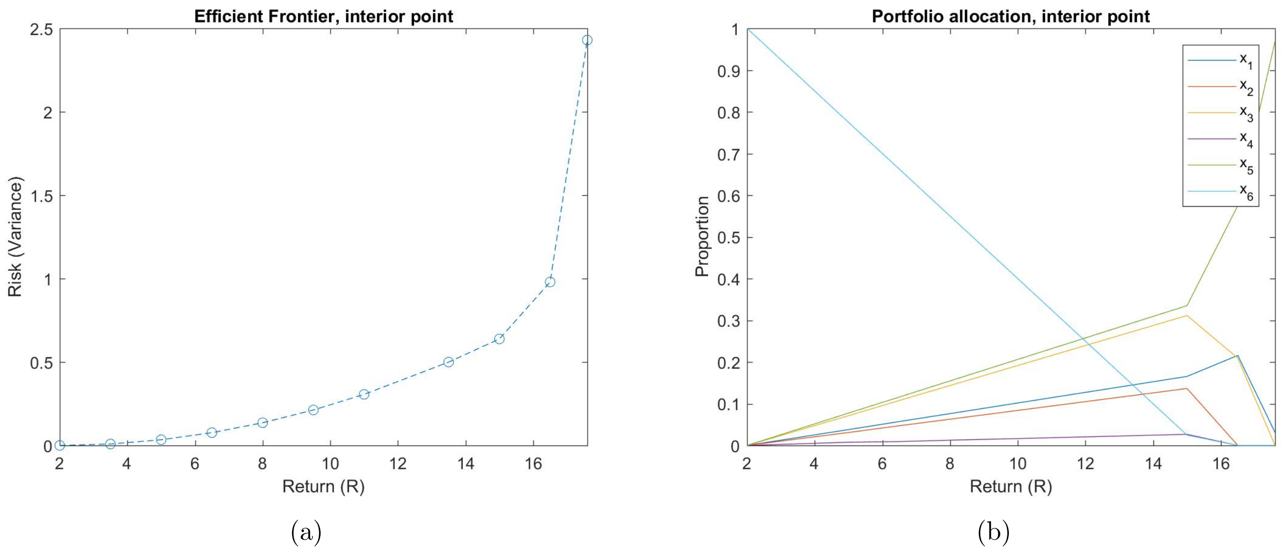


Figure 11: (a)The risk-free version of the Markowitz problem, sing the interior point method. Very fast convergence, with 7 iterations being the standard. (b)
Portfolio allocation for the risk-free Markowitz.

6.

This problem is a bit different from the previous, in that it is inequality constrained, with no equality constraints. The absence of a constraint necessitates that we use some form of branching in the code to avoid evaluating expressions with empty matrices. In this assignment, we saw it easier to make a version of the code, that did not contain the equality constraints, but the extension to the general case is trivial. The result of applying the Interior-Point, Predictor-corrector algorithm can be seen in figure 12. For completeness, we include the code used as well, so that the results can be reproduced. We immediately see the large difference from the active set method in the way the iterates behave. In short, while the active set method is restricted to the periphery of the feasible region, the interior point methods operate in the interior. This is a trivial observation, but it is nice to have it confirmed graphically.

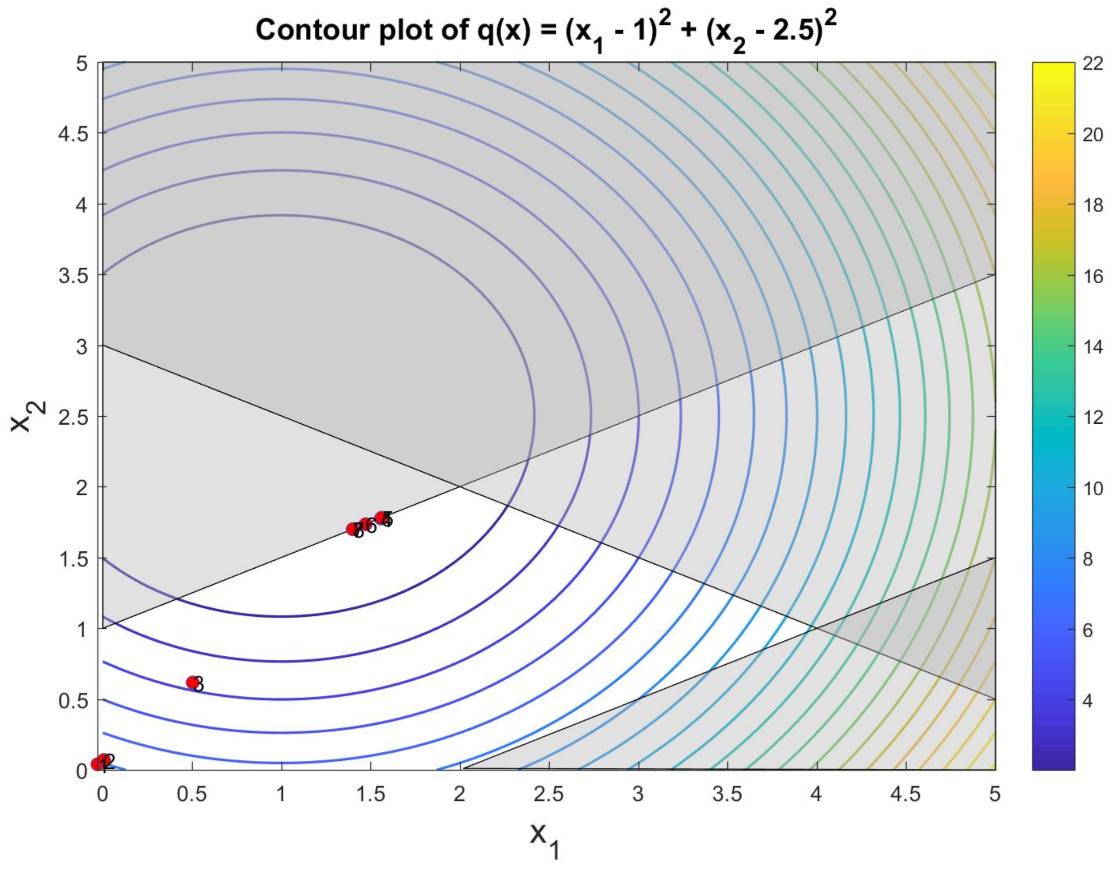


Figure 12: An iteration sequence for the Interior-Point Primal-Dual Predictor-Corrector, applied to the problem from section 3. The algorithm was started in $(x_1, x_2) = (0, 0)$ and converged in 8 steps. s and z were set as small multiples of unity vectors.

```matlab
function [x_s,z_s,s_s,k,seq] = QPippd11(H,g,C,d,x,z,s)
% QPIPPD   Primal-Dual Interior-Point QP Solver
%
%          min  (1/2)x'Hx+g'x
%           x
%          s.t. A x  = b
%                 C x >= d
%
% Syntax: [x_s,y_s,z_s,s_s,k] = QPIPPD(H,g,C,d,A,b,x,y,z,S)
%

eta = 0.995;

%residuals


[mC,nC]=size(C);
e = ones(nC,1);
Z = diag(z);
```

```matlab
S = diag(s);

rL = H*x+g-C*z;
rC = -C'*x+s+d;
rSZ = S*Z*e;
mu = (z'*s)/nC;

% iteration stopping criteria
k = 0;
maxit = 100;
tolL = 1.0e-9;
tolC = 1.0e-9;
tolmu = 1.0e-9;



while (k<=maxit && norm(rL)>=tolL &&  norm(rC)>=tolC ...
        && abs(mu)>=tolmu)
    % factorization of the lhs using LDL (use others, comment out)
    H_bar = H + C*(S\Z)*C';

    lhs = [H_bar];
    [L,D,P] = ldl(lhs,'lower','vector');

    % affine direction

    rL_bar = rL-C*(S\Z)*(rC-Z\rSZ);

    rhs = -[rL_bar];
    dxy_a(P,:) = L'\(D\(L\(rhs(P,:))));

    dx_a = dxy_a(1:length(x));


    dz_a = -(S\Z)*C'*dx_a+(S\Z)*(rC-Z\rSZ);
    ds_a = -(Z\rSZ)-(Z\S*dz_a);

    % compute largest alpha so we preserve complementarity

    alpha_a = 1;
    idx_z = find(dz_a<0);
    if (isempty(idx_z)==0)
        alpha_a = min(alpha_a,min(-z(idx_z)./dz_a(idx_z)));
    end
    idx_s = find(ds_a<0);
    if (isempty(idx_s)==0)
```

```
        alpha_a = min(alpha_a,min(-s(idx_s)./ds_a(idx_s)));
    end


    % affine duality gap

    mu_a = (z+alpha_a*dz_a)'*(s+alpha_a*ds_a)/nC;

    % centering parameter (conventional choice)

    sigma = (mu_a/mu)^3;

    % solution of system, using same factorization of lhs

    rSZ_bar = rSZ + diag(ds_a)*diag(dz_a)*e - sigma*mu*e;
    rL_bar = rL-C*(S\Z)*(rC-Z\rSZ_bar);
    rhs = -[rL_bar];
    dxy(P,:) = (L'\(D\(L\(rhs(P,:)))));
    dx = dxy(1:length(x));

    dz = -(S\Z)*C'*dx+(S\Z)*(rC-Z\rSZ_bar);
    ds = -Z\rSZ_bar-Z\S*dz;


    % for alpha

    alpha = 1;
    idx_z = find(dz<0);
    if (isempty(idx_z)==0)
        alpha = min(alpha,min(-z(idx_z)./dz(idx_z)));
    end
    idx_s = find(ds<0);
    if (isempty(idx_s)==0)
        alpha = min(alpha,min(-s(idx_s)./ds(idx_s)));
    end

    x = x + eta*alpha*dx;
    z = z + eta*alpha*dz;
    s = s + eta*alpha*ds;


    Z = diag(z);
    S = diag(s);
    k = k + 1;

    rL = H*x + g - C*z;
    rC = -C'*x + s + d;
    rSZ = S*Z*e;
    mu =(z'*s)/nC;

    seq(:,k)=x;
end

x_s = x;

z_s = z;
s_s = S;
```

```
H = [2 , 0; 0, 2];
C = [1,-1,-1,1,0;-2,-2,2,0,1];
d = [-2;-6;-2;0;0];
g = -[2;5];
x = zeros(2,1);
z = 0.1*ones(5,1);
s = 0.1*ones(size(z));
e = ones(5,1);
Z = diag(z);
S = diag(s);

[x_sol,z_sol,s_sol,iter,seq]=QPippd11(H,g,C,d,x,z,s);



%Make a contour plot of the problem

x = linspace(0,5,100);
y = linspace(0,5,100);

[X,Y] = meshgrid(x,y);

Fun = @(x,y) (x-1).^2 + (y - 2.5).^2;
F = Fun(X,Y);
v = [min(min(F)):1:-2, -1. 99:0.2:2, 2.01:max(max(F))];
contour(X,Y,F,v,'linewidth',1.2);
hold on;
yc1 = (x./2 + 1);
yc2 = (-x./2 + 3);
yc3 = (x./2 - 1);
yc3_ny = yc3(yc3>0);

fill([x x(end) x(1)],[yc1 max(y) max(y)], [0.7 0.7 0.7], 'facealpha',0.4);
fill([x x(end) x(1)],[yc2 max(y) max(y)], [0.7 0.7 0.7], 'facealpha',0.4);
fill([x(yc3>0),x(end)] ,[yc3_ny, 0],[0.7 0.7 0.7], 'facealpha',0.4);
xlabel('x_{1}','fontsize',16);
ylabel('x_{2}','fontsize',16);
title('Contour plot of q(x) = (x_{1} - 1)^{2} + (x_{2} - 2.5)^{2}','fontsize',14)
scatter(seq(1,:),seq(2,:), 'o', 'MarkerFaceColor', 'r')
labels = cellstr(["1","2","3","4","5","6","7","8"]);
text(seq(1,:),seq(2,:),labels)

colorbar;
```

References

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer.