Constrained Optimization

Georgios Tsimplis s200166

May 2020

Responsible Professor: John Bagterp Jørgensen

Contents

1 Equality Constrained Convex QP

According to Constrained Optimization's theory, a problem in which the objective function is in quadratic form and the constraints are linear is called quadratic program. Let us examine some properties of Equality – Constrained Convex Quadratic Programs which can be found in the following matrix notation.

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = \frac{1}{2}x'Hx + g'x \\ s.t \quad & A'x = b \end{aligned}$$

$$with\ H > 0.$$

We assume that we have n variables, m constraints, $H\in \mathbb{R}^{nxn}$ is symmetric and positive-definite, $g \in \mathbb{R}^n$ and $A\in \mathbb{R}^{nxm}$ is a matrix in which every column includes the coefficients of the m constraints and has full column rank.

1.1 The Lagrangian function

In mathematical optimization the Lagrangian function is a linear combination of the objective function with the functions of constraints. This function gives us a relationship between the function that we want to minimize and the constraints. For the above problem, the Lagrangian function is:

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

Where $\lambda \in \mathbb{R}^m$ is a column vector which contains the Lagrange multipliers.

1.2 First Order Optimality Conditions

The first order optimality conditions of the above problem are:

$$\begin{aligned} \nabla_x\mathscr{L} = Hx + g - A\lambda = 0 \\ A'x - b = 0 \end{aligned}$$

which arise from the partial derivatives of the Lagrangian in respect to x and λ. In addition, these conditions are sufficient because H matrix is positive definite so the scalar $x'Hx$ is strictly positive for every non-zero vector x. Due to the above, our objective function is convex so in the case that one point x satisfies the first order optimality conditions we have the optimum

solution. The above system, is a system of linear equations which can be expressed in a matrix notation as :

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

$$K \quad z \quad = d$$

This system is called KKT system and K is non-singular and symmetric.

1.3 Solving the KKT system

In order to solve the KKT system we have to implement solvers. We have the option to choose among solvers, each of which is based on different kinds of factorizations and solve the problem but is also more suitable in specific cases. Below, we will examine the pseudo-code as well as the matlab implementation of LU, LDL, Range-Space and Null Space factorizations.

LU factorization

$$PK = LU$$

Where P is the permutation matrix, L is a lower triangular matrix and U is an upper triangular matrix. To solve the KKT system we implement the following steps:
1) Factorize the K matrix according to PK=LU.
2) Obtain the solution z.

```
function [x,lambda]=EqualityQPSolverLUdense(H,g,A,b)
n=length(g);
m=length(b);
z1=zeros(m);
K=[H -A;-A' z1];
if det(K)~=0
    d=-[g;b];
    [L,U,p]=lu(K,'vector');
    z=U\(L\d(p));
    x=z(1:n);
    lambda=z(n+1:end);
else
    disp('There is not a unique minimizer')
end
end
```

This method of factoring the KKT matrix is very effective on many cases. However, this method may be expensive, if the process of choosing the permutation matrix lead us to a more dense L matrix than the KKT matrix.

LDL factorization

$$P'KP = LDL'$$

Where P is the permutation matrix, L is a lower triangular matrix and D is an block diagonal matrix.
1) Create a $(m+n) \times 1$ vector of zeros.
2) Factorize the K matrix according to $P'KP = LDL'$.
3) Obtain the solution z.

```
function [x,lambda]=EqualityQPSolverLDLdense(H,g,A,b)
n=length(g);
m=length(b);
z1=zeros(m);
K=[H -A;-A' z1];
d=-[g;b];
if det(K)~=0
    z=zeros(m+n,1);
    [L,D,p]=ldl(K,'lower','vector');
    z(p)=L'\(D\(L\d(p)));
    x=z(1:n);
    lambda=z(n+1:end);
else
    disp('There is not a unique minimizer')
end
end
```

Range-Space factorization

1) Implement Cholesky factorization $H = LL'$
2) Solve $Hv = g$ for v
3) Form $H_A = A'H^{-1}A = L_A L'_A$
4) Solve $H_A\lambda = b + A'v$ and $Hx = A\lambda - g$

```
function [x,lambda]=EqualityQPSolverRS(H,g,A,b)
if det(H)~=0
    L = chol(H,'lower');
    v = L'\(L\g);
    HA = A'*H^-1*A;
```

```
6       lambda = HA\(b+A'*v);
7       x = H\(A*lambda-g);
8   else
9       disp('There is not a unique minimizer')
10  end
11  end
```

Range-Space factorization is very useful in the case that H can be inverted easily (diagonal, or block diagonal) as well as in the case that the number of constraints m is small.

Null-Space factorization:Orthonormal basis

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_1 \; Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 \; R$$

1)QR Factorization
2) Solve $R' x_Y = b$
3) Solve $(Q_2' H Q_2) x_z = -Q_2'(H Q_1 x_Y + g)$
4) Compute $x = Q_1 x_Y + Q_2 x_z$
5) Solve $R\lambda = Q_1'(Hx + g)$

```
1  function [x,lambda]=EqualityQPSolverNS(H,g,A,b)
2  n = length(g);
3  [Q,Rbar] = qr(A);
4  m1 = size(Rbar,2);
5  Q1 = Q(:,1:m1);
6  Q2 = Q(:,m1+1:n);
7  R = Rbar(1:m1,1:m1);
8  xy = R'\b;
9  xz = (Q2'*H*Q2)\(-Q2'*(H*Q1*xy+g));
10 x = Q1*xy+Q2*xz;
11 lambda = R\(Q1'*(H*x+g));
12 end
```

The Null-Space method is quite beneficial in the case that the number of degrees of freedom n-m is small.
Concerning the Sparse methods of LU and LDL factorization, in these cases the KKT matrix is represented as a sparse matrix but the whole procedure is the same. In the appendix is provided a function with the interface:
[x,lambda]=EqualityQPSolver(H,g,A,b,solver)
which use as a flag(solver) the key-words 'LUdense' , 'LUsparse', 'LDLdense',

'LDLsparse', 'NullSpace' and 'RangeSpace' to switch among the different factorizations.

1.4 Implementation

In order to test our matlab functions we will initially solve the following example:

$$\min q(x) = 3x_1^2 + 2x_1x_2 + x_1x_3 + 2.5x_2^2 + 2x_2x_3 + 2x_3^2 - 8x_1 - 3x_2 - 3x_3$$

$$st. \quad x_1 + x_3 = 3, \quad x_2 + x_3 = 0$$

from the book "Numerical Optimization"(J. Nocedal-S. J. Wright ,2000) The matlab script for the test is the following:

```
%==============Driver File for 1.4==============%
%==============Input 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];
%==============Switch among solvers==============%
disp("==========LU-Dense========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'LUdense')
disp("==========LU-Sparce========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'LUsparse')
disp("==========LDL-Dense========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'LDLdense')
disp("==========LDL-Sparse========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'LDLsparse')
disp("==========NullSpace========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'NullSpace')
disp("==========RangeSpace========")
[x,lambda]=EqualityQPSolver(H,g,A,b,'RangeSpace')
```

The results that we had using the different solvers were the same:$(x_1, x_2, x_3) = (2, -1, 1)$ and $(\lambda_1, \lambda_2) = (3, -2)$ In addition, with purpose to compare the time performance of the different solvers as well as the behavior that they have regarding the different number of constraints we created a random symmetric and positive definite matrix and we started with one constraint. We created a loop in the script in a way that holds static the previous constraints as well as the objective function and in every next iteration adds to the problem one new constraint so that the A has a full column rank.

The above problem has $n = 6$ variables and $m = 1, 2, ..., 5$ constraints. In the appendix you can find the script of this test. The following line chart represents the performance of the different solvers in a random problem as the number constraints is increased.

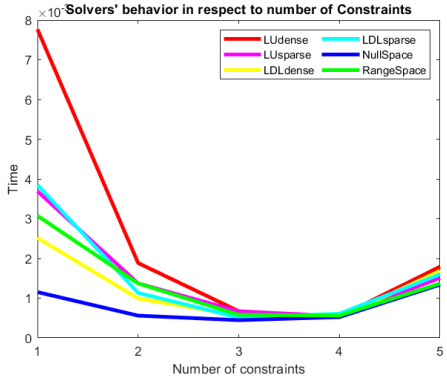


Figure 1: Solvers Performance

2 Quadratic Program (QP)

Consider the quadratic program in the following form:

$$\min_{x \in \mathbb{R}^n} \quad \phi = \frac{1}{2} x' H x + g' x$$

$$s.t \quad A'x = b$$

$$l \leq x \leq u$$

We assume that H is positive semi-definite so the objective function is convex.

For simplicity reasons: $\begin{cases} x \geq l \\ x \leq u \end{cases} \Leftrightarrow \begin{cases} x \geq l \\ -x \geq -u \end{cases} \Leftrightarrow \begin{bmatrix} I \\ -I \end{bmatrix} x \geq \begin{bmatrix} l \\ -u \end{bmatrix}$

So we have now the quadratic program in the form:

$$\min_{x \in \mathbb{R}^n} \quad \phi = \frac{1}{2} x' H x + g' x$$

$$s.t \quad A'x = b$$

$$C'x \geq d$$

where $C = [I \ \ -I]$ and $d = \begin{bmatrix} l \\ -u \end{bmatrix}$

2.1 The Lagrangian function

The Lagrangian of the above QP is:

$$\mathscr{L}(x, \lambda, z) = \frac{1}{2}x'Hx + g'x - \lambda'(A'x - b) - z'(C'x - d)$$

2.2 The Optimality Conditions

If we add slack variables to inequality constraints

$$C'x - d \geq 0 \Leftrightarrow -Cx + s + d = 0$$

where $s \geq 0$ the necessary and sufficient optimality conditions are:

$$r_L = Hx + g - Ay - Cz = 0$$
$$r_A = -A'x + b = 0$$
$$r_c = -C'x + s + d = 0$$
$$z \geq 0 \ , \ s \geq 0$$
$$s_i z_i = 0 \quad i = 1, 2, ...m_c$$

2.3 Primal-Dual Interior-Point Algorithm

The main idea of the interior point algorithm is to iteratively approach the optimal following a path into the feasible set. The algorithm is trying to find a search direction making steps in any iteration before s and z take negative values. Specifically, primal dual interior point algorithm take the advantage of duality conditions to adapt the search direction. In the following description we use the notation: S=diag(s) , Z=diag(z) and $e = [1, 1, \dots 1]^T$.

Algorithm 1 Primal-Dual Predictor-Corrector Interior-Point Algorithm

Require: (x, y, z, s) such that $(z, s) > 0$

Compute the residuals:

$r_L = Hx + g - Ay - Cz,\ r_A = b - A'x,\ r_C = s + d - C'x,\ r_{sz} = SZe,\ \mu = (z's)/m_c$

State convergence conditions:

$$||r_L|| \leq \epsilon_L,\ ||r_A|| \leq \epsilon_A,\ ||r_C|| \leq \epsilon_C,\ |\mu| \leq \epsilon_\mu$$

while not converged **do**

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

Compute LDL-factorizaton of $\begin{bmatrix} \bar{H} & -A \\ -A' & 0 \end{bmatrix}$

Find Affine Direction

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

Solve $\begin{bmatrix} \bar{H} & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$ with LDL factorization

Compute $\Delta z^{aff} = -(S^{-1}Z)C'\Delta x^{aff} + (S^{-1}Z)(r_C - Z^{-1}r_{sz})$

Compute $\Delta s^{aff} = -Z^{-1}r_{sz} - Z^{-1}S\Delta z^{aff}$

Compute the largest α^{aff} such that:

$z + \alpha^{aff}\Delta z^{aff} \geq 0$ and $s + \alpha^{aff}\Delta s^{aff} \geq 0$

Find the duality gap and the centering parameter

$\mu^{aff} = (z + \alpha^{aff}\Delta z^{aff})'(s + \alpha^{aff}\Delta s^{aff})/m_c$, $\sigma = (\mu^{aff}/\mu)^3$

Affine-Centering-Correction Direction

Compute $\bar{r}_{sz} = r_{sz} + \Delta S^{aff}\Delta Z^{aff}e - \sigma\mu e$

Compute $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$

Solve $\begin{bmatrix} \bar{H} & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$ with LDL factorization

Compute $\Delta z = -(S^{-1}Z)C'\Delta x + (S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$

Compute $\Delta s = -Z^{-1}\bar{r}_{sz} - Z^{-1}S\Delta z$

Compute the largest α such that: $z + \alpha\Delta z \geq 0$ and $s + \alpha\Delta s \geq 0$

Set $\bar{\alpha} = 0.995\alpha$ and **Update the iteration**

Compute $x = x + \bar{\alpha}\Delta x$, $y = y + \bar{\alpha}\Delta y$, $z = z + \bar{\alpha}\Delta z$, $s = s + \bar{\alpha}\Delta s$

Compute the residuals:

$r_L = Hx + g - Ay - Cz,\ r_A = b - A'x,\ r_C = s + d - C'x,\ r_{sz} = SZe,$

$\mu = (z's)/m_c$

Check convergence conditions

end while

2.4 Implementation Of Interior-Point Algorithm

With the purpose to implement the interior point algorithm it was necessary to make a Matlab function which takes as inputs the matrices H, g, A, b, C, d and some intitial value $x0$. In the appendix of this report there is the code of the function of the primal-dual interior - point algorithm. In order to test the implementation of the problem according to the given form, we will solve the following simple QP.

$$\min_{x \in \mathbb{R}^2} \quad \phi = x_1^2 + x_2^2 - 2x_1 - 4x_2$$

$$s.t \quad x_1 + x_2 = 0$$
$$-3 \leq x_1 \leq 4$$
$$-3 \leq x_2 \leq 4$$

According to pre-solving steps that are mentioned in the above paragraphs we define the following matrices:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \; g = \begin{bmatrix} -2 \\ -4 \end{bmatrix}, \; A = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \; b = 0 \, , \; C = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} d = \begin{bmatrix} -3 \\ -3 \\ -4 \\ -4 \end{bmatrix}$$

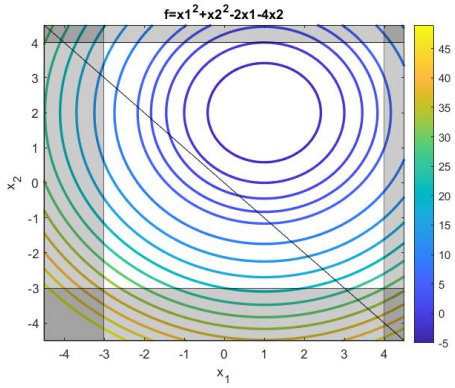


Figure 2: Contour Plot

The starting point that we provided to the algorithm was $(0, 0)^T$. After running the driver file -the script is provided at subsection 2.7- in order to test the function we had the following results:

(x_1, x_2)	$f - value$	λ_{eq}	time	iterations
$x = (-0.5, 0.5)^T$	-0.5	-3	0.0063	6

2.5 Primal Active-Set Algorithm

In this part, we will explain each step of the primal-active set algorithm in the form:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = \frac{1}{2}x'Hx + g'x \\ s.t \quad & A'x = b \\ & C'x \geq d \end{aligned}$$

After the implementation of the algorithm as a matlab function we will convert the example of the part 2.4 -which is in the form 2.1- into a QP in the above form using some pre-solving methods and we will solve it.
The general idea of the active set algorithm is to transform the original QP into an equality constrained QP for the search direction p. The algorithm starts from a feasible point and a corresponding working set W_0 which is a subset of the $A(x)$ active set. It tries to create a sequence $xk \in \Omega$ - where Ω is the feasible region- with decreasing values. The relationship between the value of the objective function before and after the step is:

$f(x_{k+1}) = f(x_k + p) = f(x_k) + (Hx_k + g)'p + \frac{1}{2}p'Hp = f(x_k) + \phi(p)$ where $\phi(p) = \frac{1}{2}p'Hp + (Hx_k + g)'p$

If $k_i(x)$ is one of the inequality constraints as a function of x then for the p direction we have: $k_i(x_k + p) = c_i'(x_k + p) - d_i = 0$ so, $c_i'x_k - d_i = c_i'p$ and the same for the equality constraints. So we end up to solve a sequence of the following Equality Convex QP in order to find the direction in each iteration.

$$\begin{aligned} \min_{p\in\mathbb{R}^n} \quad & \phi = \frac{1}{2}p'Hp + g_k'p \\ s.t \quad & \alpha_i'p = 0 \\ & c_i'p = 0 \quad , i \in W_k \end{aligned}$$

Where: $g_k = Hx_k + g$
In this way we have now transformed the original Inequality QP in a sequence of Equality QP's which can be solved by solving a sequence of KKT systems in order to obtain in each iteration the desired direction.

$$\begin{bmatrix} H & -A & -C_W \\ -A' & 0 & 0 \\ -C_W' & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ \lambda \\ \mu_W \end{bmatrix} = - \begin{bmatrix} g_k \\ 0 \\ 0 \end{bmatrix} \quad , g_k = Hx_k + g \ , C_W = [c_i]_{i\in W_k}$$

On the next page is presented the pseudocode of a Primal Active-Set Algorithm.

Algorithm 2 Primal Active-Set Algorithm

Let x_0 be a feasible point on the boundaries of some constraints and W_0 the corresponding active set. $W_0 = \{c_i x = d_i\}$

while not converged **do**

Solve the KKT system

$$\begin{bmatrix} H & -A & -C_W \\ -A' & 0 & 0 \\ -C_W' & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ \lambda \\ \mu_W \end{bmatrix} = - \begin{bmatrix} g_k \\ 0 \\ 0 \end{bmatrix} \quad , g_k = Hx_k + g \, , C_W = [c_i]_{i \in W_k}$$

if $||p|| = 0$ **then**

if $\mu_{Wi} \geq 0 \forall i \in W_k$ **then**

STOP Optimal found

set $\mu_i* = 0, \ i \notin W_k$ and $\mu_i* = \mu_i, \ i \in W_k$

else

Remove constraint i with the most negative Langarnge multiplier μ

$$i \in W_k. \ W_{k+1} := W_k \setminus \{i\}. \quad x_{k+1} := x_k$$

end if

else

Compute the distance to the nearest inactive constraint in the search direction.

$$\alpha = \min_{i \in \{i \notin W_k: \ c_i' p* < 0\}} \frac{d_i - c_i' x_k}{c_i' p^*}$$

$$j = arg \min_{i \in \{i \notin W_k: \ c_i' p* < 0\}} \frac{d_i - c_i' x_k}{c_i' p^*}$$

if $\alpha < 1$ **then**

$$x_{k+1} = x_k + \alpha p^*$$

Add constraint j to working set. $W_{k+1} = W_k \cup \{j\}$

else

$$x_{k+1} = x_k + p$$

$$W_{k+1} = W_k$$

end if

end if

end while

2.6 Implementation of Active-Set Algorithm

With the purpose to implement the active-set algorithm it was necessary to make a Matlab function which takes as inputs the matrices H, g, A, b and some intitial value x_0. In the appendix of this report you would find the code of the function of the primal active - set algorithm. In order to test the implementation of the problem according to the given form, we will solve the following simple QP.

$$\min_{x\in\mathbb{R}^2} \quad \phi = x_1^2 + x_2^2 - 2x_1 - 4x_2$$

$$s.t \quad x_1 + x_2 = 0$$
$$-3 \leq x_1 \leq 4$$
$$-3 \leq x_2 \leq 4$$

Firstly we have to implement some pre-solving steps to define the matrices that we will use in order to modify the bound constraints and finally have a QP with inequality constraints in the form: $C'x \geq b$

$$\begin{cases} x_1 \geq -3 \\ x_2 \geq -3 \\ x_1 \leq 4 \\ x_2 \leq 4 \end{cases} \Leftrightarrow \begin{cases} x_1 \geq -3 \\ x_2 \geq -3 \\ -x_1 \geq -4 \\ -x_2 \geq -4 \end{cases} \Rightarrow C = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} d = \begin{bmatrix} -3 \\ -3 \\ -4 \\ -4 \end{bmatrix}$$

The starting point that we provided to the algorithm was $(0,0)^T$. After running the driver file -the code is provided in appendix- in order to test the function we had the following results:

(x_1, x_2)	$f - value$	λ_{eq}	time	iterations
$x = (-0.5, 0.5)^T$	-0.5	-3	0.0012	1

2.7 Using Matlab's 'quadprog'

Except our implementation of Interior-Point Algorithm and Active-Set algorithm we can use Matlab's 'quadprog' library to solve Convex QPs. In this section we will solve the above problem using quadprog and we will use default options which means that the function will implement the 'interior-point-convex' algorithm. The results are presented at the following table.

(x_1, x_2)	$f - value$	λ_{eq}	time	iterations
$(-0.5, 0.5)^T$	-0.5	3	0.0119	3

We observe that the lagrange multiplier in quadprog case is the opposite from the other cases. The reason behind this, is that in some notations the linear combination of the constraints is not subtracted but added to the objective function. The script of all the above tests on matlab is the following:

```
%==========Driver File for 2.4 2.6 2.7==========%
%====================Input matrices====================%
A=[1;1]; b=0;
C=[1 0 -1 0;
   0 1 0 -1];
d=[-3 -3 -4 -4]';
H=[2,0;
    0,2];
g=[-2;-4]; x0=[0;0];
disp("==========Interior-Point Algorithm==========")
[x,fval,y,z,iter,t] = Interior_Point(H,g,A,b,C,d,x0)
G=H;
disp("==========Active-Set Algorithm==========")
[x,lambda,mu,fval,t,k] = Active_Set(G,g,A,b,C,d,x0)

Aeq=[1 1]; beq=0;
H=[2 0;0 2]; f=[-2;-4];
lb=[-3;-3]; ub=[4;4];
x0=[0;0];
t=tic;
disp("==========quadprog==========")
[x,fval,exitflag,output,lambda] = quadprog(H,f,[],[],
    Aeq,beq,lb,ub,x0)
time=toc(t)
```

2.8 Markowitz Portfolio Optimization Problem

Nowadays, several investors around the world use Markowitz Mean-Variance Portfolio Theory to choose optimally among the different weighting factors which they should invest in some securities in order to minimize risk while maximizing return. For this theory, Harry Markowitz was awarded the Nobel Price in 1990. One aspect of this theory is the Portfolio Optimization problem which has the following formulation:

$$minf(x) = x'Hx$$
$$s.t \quad \sum_{i=1}^{n} x_i = 1$$
$$\mu' x = R$$
$$0 \leq x \leq 1$$

Where x_i is the weighting factor (fraction) of our funds which we should invest in i security, H is the covariance matrix which has in the main diagonal the variances of each security, μ is the mean return and R is the return that we want to have. All the above are a consequence of the following expressions:

$$R = \sum_{i=1}^{n} r_i x_i = r'x$$
$$\bar{R} = E\{R\} = E\{r'x\} = \mu' x$$
$$V\{R\} = E\{(R - \bar{R})^2\} = E\{x'(r - \mu)(r - \mu)'x\} = x'Hx$$

Where $\bar{R}$ is the expected return. At this part of the report we will find the optimal portfolio in a financial market with the following 5 securities.

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

In order to test the algorithms that we implemented in the previous parts we will solve this problem using Interior-Point Algorithm, Active-Set Algorithm and Matlab's 'quadprog' library. We assumed that the desired return is $R = 10$. The commands that were used are presented in the appendix of this report. In the following table there are the solution along with the performance statistics.

-	Interior-Point	Active-Set	quadprog
x	(0, 0.28, 0, 0.72, 0)	(0, 0.28, 0, 0.72, 0)	(0, 0.28, 0, 0.72, 0)
λ_{eq}	(-0.5183, 7.2753)	(-0.5183, 7.2753)	(0.5183, -7.2753)
λ_{ineq}	(1.34, 0, 0.96, 0, 1.55)	(1.34, 0, 0.96, 0, 1.55)	-
fvalue	2.0922	2.0922	2.0922
iter	6	4	5
time	0.0052	0.0013	0.0081

Due to the fact that the algorithms solve problems which have the objective function in the form: $\frac{1}{2}x'Hx+g'x$ we had to multiply the output of the f-value of the algorithms (1.0461) by 2. In the next part we will discuss extensively on the results of the problem. The reasons behind λ_{ineq} for quadprog is not presented, is space reasons because we implemented the algorithm setting as inputs the 'lb' and 'ub' matrices.

3 Markowitz Portfolio Optimization

3.1 Formulation of the problem

In the previous part, we tested the 2 algorithms and 'quadprog' on Markowitz Portfolio Optimization problem so we demonstrated that the problem can be written as a QP in the form:

$$minf(x) = x'Hx$$

$$s.t \quad \sum_{i=1}^{n} x_i = 1$$

$$\mu' x = R$$

$$x \geq 0$$

We did not include completely the previous inequality constraint ($0 \leq x \leq 1$) because the above constraints are sufficient to formulate the problem.

3.2 Minimum-Maximum Return

If we have a look at the given table it is quite easy to understand that possibly we would have the minimum return if we invest the whole capital in the 4^{th} security with the value of return to be 9.02. On the other hand, we would succeed the maximum return 17.68 if we invest all the capital in the 5^{th} security.

3.3 Optimal Portfolio

We assume that we want to have $R = 10$. So the QP now is:

$$minf(x) = x'Hx$$

$$s.t \quad \sum_{i=1}^{n} x_i = 1$$

$$\mu' x = 10$$

$$x \geq 0$$

Where:

$$H = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 \end{bmatrix} \quad \mu = \begin{bmatrix} 15.1 \\ 12.5 \\ 14.7 \\ 9.02 \\ 17.68 \end{bmatrix}$$

The desired return is the same as in the 2.8 paragraph so the results are:

$$x^* = (0, 0.28, 0, 0.72, 0) \quad f(x^*) = 1.0461$$

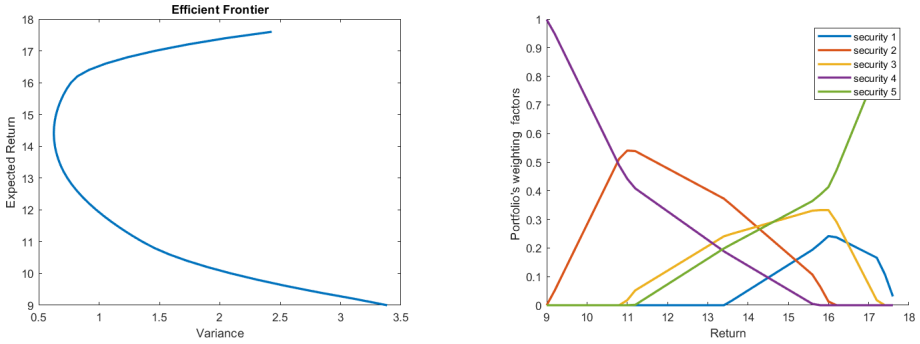
Which means that we should invest 28% of our capital in the 2nd security, 72% in the 4th and nothing to the other securities. The risk of this invest is: $2V\{R\} = 2E\{(R - \bar{R})^2\} = 2E\{x^{*\prime}(r - \mu)(r - \mu)'x^*\} = 2x^{*\prime}Hx^* = 2f(x^*) = 2.0922$

3.4 The Efficient Frontier

The efficient frontier is a set of different portfolios that provides us a clear view about the highest expected return for a given level of risk (variance). Below we can see the efficient frontier's curve as well as the allocation of the optimal portfolio's as a function of return.

3.4.1 Adding a risk free security

Let us consider that we add to the above financial market a risk free security with return $r_f = 2.0$. As we can easily understand the risk free security is not correlated with the other securities so each correlation of the last row an



(a) The efficient frontier (b) Portfolio's allocation

Figure 3

column will be zero. The new covariance matrix as well as the return vector are the following:

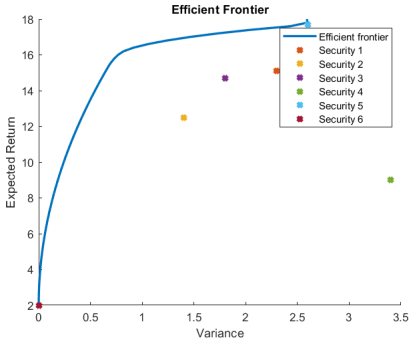
$$H = \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 & 0 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 & 0 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 & 0 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 & 0 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mu = \begin{bmatrix} 15.1 \\ 12.5 \\ 14.7 \\ 9.02 \\ 17.68 \\ 2 \end{bmatrix}$$

3.4.2 Efficient Frontier with risk free security

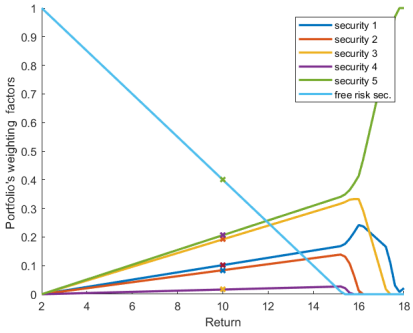
After adding the risk free security to the financial market the optimal portfolio for the return $R = 10$ is the following:

Security 1	Security 2	Security 3	Security 4	Security 5	Security 6
0.1018	0.0840	0.1917	0.0164	0.2063	0.3998

The value of the objective function after the above allocation was increased to: $2f(x*) = 0.2417$. Below, in the plots, you can see the impact of the risk free security on the efficient frontier-with the securities- as well as the optimal allocation of the securities for the different values of R.
As we can see the risk free security modified completely the efficient frontier's curve and the allocation of the capital in the different securities for a given R. Specifically, in the case that $R = 10$ we can see that the risk free security force us to invest different percentages in all the securities and not only in the 2nd and the 4th as in the previous case. In addition, we can see that the value of the objective function which represents the risk was reasonably increased.



(a) The efficient frontier



(b) Portfolio's allocation

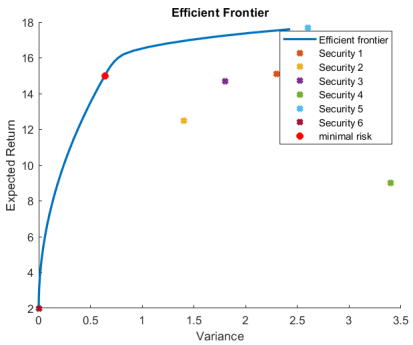
Figure 4

3.4.3 Changing the Return

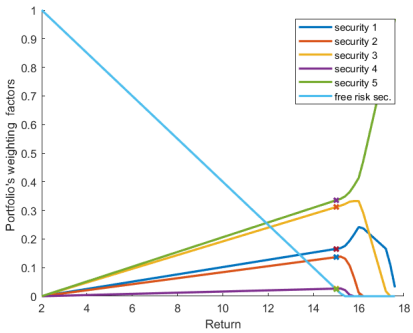
If we set the return $R = 15$ and follow the previous steps again using 'quadprog' in order to find the optimal portfolio and the minimal risk we have the risk at $2f(x^*) = 0.6383$ and the results for the optimal weighting factors are:

Security 1	Security 2	Security 3	Security 4	Security 5	Security 6
0.1655	0.1365	0.3115	0.0266	0.3352	0.0247

At the following plots we can see the minimal risk on efficient frontier's curve as well as the optimal portfolio as a function of return.



(a) The efficient frontier



(b) Portfolio's allocation

Figure 5

4 Linear Program (LP)

Consider the Linear Program in the following form:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = g'x \\ s.t \quad & A'x = b \\ & l \leq x \leq u \end{aligned}$$

We assume that $A \in \mathbb{R}^{n\times m}$ has a full column rank and $x \in \mathbb{R}^n$ is a vector with n variables.
In order to have a simple notation we will convert the above problem into its standard form. For this reason we introduce $x = x_u - x_l$ where $x_u \geq 0,\ x_l \geq 0$ and $t_u \geq 0, t_l \geq 0$ the slack variables to convert the box constraints to equality constraints. So the LP is:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = g'x_u - g'x_l \\ s.t \quad & A'x_u - A'x_l = b \\ & x_u - x_l - t_l = l \\ & x_u - x_l + t_u = u \\ & x_l \geq 0, x_u \geq 0, t_l \geq 0, t_u \geq 0 \end{aligned}$$

So in a matrix notation our problem now is in the form:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad \phi \quad &= \begin{bmatrix} -g' \\ g' \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_l \\ x_u \\ t_l \\ t_u \end{bmatrix} \\ s.t \quad & \begin{bmatrix} -A' & A' & 0 & 0 \\ -I & I & -I & 0 \\ -I & I & 0 & I \end{bmatrix} \begin{bmatrix} x_l \\ x_u \\ t_l \\ t_u \end{bmatrix} = \begin{bmatrix} b \\ l \\ u \end{bmatrix} \end{aligned}$$

For simplicity reasons now we have:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = \hat{g}'\hat{x} \\ s.t \quad & \hat{A}\hat{x} = \hat{b} \\ & \hat{x} \geq 0 \end{aligned}$$

Where: $\hat{g} = \begin{bmatrix} -g' \\ g' \\ 0 \\ 0 \end{bmatrix}, \hat{x} = \begin{bmatrix} x_l \\ x_u \\ t_l \\ t_u \end{bmatrix}, \hat{A} = \begin{bmatrix} -A' & A' & 0 & 0 \\ -I & I & -I & 0 \\ -I & I & 0 & I \end{bmatrix}, \hat{b} = \begin{bmatrix} b \\ l \\ u \end{bmatrix}$

4.1 The Lagrangian function

The Lagrangian function of the above LP is:

$$\mathscr{L}(x, \lambda, z) = \hat{g}'\hat{x} - \mu'(\hat{A}'\hat{x} - \hat{b}) - \lambda'\hat{x}$$

where $\mu \in \mathbb{R}^m$ are the lagrange multipliers of m equality constraints and $\lambda \in \mathbb{R}^n$.

4.2 The Optimality Conditions

The necessary and sufficient optimality conditions of the above LP are:

$$\nabla_x \mathscr{L}(x, \mu, \lambda) = \hat{g} - \hat{A}'\mu - \lambda = 0$$

$$\hat{A}\hat{x} = \hat{b}$$

$$\hat{x} \geq 0 \perp \lambda \geq 0$$

$$\hat{x_i}\lambda_i = 0 \quad i = 1, 2, ..., n$$

4.3 Predictor-Corrector Interior-Point Algorithm for LP

From now on, we will describe some algorithms starting from Interior-Point Algorithm for LP's in the form:

$$\min_{x \in \mathbb{R}^n} \quad \phi = g'x$$

$$s.t \quad Ax = b$$

$$x \geq 0$$

After the description of the algorithms we will solve a simple LP to test our implementations on the algorithms for LP's in the given form (4.1) while we will initially implemented some pre-solving methods. The optimality conditions for the LP in the above form are:

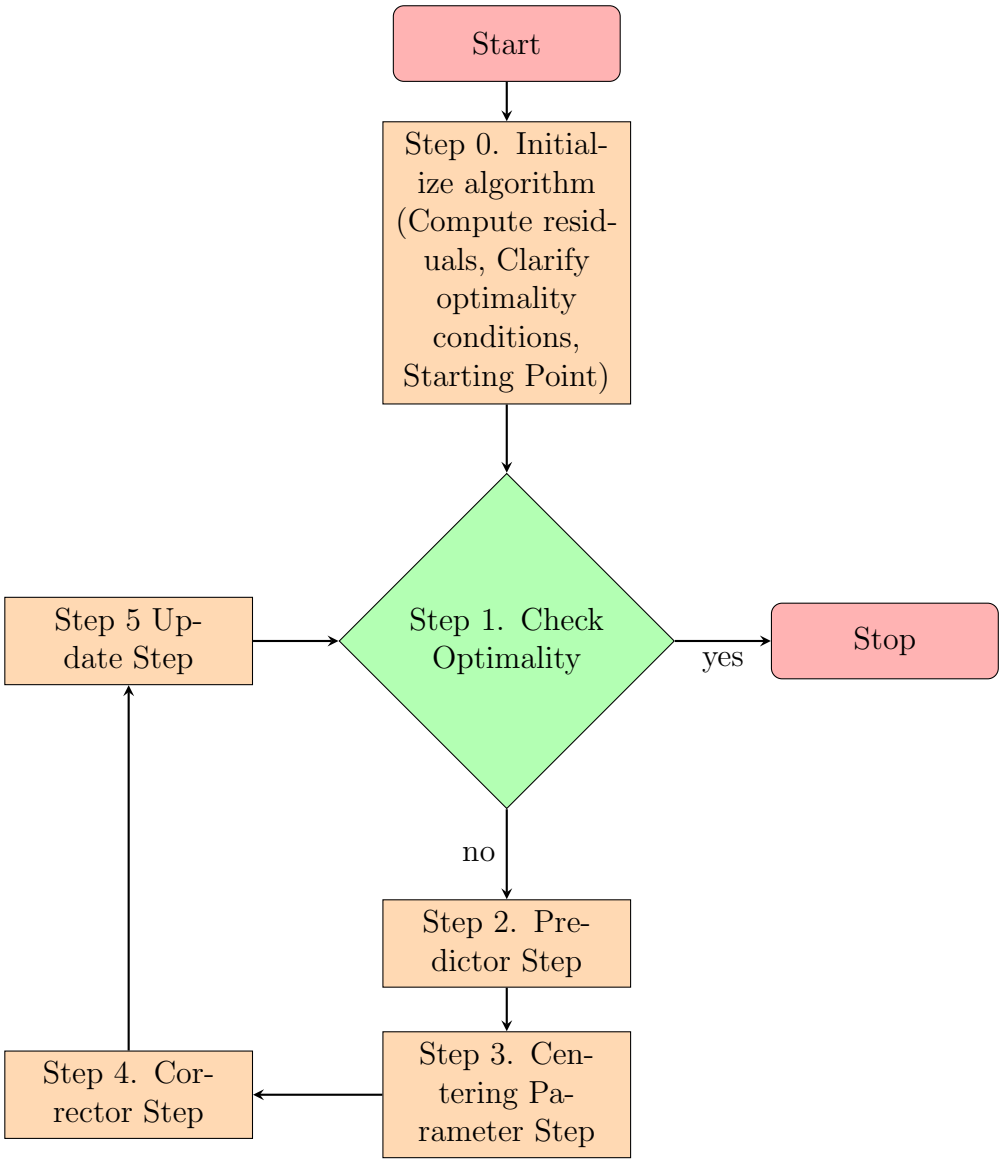
$$\nabla_x \mathscr{L}(x, \mu, \lambda) = g - A'\mu - \lambda = 0$$

$$Ax = b$$

$$x \geq 0, \perp \lambda \geq 0$$

$$x_i\lambda_i = 0 \quad i = 1, 2, ..., n$$

The algorithm loop begins from a starting point(x, μ, λ) such that $(x, \lambda) > 0$ checking the optimality conditions $(||r_L||, ||r_A||, |s|) \leq tol$(*see Algorithm 3*). At the next step computes the predictor step $(\Delta x^{aff}, \Delta\mu^{aff}, \Delta\lambda^{aff})$ solving a system - which is presented in the below pseudo-code-. Computes the largest possible step $\alpha^{aff}, \beta^{aff}$ such that $x+\alpha^{aff}\Delta x^{aff} \geq 0,\ \lambda+\beta^{aff}\Delta\lambda^{aff} \geq 0$. After, computes the affine duality gap $s^{aff} = (x+\alpha^{aff}\Delta x^{aff})'(\lambda+\beta^{aff}\Delta\lambda^{aff})/n$ and the centering parameter $\sigma = (s^{aff}/s)^3$ with $s = x'\lambda/n$. At the next step computes the corrector step $(\Delta x, \Delta\mu, \Delta\lambda)$ solving a system which you can find in the following pseudo-code. Finally calculates the largest primal and dual step such that $x + \alpha\Delta x \geq 0,\ \lambda + \beta\Delta\lambda^{aff} \geq 0$ and updates the point. At the following flowchart we have the opportunity to make a general view of the steps.


Start
Step 0. Initialize algorithm (Compute residuals, Clarify optimality conditions, Starting Point)
Step 1. Check Optimality
yes
Stop
no
Step 2. Predictor Step
Step 3. Centering Parameter Step
Step 4. Corrector Step
Step 5 Update Step

Algorithm 3 Primal-Dual Predictor-Corrector Interior-Point Algorithm for LP

Require: (x, μ, λ) such that $(x, \lambda) > 0$
Compute the residuals:
$r_L = g - A'\mu^k - \lambda^k,\ r_A = Ax^k - b,\ s = x'\lambda/n$
State convergence conditions:

$$||r_L|| \leq \epsilon_L,\ ||r_A|| \leq \epsilon_A,\ |s| \leq \epsilon_\mu$$

while not converged **do**
Solve to find affine direction $\begin{bmatrix} 0 & -A' & -I \\ A & 0 & 0 \\ \Lambda & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta\mu^{aff} \\ \Delta\lambda^{aff} \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -X\Lambda e \end{bmatrix}$
Compute the largest α^{aff} and β^{aff} such that: $z + \alpha^{aff}\Delta x^{aff} \geq 0$ and $\lambda + \beta^{aff}\Delta\lambda^{aff} \geq 0$

Find the duality gap and the centering parameter
$s^{aff} = (x + \alpha^{aff}\Delta x^{aff})'(\lambda + \beta^{aff}\Delta\lambda^{aff})/n$, $\sigma = (s^{aff}/s)^3$

Centering-Correction Direction

Solve $\begin{bmatrix} 0 & -A' & -I \\ A & 0 & 0 \\ \Lambda & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\mu \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -X\Lambda e - \Delta X^{aff}\Delta\Lambda^{aff}e + \sigma se \end{bmatrix}$

Compute the largest α and β such that: $x + \alpha\Delta x \geq 0$ and $\lambda + \beta\Delta\lambda \geq 0$
Update: $x^{k+1} = x^k + 0.995\alpha\Delta x$, $\mu^{k+1} = \mu^k + 0.995\beta\Delta\mu$, $\lambda^{k+1} = \lambda^k + \beta\Delta\lambda$
Compute the residuals:
$r_L = g - A'\mu^{k+1} - \lambda^{k+1},\ r_A = Ax^{k+1} - b,\ s = (x^{k+1})'\lambda/n$
end while

4.4 Implementation of Interior-Point Algorithm

In this section we will use the Primal-Dual Interior-Point Algorithm which was provided as teaching material for the course "Constrained Optimization". We will use the matlab function to solve the following LP:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = 2x_1 - 5x_2 \\ s.t \quad & x_1 + x_2 = 210 \\ & 100 \leq x_1 \leq 200 \end{aligned}$$

$$50 \leq x_2 \leq 150$$

We apply the following pre-solving steps -slack variables- to have the LP in its standard form:

$$\begin{cases} x_1 \geq 100 \\ x_2 \geq 50 \\ x_1 \leq 200 \\ x_2 \leq 150 \end{cases} \Leftrightarrow \begin{cases} x_1 - x_3 = 100 \\ x_2 - x_4 = 50 \\ x_1 + x_5 = 200 \\ x_2 + x_6 = 150 \end{cases}$$

So the input matrices will be:

$$g = \begin{bmatrix} 2 \\ -5 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 210 \\ 100 \\ 50 \\ 200 \\ 150 \end{bmatrix}$$

At the following table we can see the results of the solution of the above LP.

x	λ	μ	time	iterations
$(100, 110)$	$(0, 0, 7, 0, 0, 0)$	$(-5, 7, 0, 0, 0)$	0.0054	24

We did not include in the table the values of the slack variables which are:

$$x_3 = 0, x_4 = 60, x_5 = 100, x_6 = 40$$

4.5 Primal Active-Set Algorithm (Simplex)

The primal active-set (Simplex) algorithm is one of the top ten algorithms with the greatest influence in the 20^{th} century. Below we will describe the steps of the algorithm as well as we will provide a pseudo-code of the algorithm.

Step 1
Choose an initial basic solution from a basic set. If the solution is feasible continue to the next step.

Step 2
Compute μ and λ.

Step 3.1(Test Optimality)
If all λ from the non-basic set are positive or zero then STOP "optimal found".

Step 3.2
Else select the "s" minimum and negative λ to enter in the non-basic set.

Step 3.3
Compute the pivot column $h = (A_{basic})^{-1} A_s$
Step 3.3
If all "h" are non-positive then STOP the LP is unbounded.
Step 3.4
Else select the leaving variable using the rule: $j = \{arg\min_{h_i>0} \frac{x_{basic}}{h}\}$
Step 3.5
Update the Basic and the non-Basic sets.

Algorithm 4 Primal Active-Set (Revised Simplex) Algorithm for LP

Require: an initial Basic and non Basic set(B,N) such that:
$x_B = B^{-1}b \geq 0$.
Clarify that: $B = [a_i]_{i\in\mathbb{B}}, N = [a_i]_{i\in N}, x_B = (x_i)_{i\in\mathbb{B}}, x_N = (x_i)_{i\in N}, A = [a_1, a_2, ..a_n]$
while not STOP **do**
  Solve for μ: $B'\mu = g_B$ where $g_B = (g_i)_{i\in B}$
  Compute $\lambda_N = g_N - N'\mu$
  if all $(\lambda_N)_i \geq 0$ **then**
    STOP (Optimal- found)
  else
    Select $s:$ $(\lambda_N)_s < 0$ to enter the non-basic set
    Solve for h: $Bh = a_{is}$
    Set

$$J = \{arg\min_{h_i>0} \frac{(x_B)_i}{h_i}\}$$

    if J is empty **then**
      STOP the problem is unbounded
    else
      Select $j \in J$ and $\alpha = \frac{(x_B)_j}{h_j}$
      $x_B := x_B - \alpha h$
      Enter $j \in J$ in the basic set.
      Update Basic and non-Basic sets
    end if
  end if
end while
$\lambda_B \longleftarrow 0$

4.6 Implementation of Active-Set Algorithm

In order to implement the active-set algorithm for LP's we created a matlab function. We tested the function on the example of the paragraph 4.4 but before the implementation we applied the same pre-solving steps.

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \phi = 2x_1 - 5x_2 \\ s.t \quad & x_1 + x_2 = 210 \\ & 100 \leq x_1 \leq 200 \\ & 50 \leq x_2 \leq 150 \end{aligned}$$

The results of the implementation are included in the following table:

x	λ	μ	time	iterations
$(100, 110)$	$(0, 0, 7, 0, 0, 0)$	$(-5, 7, 0, 0, 0)$	0.0045	2

We did not include in the table the values of the slack variables which are:

$$x_3 = 0, x_4 = 60, x_5 = 100, x_6 = 40$$

4.7 Using Matlab's 'linprog'

Except our implementation of Interior-Point Algorithm and Active-Set algorithm we can use Matlab's 'linprog' library to solve LPs. In this section we will solve the above problem using linprog and we will use default options which means that the function will implement the 'Dual-Simplex' algorithm. The summary results are presented at the following table.

Alg-results	x	λ	μ	time	iterations
Int-Point	$(100, 110)$	$(0, 0, 7, 0, 0, 0)$	$(-5, 7, 0, 0, 0)$	0.0054	24
Simplex	$(100, 110)$	$(0, 0, 7, 0, 0, 0)$	$(-5, 7, 0, 0, 0)$	0.0045	2
linprog	$(100, 110)$	$*lower \quad (7, 0)$	$*eq \quad 5$	0.0980	1

From the above table we can realize the big disadvantage of the Interior-Point algorithm compared to Simplex. The reason behind the difference on the number of iterations between Interior-Point Algorithm and Simplex Algorithm is that if we begin from an initial point -here $x = (1, 1, 1, 1, 1, 1)$- quite far from the optimal the Interior Point Algorithm needs many iterations to reach the optimal. Below, there is the matlab script with the appropriate commands to solve the above problem using the 3 functions. In the appendix you can find the code of the 2 algorithms.

```
%=======Driver File 4.4, 4.6, 47================
%         min f(x)=2*x1-5*x2
%          s.t     x1+x2=210
%                  100<=x1<=200
%                  50<=x2<=150
%================================================
A=[1 1 0 1 0;
     1 0 1 0 1
     0 -1 0 0 0;
     0 0 -1 0 0;
     0 0 0 1 0;
     0 0 0 0 1]';
b=[210 100 50 200 150]'; g=[2;-5;0;0;0;0];
disp("=========Simplex Algorithm==========")
t1=tic;
[x,mu,lambda,optimal,iter]=Simplex(g,A,b)
time1=toc(t1)
disp("=========Interior-Point Algorithm==========")
x=ones(6,1);
t2=tic;
[x,info,mu,lambda,iter] = LPippd(g,A,b,x)
time2=toc(t2)
disp("==============linprog=============")
f=g(1:2); Aeq=[1 1]; beq=210; lb=[100;50]; ub
   =[200;150];
t3=tic;
[x,fval,exitflag,output,lambda] = linprog(f,[],[],Aeq,
   beq,lb,ub)
time3=toc(t3)
```

4.8 Markowitz Porfolio with Risk Parameter

As we discussed in the paragraph 3, Markowitz Portfolio optimization problem can be expressed as a QP. Below we present another formulation of the problem which allows to maximize our returns choosing a risk-tolerance parameter.

$$\max f(x) = x'\mu - \kappa x' H x$$

$$s.t \quad \sum_{i=1}^{n} x_i = 1$$

$$0 \leq x \leq 1 \qquad where \quad \kappa \in [0, \infty)$$

If set the risk parameter $\kappa = 0$, which means that we do not care about the risk the QP is converted to the following LP:

$$\max f(x) = x'\mu$$

$$s.t \quad \sum_{i=1}^{n} x_i = 1$$

$$0 \leq x \leq 1$$

In order to use our matlab functions we have to express the problem as a minimizing problem. In addition, we can skip the inequality $x \leq 1$ as the expressions $\{x \geq 0, \sum_{i=1}^{n} x_i = 1\}$ are sufficient to formulate the problem. After this step we have to solve the below LP:

$$\min\{-f(x)\} = -x'\mu$$

$$s.t \quad \sum_{i=1}^{5} x_i = 1$$

$$x \geq 0$$

So, we have the LP in its standard form. The input matrices are provided in the below script:

```
%==============Driver File for 48=====================%
%==================Insert Data=======================%
clc
clear
g=-1*[15.10;12.50;14.70;9.02;17.68];
A=[1 1 1 1 1];
b=[1]; x=ones(5,1);
disp("==========Interior-Point Algorithm==========")
[x,info,mu,lambda,iter] = LPippd(g,A,b,x)
disp("==========Simplex Algorithm==========")
[x,mu,lambda,optimal,iter]=Simplex(g,A,b)

```

```
Aeq=[1  1  1  1  1];
beq=1;
lb=zeros(5,1);
ub=[];
f=g;
disp("==========='linpgrog'==========")
[x,fval,exitflag,output,lambda] = linprog(f,[],[],Aeq,
    beq,lb,ub)
```

The result was quite expected from the beginning as the algorithm advises to invest all the funds in the security with the maximum return $x^* = (0, 0, 0, 0, 1)$.

5 Nonlinear Program(NLP)

Let us consider the nonliner program in the form:

$$\min_{x \in \mathbb{R}^n} f(x)$$

$$s.t \quad g(x) = 0$$

$$l \leq x \leq u$$

Where $f(x)$, $g(x)$ are sufficiently smooth and $\nabla g(x)$ has a full column rank.

5.1 The Lagrangian

For simplicity reasons we proceed in the following way:

$$\begin{cases} x \geq l \\ x \leq u \end{cases} \Leftrightarrow \begin{cases} c_1(x) = x - l \geq 0 \\ c_2(x) = -x + u \geq 0 \end{cases}$$

So we have now the NLP in the form:

$$\min_{x \in \mathbb{R}^n} f(x)$$

$$\begin{aligned} s.t \qquad & g_i(x) = 0 \qquad i \in \mathscr{E} \\ & c_i(x) \geq 0 \qquad , i \in \mathscr{I} \end{aligned}$$

According to the above form the lagrangian of the NLP is the following function:

$$\mathscr{L}(x, y, z) = f(x) - \sum_{i \in \mathscr{E}} y_i g_i(x) - \sum_{i \in \mathscr{I}} z_i c_i(x) = f(x) - y^T g(x) - z^T c(x)$$

5.2 First Order Optimality Conditions

The gradient of the Lagrangian in respect to x is:

$$\nabla \mathscr{L}(x,y,z) = \nabla f(x) - \sum_{i\in\mathscr{E}} y_i \nabla g_i(x) - \sum_{i\in\mathscr{I}} z_i \nabla c_i(x) = \nabla f(x) - \nabla g(x) y - \nabla c(x) z$$

So as 1st order KKT conditions we have the following sentence:
If x^* is a local optimum then:

$$\nabla_x \mathscr{L}(x^*, y^*, z^*) = 0 \qquad (stationarity)$$

$$g_i(x^*) = 0, \quad i \in \mathscr{E} \quad (feasibility)$$

$$c_i(x^*) \geq 0, \quad i \in \mathscr{I}$$

$$z_i^* \geq 0 \quad i \in \mathscr{I} \quad (complementarity)$$

$$z_i^* c_i(x^*) = 0, \quad i \in \mathscr{I}$$

5.3 Second Order Sufficient Conditions

We assume that (x^*, y^*, z^*) satisfy the 1st order KKT conditions. If for all feasible active directons, $h \neq 0$

$$h^T \nabla^2_{xx} \mathscr{L}(x^*, y^*, z^*) h > 0$$

Then x^* is a strict local minimizer.

5.4 Himmelblau's Function

In the following sections we will deal with Himmelblau's function which is a continuous and non-convex function. In the region $x_i \in [-6, 6]$ for $i = 1, 2$ it has 4 local minimizers. This function is widely used in Optimization in order to test the performance of the algorithms. Specifically, the NLP is the following:

$$\min_{x,y} f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

$$s.t \quad h(x,y) = (x+2)^2 - y$$

$$-5 \leq x \leq -1$$

$$0 \leq y \leq 5$$

The contour plot of the function in the region that we will work is:

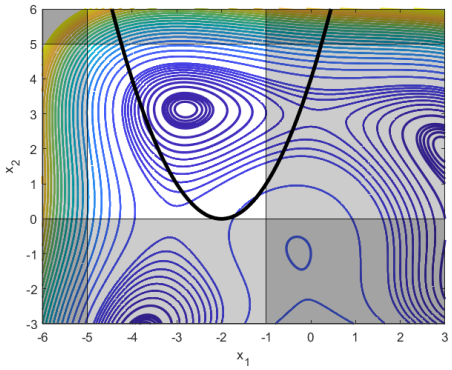


Figure 6: Contour Plot

5.5 Solving with Matlab's 'fmincon'

In order to solve this problem with matlab's library 'fmincon' it was necessary to implement the objective function as well as the function of the constraint and then to call 'fmincon'. We set default options of the algorithm which means that the used algorithm is 'Interior-Point' algorithm. The script of the aforementioned procedure is the following:

```
%==========Driver file for 55==========%
%==========="fmincon"==========%
%==========Initial data==========%
clc; clear;
LB=[-5;0]; UB=[-1;5];
x0=[-5;-5];
A=[]; B=[]; Aeq=[]; Beq=[];
%==========Call function==========%
options = optimoptions('fmincon','Display','iter',...
    'SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true,'FunctionTolerance
      ',10e-7);

[x,fval,exitflag,output]= fmincon(@(x) fun2(x),x0,A,B,
   Aeq,Beq,LB,UB,...
    @(x) con12(x),options)
%==========Specify obj function & gradient=======%
function [f,df] = fun2(x)
x1=x(1,1);
x2=x(2,1);
```

```
19
20 tmp1=x1^2+x2-11;
21 tmp2=x1+x2^2-7;
22
23 f=tmp1^2+tmp2^2;       %Gradient%
24 if nargout > 1
25     df = zeros(2,1);
26     df(1,1) = 4*tmp1*x(1) + 2*tmp2;
27     df(2,1) = 2*tmp1 + 4*tmp2*x(2);
28 end
29 end
30 %============Specify constraint function=========%
31 function [c,ceq,dc,dceq]=con12(x)
32 x1=x(1,1);
33 x2=x(2,1);
34 tmp=x1+2;
35 ceq=tmp^2-x2;
36 c=[];
37 if nargout > 2
38     dc =[];
39     dceq = zeros(2,1);
40     dceq(1,1) = 2*tmp;
41     dceq(2,1) = -1.0;
42 end
43 end
```

We set the 'FunctionTolerance' at '$10e-7$' and after 8 iterations of the algorithm from the starting point $(x, y) = (-5, -5)$ which is not in the feasible region, we finally got the minimizer $x^* = (-3.6546, 2.7377)$

5.6 SQP Algorithm

Sequential Quadratic Programming is one of the most significant approaches for solving nonlinear constrained optimization problems. This method generates steps by giving solutions in quadratic sub-problems and is quite effective for small or large problems with nonlinear constraints. In this section we will explain local SQP method for the above problem. First of all we set the box constraints as an inequality constraint function because we will need the

values of this function during the iterations of the algorithm:

$$\begin{cases} x \geq -5 \\ y \geq 0 \\ x \leq -1 \\ y \leq 5 \end{cases} \Leftrightarrow \begin{cases} x + 5 \geq 0 \\ y \geq 0 \\ -x - 1 \geq 0 \\ -y + 5 \geq 0 \end{cases} \Rightarrow g(x) = \begin{bmatrix} g_1(x,y) \\ g_2(x,y) \\ g_3(x,y) \\ g_4(x,y) \end{bmatrix} = \begin{bmatrix} x+5 \\ y \\ -x-1 \\ -y+5 \end{bmatrix} \geq 0$$

The algorithm requires some initial values (x_0, y_0, z_0) and a first computation of the Hessian matrix of the Lagrangian function B_0. After this we enter the loop by solving the following QP in order to obtain the next step Δx.

$$\min_{\Delta x \in \mathbb{R}^n} \quad \frac{1}{2}\Delta x' B^k \Delta x + \nabla f(x^k)' \Delta x$$

$$s.t \quad \nabla h(x^k)' \Delta x = -h(x^k)$$
$$l - x^k \leq \Delta x \leq u - x^k$$

At the next step it computes the gradient of the lagrangian at the starting point combined with the new lagrange multipliers and the new point after adding the step Δx

$$\nabla L(x^k, y^{k+1}, z^{k+1}) = \nabla f(x^k) - \nabla h(x^k) y^{k+1} - \nabla g(x^k) z^{k+1}$$

$$x^{k+1} = x^k + \Delta x^k$$

After this step we have to compute the gradient of the Lagrangian at the new point and the difference of the two Lagrangians in order to be ready to update the Hessian matrix by Damped BFGS procedure.

$$\nabla L(x^{k+1}, y^{k+1}, z^{k+1}) = \nabla f(x^{k+1}) - \nabla h(x^{k+1}) y^{k+1} - \nabla g(x^{k+1}) z^{k+1}$$

$$q^k = \nabla L(x^{k+1}, y^{k+1}, z^{k+1}) - \nabla L(x^k, y^{k+1}, z^{k+1})$$

Damped BFGS
Given a symmetric and and positive definite matrix B_k we initially compute:

$$r^k = \theta^k q^k + (1 - \theta_k) B^k \Delta x^k$$

Where:

$$\theta^k = \begin{cases} 1, & \text{if } (\Delta x^k)' q^k \geq 0.2 (\Delta x^k)' B^k \Delta x^k \\ \frac{0.8 (\Delta x^k)' B^k \Delta x^k}{(\Delta x^k)' B^k \Delta x^k - (\Delta x^k)' q^k}, & \text{if } (\Delta x^k)' q^k < 0.2 (\Delta x^k)' B^k \Delta x^k \end{cases}$$

Finally we compute the new Hessian approximation using the following formula:

$$B^{k+1} = B^k + \frac{r^k(r^k)'}{(\Delta x^k)'r^k} - \frac{(B^k\Delta x^k)(B^k\Delta x^k)'}{(\Delta x^k)'(B^k\Delta x^k)}$$

The convergence criteria are satisfied when the gradient of the Lagrangian as well as the value of the equality constraint are close to zero.
In order to implement the SQP algorithm for the given problem we created the following script:

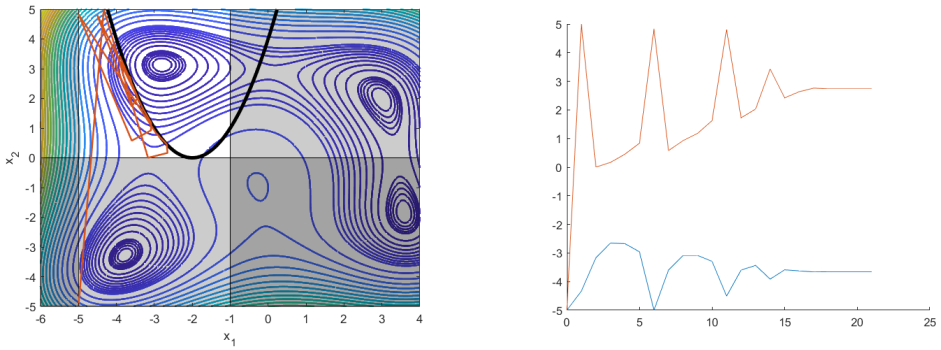
```
%==========SQP Algorithm with Damped BFGS==========%

%================Initialize data=================%
clc; clear;
x=[-5;-5]; y=1;
%Lagange multipliers for lower-upper bound constraints
z1=ones(2,1); z2=ones(2,1);
k=0;%Iterations
kmax=80;
lb=[-5;0]; ub=[-1;5];
%================call functions=================%
[f,df,d2f]=fun1(x);
[ceq,dceq,d2ceq]=con1(x);
[c,dc]=con2(x);

H=eye(2); %Initial Hessian
tol=10e-7;
converged=false; t1=tic;
while ~converged && k<kmax
    k=k+1;
 %==========Set the lower and upper bounds==========%
    lb1=lb-x(:,end);
    ub1=ub-x(:,end);
 %================Solve QP=======================%
    [p,~,~,~,lambda] = quadprog(H,df,[],[],dceq',-ceq,
        lb1,ub1);
 %================Obtain multipliers=============%
    y=-lambda.eqlin;
    z1=lambda.lower;
    z2=lambda.upper;
 %================Gradient of Lagrangian=========%
```

```
31      dL=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc(:,3)*z2
            (1)-dc(:,4)*z2(2);
32      x(:,k+1)=x(:,k)+p; %take step
33  %==========Function values at new point=========%
34      [f,df,d2f]=fun1(x(:,end));
35      [ceq,dceq,d2ceq]=con1(x(:,end));
36      [c,dc]=con2(x(:,end));
37
38      dL1=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc(:,3)*
            z2(1)-dc(:,4)*z2(2);
39      q=dL1-dL;
40  %=========Call Damped BFGS to update Hessian======%
41      H= Damped_BFGS(H,q,p);
42  %============Set convergence criteria============%
43      converged= ( norm(dL1,inf)<tol) && (norm(ceq,inf)<
            tol);
44
45  end
46  time=toc(t1)
47  x(:,end)
48  k
49  %%
```

The results that we had are presented in the following table:

Starting Point	solution	time	iterations
$(-5,-5)$	$(-3.6546, 2.7377)$	0.1041	21

At the following figures you can see the movements of the point as well as the convergence of the point:



(a) Iteration sequence (b) Convergence

Figure 7

We tested the algorithm from the different staring points $x1(-5,-5)$ and $x2 = (-5,2)$ and the sequence of the first as well as the final iterations are presented below:

iter	0	1	2	3	....	18	19	20	21
x1	-5	-4.3333	-3.1667	-2.6537	....	-3.6541	-3.6545	-3.6546	-3.6546
x2	-5	4.9998	6.59e-10	0.1643	....	2.7363	2.7376	2.7377	2.7377
iter	0	1	2	3	.....	15	16	17	18
x1	-5	-3.5	-2.75	-2.8784	....	-3.6558	-3.6545	-3.6546	-3.6546
x2	2	5.77e-11	1.26e-12	0.7552	....	2.7415	2.7376	2.7377	2.7377

At the appendix of the report you can find the implementation of Damped BFGS algorithm on Matlab.

5.7 Line Search Approach

Line search method is used to increase the rate of convergence from remote points. In this part we will implement a line search function for the given problem which will modify the search direction p if Armijo's condition is satisfied.
After the solution of the quadratic sub-problem we obtain the search direction p. This approach modify the direction with a scalar a. Firstly, we set $a = 1$. We set:

$$\phi(a) = f(x^k + ap^k) + y'|h(x^k + ap^k)| + z'|\min\{0, g(x^k + ap^k)\}$$

$$c = \phi(0) = f(x^k) + y'|h(x^k)| + z'|\min\{0, g(x^k)\}|$$

$$b = \phi'(0) = \frac{d\phi}{da}(0) = \nabla f(x^k)'p^k - y'|h(x^k)| - z'|\min\{0, g(x^k)\}|$$

If Armijo condition - $\phi(a) \leq \phi(0) + c_1 a\frac{d\phi}{da}(0)$, $c_1 \in (0,1)$ - is satisfied we accept the spacific direction. If not, then we compute the next iteration's scalar a using the following formulas:

$$\alpha_1 = \frac{\phi(a) - (c + ba)}{a^2} \text{ and } a_{min} = \frac{-b}{2a_1}$$

And then we accept as a the above:

$$a = \min\{0.9a, \max\{a_{min}, 0.1a\}\}$$

Finally, we compute the next point and the values of the functions in this point and the algorithm proceeds as described above until Armijo condition is satisfied. The following function is the implementation of the Line search method on Matlab.

```
%==============Line search method==============%
function [xk,initer]=linesearch(x,y,z,p)
%==============Initialize data==================%
alpha=1; initer=0;
flag=0;
[f,df,~]=fun1(x);
[ceq,dceq,~]=con1(x);
[c,dc]=con2(x);
f0=f+y'*abs(ceq)+z'*abs(min(zeros(length(z),1),c));
f01=df'*p-y'*abs(ceq)-z'*abs(min(zeros(length(z),1),c))
    ;
while flag==0
%====Compute new point & functions' value====%
      xk=x+alpha*p;
      [f,df,~]=fun1(xk);
      [ceq,dceq,~]=con1(xk);
      [c,dc]=con2(xk);
      fa=f+y'*abs(ceq)+z'*abs(min(zeros(length(z),1),c));
%==============Check armijo condition==========%
      if fa<=f0+0.1*f01*alpha
          flag=1;
      else
%==============compute new alpha===============%
          a1=(fa-(f0+f01*alpha))/(alpha*alpha);
          amin=-f01/(2*a1);
          alpha=min(0.9*alpha,max(amin,0.1*alpha));
          initer=initer+1;
      end
end
end
```

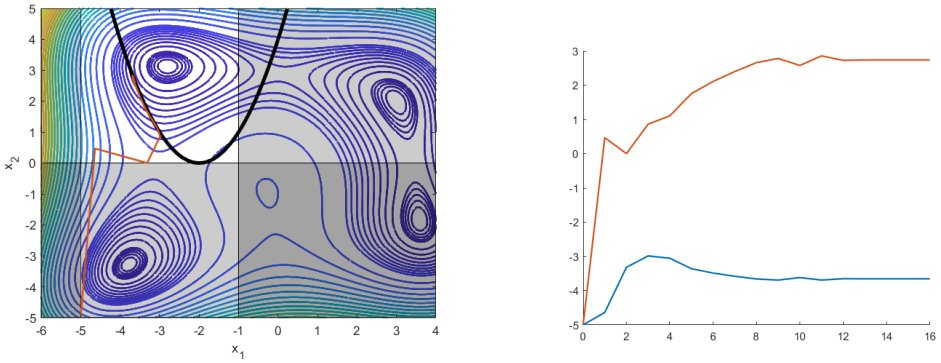
The results that we had after implementing SQP algorithm with Line Search method on Matlab were the following:

Starting Point	solution	time	iterations
$(-5,-5)$	$(-3.6546, 2.7377)$	0.0899	16 (6)*

(*)=line search function calls At the following table you can see the iteration sequence from the starting point to our solution. Due to space reasons we do not include all the sequence.

iter	0	1	2	3	....	13	14	15	16
x1	-5	-4.635	-3.317	-2.985	....	-3.654	-3.654	-3.6546	-3.6546
x2	-5	0.467	1.423e-11	0.862	....	2.737	2.737	2.737	2.7377

At the following plots we can see the progress of the convergence as well as the iteration sequence from the starting point to the solution.



(a) Iteration sequence (b) Convergence

Figure 8

5.8 Trust Region SQP Algorithm

Trust-Region SQP approach is a quite effective method because does not require the Hessian matrix of the Lagrangian to be positive definite. In our case we add to the problem a box constraint from a trust region Δ_k $(-\Delta_k e \leq p \leq \Delta_k e)$ so the quadratic sub-problem is:

$$\min_p \quad \frac{1}{2} p' H_k p + \nabla f(x_k)' p$$

$$s.t \quad \nabla h(x^k)' p = -h(x^k)$$

$$l - x^k \leq p \leq u - x^k$$

$$\Delta_k e \leq p \leq \Delta_k e$$

After the solution of the sub-problem choose a scalar ρ using the following formula:

$$\rho = \begin{cases} \frac{f(x_k+p)-f(x_k)}{0.5 p' H_k p + \nabla f(x_k)' p}, & \text{if } p \neq 0 \\ 1 & \text{if } p = 0 \end{cases}$$

At the next step the algorithm computes:

$$\gamma(\rho) = \begin{cases} 0.25, & \text{if } \rho < 0.25 \\ 1 & \text{if } 0.25 \leq \rho \leq 0.75 \\ 2, & \text{if } \rho > 0.75 \end{cases}$$

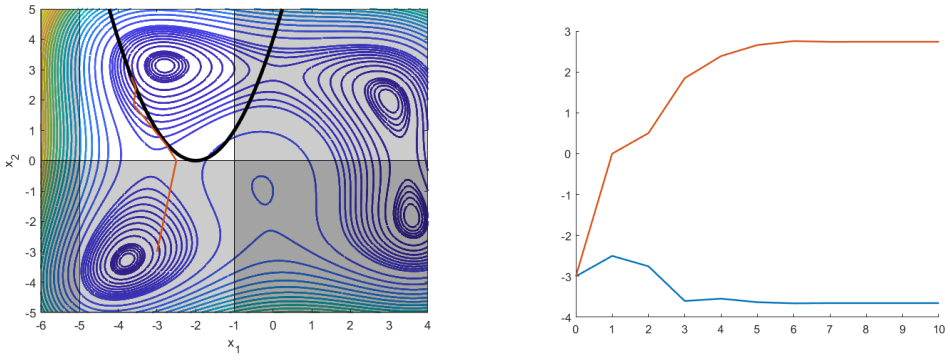
Finally, if $\rho > 0$ we accept the step, we compute the values of the functions, the gradient of the Lagrangian and we update the Hessian. Else, we reject the step and we compute the scalar $\Delta_{k+1} = \gamma(\rho)||p||_\infty$ to obtain a new trust region.
The results that we had after the implementation on Matlab were the following:

Starting Point	solution	time	iterations
$(-3, -3)$	$(-3.6546, 2.7377)$	0.0675	10 (2)*

The following table represents the iteration sequence of the algorithm.

iter	0	1	2	3	4	....	7	8	9	10
x1	-3	-2.5	-2.75	-3.60	-3.546	....	-3.654	-3.654	-3.654	-3.6546
x2	-3	5.25e-10	0.50	1.848	2.388	....	2.7368	2.7377	2.7377	2.7377

Below we can see graphically the progress of the iterations.



(a) Iteration sequence (b) Convergence

Figure 9

5.9 Interior-Point Algorithm for NLP

Interior-Point approach for Nonlinear Programming is at the same level effective as for Linear or Quadratic Programming. The structure of the algorithm

is quite similar with the approaches for LPs and even if it differs from SQP algorithms are considered together as the most successful algorithms for NLP. The algorithm use Error measure as convergence criteria which is described as:

$$E(x,s,y,z;\tau) = ||F(x,s,y,z;\tau)|| =$$
$$\max\{||\nabla L(x,y,z)||, ||Sz-\tau e||, ||c_E(x)||, ||c_I(x)-s||\}$$

Where y is the multiplier of the equality constraints, z is the multiplier of inequality constraints, s is the vector with the slack variables and τ is a positive parameter close to zero. The procedure begins from the solution of the above system:

$$\begin{bmatrix} \nabla^2_{xx}L(x,y,z) & 0 & \nabla c_E(x) & \nabla c_I(x) \\ 0 & \Sigma & 0 & -I \\ \nabla c_E(x)' & 0 & 0 & 0 \\ \nabla c_I(x)' & -I & 0 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_s \\ -p_y \\ -p_z \end{bmatrix} = \begin{bmatrix} \nabla_x L(x,y,z) \\ z-\tau S^{-1}e \\ c_E(x) \\ c_I(x)-s \end{bmatrix}$$

Where $\Sigma = S^{-1}Z$, $S = diag(s)$ and $Z = diag(z)$. Solving the above system we obtain the direction. After this we compute the step length:

$$\alpha_s = \max\{\alpha \in (0,1] \ : s + \alpha p_s \geq \eta s\}$$

$$\alpha_z = \max\{\alpha \in (0,1] \ : s + \alpha p_z \geq \eta z\}$$

Where $\eta = 0.005$ and then we take the step:

$$x_{k+1} = x_k + \alpha_s p_x, \quad s_{k+1} = s_k + \alpha_s p_s, \ y_{k+1} = y_k + \alpha_z p_y, \ z_{k+1} = z_k + \alpha_z p_z$$

5.10 Comparison

As we can easily understand all the above methods of solving Nonlinear Programs are quite effective but differs in the way that the minimum value is approached. From the tables of sequence as well as from the contour plots we can see the different paths that the point follows to reach the minimizer. Local SQP algorithm is a general method where the convergence is coming after more iterations compared to the other methods. The point does not use a quite predictable path to end up the minimizer. Line-search method takes the advantage of the constraints and tries to converge using the paths that are provided from the constraints doing more reasonable and predictable steps. Finally, Trust-region approach is trying to encircle the working region in even smaller regions using some extra box constraints in every next iteration to help the point reach to the minimizer. Even if the last two aforementioned approaches succeed to be converged more quickly, this is due to the fact that the algorithms use some inner loops to adjust the direction of the point.

Appendix

EqualityQPSolver

```
function [x,lambda]=EqualityQPSolver(H,g,A,b,solver)
expectedsolvers = {'NullSpace','RangeSpace','LUdense'
    ,...
    'LUsparse','LDLdense','LDLsparse'};

solvername = validatestring(solver,expectedsolvers,6);
switch solvername
    case {'LUdense'}
        [x,lambda]=EqualityQPSolverLUdense(H,g,A,b);
    case {'LUsparse'}
        [x,lambda]=EqualityQPSolverLUsparse(H,g,A,b);
    case {'LDLdense'}
        [x,lambda]=EqualityQPSolverLDLdense(H,g,A,b);
    case {'LDLsparse'}
        [x,lambda]=EqualityQPSolverLDLsparse(H,g,A,b);
    case {'NullSpace'}
        [x,lambda]=EqualityQPSolverNS(H,g,A,b);
    case {'RangeSpace'}
        [x,lambda]=EqualityQPSolverRS(H,g,A,b);
    otherwise
        error('Unknown Solver')
end
```

Test on a random EQP Problem

```
%==============Driver file for 14=============================%
clc;clear;
%=====Create symmetric positive definite matrix H=======%
H=randi([1,3],6);
H=H*H';
%=========Create random column vectors g, A, b==========%
g=randi([1,9],6,1);
A=randi([1,9],6,1);
b=randi([1,9]);
k=zeros(5,1);
%==========for loop to solve in sequence QP's==========%
for i=1:5
```

```
13      %=======Check-Create A with full column rank=======%
14      [U,S,V]=svd(A);
15      ra=i;
16      A=U(:, 1: ra)* S(1: ra, 1: ra)* V(:, 1: ra)';
17      k(i,1)=i;
18      %=========Solve the QP's and store time=========%
19      tic;
20      [x,lambda]=EqualityQPSolver(H,g,A,b,'LUdense');
21      t(i,1)=toc;
22      tic;
23      [x,lambda]=EqualityQPSolver(H,g,A,b,'LUsparse');
24      t(i,2)=toc;
25      tic;
26      [x,lambda]=EqualityQPSolver(H,g,A,b,'LDLdense');
27      t(i,3)=toc;
28      tic;
29      [x,lambda]=EqualityQPSolver(H,g,A,b,'LDLsparse');
30      t(i,4)=toc;
31      tic;
32      [x,lambda]=EqualityQPSolver(H,g,A,b,'NullSpace');
33      t(i,5)=toc;
34      tic;
35      [x,lambda]=EqualityQPSolver(H,g,A,b,'RangeSpace');
36      t(i,6)=toc;
37      %=========Add another column constraint=========%
38      A1=randi([1,9],6,1);
39      b1=randi([1,9]);
40      A=[A A1];
41      b=[b;b1];
42  end
43  %=================Plot the results=====================%
44  plot(k,t(:,1),'r',k,t(:,2),'m',k,t(:,3),'y',k,t(:,4),'c
        ',k,t(:,5),...
45        'b',k,t(:,6),'g','Linewidth',2)
46  lgd=legend('LUdense','LUsparse','LDLdense','LDLsparse',
        'NullSpace',...
47        'RangeSpace');
48  lgd.NumColumns = 2;
49  title("Solvers' behavior in respect to number of
        Constraints");
50  xticks([1,2,3,4,5])
```

```
xlabel('Number of constraints');
ylabel('Time');
```

Interior-Point QP function

```
%          Convex Quadratic Programming
%      Primal-Dual Interior Point Algorithm
%==========================================================
%                 min (1/2)*x'*H*x + g'*x
%          s.t          A'*x = b
%                       C'*x >= d
%==========================================================

function [x,fval,y,z,iter,t] = Interior_Point(H,g,A,b,C,
    d,x0)
tStart=tic;
n = length(H); [~,mc] = size(C); [~,n1] = size(A);
x = x0; y = ones(n1,1);z = ones(mc,1);s = ones(mc,1);
itermax = 150;
iter = 0;

tol = 10e-9;
%compute the residuals
rL = H*x+g-A*y-C*z;
rA = b-A'*x;
rC = d-C'*x+s;
S = diag(s);Z=diag(z);e=ones(mc,1);
rsz = S*Z*e;
mu = (z'*s)/mc;
 %Stopping Criteria
Converged = (norm(rL, inf) < tol ) && (norm(rA , inf) <
     tol) &&...
     (abs(mu)<tol);
%Start the loop
while ~Converged && (iter<itermax)
     iter = iter+1;
%First System to affine direction
     H1=H+C*(inv(S)*Z)*C';
     KKT=[H1 -A;-A' zeros(n1)];
     rL1=rL-C*(inv(S)*Z)*(rC-inv(Z)*rsz);
     RH=-[rL1;rA];
```

```
35      %LDL factorization
36      [L,D,p] = ldl(KKT,'lower','vector');
37      sol(p) = L'\(D\(L\RH(p)));
38      %Affine direction
39      dxaff = sol(1:n)'; dyaff = sol(n+1:n+n1)';
40      dzaff=-(inv(S)*Z)*C'*dxaff+(inv(S)*Z)*(rC-inv(Z)*
            rsz);
41      dsaff=-inv(Z)*rsz-inv(Z)*S*dzaff;
42      idx = find(dzaff < 0.0);
43      alpha = min([1.0; -z(idx,1)./dzaff(idx,1)]);
44
45      idx = find(dsaff < 0.0);
46      beta = min([1.0; -s(idx,1)./dsaff(idx,1)]);
47      alpha_aff = min(alpha,beta);
48     %Duality gap
49      muaff = ((z+alpha_aff*dzaff)'*(s+alpha_aff*dsaff))/
            mc;
50
51      sigma = (muaff/mu)^3;
52      %Affine-Centering-Correction Direction
53      rsz1=rsz+diag(dsaff)*diag(dzaff)*e-sigma*mu*e;
54      rL1=rL-C*(inv(S)*Z)*(rC-inv(Z)*rsz1);
55      RH=-[rL1;rA];
56      [L,D,p] = ldl(KKT,'lower','vector');
57      sol(p) = L'\(D\(L\RH(p)));
58  % Solution to obtain direction toward optimal
59      dx = sol(1:n)'; dy = sol(n+1:n+n1)';
60      dz=-(inv(S)*Z)*C'*dx+(inv(S)*Z)*(rC-inv(Z)*rsz1);
61      ds=-inv(Z)*rsz1-inv(Z)*S*dz;
62
63
64
65      idx = find(dz < 0.0);
66      alpha = min([1.0; -z(idx,1)./dz(idx,1)]);
67
68      idx = find(ds < 0.0);
69      beta = min([1.0; -s(idx,1)./ds(idx,1)]);
70      alpha = min(alpha,beta);
71    %Move to the new point
72      x=x+0.995*alpha*dx;
73      y=y+0.995*alpha*dy;
```

```
74      z=z+0.995*alpha*dz;
75      s=s+0.995*alpha*ds;
76
77      rL = H*x+g-A*y-C*z;
78      rA = b-A'*x;
79      rC = d-C'*x+s;
80      S = diag(s);Z = diag(z);e = ones(mc,1);
81      rsz = S*Z*e;
82      mu = (z'*s)/mc;
83
84      Converged = (norm(rL, inf) < tol ) && (norm(rA ,
            inf) < tol) &&...
85          (abs(mu)<tol);
86  end
87  x = x(:,end);
88  fval = 0.5*x'*H*x + g'*x;
89  t = toc(tStart);
90  end
```

Active-Set QP function

```
1  %              Active Set Algorithm
2  %              Quadratic Programming
3  %=========================================================
4  % min (1/2)*x'*G*x + g'*x
5  %  s.t A'*x = b
6  %       C'*x >=d
7  %=========================================================
8  % k = iterations
9  % t = cputime
10 function [x,lambda,mu,fval,t,k] = Active_Set(G,g,A,b,C,
       d,x0)
11 tStart = tic;
12 workingSet = find(C'*x0-d == 0);
13 nonworkingSet = find(C'*x0-d ~= 0);
14 flag = 0;
15 k = 0;
16 x = x0;
17 mu=zeros(length(C),1);
18 maxiter=100;
19 [~,n2]=size(A);
```

```
20
21
22 while flag == 0 && k<maxiter
23     k = k+1;
24     n1=length(workingSet);
25
26
27     KKT = [G -A -C(:,workingSet);
28            -A' zeros(n2) zeros(n2,n1);
29            -C(:,workingSet)' zeros(n1,n2) zeros(n1)];
30     RHS = -[G*x(:,k)+g; b-A'*x(:,k); d(workingSet)-C(:,
           workingSet)'*x(:,k)];
31
32     sol = KKT\RHS;
33     p = sol(1:length(G));
34     lambda=sol(length(G)+1 : length(G)+n2);
35     mu(workingSet) = sol(length(G)+n2+1 : end);
36
37     if norm(p)<10e-9
38         if all(mu(workingSet)>=0)
39             flag = 1;
40             k = k-1;
41             mu(nonworkingSet)=0;
42         else
43             [minimum,j1] = min(mu(workingSet));
44             j=find(mu==minimum);
45             nonworkingSet = sort([nonworkingSet;j]);
46             workingSet(j1) = [];
47             x(:,k+1) = x(:,k);
48         end
49     else
50         temp1 = C(:,nonworkingSet)'*p;
51         index = find(temp1<0);
52         i = nonworkingSet(index);
53         temp = (d(i)-C(:,i)'*x(:,k))./(C(:,i)'*p);
54         [alpha,j] = min(temp);
55         j = i(j);
56         if alpha<1
57             x(:,k+1) = x(:,k)+alpha*p;
58             workingSet = sort([workingSet;j]);
59             j = find(nonworkingSet==j);
```

```
60                  nonworkingSet(j) = [];
61              else
62                  x(:,k+1) = x(:,k)+p;
63              end
64          end
65      end
66      fval=0.5*x(:,end)'*G*x(:,end) + g'*x(:,end);
67      x=x(:,end);
68      t=toc(tStart);
69      mu;
70      end
```

Markowitz Problem using 3 Algorithms

```
1   %=========================================%
2   %          Driver file for 2.8            %
3   %=========================================%
4   clc; clear;
5   %=====Initialize data=====================%
6   G = [2.3  0.93  0.62  0.74  -0.23;
7         0.93  1.4  0.22  0.56  0.26;
8         0.62  0.22  1.8  0.78  -0.27;
9         0.74  0.56  0.78  3.4  -0.56;
10        -0.23  0.26  -0.27  -0.56  2.6];
11  A = [15.1  1;
12       12.5  1;
13       14.7  1;
14       9.02  1;
15       17.68  1];
16  g=zeros(5,1); C=eye(5); b = [10; 1]; d=zeros(5,1); x0
        =0.2*ones(5,1);
17  %=======Create matrix for results========%
18  matrix=cell(4,7);
19  matrix(2,1)={"Active-Set Algorithm"};
20  matrix(3,1)={"Interior-Point Algorithm"};
21  matrix(4,1)={"quadprog library"};
22  matrix(1,2)={"Optimal"};
23  matrix(1,3)={"lambda_Eq"};
24  matrix(1,4)={"lambda_Ineq"};
25  matrix(1,5)={"f-value"};
26  matrix(1,6)={"time"};
```

```
matrix(1,7)={"Iterations"};

%======Solution Active-Set & store data=====%
disp("==========Active-Set Algorithm==========")
[x,lambda,mu,fval,t,k] = Active_Set(G,g,A,b,C,d,x0)
matrix(2,2)={x};
matrix(2,3)={lambda};
matrix(2,4)={mu};
matrix(2,5)={fval};
matrix(2,6)={t};
matrix(2,7)={k};
%======Solution Interior-Point & store data=====%
disp("==========Interior-Point Algorithm==========")
H=G;
[x,fval,y,z,iter,t] = Interior_Point(H,g,A,b,C,d,x0)
matrix(3,2)={x};
matrix(3,3)={y};
matrix(3,4)={z};
matrix(3,5)={fval};
matrix(3,6)={t};
matrix(3,7)={iter};
%======Solution quadprog & store data=====%
disp("==========quadprog library==========")
f=g; Aeq=A'; beq=b; lb=zeros(5,1); ub=ones(5,1);
t1=tic;
[x,fval,exitflag,output,lambda] = quadprog(H,f,[],[],
    Aeq,beq,lb,ub,x0)
time=toc(t1)
matrix(4,2)={x};
matrix(4,3)={lambda.eqlin};
matrix(4,4)={'-'};
matrix(4,5)={fval};
matrix(4,6)={time};
matrix(4,7)={output.iterations};
disp("For summary solution click on matrix at the
    Workspace!!!!")

matrix
```

Markowitz script for R=10

```
%==========Driver file for 3.3-3.4==========%
clc; clear;
H = [2.3  0.93  0.62  0.74  -0.23;
      0.93  1.4  0.22  0.56  0.26;
      0.62  0.22  1.8  0.78  -0.27;
      0.74  0.56  0.78  3.4  -0.56;
      -0.23  0.26  -0.27  -0.56  2.6];
Aeq = [15.1  1;
     12.5  1;
     14.7  1;
     9.02  1;
     17.68  1]';
f=zeros(5,1); x0=0.2*ones(5,1);
lb=zeros(5,1); ub=ones(5,1);

risk=zeros(10,2);

i=1;
%==========collect data to plot==========%
for R=9:0.2:17.6
     beq=[R;1];
     options = optimoptions('quadprog','Display','off');
     [x,fval,exitflag,output,lambda] = quadprog(H,f
         ,[],[],Aeq,beq,lb,ub,x0,options);
     if R==10
          disp(x)
          disp(2*fval)
     end
     risk(i,1)=R;
     risk(i,2)=2*fval;
     sol1(1,i)=x(1);sol1(2,i)=R;
     sol2(1,i)=x(2);sol2(2,i)=R;
     sol3(1,i)=x(3);sol3(2,i)=R;
     sol4(1,i)=x(4);sol4(2,i)=R;
     sol5(1,i)=x(5);sol5(2,i)=R;
     i=i+1;
end
figure(1)
plot(risk(:,2),risk(:,1),'-','LineWidth',2)
title("Efficient Frontier")
xlabel("Variance")
```

```
ylabel("Expected Return")

figure(2)
hold on
plot(sol1(2,:),sol1(1,:),'-','LineWidth',2,'DisplayName','security 1')
plot(sol2(2,:),sol2(1,:),'-','LineWidth',2,'DisplayName','security 2')
plot(sol3(2,:),sol3(1,:),'-','LineWidth',2,'DisplayName','security 3')
plot(sol4(2,:),sol4(1,:),'-','LineWidth',2,'DisplayName','security 4')
plot(sol5(2,:),sol5(1,:),'-','LineWidth',2,'DisplayName','security 5')
legend('security 1','security 2','security 3','security 4','security 5')
ylabel("Portfolio's weighting factors")
xlabel("Return")
hold off
```

Markowitz Problem with risk free security

```
%========Driver file for 3.5=======%
clc;clear;
H = [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];
Aeq = [15.1 1;
    12.5 1;
    14.7 1;
    9.02 1;
    17.68 1;
    2 1]';
f=zeros(6,1); x0=[0.1 0.1 0.2 0.2 0.2 0.2]';
lb=zeros(6,1); ub=ones(6,1);

risk=zeros(10,2);

```

```
20 i=1;
21 options = optimoptions('quadprog','Display','off');
22 for R=2:0.2:17.6
23     beq=[R;1];
24     [x,fval] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,
           options);
25     risk(i,1)=R;
26     %==== Compute Variance=========%
27     risk(i,2)=2*fval;
28     %============ Store data ================%
29     sol1(1,i)=x(1);sol1(2,i)=R;
30     sol2(1,i)=x(2);sol2(2,i)=R;
31     sol3(1,i)=x(3);sol3(2,i)=R;
32     sol4(1,i)=x(4);sol4(2,i)=R;
33     sol5(1,i)=x(5);sol5(2,i)=R;
34     sol6(1,i)=x(6);sol6(2,i)=R;
35     i=i+1;
36 end
37
38 %====Efficient frontier plot with securities===%
39 figure(1)
40 hold on
41 plot(risk(:,2),risk(:,1),'LineWidth',2)
42 legendInfo{1} = ['Efficient frontier'];
43
44 s=diag(H);
45 for j=1:6
46    plot(s(j),Aeq(1,j),'X','LineWidth',3)
47    legendInfo{j+1} = ['Security ' num2str(j)];
48 end
49 legend(legendInfo)
50 title("Efficient Frontier")
51 xlabel("Variance")
52 ylabel("Expected Return")
53 hold off
54 %============ optimal portfolio=============%
55 figure(2)
56 hold on
57 plot(sol1(2,:),sol1(1,:),'-','LineWidth',2,'DisplayName
       ','security 1')
58 plot(sol2(2,:),sol2(1,:),'-','LineWidth',2,'DisplayName
```

```
    ','security 2')
59 plot(sol3(2,:),sol3(1,:),'-','LineWidth',2,'DisplayName
    ','security 3')
60 plot(sol4(2,:),sol4(1,:),'-','LineWidth',2,'DisplayName
    ','security 4')
61 plot(sol5(2,:),sol5(1,:),'-','LineWidth',2,'DisplayName
    ','security 5')
62 plot(sol6(2,:),sol6(1,:),'-','LineWidth',2,'DisplayName
    ','free risk sec.')
63 plot(sol1(2,41),sol1(1,41),'X','LineWidth',2)
64 plot(sol2(2,41),sol2(1,41),'X','LineWidth',2)
65 plot(sol3(2,41),sol3(1,41),'X','LineWidth',2)
66 plot(sol4(2,41),sol4(1,41),'X','LineWidth',2)
67 plot(sol5(2,41),sol5(1,41),'X','LineWidth',2)
68 plot(sol6(2,41),sol6(1,41),'X','LineWidth',2)
69
70 legend('security 1','security 2','security 3','security
    4','security 5','free risk sec.')
71 ylabel("Portfolio's weighting  factors ")
72 xlabel("Return")
73 hold off
74 %========optimal portfolio results for R=10=========%
75 H = [2.3  0.93  0.62  0.74  -0.23  0;
76       0.93  1.4  0.22  0.56  0.26  0;
77       0.62  0.22  1.8  0.78  -0.27  0;
78       0.74  0.56  0.78  3.4  -0.56  0;
79       -0.23  0.26  -0.27  -0.56  2.6  0;
80       0  0  0  0  0  0];
81 Aeq = [15.1  1;
82      12.5  1;
83      14.7  1;
84      9.02  1;
85      17.68  1;
86      2  1]';
87 f=zeros(6,1); x0=[0.1  0.1  0.2  0.2  0.2  0.2]';
88 lb=zeros(6,1); ub=ones(6,1);
89 R=10; beq=[R;1];
90 disp("==== Optimal Portfolio with risk free security
    ====")
91 [x,fval] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,options)
```

Markowitz Problem with risk free security for R=15

```
%========Driver file for 3.5========%

H = [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];
Aeq = [15.1 1;
     12.5 1;
     14.7 1;
     9.02 1;
     17.68 1;
     2 1]';
f=zeros(6,1); x0=[0.1 0.1 0.2 0.2 0.2 0.2]';
lb=zeros(6,1); ub=ones(6,1);

risk=zeros(10,2);

i=1;
options = optimoptions('quadprog','Display','off');
for R=2:0.2:17.6
     beq=[R;1];
     [x,fval] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,
        options);
     risk(i,1)=R;
     %==== Compute Variance=========%
     risk(i,2)=2*fval;
     %========== Store data ==============%
     sol1(1,i)=x(1);sol1(2,i)=R;
     sol2(1,i)=x(2);sol2(2,i)=R;
     sol3(1,i)=x(3);sol3(2,i)=R;
     sol4(1,i)=x(4);sol4(2,i)=R;
     sol5(1,i)=x(5);sol5(2,i)=R;
     sol6(1,i)=x(6);sol6(2,i)=R;
     i=i+1;
end

%====Efficient frontier plot with securities==%
```

```
figure(1)
hold on
plot(risk(:,2),risk(:,1),'LineWidth',2)
legendInfo{1} = ['Efficient frontier'];
s=diag(H);
for j=1:6
    plot(s(j),Aeq(1,j),'X','LineWidth',3)
    legendInfo{j+1} = ['Security ' num2str(j)];
end
plot(0.6383,15,'Xr','LineWidth',4)
legendInfo{8}=['minimal risk'];
legend(legendInfo);
title("Efficient Frontier")
xlabel("Variance")
ylabel("Expected Return")
hold off
%============ optimal portfolio============%
figure(2)
hold on
plot(sol1(2,:),sol1(1,:),'-','LineWidth',2,'DisplayName','security 1')
plot(sol2(2,:),sol2(1,:),'-','LineWidth',2,'DisplayName','security 2')
plot(sol3(2,:),sol3(1,:),'-','LineWidth',2,'DisplayName','security 3')
plot(sol4(2,:),sol4(1,:),'-','LineWidth',2,'DisplayName','security 4')
plot(sol5(2,:),sol5(1,:),'-','LineWidth',2,'DisplayName','security 5')
plot(sol6(2,:),sol6(1,:),'-','LineWidth',2,'DisplayName','free risk sec.')
plot(sol1(2,66),sol1(1,66),'X','LineWidth',2)
plot(sol2(2,66),sol2(1,66),'X','LineWidth',2)
plot(sol3(2,66),sol3(1,66),'X','LineWidth',2)
plot(sol4(2,66),sol4(1,66),'X','LineWidth',2)
plot(sol5(2,66),sol5(1,66),'X','LineWidth',2)
plot(sol6(2,66),sol6(1,66),'X','LineWidth',2)

legend('security 1','security 2','security 3','security 4',...
    'security 5','free risk sec.')
```

```
73 ylabel('Portfolio's weighting  factors ')
74 xlabel('Return')
75 hold off
76 %=========optimal portfolio results for R=15==========%
77 H = [2.3  0.93  0.62  0.74  -0.23  0;
78       0.93  1.4  0.22  0.56  0.26  0;
79       0.62  0.22  1.8  0.78  -0.27  0;
80       0.74  0.56  0.78  3.4  -0.56  0;
81       -0.23  0.26  -0.27  -0.56  2.6  0;
82       0  0  0  0  0  0];
83 Aeq = [15.1  1;
84      12.5  1;
85      14.7  1;
86      9.02  1;
87      17.68  1;
88      2  1]';
89 f=zeros(6,1); x0=[0.1  0.1  0.2  0.2  0.2  0.2]';
90 lb=zeros(6,1); ub=ones(6,1);
91 R=15; beq=[R;1];
92 disp('==== Optimal Portfolio with risky security at R
      =15 ====')
93 [x,fval] = quadprog(H,f,[],[],Aeq,beq,lb,ub,x0,options)
```

Interior-Point QP function

```
1  function [x,info,mu,lambda,iter] = LPippd(g,A,b,x)
2  % LPIPPD   Primal-Dual Interior-Point LP Solver
3  %
4  %           min  g'*x
5  %            x
6  %           s.t. A x  = b       (Lagrange multiplier: mu)
7  %                  x >= 0       (Lagrange multiplier:
       lamba)
8  %
9  % Syntax: [x,info,mu,lambda,iter] = LPippd(g,A,b,x)
10 %
11 %         info = true   : Converged
12 %              = false  : Not Converged
13
14 % Created: 04.12.2007
15 % Author : John Bagterp Jorgensen
```

```
16 %           IMM, Technical University of Denmark
17
18 %%
19 [m,n]=size(A);
20
21 maxit = 100;
22 tolL = 1.0e-9;
23 tolA = 1.0e-9;
24 tols = 1.0e-9;
25
26 eta = 0.99;
27
28 lambda = ones(n,1);
29 mu = zeros(m,1);
30
31 % Compute residuals
32 rL = g - A'*mu - lambda;      % Lagrangian gradient
33 rA = A*x - b;                 % Equality Constraint
34 rC = x.*lambda;               % Complementarity
35 s = sum(rC)/n;                % Duality gap
36
37 % Converged
38 Converged = (norm(rL,inf) <= tolL) && ...
39              (norm(rA,inf) <= tolA) && ...
40              (abs(s) <= tols);
41 %%
42 iter = 0;
43 while ~Converged && (iter<maxit)
44     iter = iter+1;
45
46     %
          ==========================================================
47     % Form and Factorize Hessian Matrix
48     %
          ==========================================================
49     xdivlambda = x./lambda;
50     H = A*diag(xdivlambda)*A';
51     L = chol(H,'lower');
52
```

```
53      %
            ================================================================

54      % Affine Step
55      %
            ================================================================

56      % Solve
57      tmp = (x.*rL + rC)./lambda;
58      rhs = -rA + A*tmp;
59
60      dmu = L'\(L\rhs);
61      dx = xdivlambda.*(A'*dmu) - tmp;
62      dlambda = -(rC+lambda.*dx)./x;
63
64      % Step length
65      idx = find(dx < 0.0);
66      alpha = min([1.0; -x(idx,1)./dx(idx,1)]);
67
68      idx = find(dlambda < 0.0);
69      beta = min([1.0; -lambda(idx,1)./dlambda(idx,1)]);
70
71      %
            ================================================================

72      % Center Parameter
73      %
            ================================================================

74      xAff = x + alpha*dx;
75      lambdaAff = lambda + beta*dlambda;
76      sAff = sum(xAff.*lambdaAff)/n;
77
78      sigma = (sAff/s)^3;
79      tau = sigma*s;
80
81      %
            ================================================================

82      % Center-Corrector Step
83      %
```

```
84      rC = rC + dx.*dlambda - tau;
85
86      tmp = (x.*rL + rC)./lambda;
87      rhs = -rA + A*tmp;
88
89      dmu = L'\(L\rhs);
90      dx = xdivlambda.*(A'*dmu) - tmp;
91      dlambda = -(rC+lambda.*dx)./x;
92
93      % Step length
94      idx = find(dx < 0.0);
95      alpha = min([1.0; -x(idx,1)./dx(idx,1)]);
96
97      idx = find(dlambda < 0.0);
98      beta = min([1.0; -lambda(idx,1)./dlambda(idx,1)]);
99
100     %
        ================================================================

101     % Take step
102     %
        ================================================================

103     x = x + (eta*alpha)*dx;
104     mu = mu + (eta*beta)*dmu;
105     lambda = lambda + (eta*beta)*dlambda;
106
107     %
        ================================================================

108     % Residuals and Convergence
109     %
        ================================================================

110     % Compute residuals
111     rL = g - A'*mu - lambda;      % Lagrangian gradient
112     rA = A*x - b;                 % Equality Constraint
113     rC = x.*lambda;               % Complementarity
114     s = sum(rC)/n;                % Duality gap
```

```
115
116     % Converged
117     Converged = (norm(rL,inf) <= tolL) && ...
118                 (norm(rA,inf) <= tolA) && ...
119                 (abs(s) <= tols);
120 end
121
122 %%
123 % Return solution
124 info = Converged;
125 if ~Converged
126     x=[];
127     mu=[];
128     lambda=[];
129 end
```

Simplex function

```
1  %   Revised Simplex Algorithm
2  %=========================================
3  %  LP in Standard form
4  %       min f(x)= g'*x
5  %         s.t   A*x=b
6  %=========================================
7
8
9  function [x,mu,lambda,optimal,iter]=Simplex(g,A,b)
10 %=====Initialize Data=======================
11 [m,n] = size(A);
12 basic=1;
13 lambda = zeros(n,1);
14 iter = 0;
15 set=[1:n];
16 x=-1*ones(n,1);
17 flag = 0;
18 %slack=find(g==0)
19 %=====Choose Basic and Non Basic Sets=====
20 while ~all(x(basic)>=0)
21     non_basic = randperm(n,n-m);
22     basic = setdiff(set,non_basic);
23     B = A(:,basic);
```

```
24      N = A(:,non_basic);
25      if det(B)~=0
26          x(basic) = pinv(B)*b;
27          x(non_basic)=0;
28          mu = B'\g(basic);
29          lambda(non_basic) = g(non_basic)-N'*mu;
30          lambda(basic)=0;
31      else
32          x=-1*ones(n,1);
33      end
34  end
35
36  %=====Simplex Loop=======
37  while flag==0
38      iter = iter + 1;
39      B = A(:,basic);
40      N = A(:,non_basic);
41      x(basic) = B\b;
42      x(non_basic)=0;
43      mu = B'\g(basic);
44      lambda(non_basic) = g(non_basic)-N'*mu;
45      lambda(basic)=0;
46  %=====Condition for break=====
47      if lambda(non_basic)>=0
48          flag = 1;
49          optimal = g(basic)'*x(basic);
50      else
51  %=====Conditions to change Sets====
52          [lambda_min,s] = min(lambda(non_basic));
53          s = find(lambda==lambda_min);
54          h = B\A(:,s);
55          index = find(h>0);
56          [~,J] = min(x(basic(index))./h(index));
57          J = index(J);
58          J = basic(J);
59          if isempty(J)
60              flag = 1;
61              disp("*** The LP is Unbounded ***")
62              x=[]; mu=[]; lambda=[]; optimal=[];
63          else
64  %=====Change The Sets=====
```

```
65                  non_basic(end+1) = J;
66                  basic(end+1) = s;
67                  non_basic = non_basic(non_basic ~= s);
68                  basic = basic( basic ~= J);
69              end
70          end
71      end
```

Used functions

```
1  function [f,df,d2f]=fun1(x)
2  x1=x(1,1);
3  x2=x(2,1);
4
5  tmp1=x1^2+x2-11;
6  tmp2=x1+x2^2-7;
7
8  f=tmp1^2+tmp2^2;
9  df=zeros(2,1);
10 df(1,1)=4*x1*tmp1+2*tmp2;
11 df(2,1)=2*tmp1+4*x2*tmp2;
12
13 d2f=zeros(2,2);
14 d2f(1,1)=4*tmp1+8*x1^2+2;
15 d2f(2,1)=4*(x1+x2);
16 d2f(1,2)=d2f(2,1);
17 d2f(2,2)=4*tmp2+8*x2^2+2;
```

```
1  function [ceq,dceq,d2ceq]=con1(x)
2  x1=x(1,1);
3  x2=x(2,1);
4
5  tmp=x1+2;
6
7  ceq=tmp^2-x2;
8
9  dceq=zeros(2,1);
10 dceq(1,1)=2*tmp;
11 dceq(2,1)=-1;
12
13 d2ceq=zeros(2,2);
14 d2ceq(1,1)=2;
```

```
function [c,dc]=con2(x)
x1=x(1,1);
x2=x(2,1);

c=zeros(4,1);
c(1,1)=x1+5;
c(2,1)=x2;
c(3,1)=-x1-1;
c(4,1)=-x2+5;

dc=[eye(2) -eye(2)];
```

Damped BFGS function

```
function H= Damped_BFGS(H,q,p);

if p'*q>=0.2*p'*(H*p)
    theta=1;
else
    theta=(0.8*p'*(H*p))/(p'*(H*p)-p'*q);
end
ro=theta*q+(1-theta)*(H*p);

H=H-(((H*p)*(H*p)')/(p'*(H*p)))+(ro*ro')/(p'*ro);
end
```

SQP with Line-Search approach

```
%==========SQP Algorithm with Damped BFGS=======%
%              with line-search approach
%================Initialize data================%
clc; clear;
x=[-5;-5]; y=1;
%Lagange multipliers for lower-upper bound constraints
z1=ones(2,1); z2=ones(2,1);
k=0;%Iterations
kmax=80;
lb=[-5;0]; ub=[-1;5];
%================call functions================%
[f,df,d2f]=fun1(x);
[ceq,dceq,d2ceq]=con1(x);
[c,dc]=con2(x);
```

```
15 options =  optimoptions('quadprog','Display','off');
16 H=eye(2); %Initial Hessian
17 tol=10e-7;
18 converged=false; t1=tic; line_iter=0;
19 while ~converged && k<kmax
20     k=k+1;
21  %=========Set the lower and upper bounds=======%
22     lb1=lb-x(:,end);
23     ub1=ub-x(:,end);
24  %====================Solve QP==================%
25     [p,~,~,~,lambda] = quadprog(H,df,[],[],dceq',-ceq,
           lb1,ub1,[],options);
26  %================Obtain multipliers============%
27     y=-lambda.eqlin;
28     z1=lambda.lower;
29     z2=lambda.upper;
30  %================Gradient of Lagrangian========%
31     dL=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc(:,3)*z2
           (1)-dc(:,4)*z2(2);
32     z=[z1;z2];
33     [x(:,end+1),initer]=linesearch(x(:,end),y,z,p);
34     line_iter=line_iter+initer;
35  %==========Function values at new point=======%
36     [f,df,d2f]=fun1(x(:,end));
37     [ceq,dceq,d2ceq]=con1(x(:,end));
38     [c,dc]=con2(x(:,end));
39
40     dL1=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc(:,3)*
           z2(1)-dc(:,4)*z2(2);
41     q=dL1-dL;
42 %========Call Damped BFGS to update Hessian=====%
43     H= Damped_BFGS(H,q,p);
44 %==========Set convergence criteria============%
45     converged= ( norm(dL1,inf)<tol) && (norm(ceq,inf)<
           tol);
46
47 end
48 time=toc(t1)
49 x(:,end)
50 k
51 line_iter
```

```
52 %%
53 %==========PLOT 1- point movements==========%
54 figure(1)
55 X1=-6:0.005:4;
56 X2=-5:0.005:5;
57 [X1,X2]=meshgrid(X1,X2);
58 f=(X1.^2+X2-11).^2+(X1+X2.^2-7).^2;
59 v = [0:2:10 10:10:100 100:40:1000];
60 contour(X1,X2,f,v,'LineWidth',1.5)
61 xlabel('x_{1}')
62 ylabel('x_{2}')
63 zlabel('f')
64
65 x2=-6:0.01:4;
66 y2=(x2+2).^2;
67 hold on
68 plot(x2,y2,'k','LineWidth',3)
69 xlim([-6 4])
70 ylim([-5 5])
71
72 fill([X1(1) -5 -5 X1(1)],[X2(1) X2(1) X2(end) X2(end)
     ],...
73      'k','facealpha',0.2)
74 fill([-1 X1(end) X1(end) -1],[X2(1) X2(1) X2(end) X2(
     end)],...
75      'k','facealpha',0.2)
76 fill([X1(1) X1(end) X1(end) X1(1)],[X2(1) X2(1) 0
     0],...
77      'k','facealpha',0.2)
78 fill([X1(1) X1(end) X1(end) X1(1)],[5 5 X2(end) X2(end)
     ],...
79      'k','facealpha',0.2)
80 hold off
81
82 hold on
83 plot(x(1,:),x(2,:),'Linewidth',1.5)
84 hold off
85 %==========Convergence of the point==========%
86 k1=[0:k];
87 figure(2)
88 hold on
```

```
89 plot(k1,x(1,:),'Linewidth',1.5)
90 plot(k1,x(2,:),'Linewidth',1.5)
91 hold off
```

Trust-Region approach

```
1  %==========SQP Trust region algorithm=========%
2  %================Initialize data================%
3  clc; clear;
4  x=[-3;-3]; y=1;
5  z1=ones(2,1); z2=ones(2,1);
6  k=0; kmax=50;
7  [f,df,d2f]=fun1(x); %functions at starting point
8  [ceq,dceq,d2ceq]=con1(x);
9  [c,dc]=con2(x);
10 lb=[-5;0];
11 ub=[-1;5];
12 b1=[0;5;5;-1];
13
14 H=d2f; %Hessian
15 tol=10e-7;
16 options = optimoptions('quadprog','Display','off');
17 dk=inf; %Starting trust region
18 converged=false;
19 initer=0;
20 t1=tic;
21 %=====Enter loop to solve sub-problems=====%
22 while ~converged &&(k<kmax)
23 %=========Combine LB-UB constraints========%
24     lb1=lb-x(:,end);
25     ub1=ub-x(:,end);
26     lbb=-dk*ones(2,1);
27     ubb=dk*ones(2,1);
28     lb1=max(lb1,lbb);
29     ub1=min(ub1,ubb);
30     [p,~,~,~,lambda] = quadprog(H,df,[],[],dceq',-ceq,
           lb1,ub1,[],options);
31     y=-lambda.eqlin;
32     z1=lambda.lower;
33     z2=lambda.upper;
34
```

```
35      [f1,df1,d2f1]=fun1(x(:,end)+p);
36  %============Conditions for ro============%
37      if p~=0
38          ro=(f1-f)/(0.5*p'*H*p+df'*p);
39      else
40          ro=1;
41      end
42  %============Conditions for gamma============%
43      if ro<0.25
44          gamma=0.25;
45      elseif (ro<=0.75) && (ro>=0.25)
46          gamma=1;
47      else
48          gamma=2;
49      end
50  %================Accept step================%
51      if ro>0
52          k=k+1;
53          dk=gamma*dk;
54          x(:,end+1)=x(:,end)+p;
55          gL=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc
                (:,3)*z2(1)-dc(:,4)*z2(2);
56          [f,df,d2f]=fun1(x(:,end));
57          [ceq,dceq,d2ceq]=con1(x(:,end));
58          [c,dc]=con2(x(:,end));
59          gL1=df-dceq*y-dc(:,1)*z1(1)-dc(:,2)*z1(2)-dc
                (:,3)*z2(1)-dc(:,4)*z2(2);
60          q=gL1-gL;
61          H= Damped_BFGS(H,q,p);
62          converged=(norm(gL1,inf)<tol) &&(norm(p,inf)<
                tol);
63      else
64  %================Reject step================%
65          dk=norm(p,inf)*gamma;
66          initer=initer+1;
67      end
68  end
69  time=toc(t1)
70  k
71  x
72  initer
```

```
73 %%
74 %==========PLOT 1- point movements==========%
75 figure(1)
76 X1=-6:0.005:4;
77 X2=-5:0.005:5;
78 [X1,X2]=meshgrid(X1,X2);
79 f=(X1.^2+X2-11).^2+(X1+X2.^2-7).^2;
80 v = [0:2:10 10:10:100 100:40:1000];
81 contour(X1,X2,f,v,'LineWidth',1.5)
82 xlabel('x_{1}')
83 ylabel('x_{2}')
84 zlabel('f')
85
86 x2=-6:0.01:4;
87 y2=(x2+2).^2;
88 hold on
89 plot(x2,y2,'k','LineWidth',3)
90 xlim([-6 4])
91 ylim([-5 5])
92
93 fill([X1(1) -5 -5 X1(1)],[X2(1) X2(1) X2(end) X2(end)
    ],...
94     'k','facealpha',0.2)
95 fill([-1 X1(end) X1(end) -1],[X2(1) X2(1) X2(end) X2(
    end)],...
96     'k','facealpha',0.2)
97 fill([X1(1) X1(end) X1(end) X1(1)],[X2(1) X2(1) 0
    0],...
98     'k','facealpha',0.2)
99 fill([X1(1) X1(end) X1(end) X1(1)],[5 5 X2(end) X2(end)
    ],...
100     'k','facealpha',0.2)
101 hold off
102
103 hold on
104 plot(x(1,:),x(2,:),'Linewidth',1.5)
105 hold off
106 %==========Convergence of the point==========%
107 k1=[0:k];
108 figure(2)
109 hold on
```

```
110 plot(k1,x(1,:),'Linewidth',1.5)
111 plot(k1,x(2,:),'Linewidth',1.5)
112 hold off
```

References

Nocedal, J., Wright, S. J. (2000). *Numerical optimization.* New York: Springer.

Ploskas, N., Samaras, N. (2017). *Linear programming using Matlab.* Cham, Switzerland: Springer.

Jørgensen, B. J. (2020). *Constrained Optimization* [PowerPoint slides]. Retrieved from https://www.inside.dtu.dk/da/undervisning