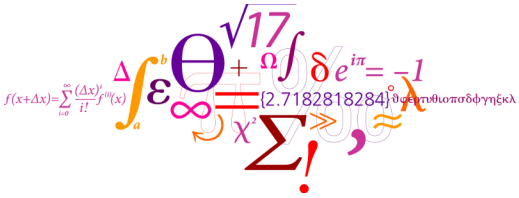
DTU Technical University of Denmark

Exam Assignment

- 02612 Constrained Optimization 2020 -



Kgs. Lyngby, 2020

Author:

Bjørn Hansen, s193035

Collaborator:

Shuai Wang, s200108

Contents

1 Equality Constrained Convex QP

This section considers the following equality constrained convex QP:

$$\begin{aligned} \underset{min\, x}{\phi} &= \frac{1}{2}x'Hx + g'x \\ s.t.\, A'x &= b \\ where,\, H &> 0 \end{aligned} \tag{1}$$

1.1 1. Lagrangian

Question: *What is the Lagrangian function for this problem?*

The Lagrangian for the QP stated above as equation 1 in matrix form is:

$$F(x,\lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ c(x) \end{bmatrix} = \begin{bmatrix} \nabla f(x) - \nabla c(x)\lambda \\ c(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{2}$$

The above equation with inputs from equation 1 gives:

$$F(x,\lambda) = \begin{bmatrix} Hx + g - A\lambda \\ A'x - b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3}$$

This can be simplified into the desired system of equations to solve for x and λ as follows:

$$F(x,\lambda) = \begin{bmatrix} H & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{4}$$

1.2 First order necessary optimality conditions

Question: *What is the first order necessary optimality conditions for this problem? Are they also sufficient and why?*
The first order optimality conditions is that the program is convex. Since equation 1 is a convex QP the condition is necessary but also sufficient. Equation 16 is the KKT-matrix and solving it will yield the minimum values that we want.

1.3 3. Implement solvers for solution of the problem

Question: *Implement solvers for solution of the problem (1.1) that are based on an LU-factorization (dense), LU-factorization (sparse), LDL-factorization (dense), LDL- factorization (sparse), a range-space factorization, and a null-space factorization. You must provide pseudo-code and source code for your implementation. The solvers for the individual factorizatons must have the interface [x,lambda]=EqualityQPSolverXX(H,g,A,b) where XX can be e.q. LUdense, LUsparse, etc. You must make a system that can switch between the different solvers as well. It should have an interface like [x,lambda]=EqualityQPSolver(H,g,A,b,solver), where solver is a flag used to switch between the different factorizations.*

To make sure each solver receives the same information a function which constructs a KKT system of equations similar to equation 16 above. The function was developed based on problem 1 (recycled system) presented in the exercises from week five. The inputs to this KKT constructing function are n, $\overline{u}\, and\, \mathrm{d}_0$. The system of equations is then given as inputs for the solvers to solve.

1.3.1 Pseudo-code for LU-factorization (dense/ sparse)

1) Construct system of equations (left hand side, right hand side).

2) Perform LU matrix factorization so that the left hand side of the system of equations is factorized into [L, U].

3) Solve the system by first solving $L[z] = C$, (where C is the right hand side of the system) and then $U[x, \lambda] = z$ to find the given x and λ.

Note: The sparse LU- factorization works in the same manner as above, the only difference is it utilizes the MATLAB sparse storage organization. The sparse function is applied to the left hand side of the system before step 2), this in essence converts a full matrix into a sparse one by eliminating any zero elements. The sparse matrix however still holds the same information as the original full matrix but is less computationally taxing to store.

1.3.2 Pseudo-code for LDL-factorization (dense/ sparse)

1) Construct system of equations (left hand side, right hand side).

2) Perform LDL factorization so that the left hand side of the system is factorized into [L, D, L'].

3) Solve the system by first solving $DL[y] = C$ (where C is the right hand side of the system), next solve for $D[z] = y$, lastly $L'[x, \lambda] = z$ is solved to find x and λ.

Note: The sparse LDL-factorization works in the same manner as above, the only difference is it utilizes the MATLAB sparse storage organization. The sparse function is applied to the left hand side of the system before step 2), this in essence converts a full matrix into a sparse one by eliminating any zero elements. The sparse matrix however still holds the same information as the original full matrix but is less computationally taxing to store.

1.3.3 Pseudo-code for Range-Space method/ Schur-Complement method

1) Find the [H,g,A,b] elements of the system.

2) Cholesky factorize H into $[LL']$.

3) Solve for v: H[v] = g.

4) Form and Cholesky factorize $H_a = A'H^{-1}A$ into L_aL_a'.

5) Solve for x and λ via solving the following: $H_a[\lambda] = b + A'v$ and $H[x] = A\lambda - g$.

1.3.4 Pseudo-code for Null-Space method

1) Find the [H,g,A,b] elements of the system as well as define the non-singular matrix [Y Z].

2) Solve $(A'Y)x_Y = b$.

3) Solve $(Z'HZ)x_Z = -Z'(HYx_Y + g)$.

4) Compute $x = Yx_Y + Zx_Z$.

5) solve for lambda $(A'Y)'[\lambda] = Y'(Hz + g)$.

1.4 4. Testing solvers

The above mentioned solvers are tested in this section. Their solutions are presented below as MATLAB syntax. The solvers are implemented as their own functions which can be called upon. A function that can call on any of the solvers via a flag argument was also implemented. The flag argument is called "solver" and accepts the following as input: "LDL", "LU", "NullSpace", "RangeSpace", "SparseLDL", "SparseLU".

Calling individual solvers looks as follows:

```
>> EqualityQPSolverLDL(5,0.2,1)

ans =

    2.2000
    2.2000
    2.2000
    2.2000
    1.2000
    1.0000

>> EqualityQPSolverSparseLU(5,0.2,1)

ans =

    2.2000
    2.2000
    2.2000
    2.2000
    1.2000
    1.0000
```

Calling upon the function that can switch between all solvers looks as follows.

```
>> EqualityQPSolverMaster(5,0.2,1,"RangeSpace")

ans =

    2.2000
    2.2000
    2.2000
    2.2000
    1.2000
    1.0000

>> EqualityQPSolverMaster(5,0.2,1,"NullSpace")

ans =

    2.2000
    2.2000
    2.2000
    2.2000
    1.2000
    1.0000
```

To test how the performance of each solver changes over varying input sizes, the CPU-time is measured and plotted as function of problem size as seen in figure 1. As seen in the plot, the sparse LU and LDL solvers are the least affected by an increase in input size, where as the range-space and null-space are heavily affected by the increase in input size. This makes sense because the LU and LDL implementations have fewer equations to compute compared to the range-space and null-space implementations. This effectively makes the range-space and null-space implementations comparatively much more expensive to compute.

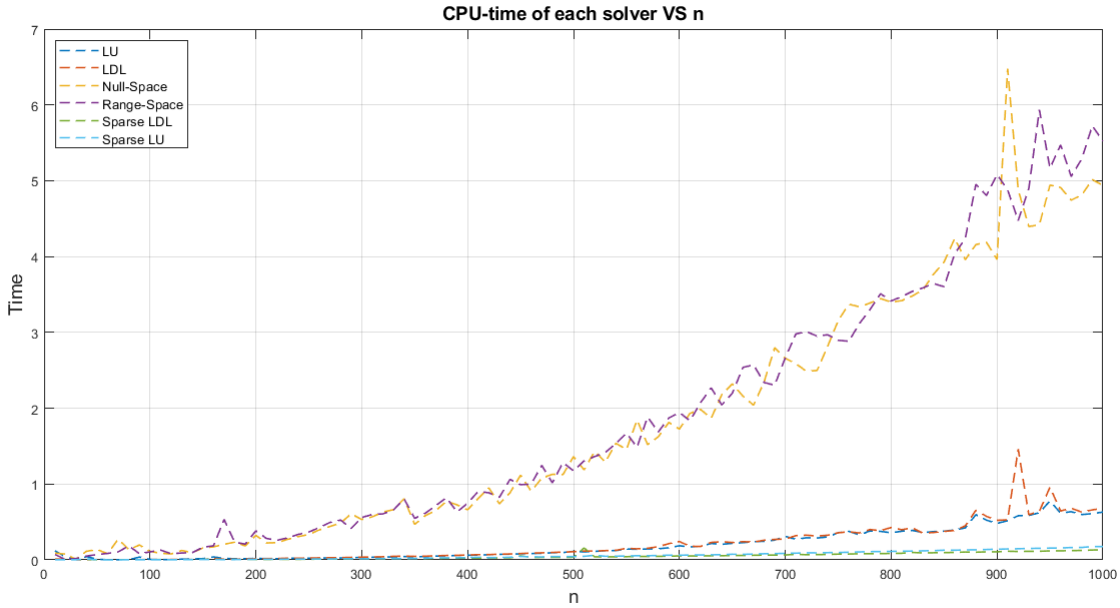


Figure 1: CPU-time for each solver as a function of input size.

2 Quadratic Program (QP)

In this section a quadratic program of the following form will be considered:

$$\begin{aligned} \min_{x} \quad & \phi = \frac{1}{2}x'Hx + g'x \\ s.t. \quad & A'x = b \\ where\ H \succ 0 & \end{aligned} \tag{5}$$

The problem can equivalently be formulated as such:

$$\begin{aligned} \min_{x} \quad & f(x) = \frac{1}{2}x'Hx + g'x \\ s.t. \quad & c_i(x) = A'x - b = 0 \quad i \in \varepsilon\{1, \ldots, r\} \\ & c_i(x) = C'x - d \geq 0 \quad i \in I\{r+1, \ldots, m\} \\ where, & \\ & l \leq x \leq u \Rightarrow \begin{bmatrix} I \\ -I \end{bmatrix} x \geq \begin{bmatrix} l \\ -u \end{bmatrix} \\ & C' = \begin{bmatrix} I \\ -I \end{bmatrix} \\ & d = \begin{bmatrix} l \\ -u \end{bmatrix} \end{aligned} \tag{6}$$

2.1 Lagrangian

Question: What is the Lagrangian function for this problem.

As stated during the lecture of week six, the Lagrangian of a quadratic program looks as follows:

$$\begin{aligned} L(x, \lambda) &= f(x) - \sum_{i \in \varepsilon} y_i c_i(x) - \sum_{i \in I} z_i c_i(x) \\ &= \frac{1}{2}x'Hx + g'x - y'(A'x - b) - z'(C'x - d) \end{aligned} \tag{7}$$

2.2 Necessary and sufficient optimality conditions

Question: Write the necessary and sufficient optimality conditions for this problem.

The optimality conditions for a quadratic problem involve introducing slack variables s so that the inequality constraints can be transformed to equality. This is done so that the quadratic problem can be represented and solved as a linear system of equations. The mathematical notation is presented below:

$$\begin{aligned} s \triangleq C'x - d \geq 0 \\ \Leftrightarrow \\ -C'x + s + d = 0 \\ s \geq 0 \end{aligned} \tag{8}$$

$$S = \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_{m_c} \end{bmatrix} \quad Z = \begin{bmatrix} z_1 & & & \\ & z_2 & & \\ & & \ddots & \\ & & & z_{m_c} \end{bmatrix} \quad z = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

The sizes of S, Z *and* z are chosen such that,

$$SZe = 0$$

The linear system which results from the optimality conditions can be expressed as:

$$F(x, y, z, s) = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} Hx + g - Ay - Cz \\ -A'x + b \\ -C'x + s + d \\ SZe \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{9a}$$

$$(z, s) \geq 0 \tag{9b}$$

2.3 Pseudo code for a primal-dual interior-point algorithm

Question: Write pseudo-code for a primal-dual interior-point algorithm which solves the problem. Explain each major step in your algorithm.

Steps in pseudo code:

1) Set starting values for: (H, g, C, d, A, b, x, y, z, s) where, x, y, z, s are initial guesses.

2) Compute residuals: r_L, r_A, r_C, r_{sz} and duality gap μ.

While: Euclidean length of $r_L, r_A, r_C, r_{sz}, \mu \geq \epsilon$ and $\Sigma\, iteration \leq\ iteration_{max}$:

3) Calculate the affine step direction $\triangle x^{aff}, \triangle z^{aff}$ and $\triangle s^{aff}$

4) Calculate the affine step size α^{aff}

5) Calculate the affine duality gap μ^{aff}

6) Calculate centering parameter σ

7) Calculate affine-centering-correction direction

8) Update x, y, z and s by taking the actual step $*\alpha * \eta$

9) Add new x into history

10) Updata $r_L, r_A, r_C, r_{sz}, \mu$

End While

Return: $(x_r, y_r, z_r, s_r, iter, his)$

Where,
x_r: the optimal solution,
y_r: Lagrangian coefficient of equality constraints,
z_r: Lagrangian coefficient of inequality constraints,
s_r: finally slack variables of inequality constraints,
iter: the loop number,
his: result x in each step.

2.4 Implementation of the primal-dual interior-point algorithm

Question: Implement the primal-dual interior-point algorithm and test it.

The inter-point algorithm discussed above was implemented and tested on the optimization of a portfolio where return is equal to two. The formulation of the problem is shown below.

$$H = 2\Sigma = 2 * \begin{bmatrix} 2.30 & 0.93 & 0.62 & 0.74 & -0.23 \\ 0.93 & 1.40 & 0.22 & 0.56 & 0.26 \\ 0.62 & 0.22 & 1.80 & 0.78 & -0.27 \\ 0.74 & 0.56 & 0.78 & 3.40 & -0.56 \\ -0.23 & 0.26 & -0.27 & -0.56 & 2.60 \end{bmatrix} \tag{10a}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} g = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} A = \begin{bmatrix} 1 & 15.1 \\ 1 & 12.5 \\ 1 & 14.7 \\ 1 & 9.02 \\ 1 & 17.68 \end{bmatrix} d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10b}$$

$$b = \begin{bmatrix} 1 \\ 2 \end{bmatrix} x = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} y = \begin{bmatrix} 1 \\ 1 \end{bmatrix} z = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} s = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{10c}$$

From testing the algorithm is seen that the optimal solution for a return of R = 2 is found after 102 iterations. The optimal solution found is:

$$x_{optimal} = \begin{bmatrix} 0.0009 & 0.2458 & 0.0218 & 0.8138 & 0.2650 \end{bmatrix}' \tag{11}$$

The compute time of this problem is 0.1638 time units. For comparison this same problem was tested with MATLAB's quadprog function which solved the problem about 80 times faster using 0.0023 time units.

2.5 Pseudo code for an active set algorithm

Question: Write pseudo-code for an active set algorithm which solves the problem. Explain each major step in your algorithm.

Step in pseudo code:
1) Set inputs and starting values: $(H, g, A, b, xtilde)$ where, xtilde is starting guess of x

2) Calculate a feasible starting point $x0$, set $W_0 \leftarrow W_0 \subset A(x_0)$, Initialize $k = 0$

While True:

Find p_k
if $p_k = 0$:
    if $\lambda_i \geq 0, \forall i \in W_k \cap I$
        Optimal solution is $x_* = x_k$
        Break
    Else:
        $W_{k+1} = W_k \setminus \arg\min \{\lambda_i, \forall i \in W_k \cap I\}$
        $x_{k+1} = x_k$
        Break
    Else:
        Calculate α_k

$x_{k+1} = x_k + \alpha_k p_k$
If $\alpha_k < 1$:
add one of the blocking constraints to W_k
$W_{k+1} = W_k \cup i$
Else:
$W_{k+1} = W_k$
End
End
End
Return: (x, λ)

2.6 Implementation of the active set algorithm

Question: Implement the active set algorithm and test it.

To test the implementation of the active set algorithm the following example was tested:

$$\begin{aligned} \min_x \quad & q(x) = (x_1 - 1)^2 + (x_2 - 2.5)^2 && \text{(12a)} \\ s.t. \quad & x_1 - 2x_2 + 2 \geq 0 && \text{(12b)} \\ & -x_1 - 2x_2 + 6 \geq 0 && \text{(12c)} \\ & -x_1 + 2x_2 + 2 \geq 0 && \text{(12d)} \\ & x_1 \geq 0 && \text{(12e)} \\ & x_2 \geq 0 && \text{(12f)} \end{aligned}$$

Equation 12 can be formulated as quadratic program inputs as shown below:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} g = \begin{bmatrix} -2 \\ -5 \end{bmatrix} A = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 \\ -2 & -2 & 2 & 0 & 1 \end{bmatrix} b = \begin{bmatrix} -2 \\ -6 \\ -2 \\ 0 \\ 0 \end{bmatrix} x_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \tag{13}$$

The optimal solution was found after five iterations, the total compute time for the problem was 0.00034 time units. The optimal solution is shown below.

$$x_{optimal} = \begin{bmatrix} 1.4 & 1.7 \end{bmatrix}' \tag{14}$$

2.7 Comparison of algorithms

Question: Compare the performance of your primal-dual interior-point algorithm, active-set algorithm, and quadprog from Matlab

Both the primal-dual interior-point algorithm and active-set algorithm were compared with quadprog for their respective test problems. The compute time comparison with quadprog are shown below:

Table 1: CPU usage when $R = 10, \sum x_i = 1$

interior-point	quadprog
9.390e-04	0.0034

Table 2: CPU usage when $R \geq 10, \sum x_i \leq 1$

active-set	quadprog
2.456e-04	0.0044

2.8 Markowitz portfolio optimization problem

Demonstrate that the Markowitz' portfolio optimization problem can be expressed as a QP. Test the primal-dual interior-point QP algorithm, the active-set QP algorithm, and quadprog on the Markowitz problem.

The Markowitz optimzation problem has already been described and tested above so this section will focus mainly on the results. A more in depth description can be found in the next section.

A comparison of the methods are shown below in the form of an efficient frontier seen in Figure 2 and a results table seen in Table 3. As can be seen, both methods give equal results.

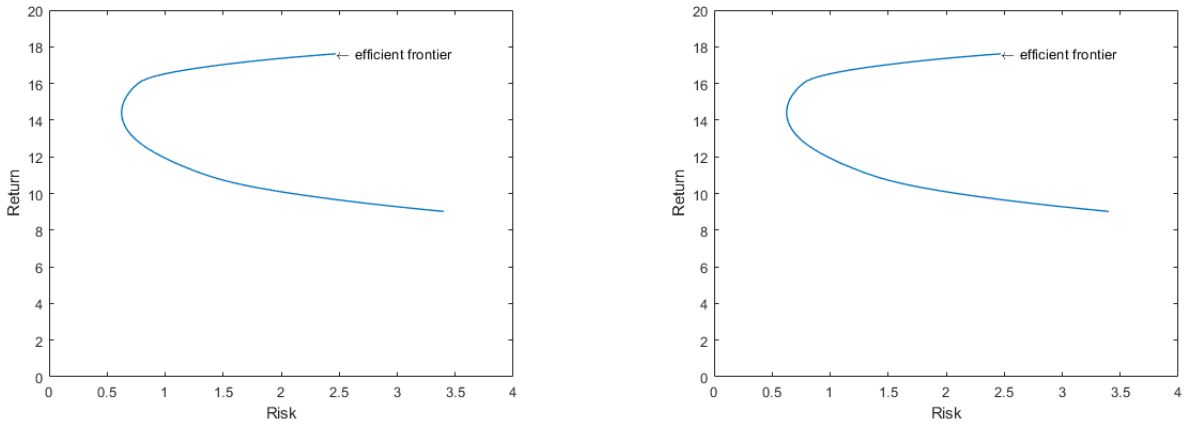


Figure 2: Efficient frontier using quadprog (left), efficient frontier using interior-point (right)

Table 3: Comparison of results between quadprog and active set.

	quadprog	Active set
Opt. solution	0.1042, 0.1022, 0.2067, 0.0249, 0.2198	0.1042, 0.1022, 0.2067, 0.0249, 0.2198
Iterations	5	4
Compute time	0.0044	2.417e-04

3 Markowitz Portfolio Optimization

Question 1 (without risk free security):
1. For a given return, R, formulate Markowitz' Portfolio optimization problem as a quadratic program.
2. What is the minimal and maximal possible return in this financial market?
3. Compute a portfolio with return, R = 10.0, and minimal risk. What is the optimal portfolio and what is the risk (variance)?
4. Compute the efficient frontier, i.e. the risk as function of the return. Plot the efficient frontier as well as the optimal portfolio as function of return.

Question 2 (including risk free security):
In the following we add a risk free security to the financial market. It has return rf = 2:0.
1. What is the new covariance matrix and return vector.
2. Compute the efficient frontier, plot it as well as the (return,risk) coordinates of all the securities. Comment on the effect of a risk free security. Plot the optimal portfolio as function of return.
3. What is the minimal risk and optimal portfolio giving a return of R = 15:00. Plot this point in your optimal portfolio as function of return as well as on the efficient frontier diagram.

3.1 Without risk free security

3.1.1 Formulation of Markowitz' Portfolio optimization problem

For this problem we will be working with the following financial market consisting five securities and the estimated return presented in the table below.

Table 4: Security covariance matrix and return (financial market).

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

We want to formulate the problem in terms of a constrained QP $\min_x \frac{1}{2}x'Hx$, where the hessian in this case is the covariance matrix between securities and x is a vector of length five which represents the proportion funds allocated into each security. The overall goal is to minimize the variance in the portfolio whilst being subject to aproduucing a return equal or higher than the chosen desired return. The completely defined problem including its constraints looks as follows:

$$\begin{aligned} \min_x \frac{1}{2}x'Hx \\ s.t. \\ Ax = b \\ Ix \geq \overline{0s} \end{aligned} \tag{15}$$

To clarify the constraints, A is a vector containing μ which is the return expected followed by a vector of ones, b is a vector containing R which defines the wanted return. Constraint $Ax = b$ ensures the allocation of funds must be equal to the defined return column in table Table 4 as well as ensuring that the total allocation must be equal to a total of one, i.e. 100%. Constraint Ix $\geq \overline{0s}$ ensures that the allocation amount can not be negative, where I is an identity matrix of size five and $\overline{0s}$ is a vector of zeros.

The individual terms discussed above are explicitly defined as follows:

$$x = [x_1\ x_2\ x_3\ x_4\ x_5]$$

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

$$A = [\mu' \; \overline{1s}]$$

$$b = [R]$$

$$\mu = [15.10 \; 12.50 \; 14.70 \; 9.02 \; 17.68]$$

3.1.2 Minimal and Maximal return in financial market

In this section only a strict financial market as described by Table 4 is considered. Without any specific financial trends such as market drops, interest rates etc. the minimal return should be if all fund were allocated into the security giving the lowest return. This would account to allocating all funds into security four, giving a return of 9.02. The same principle would apply for the maximal return and would account to allocating all funds into security five, giving a return of 17.68.

3.1.3 Optimal portfolio with return, R = 10.0

The optimal portfolio was calculated and shown in the figure Figure 3 below. This was the optimal portfolio calculated with the help of the quadprog package in MATLAB. The exact allocation is 28.16% for security two and 71.84% for security four. The variance of this portfolio is 1.046.

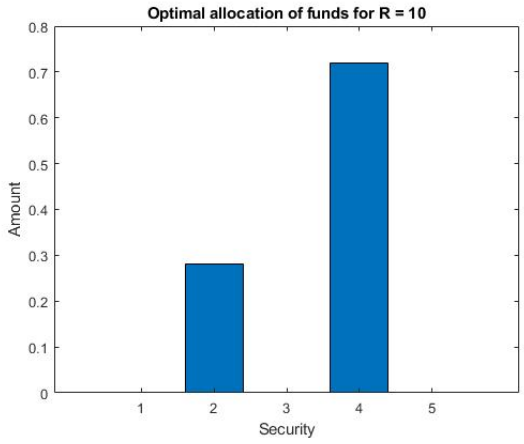


Figure 3: Optimal portfolio composition for R = 10.

3.1.4 Efficient frontier and optimal portfolio for given financial market

The efficient frontier is presented in Figure 4 to the left below. The efficient frontier shows that the optimal portfolio is achieved for a return of approximately 14.4. This is the portfolio with the highest return/variance ratio. The

composition of the this optimal portfolio is relatively balanced and allocates funds to all securities. The optimal portfolio is shown to the right below in Figure 4. The exact allocation of funds for this portfolio is as follows: 8.71% to security one, 25.24% to security two, 28.19% to security three, 10.48% to security four and 27.38% to security five. The total variance of the optimal portfolio is 0.312.

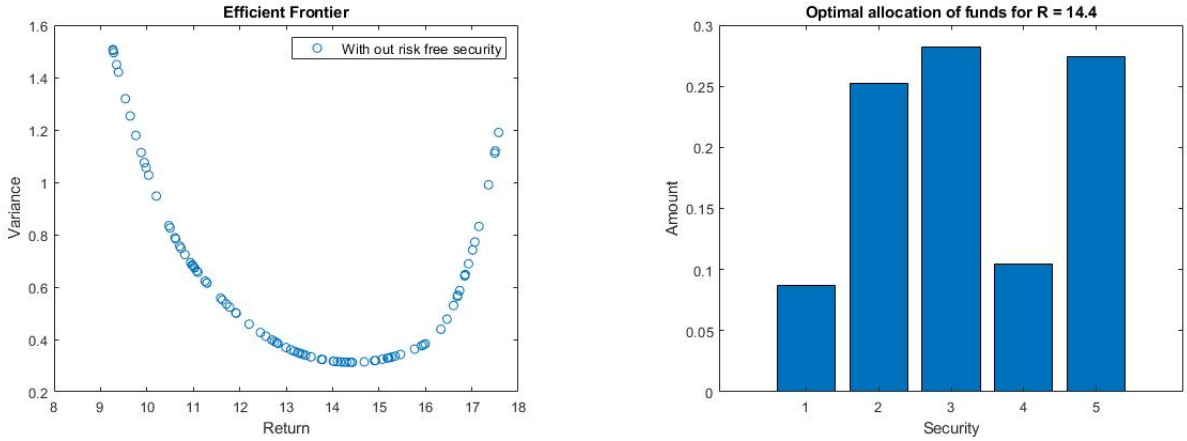


Figure 4: Efficient frontier of given financial market (left) and Optimal portfolio composition for R = 14.4 (right)

3.2 With risk free security

3.2.1 New financial market description

The financial market being dealt with on this section is the same as presented before but now also includes one risk free security with a return of 2.0. This means that this new security will be included in the covariance matrix as zeros in all pairing positions, because is has zero covariance with all other securities in the financial market. The new covariance matrix and expected return vector including the risk free security is shown below.

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

$$\mu = [15.10\ 12.50\ 14.70\ 9.02\ 17.68\ 2]$$

3.2.2 Efficient frontier and optimal portfolio for new financial market

The efficient frontier including the risk free security is shown in Figure 5 below and shows that the new financial market has changed the shape of the frontier quite drastically. Both lines converge and rise similarly but are otherwise different. With a risk free security the efficient frontier is now linear until a return of approximately 16 where variance grows much faster than return. A well known way to calculate how optimal a portfolio is, is by using the so called Shapes ratio (1). The higher this ratio is the better the portfolio is considered to be. As was mentioned before, the new efficient frontier is linear meaning that any arbitrary portfolio giving a return smaller than about 16 will have an equal Shapes ratio value. Thus the *true* optimal portfolio depends on the amount of risk one is willing to take. For the sake of example the portfolio with a return of 9 is analyzed. As can be seen, the composition of this portfolio is heavily reliant on the risk free security. Funds allocated in this portfolio are distributed as follows: 7.35% to security one, 16.77% to security two, 1.43% to security three, 18.05% to security

four and 47.48% to security five. This portfolio has a variance of 0.092 meaning it is a relatively safe portfolio option.

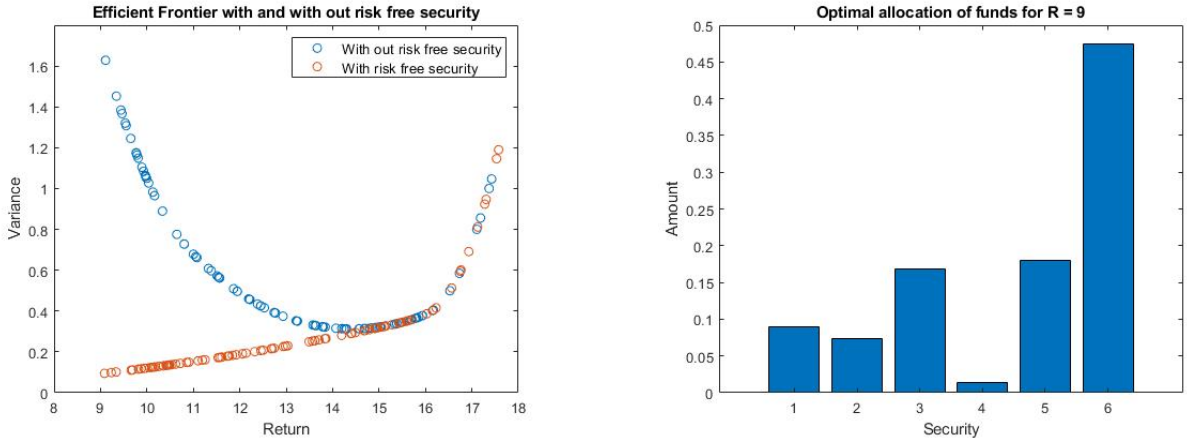


Figure 5: Efficient frontier of new financial market (left) and Optimal portfolio composition for R = 9.0 (right)

3.2.3 Optimal portfolio giving return of R = 15.0

The same efficient frontier presented previously now with a marker over a return of 15 is shown in Figure 6. The optimal portfolio with a return of 15.0 has a find distribution as follows: 16.55% to security one, 31.15% to security two, 2.66% to security three, 33.52% to security four and 2.47% to security five. This portfolio has a variance of 0.319.

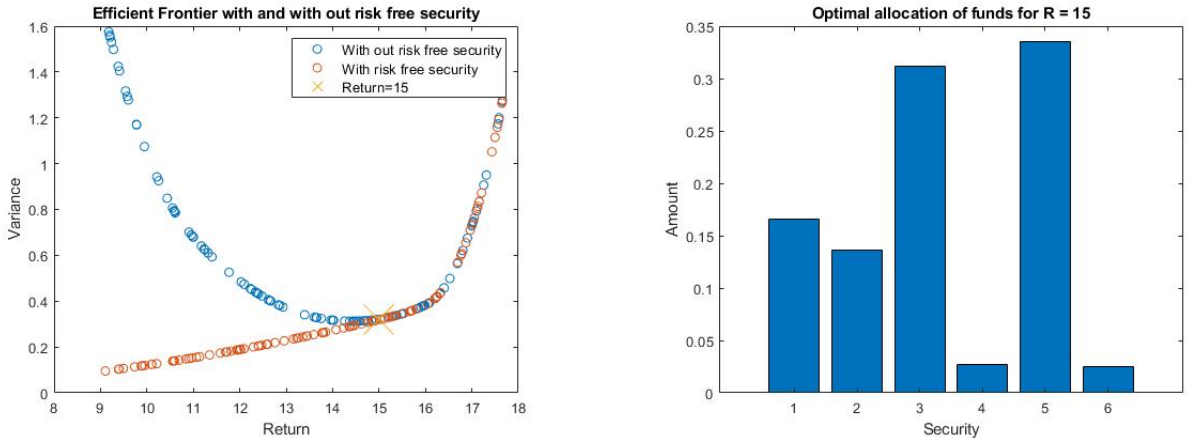


Figure 6: Efficient frontier of new financial market (left) and Optimal portfolio composition for R = 15.0 (right)

4 Linear Program (LP)

This section considers the a linear program in the following form:

$$\begin{aligned} \underset{min\, x}{\phi} &= g'x \\ s.t.\ A'x &= b \\ where,\ l \leqslant x &\leqslant u \end{aligned} \tag{16}$$

4.1 Lagrangian

Question: *What is the Lagrangian function for the linear problem?*

During week seven linear programs was discussed, according to that lecture the Lagrangian for the standard form LP stated above as Equation 16 is as follows:

$$\mathcal{L}(x, \mu, \lambda) = g'x - \mu'(Ax - b) - \lambda'x \tag{17}$$

4.2 Necessary and sufficient optimality conditions

Question: *What are the necessary and sufficient optimality conditions for the problem?*

The necessary and sufficient optimal conditions look as follows:

$$\begin{aligned} \nabla_x \mathcal{L}(x, \mu, \lambda) = g - A'\mu - \mu = 0, \\ Ax = b \\ \Leftrightarrow \begin{bmatrix} 0 & A' \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \mu \end{bmatrix} = \begin{bmatrix} g - \mu \\ b \end{bmatrix} \\ s.t.\ x \geqslant 0 \perp \lambda \geqslant 0 \end{aligned} \tag{18}$$

4.3 Pseudo code for primal-dual interior-point

Question: *Write pseudo-code for a primal-dual interior-point algorithm which solves the LP problem.*

The main resources for the primal-dual interior point algorithm are the week seven materials and chapter 14 in the book Numerical Optimization (2). The pseudo code over the main steps of the algorithm look as follows:

1) Set g, A, b, x, λ, μ
2) Compute residuals: r_L, r_A
3) Activate/ Deactivate STOP condition: set convergence values to be fulfilled
4) While STOP do:
5) Compute hessian matrix H and Cholesky factorize
6) Compute affine step: $\Delta\mu, \Delta x, \Delta\lambda, \beta, \alpha$
7) Compute centering parameter and duality gap
8) Compute center corrector step: $\Delta\mu, \Delta x, \Delta\lambda, \beta, \alpha$
9) Update values for iteration
10) Give forth updated x'
11) Check if convergence values are fulfilled, if not continue to next iteration
12) end While

4.4 Implementation of primal-dual interior-point

Question: *Implement the primal-dual interior-point algorithm and test it.*

The problem chosen to test the linear programs on will a maximization problem with two outputs x_1, x_2 as well as two constraints. The thought is that the x_1, x_2 are amounts of two different products (both have different value) with the constraints being the amounts and cost of the materials that must be used produce each of these two products. The problem is then to maximize profit by producing the appropriate number of each product. The mathematical notation of this problem will be:

$$\begin{aligned} \underset{min\, x}{\phi} &= -3x_1 - 5x_2 \\ &s.t.: \\ 3x_1 + x_2 &\leqslant 7, \\ x_1 + 2x_2 &\leqslant 9 \\ for\ x &\geqslant 0 \\ &\Leftrightarrow \\ \underset{min\, x}{\phi} &= \underset{x}{min} \begin{bmatrix} -3 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &s.t.: \\ \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} &\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \end{bmatrix} \end{aligned} \tag{19}$$

Equation 19 is solved with the syntax and produces the output as shown below. The algorithm has come to the optimal solution that a single product x_1 should be produced for every fourth product x_2. The performance of the algorithm was also measured using the MATLAB timeit function, which showed a compute time of $1.7997 \cdot 10^{-4}$ time units.

```
>> TestLPippdsolver

iter =

     7


xlp =

    1.0000
    4.0000


cpu_time =

   1.7997e-04
```

Equation 19 was also solved using the MATLAB library linprog (using the interior-point method under options). The output below shows that the same optimal solution was found but was slower than the self interior-point implementation, as linprog took $45 \cdot 10^{-4}$ time units to compute. It should be noted however that linprog used five iterations vs seven of the self implemented interior-point algorithm.

```
>> Test_problem_4_linprog

Optimal solution found.


x =

    1.0000
    4.0000
```

```
fval =

   -23


exitflag =

     1


cpu_time =

    0.0045
```

4.5 Pseudo code for simplex algorithm

Question: *Write pseudo-code for a simplex algorithm which solves the LP problem.*
The main resourced used for researching the simplex algorithm have been (2) (3) (4). The implementation of the simplex method utilizes the tableau format. The algorithm has the optimal solution when the associated basis obtains two features simultaneously namely, primal feasibility and dual feasibility. The algorithm must however start by fully fulfilling one of these features and striving to obtain the other iteratively. It does not matter which feature, primal or dual feasibility is started with as both methods should end at the same optimal solution. Primal feasibility is obtained for the basis of the tableau when all elements on the right hand side of the tableau are non negative, otherwise the basis is considered primal infeasible. Dual feasibility of the basis is obtained when all elements in row 0 are non negative, otherwise the basis is considered dual infeasible. Pseudo code for the algorithm looks as follows:

1) Start with a dual feasible basis. Create a tableau for the basis in simplex format.
2) Check if right hand side elements are all non negative, if they are stop. (optimal solution found)
3) Start iterations, select leaving variable: select a row with the most negative right hand side element. This row will be the current pivoting row.
4) Select entering variable: For each negative coefficient in the pivot row, compute the negative of the ratio between the reduced cost in row 0 and the structural coefficient in the current pivoting row. If there is no negative coefficient then STOP as there no feasible solution.
5) The column which has the minimum ratio will be the pivoting column. The element where pivoting row and pivoting column intersect will be the entering variable.
6) Change basis: Replace the leaving variable with the entering variable. Create new tableau by performing Gaussian elimination on the entire tableau such that the pivot column will have a single value of 1 where it intersects with the pivot row and zeros elsewhere.
7) Check if optimal solution has been obtained by checking if all right hand side elements are non negative (primal feasible), if not iterate again.

4.6 Implementation of simplex algorithm

Question: *Implement the simplex algorithm and test it.*

The above described tableau simplex method was used to solve Equation 19, the optimal solution, final tableau and compute time is shown below. As can be seen, once again the solution is the same as for the previously mentioned methods (interior-point, linprog). The compute time is however the smallest of all the previously tested methods with $0.94118 \cdot 10^{-4}$ time units.

```
>> TestLPsimplex

t =
```

```
    1.0000         0    1.0000
         0    1.0000    4.0000
         0         0   23.0000


x =

    1.0000
    4.0000


cpu_time =

   9.4118e-05
```

4.7 Comparison of implementations

Question: *Compare the performance of your primal-dual interior-point algorithm, simplex algorithm), and linprog from Matlab.*

It was shown in the previous sections that all methods produce the same optimal solution of $x = [1, 4]$. However all methods had various compute times as shown in Table 5 with the simplex being the fastest computationally and lingprog being the slowest. Linprog is most likely the slowest as it is by far the most robust method of all. It can take in various types of inputs and has multiple options to choose from. It is also able to solve a broader range of problems than the self implemented inter-point and simplex algorithms are able to.

Table 5: Compute time comparison of Linear Program solving methods.

Interior-point	Simplex	MATLAB linprog
$1.79 \cdot 10^{-4}$	$0.94 \cdot 10^{-4}$	$45 \cdot 10^{-4}$

4.8 Special Markowitz portfolio optimization problem

Question: *Test on a special Markowitz portfolio optimization problem where we do not care about risk but just want to maximize the return. Formulate this Markowitz portfolio optimization problem and test your algorithms. Discuss your tests and the results.*

The returns vector as presented in Table 4 will be considered and denoted as x. Because no risk is being accounted for and the only aim is to maximize return the mathematical formulation of the problem will look as follows:

$$\begin{aligned} \underset{min\, x}{\phi} &= -15.1x_1 - 12.5x_2 - 14.7x_3 - 9.02x_4 - 17.68x_5 \\ &s.t.: \\ x_1 + x_2 + x_3 + x_4 + x_5 &\leqslant 1, \\ for\ x &\geqslant 0 \end{aligned} \tag{20}$$

The outputs of each three methods, primal-dual interior-point, simplex and linprog is presented below.

Primal-dual interior-point:

```
>> TestLPippdsolver

iter =

     7
```

```
xlp =

    0.0000
    0.0000
    0.0000
    0.0000
    1.0000


cpu_time =

   3.0037e-04
```

Simplex:

```
>> TestLPsimplex

t =

    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    2.5800    5.1800    2.9800    8.6600         0   17.6800


x =

     0
     0
     0
     0
     1


cpu_time =

   7.2468e-05
```

linprog:

```
>> Test_problem_4_linprog

Optimal solution found.


x =

    0.0000
    0.0000
    0.0000
    0.0000
    1.0000


fval =

   -17.6800
```

```
exitflag =

     1


cpu_time =

    0.0045
```

It is seen that all methods have again come to the same optimal solution of $x = [0, 0, 0, 0, 1]$, which would mean allocating all funds to security five. This makes sense because the goal is to maximize the return and as is seen in Table 4, security five has the largest expected return. Considering computation time it is again seen that the simplex algorithm has the fastest compute time of all implemented methods and linprog the slowest.

5 Nonlinear Program (NLP)

This section considers a nonlinear program in the form:

$$\begin{aligned} \min_{x} &= f(x) \\ s.t.\ & g(x) = 0 \\ where,\ & l \leqslant x \leqslant u \end{aligned} \tag{21}$$

5.1 Lagrangian, first and second order optimality conditions

Question: *What is the Lagrangian function, first order and second order optimality conditions for the nonlinear problem?*

The Lagrangian of Equation 21 looks as follows:

$$\mathcal{L}(x, \lambda) = f(x) - \lambda' g(x) \tag{22}$$

The fist order optimality conditions of the nonlinear program is the result of taking the gradient of the Lagrangian with respect to x and y equal to zero, this looks as follows:

$$\begin{aligned} F(x,y) = \nabla_x \mathcal{L}(x,y) = \nabla f(x) - \nabla g(x)\lambda = 0 \\ \nabla_\lambda \mathcal{L}(x,\lambda) = -g(x) = 0 \\ \Leftrightarrow F(x,\lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ \nabla_\lambda \mathcal{L}(x,\lambda) \end{bmatrix} \begin{bmatrix} \nabla f(x) - \nabla g(x)\lambda \\ -g(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{23}$$

The second order optimality conditions of the nonlinear program is the result of taking the derivative of the first order optimality conditions. This looks as follows:

$$\nabla F(x,\lambda) = \begin{bmatrix} \nabla_x F(x,\lambda) \\ \nabla_\lambda F(x,\lambda) \end{bmatrix} = \begin{bmatrix} \nabla^2_{xx} \mathcal{L}(x,\lambda) & -\nabla g(x) \\ -g(x)' & 0 \end{bmatrix} \tag{24}$$

5.2 Nonlinear problem formulation

Question: *Present the problem and argue why it was chosen.*

The nonlinear problem chosen to solve will be the Himmelblau function with one constraint. The mathematical formulation is the following:

$$\begin{aligned} \min_{x_1, x_2}\ & f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ s.t.\ & c(x_1, x_2) = (x_1 + 2)^2 - x_2 = 0 \end{aligned} \tag{25}$$

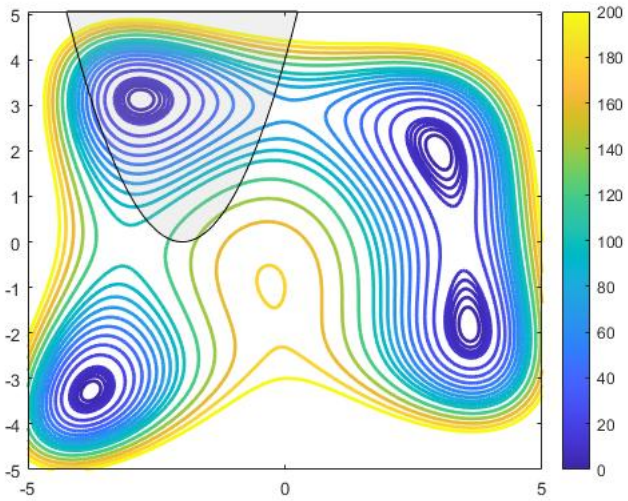


Figure 7: Contour plot of Himmelblau problem with constraint.

This problem was used as it is a well known benchmark function for testing algorithms as it presents various challenges such as multiple saddle points. It is also a multi nodal optimization problem as it has four minima (one taken out due to the constraint).

5.3 Solving problem with MATLAB's fmincon function

Question: *Solve the test problem using a library function for nonlinear programs in Matlab.*

The library function fmincon was used to solve the problem described in Equation 25. Different starting points returned different optimal solutions, which can be seen in Table 6 below. The syntax used to produce the fmincon results can be found in the appendix.

Table 6: Optimal solutions found with fmincon.

Starting point	Optimal solution
[-2, 3]	[-3.6546, 2.7377]
[0, 0]	[-0.2983, 2.8956]
[3, 3]	[-0.2983, 2.8956]
[-1, 4]	[-3.6546, 2.7377]

5.4 SQP with a damped BFGS approximation to the Hessian

Question: *Explain, discuss and implement SQP procedure with BFGS approximation to the Hessian matrix. Make a table with the iteration sequence and plot it in a contour plot.*

The optimization techniques used to solve non linear program are not the same as the techniques used to solve linear or quadratic problems as the non linear constraint does not comply with the standard KKT conditions. Non linear problems can however be solved by breaking the full problem up into a sequence of quadratic problems which are solved iteratively. To do this the Newtons method can be applied to the gradient Lagrangian function as stated in Equation 24. The goal of the Newton method is to find roots of a given function using initial guess inputs for the function and the functions derivative. This will give a continuously better approximation of the roots over multiple iterations. The roots which are of interest to find are the x *and* λ stated in the optimality conditions above, the iterative process to find them is shown below.

$$\begin{bmatrix} x_{iteration+1} \\ \lambda_{iteration+1} \end{bmatrix} = \begin{bmatrix} x_{iteration} \\ \lambda_{iteration} \end{bmatrix} + \begin{bmatrix} p_{iteration} \\ p_{iteration} \end{bmatrix}$$

Each sequence of the transformed nonlinear problem will be a quadratic problem which looks as follows:

$$\min_{\Delta x} \frac{1}{2}\Delta x[\nabla^2_{xx}\mathcal{L}(x,\lambda)]\Delta x + [\nabla_x \mathcal{L}(x,\lambda)]'\Delta x \\ s.t.\ \nabla g(x)'\Delta x = -g(x) \tag{26}$$

The above formulation is theoretically a possible way to solve a nonlinear problem, however there is a drawback in practice. That is that the Hessian can be difficult to obtain, a BFGS approximation can instead be used for the true Hessian. As is stated in Figure 8 which was taken from the week nine slides, B is the approximation of the Hessian matrix. To maintain the so called curvature condition $p'q > 0$ and positive definiteness of the approximated Hessian the scalar value theta is introduced. The iterative process shown below ensures positive eigenvalues and produces closer approximations of the Hessian matrix over iterations.

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

BFGS update

$$B \leftarrow B + \frac{qq'}{p'q} - \frac{(Bp)(Bp)'}{p'(Bp)}$$

Modified BFGS update

$$\theta = \begin{cases} 1 & p'q \geq 0.2p'(Bp) \\ \frac{0.8p'(Bp)}{p'(Bp) - p'q} & p'q < 0.2p'(Bp) \end{cases}$$
$$r = \theta q + (1-\theta)(Bp)$$
$$B \leftarrow B + \frac{rr'}{p'r} - \frac{(Bp)(Bp)'}{p'(Bp)}$$

Figure 8: Mathematical notation of the modified BFGS approximation.

MATLAB has been used to run the code which solves the problem described above and can be found in the appendix. The solutions obtained for four different starting points can be seen below in Figure 9 and Figure 10. The quickest convergence was seen after 13 iterations with a starting value of [3, 3].

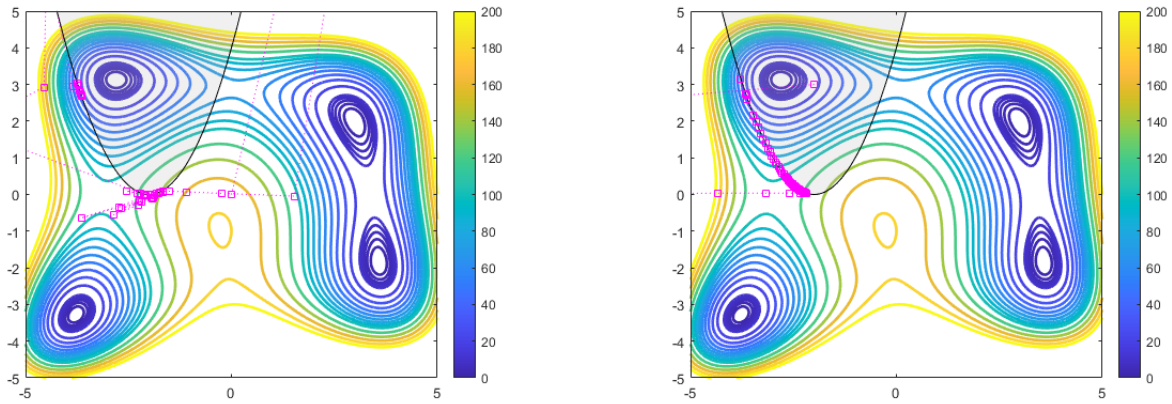


Figure 9: Contour plot of SQP BFGS solution: starting point: [0, 0], nr. iterations: 41. (left), starting point: [-2, 3], nr. iterations: 97. (right)

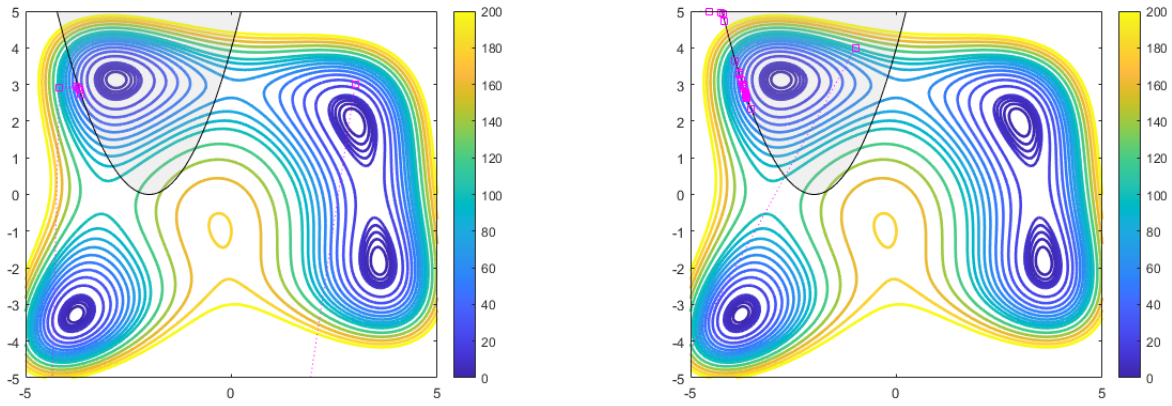


Figure 10: Contour plot of SQP BFGS solution: starting point: [3, 3], nr. iterations: 13 (left), starting point: [-1, 4], nr. iterations: 36. (right)

Table 7: Optimal solutions found with SQP BFGS.

Starting point	Optimal solution	Nr. iterations
[0, 0]	[-3.6546, 2.7376]	41
[-2, 3]	[-3.6546, 2.7377]	97
[3, 3]	[-3.6546, 2.7377]	13
[-1, 4]	[-3.6546, 2.7377]	36

5.5 SQP with a damped BFGS approximation to the Hessian and linesearch

Question: *Explain, discuss and implement SQP procedure with BFGS approximation to the Hessian matrix and line search. Make a table with the iteration sequence and plot it in a contour plot.*

The same type of BFGS approximation is considered in this section as was considered in the previous, although now a backtracking line search method is implemented to control step length at each iteration. The backtracking condition considered Goldstein-Armijo condition as discussed in (2), which is based on the l_1 merit function. The merit function (where μ is a penalty parameter) is defined as:

$$\phi(x, \mu) = f(x) + \mu||c(x)||_1$$

The direction of of the back tracking method towards the direction of $p_{iteration}$ is defined as:

$$Dir(\phi(x, \mu), p) = \nabla f(x)'p - \mu||c(x)||_1$$

The same four starting points as used before are tested on the SQP with BFGS approximation and line search, the solutions and iteration sequences are shown below in Figure 11 and Figure 12. The quickest convergence takes 41 iterations and is for a starting point of [0, 0]. It is seen that the addition of the line search method increases the required iterations in general for some starting points full convergence is never reached. It is however seen that by adding the line search method, different optimal results are achieved where as without line search as seen in Table 7, the same optimal solution is always produced.

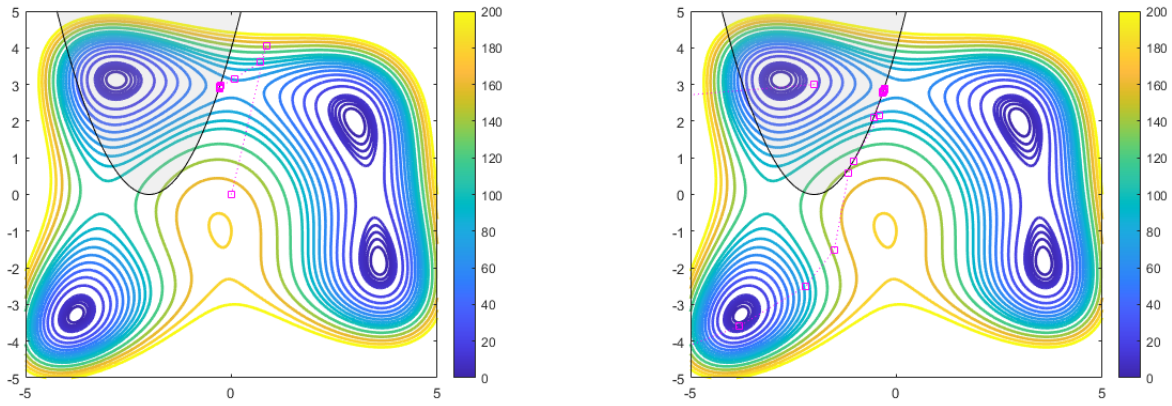


Figure 11: Contour plot of SQP BFGS with line search solution: starting point: [0, 0], nr. iterations: 41. (left), starting point: [-2, 3], nr. iterations: 97. (right)

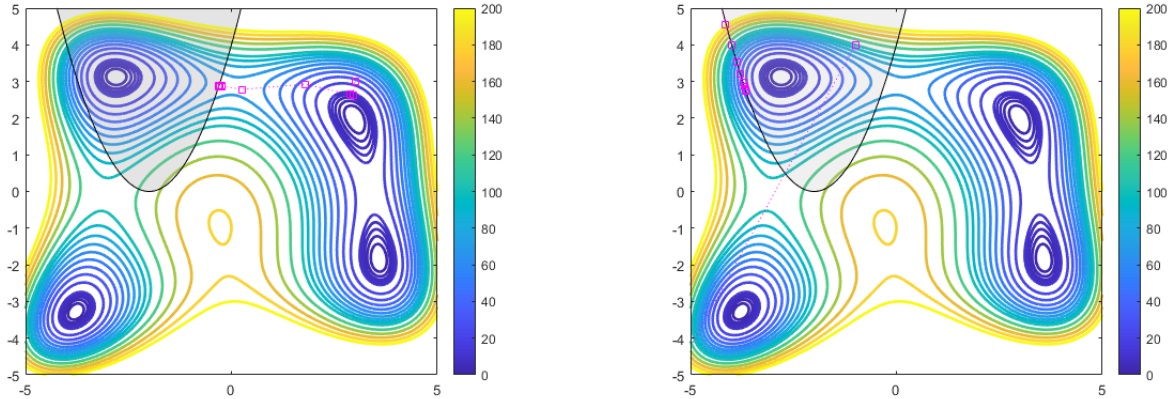


Figure 12: Contour plot of SQP BFGS solution: starting point: [3, 3], nr. iterations: 13 (left), starting point: [-1, 4], nr. iterations: 36. (right)

Table 8: Optimal solutions found with SQP BFGS.

Starting point	Optimal solution	Nr. iterations
[0, 0]	[-0.3290, 2.7921]	41
[-2, 3]	[-0.3290, 2.7921]	No Convergence
[3, 3]	[-0.2986, 2.8946]	64
[-1, 4]	[-3.6549, 2.7386]	70

5.6 Trust Region based SQP

Question: *Explain, discuss and implement SQP procedure fpr a Trust Region based algorithm. Make a table with the iteration sequence and plot it in a contour plot.*

The trust region algorithm which was followed is shown in Figure 13 and is algorithm 18.4 of chapter 18 in (2). The idea of the algorithm was to define a region around the current solution, calculate step by computing quadratic program. If the step minimized the objective function then the model is believed to be a good representation otherwise a new trust region must be determined. The trust region was used by setting boundaries to MATLABS quadprog function.

Choose constants $\epsilon > 0$ and $\eta, \gamma \in (0, 1)$;
Choose starting point x_0, initial trust region $\Delta_0 > 0$;
for $k = 0, 1, 2, \ldots$
    Compute $f_k, c_k, \nabla f_k, A_k$;
    Compute multiplier estimates $\hat{\lambda}_k$ by (18.21);
    if $\|\nabla f_k - A_k^T \hat{\lambda}_k\|_\infty < \epsilon$ **and** $\|c_k\|_\infty < \epsilon$
        stop with approximate solution x_k;
    Solve normal subproblem (18.45) for v_k and compute r_k from (18.46);
    Compute $\nabla^2_{xx} \mathcal{L}_k$ or a quasi-Newton approximation;
    Compute p_k by applying the projected CG method to (18.44);
    Choose μ_k to satisfy (18.35);
    Compute $\rho_k = \text{ared}_k / \text{pred}_k$;
    if $\rho_k > \eta$
        Set $x_{k+1} = x_k + p_k$;
        Choose Δ_{k+1} to satisfy $\Delta_{k+1} \geq \Delta_k$;
    else
        Set $x_{k+1} = x_k$;
        Choose Δ_{k+1} to satisfy $\Delta_{k+1} \leq \gamma \|p_k\|$;
end (for).

Figure 13: Trust Region algorithm pseudo code.

The implementation of the trust region algorithm was correct and the algorithm was nonfunctional. The implementation as seen in Figure 14 gave unsuccessful results as the algorithm did not iterate correctly and would get stuck after a few iterations independent of starting position.

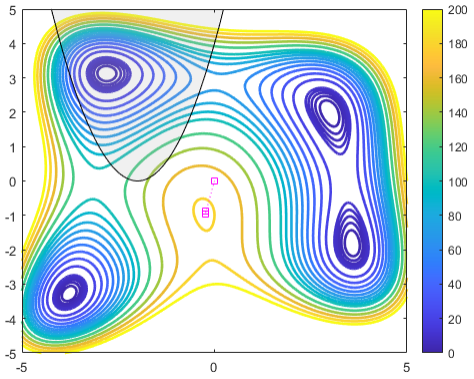


Figure 14: Trust Region iteration sequence, starting at [0, 0].

5.7 Interior- Point based Algorithm

Question: *Explain, discuss and implement a Interior- Point based algorithm. Make a table with the iteration sequence and plot it in a contour plot.*

A line search inter-point algorithm was attempted to be implemented. The line search algorithm followed was 19.2 in chapter from (2), which can be seen in Figure 15. The thought was to set up a quadratic programming problem which uses gradients of the objective to find the positive definite Hessian. From there initial values and multipliers would be chosen to compute primal-dual direction, alphas.

repeat until $E(x_k, s_k, y_k, z_k; 0) \leq \epsilon_{\text{TOL}}$
  repeat until $E(x_k, s_k, y_k, z_k; \mu) \leq \epsilon_\mu$
    Compute the primal-dual direction $p = (p_x, p_s, p_y, p_z)$ from
      (19.12), where the coefficient matrix is modified as in
      (19.25), if necessary;
    Compute $\alpha_s^{\max}$, $\alpha_z^{\max}$ using (19.9); Set $p_w = (p_x, p_s)$;
    Compute step lengths α_s, α_z satisfying both (19.27) and
    $\phi_\nu(x_k + \alpha_s p_x, s_k + \alpha_s p_s) \leq \phi_\nu(x_k, s_k) + \eta\alpha_s D\phi_\nu(x_k, s_k; p_w)$;
    Compute $(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1})$ using (19.28);
    if a quasi-Newton approach is used
      update the approximation B_k;
    Set $k \leftarrow k + 1$;
  end
  Set $\mu \leftarrow \sigma\mu$ and update ϵ_μ;
end

Figure 15: Interior algorithm pseudo code.

The attempted interior-point algorithm was however not successful and did not iterate correctly. Figure 16 shows the sequence which the algorithm iterates, it is clear that something is wrong with the updating step direction. Similar behaviour was seen for various starting points.

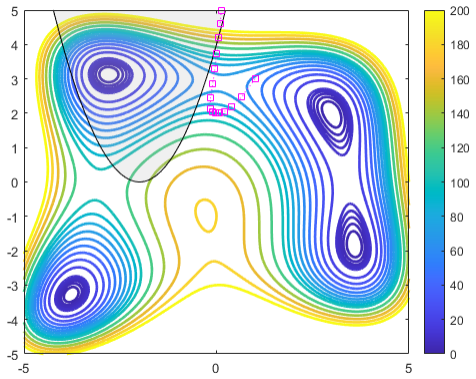


Figure 16: Interior algorithm iteration sequence code.

5.8 MATLAB Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Question 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that constructs H, g, A, and b as function of n,
% ubar, and d0.

function[H,g,A,b] = construct_input_func(n,ubar,d0)
H = ubar*eye(n+1);
b = [-d0;zeros(n-1,1)];
g = -2*ubar*(ones(n+1,1)); % Transposing this gives error
A = zeros(n+1,n); % Create empty A matrix of correct dimensions to be populated later
for i = 1:n-1 % Populate the diagonal of A matrix with respective -1,1
    A(i,i:i+1) = [-1,1];
end
A(n,1) = 1; % Complete constraint 1 row
A(n:n+1,n) = [-1,-1]; % Complete constraint 3 row
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that constructs the KKT-matrix as function of n,
% ubar, and d0.

function[eq1,eq2] = construct_KKT_func(n,ubar,d0)
% Run previous function to find H,g,A,b
[H,g,A,b] = construct_input_func(n,ubar,d0);
% Construct normal KKT condition linear system
% eq1 is the left side of the system and eq2 is the right (eq1=eq2)
eq1 = [H,-A;-A',zeros(n,n)];
eq2 = -[g;b];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that solves (1.1) using an LDL factorization.

function[x,lambda] = EqualityQPSolverLDL(n,ubar,d0)
% Use previous KKT function to produce the linear system to solve
[eq1,eq2] = construct_KKT_func(n,ubar,d0);
% Information on how the LDL function works:
% https://se.mathworks.com/help/matlab/ref/ldl.html
[L,D,P] = ldl(eq1,"lower","vector");
s(P,:) = L'\(D\(L\(eq2(P,:))));
x = s(1:n+1)';
lambda = s(n+2:end)';
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that solves (1.1) using an LU factorization.

function[x,lambda] = EqualityQPSolverLU(n,ubar,d0)
% Use previous KKT function to produce the linear system to solve
[eq1,eq2] = construct_KKT_func(n,ubar,d0);
% Information on how the lu function works:
% https://se.mathworks.com/help/matlab/ref/lu.html
[L,U,P] = lu(eq1,"vector"); % Perform the LU matrix factorization
s = U\(L\eq2(P)); % Find solutions for the equations
x = s(1:n+1);
lambda = s(n+2:end);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Make a Matlab function that solves (1.1) using the Null-Space procedure
% based on QR-factorizations.

function[x,lambda] = EqualityQPSolverNullSpace(n,ubar,d0)
% Use previous construct function to produce the H,g,A,b inputs of the
% system. These will then be used to solve the system with QR decomp.
[H,g,A,b] = construct_input_func(n,ubar,d0);
% Information on how the QR function works:
% https://se.mathworks.com/help/matlab/ref/qr.html
% See end of lecture 5 EqualityConstrainedQP slides for solution to a system
% with QR decmomposition
[Q,R_] = qr(A);
ncol = size(R_,2); % Find nr. columns of R matrix
Q1 = Q(:,1:ncol);
Q2 = Q(:,ncol+1:end);
R = R_(1:ncol,1:ncol);
xy = R'\b;
xz = (Q2'*H*Q2)\(-Q2'*(H*Q1*xy+g));
x = Q1*xy+Q2*xz;
lambda = R\(Q1'*(H*x+g));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function[x,lambda] = EqualityQPSolverRangeSpace(n,ubar,d0)
% Use previous construct function to produce the H,g,A,b inputs of the
% system. These will then be used to solve the system via range space.
% See slide 7 in week 5 EqualityConstrainedQP pdf for details.
[H,g,A,b] = construct_input_func(n,ubar,d0);

L = chol(H,"lower");
v = (L*L')\g;
Ha = A'*inv(H)*A;
La = chol(Ha,"lower");
lambda = (La*La')\(b+A'*v);
x = H\(A*lambda-g);

% A_ = A'*A;
% lambda = A_\(b+A'*g);
% x = H*(A*lambda-g);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that solves (1.1) using an LDL factorization.

function[x,lambda] = EqualityQPSolverSparseLDL(n,ubar,d0)
% Use previous KKT function to produce the linear system to solve
[eq1,eq2] = construct_KKT_func(n,ubar,d0);
sparse_eq1 = sparse(eq1);
% Information on how the LDL function works:
% https://se.mathworks.com/help/matlab/ref/ldl.html
[L,D,P] = ldl(sparse_eq1,"lower","vector");
s = L'\(D\(L\eq2(P)));
x = s(1:n+1)';
lambda = s(n+2:end)';
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make a Matlab function that solves (1.1) using an LU factorization.
```

```
function[x,lambda] = EqualityQPSolverSparseLU(n,ubar,d0)
% Use previous KKT function to produce the linear system to solve
[eq1,eq2] = construct_KKT_func(n,ubar,d0);
sparse_eq1 = sparse(eq1); % create sparse matrix
% Information on how the lu function works:
% https://se.mathworks.com/help/matlab/ref/lu.html
[L,U,P] = lu(sparse_eq1,"vector"); % Perform the LU matrix factorization
s = U\(L\eq2(P)); % Find solutions for the equations
x = s(1:n+1);
lambda = s(n+2:end);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function[x,lambda] = EqualityQPSolverMaster(n,ubar,d0,solver)
n = n;
ubar = ubar;
d0 = d0;

if strcmpi(solver, "LDL")
[x,lambda] = EqualityQPSolverLDL(n,ubar,d0)
return

elseif strcmpi(solver, "LU")
[x,lambda] = EqualityQPSolverLU(n,ubar,d0)
return

elseif strcmpi(solver, "NullSpace")
[x,lambda] = EqualityQPSolverNullSpace(n,ubar,d0)
return

elseif strcmpi(solver, "RangeSpace")
[x,lambda] = EqualityQPSolverRangeSpace(n,ubar,d0)
return

elseif strcmpi(solver, "SparseLDL")
[x,lambda] = EqualityQPSolverSparseLDL(n,ubar,d0)
return

elseif strcmpi(solver, "SparseLU")
[x,lambda] = EqualityQPSolverSparseLDL(n,ubar,d0)
return

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Question 2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x_r,y_r,z_r,s_r,iter,his] = QP_IP_PrimalDual(H,g,C,d,A,b,x,y,z,s)
% Implement the Primal-Dual Interior-Point Algorithm for this convex quadratic program
% min (1/2)x'Hx+g'x
% s.t. A'x = b =>y
% C'x >= d =>z
eta = 0.995;
% iteration and stopping criteria
k = 0;
maxit = 100;
stopL = 1.0e-9;
stopA = 1.0e-9;
stopC = 1.0e-9;
```

```
stopmu = 1.0e-9;
his = [x'];
% Compute the residuals rL,rA,rC,rsz and duality gap mu

[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+d+s;
rSZ = S*Z*e;
mu = z'*s/nC;

while (k<=maxit && norm(rL)>=stopL && norm(rA)>=stopA && norm(rC) >= stopC ...
        && abs(mu)>=stopmu)
    Hbar = H+C*(S\Z)*C';
    % Solve for step using LDL
    eq1 = [Hbar,-A; -A',zeros(size(A,2))];
    [L,D,P,] = ldl(eq1,"lower","vector");
    % Calculate the affine step direction
    rLbar = rL-C*(S\Z)*(rC-Z\rSZ);
    eq2 = -[rLbar; rA];
    delta_xy_a(P,:) = L'\(D\(L\(eq2(P,:))));

    delta_x_a = delta_xy_a(1:length(x));
    delta_y_a = delta_xy_a(length(x)+1:length(x)+length(y));
    delta_z_a = -(S\Z)*C'*delta_x_a+(S\Z)*(rC-Z\rSZ);
    delta_s_a = -(Z\rSZ)-(Z\(S*delta_z_a));
    % Calculate alpha
    alpha_a = 1;
    idelta_x_z = find(delta_z_a<0);
    if (isempty(idelta_x_z) == 0)
        alpha_a = min(alpha_a, min(-z(idelta_x_z)./delta_z_a(idelta_x_z)));
    end
    idelta_x_s = find(delta_s_a<0);
    if (isempty(idelta_x_s) == 0)
        alpha_a = min(alpha_a, min(-s(idelta_x_s)./delta_s_a(idelta_x_s)));
    end
    % Calculate the affine duality gap mu
    mu_a = ((z+alpha_a*delta_z_a)'*(s+alpha_a*delta_s_a))/nC;
    % centering
    sigma = (mu_a/mu)^3;
    % Correcting step
    rSZbar = rSZ+diag(delta_s_a)*diag(delta_z_a)*e-sigma*mu*e;
    rLbar = rL-C*(S\Z)*(rC-Z\rSZbar);
    % Solve for step using LDL as before
    eq1 = -[rLbar; rA];
    delta_xy(P,:) = (L'\(D\(L\(eq1(P,:)))));
    delta_x = delta_xy(1:length(x));
    delta_y = delta_xy(length(x)+1:length(x)+length(y));

    delta_z = -(S\Z)*C'*delta_x+(S\Z)*(rC-Z\rSZbar);
    delta_s = -Z\rSZbar-Z\S*delta_z;
```

```
    % Update alpha
    alpha = 1;
    idelta_x_z = find(delta_z<0);
    if (isempty(idelta_x_z) == 0)
        alpha = min(alpha, min(-z(idelta_x_z)./delta_z(idelta_x_z)));
    end
    idelta_x_s = find(delta_s<0);
    if (isempty(idelta_x_s) == 0)
        alpha = min(alpha, min(-s(idelta_x_s)./delta_s(idelta_x_s)));
    end

    % Update x, y, z and s by taking the actual step
    x = x+eta*alpha*delta_x;
    y = y+eta*alpha*delta_y;
    z = z+eta*alpha*delta_z;
    s = s+eta*alpha*delta_s;

    %Add new x, y, z and s into history
    his = [his;x'];
    Z = diag(z);
    S = diag(s);
    k = k+1;

    % Update residuals
    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
% Final outputs
x_r = x;
y_r = y;
z_r = z;
s_r = s;
iter = k;

end

function [x ,his,iter] = ActiveSet(H,g,A,b,x0)
 %{
min f(x)=1/2*x'Hx+gx
s.t. A'x>=b
x0 is initial point
            %}
K=0;
m = size(A',1); % Number of constraints
n = size(A',2); % Number of variables ... p in R^n

k = 1;
not_converged = 1;
error = 1e-10;
maxIter = 100;
% Set the initial working set
w = abs(A'*x0 - b) < error;
w_index = find(w);
x_old = x0;
```

```
while (not_converged && k < maxIter)
    g_k = H * x_old + g;
    [ p, lambda ] = LDL_solver(H,g_k,A,w_index);
    if all(abs(p)<error)
        if all(lambda(w_index)>= 0)
            not_converged = 0;
            x_star = x_old;
        else
            [~,i_min_lambda] = min(lambda(w_index));
            x_new = x_old;
            w_index(i_min_lambda,:) = [];
        end
    else
        [ alpha , blockers ] = AlphaCal(p,x_old,A,b,w_index);
        x_new = x_old + alpha * p;
        if alpha < 1
            w_index(blockers(1),:) = blockers(1);
        end
        k+1 ;
    end
    x_old = x_new;
    out.x(:,k) = x_old;
    k = k + 1;
end
iter = k-2;
x=x_star;
his = out.x;
end

% Compute the distance to the nearest
% inactive constraint in the search direction
function [ alpha , blocker ] = AlphaCal(p,x_old,A,b,w_index)
stepset = A' * p < 0;
stepset(w_index,:) = 0;
alphaset = (b(stepset,:)- A(:,stepset)'*x_old)./(A(:,stepset)'*p);
[ alpha , blocker ] = min([alphaset;1]);
end

% just culate the x in W_i, index is the record of i colomn
function[x,lambda] = LDL_solver(H,g,A,index)
% Calculate matrix A, number of rows and columns
[ra,ca]=size(A);
[ri,ci]=size(index);
b = zeros(ri,1);
A = A(:,index');
[r,c]=size(A);
% AA*[x,lambda] = bb
% change the form of first order necessary optimality conditions
% to the form of symmetric positive definite matrix
AA = [H,-A;-A',zeros(c,c)];
bb = [-g ; -b];
% LDL matrix decomposition
[LA,DA,P] = ldl(AA,'lower','vector');
% caculate the result of x and lambda
xx(P,:) = LA'\(DA\(LA\bb(P,:)));

% take the x from xx
```

```
x = xx(1:r,:);
% take lambda from xx
la = xx(r+1:r+c,:);
lambda = zeros(ca,1);
for i = 1:ri
    lambda(index(i))=la(i);
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test driver for QP IP algorithm
% build the input matrix of MarkowitzMarkowitz' portfolio
% optimization problem portfolio optimization problem
% When R=2, caculate the optimal portfolio
sigma = [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];
H = 2*sigma;
C = eye(5,5);
R = [ 15.1; 12.5; 14.7; 9.02; 17.68];
A = [R'; ones(5,1)'];
b = [2; 1];
g = zeros(5,1);
d = zeros(5,1);
x=zeros(5,1);
y=ones(2,1);
z=ones(5,1);
s=ones(5,1);
[x_result,y_result,z_result,s_result,iter,his] = QP_IP_PrimalDual(H,g,C,d,A',b,x,y,z,s)

% f = @() QP_IP_PrimalDual(H,g,C,d,A',b,x,y,z,s);
% t = timeit(f) % t = 0.1638

%% Test compute time with quadprog.
% f = @() quadprog(H,g,-C',-d,A,b,[],[]);
% t = timeit(f)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Active set algorithm for QP.
function [x ,his,iter] = ActiveSet(H,g,A,b,x0)
 %{
min f(x)=1/2*x'Hx+gx
s.t. A'x>=b
x0 is initial point
            %}
K=0;
m = size(A',1); % Number of constraints
n = size(A',2); % Number of variables ... p in R^n

k = 1;
not_converged = 1;
error = 1e-10;
maxIter = 100;
% Set the initial working set
w = abs(A'*x0 - b) < error;
w_index = find(w);
```

```
x_old = x0;
while (not_converged && k < maxIter)
    g_k = H * x_old + g;
    [ p, lambda ] = LDL_solver(H,g_k,A,w_index);
    if all(abs(p)<error)
        if all(lambda(w_index)>= 0)
            not_converged = 0;
            x_star = x_old;
        else
            [~,i_min_lambda] = min(lambda(w_index));
            x_new = x_old;
            w_index(i_min_lambda,:) = [];
        end
    else
        [ alpha , blockers ] = AlphaCal(p,x_old,A,b,w_index);
        x_new = x_old + alpha * p;
        if alpha < 1
            w_index(blockers(1),:) = blockers(1);
        end
        k+1 ;
    end
    x_old = x_new;
    out.x(:,k) = x_old;
    k = k + 1;
end
iter = k-2;
x=x_star;
his = out.x;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test driver for Active set algorithm.
% H = [2 0;0 2];
% g = [-2;-5];
% ga = 0;
% A = [1 -1 -1 1 0;
%     -2 -2 2 0 1];
% b = [-2;-6;-2;0;0];
% x0=[2;0];
% [x,his,iter] = ActiveSet(H,g,A,b,x0)

 %%
% implement test of Active-Set Algorithm
H = [2 0;0 2];
g = [-2;-5];
ga = 0;
A = [1 -1 -1 1 0;
    -2 -2 2 0 1];
b = [-2;-6;-2;0;0];
x0=[2;0];
[x,his,iter] = ActiveSet(H,g,A,b,x0)

f = @() ActiveSet(H,g,A,b,x0);
t = timeit(f)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Comparison of Interior-point and Active set with quadprog.
% 2.8 QP Markowitz portfolio optimization problem
% quadprog
```

```
n = 17.68;
step = 0.1;
start = 9.02;
mm = start:step:n;
size =length(mm);

risk = eye(1,size);

sigma = [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];
H = 2*sigma;

R = [ 15.1; 12.5; 14.7; 9.02; 17.68];
A = [R'; ones(5,1)'];

g = zeros(5,1);


% when r between 8~18

k=1;
for i = start:step:n %loop for calculating risk with Return change
b = [i; 1];
[x,fval] = quadprog(H,[],[],[],A,b,g,[]);
risk(k) = fval;
k = k+1;
end
figure
plot(risk, mm,'LineWidth',1);
xlabel('Risk')
ylabel('Return')
xlim([0 4])
ylim([0 20])

xx = eye(5,5);
risks = eye(5,1);
for i = 1:5
risks(i) = xx(i,:)*sigma*xx(i,:)';
end
hold on
text(risk(size),n,'\leftarrow efficient frontier')
hold off

%

%2.8 QP Markowitz portfolio optimization problem
%QP_IP_PrimalDual
n = 17.68;
step = 0.1;
start = 9.02;
mm = start:step:n;
size =length(mm);

risk = eye(1,size);
```

```
sigma = [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];
H = 2*sigma;

R = [ 15.1; 12.5; 14.7; 9.02; 17.68];
A = [R'; ones(5,1)']';

g = zeros(5,1);

x0=zeros(5,1);
y=ones(2,1);
z=ones(5,1);
s=ones(5,1);

C = eye(5,5);
d = zeros(5,1);
% when r between 8~18

k=1;
for i = start:step:n %loop for calculating risk with Return change
b = [i; 1];
[x] = QP_IP_PrimalDual(H,g,C,d,A,b,x0,y,z,s);
fval = x'*sigma*x;
risk(k) = fval;
k = k+1;
end
figure
plot(risk, mm,'LineWidth',1);
xlabel('Risk')
ylabel('Return')
xlim([0 4])
ylim([0 20])

xx = eye(5,5);
risks = eye(5,1);
for i = 1:5
risks(i) = xx(i,:)*sigma*xx(i,:)';
end
hold on
text(risk(size),n,'\leftarrow efficient frontier')
hold off
%%
% 2.8 QP Markowitz portfolio optimization problem
% sum(x_i) <= 1,R'x>=10
% the portfolio when mini risk
% Active Set
n = 17.68;
% n=14.22;
step = 0.4;
% start = 9.02;
start = 14.62;
mm = start:step:n;
size =length(mm);
```

```
risk = eye(1,size);

sigma = [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];
H = 2*sigma;

R = [ 15.1; 12.5; 14.7; 9.02; 17.68];
A = [R'; -ones(5,1)';eye(5,5)]';

g = zeros(5,1);

x0=[0.2;0.2;0.2;0.2;0.2];
y=ones(2,1);
z=ones(5,1);
s=ones(5,1);

C = eye(5,5);
d = zeros(5,1);
% when r between 8~18
b = [10; -1;zeros(5,1)];

[x,his,iter] = ActiveSet(H,g,A,b,x0)
f = @() ActiveSet(H,g,A,b,x0);
t1 = timeit(f)

risks = x'*sigma*x
retur = R'*x
[x,fval,exitflag,output,lambda] = quadprog(H,[],-A',-b)
f = @() quadprog(H,[],-A',-b);
t2 = timeit(f)

qr = R'*x
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Question 3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Week 5 Problem 3.3/ Problem 3 Exam Assignment 3.1.3
% Use quadprog to and a portfolio with return, R = 10:0, and minimal risk.
% What is the optimal portfolio and what is the risk (variance)?
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 = [15.1;12.5;14.7;9.02;17.68];
A = [mu';ones(5,1)'];
b = [10;1]; % We would like a return of 10 in this example
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2 = -eye(5);
b2 = zeros(5,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.

[x,variance] = quadprog(H,[],A2,b2,A,b);
```

```
% Results in allocation vector x = [0.0000 0.2816 0.0000 0.7184 0.0000]'
% Meaning of our fund we allocate 28.16% to x2 and 71.84% to x4.
% Variance = 1.0461
% Plot results
bar(x)
xlabel("Security")
ylabel("Amount")
title("Optimal allocation of funds for R = 10")
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Week 5 Problem 3.4/ Exam Assignment Problem 3.1.4
% Compute the efficient frontier, i.e. the risk as function of the return.
% Plot the efficient frontier as well as the optimal portfolio as function
% of return.
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 = [15.1;12.5;14.7;9.02;17.68];
A = [mu';ones(5,1)'];
%b = [10;1]; % We would like a return of 10 in this example
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2 = -eye(5);
b2 = zeros(5,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.
x_ = zeros(5,100); % create zero matrix for 30 different allocation vector
% x to fill
variance_ = zeros(1,100); % create zero vector for 100 different variances.
% Make a sort of monte carlo simulation with various portfolios.
% create 100 random return values within interval [9.02,17.68]
R = ((17.68-9.02).*rand(100,1) + 9.02)';

for i = 1:length(R)
    b = [R(i),1];
    [x_(:,i),variance_(1,i)] = quadprog(H,[],A2,b2,A,b);
end

plot(R,variance_,"o")
title("Efficient Frontier")
xlabel("Return")
ylabel("Variance")
legend(["With out risk free security"],'location','northEast','fontsize',10);
xlim([8,18])
% It seems that the optimal portfolio is obtained when return is about 14.4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Week 5 Problem 3.4/ Exam Assignment Problem 3.2.2 amd 3.2.3
% Compute the efficient frontier, i.e. the risk as function of the return.
% Plot the efficient frontier as well as the optimal portfolio as function
% of return.
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 = [15.1;12.5;14.7;9.02;17.68];
A = [mu';ones(5,1)'];
```

```
%b = [10;1]; % We would like a return of 10 in this example
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2 = -eye(5);
b2 = zeros(5,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.
x_ = zeros(5,100); % create zero matrix for 30 different allocation vector
% x to fill
variance_ = zeros(1,100); % create zero vector for 30 different variances.
% create 30 random return values within interval [9.02,17.68]
R = ((17.68-9.02).*rand(100,1) + 9.02)';

for i = 1:length(R)
    b = [R(i),1];
    [x_(:,i),variance_(1,i)] = quadprog(H,[],A2,b2,A,b);
end

% With risk free security:
H2 = [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];
mu2 = [15.1;12.5;14.7;9.02;17.68;2];
A_2 = [mu2';ones(6,1)'];
%b = [10;1]; % We would like a return of 10 in this example
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2_2 = -eye(6);
b2_2 = zeros(6,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.
x_2 = zeros(6,100); % create zero matrix for 30 different allocation vector
% x to fill
variance_2 = zeros(1,100); % create zero vector for 30 different variances.
% create 30 random return values within interval [9.02,17.68]
R2 = ((17.68-9.02).*rand(100,1) + 9.02)';

for i = 1:length(R2)
    b_2 = [R2(i),1];
    [x_2(:,i),variance_2(1,i)] = quadprog(H2,[],A2_2,b2_2,A_2,b_2);
end

% What is the minimal risk and optimal portfolio giving a return of
% R = 15.0? Plot this point on the Efficient Frontier plot.
b_3 = [15,1]; % Want the Return = 15
A_2 = [mu2';ones(6,1)'];
%b = [10;1]; % We would like a return of 10 in this example
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2_2 = -eye(6);
b2_2 = zeros(6,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.
x_2 = zeros(6,100); % create zero matrix for 30 different allocation vector
% x to fill
[x3,variance3] = quadprog(H2,[],A2_2,b2_2,A_2,b_3);
x3
```

```
variance3

plot(R,variance_,"o")
title("Efficient Frontier")
xlabel("Return")
ylabel("Variance")
hold on

plot(R2,variance_2,"o")

plot(b_3,variance3,"x",'MarkerSize',30)
hold off
title("Efficient Frontier with and with out risk free security")
xlabel("Return")
ylabel("Variance")
legend(["With out risk free security","With risk free security", "Return=15"],'location','northEast','font
xlim([8,18])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Week 5 Problem 3.3/ Problem 3 Exam Assignment 3.2.2 and 3.2.3
% Use quadprog to and a portfolio with return, R = 10:0, and minimal risk.
% What is the optimal portfolio and what is the risk (variance)?
retur = 9; % Change to whatever return is needed.
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 = [15.1;12.5;14.7;9.02;17.68;2];
A = [mu';ones(6,1)'];
b = [retur;1]; % We would like a return as chosen above.
% Second constraint: We need allocation vector x to be positive,
% i.e. A2*x >= b2
A2 = -eye(6);
b2 = zeros(6,1);
% variance = x'*H*x, where x is allocation vector and H is covar matrix.

[x,variance] = quadprog(H,[],A2,b2,A,b);
% Results in allocation vector x = [0.0000 0.2816 0.0000 0.7184 0.0000]'
% Meaning of our fund we allocate 28.16% to x2 and 71.84% to x4.
% Variance = 1.0461
% Plot results
bar(x)
xlabel("Security")
ylabel("Amount")
title("Optimal allocation of funds for R = 9")
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Question 4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Primal-Dual Interior-Point algorithm for LP
function [x,info,mu,lambda,iter] = LPippd(g,A,b,x)
% LPIPPD   Primal-Dual Interior-Point LP Solver
%
%          min  g'*x
%           x
%          s.t. A x  = b      (Lagrange multiplier: mu)
%                 x >= 0      (Lagrange multiplier: lamba)
```

```
%
% Syntax: [x,info,mu,lambda,iter] = LPippd(g,A,b,x)
%
%         info = true   : Converged
%              = false  : Not Converged

% Created: 04.12.2007
% Author : John Bagterp Jørgensen
%          IMM, Technical University of Denmark

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

maxit = 100;
tolL = 1.0e-9;
tolA = 1.0e-9;
tols = 1.0e-9;

eta = 0.99;

lambda = ones(n,1);
mu = zeros(m,1);

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

% Converged
Converged = (norm(rL,inf) <= tolL) && ...
            (norm(rA,inf) <= tolA) && ...
            (abs(s) <= tols);
%%
iter = 0;
while ~Converged && (iter<maxit)
    iter = iter+1;

    % ======================================================================
    % Form and Factorize Hessian Matrix
    % ======================================================================
    xdivlambda = x./lambda;
    H = A*diag(xdivlambda)*A';
    L = chol(H,'lower');

    % ======================================================================
    % Affine Step
    % ======================================================================
    % Solve
    tmp = (x.*rL + rC)./lambda;
    rhs = -rA + A*tmp;

    dmu = L'\(L\rhs);
    dx = xdivlambda.*(A'*dmu) - tmp;
    dlambda = -(rC+lambda.*dx)./x;

    % Step length
```

```
    idx = find(dx < 0.0);
    alpha = min([1.0; -x(idx,1)./dx(idx,1)]);

    idx = find(dlambda < 0.0);
    beta = min([1.0; -lambda(idx,1)./dlambda(idx,1)]);

    % ================================================================
    % Center Parameter
    % ================================================================
    xAff = x + alpha*dx;
    lambdaAff = lambda + beta*dlambda;
    sAff = sum(xAff.*lambdaAff)/n;

    sigma = (sAff/s)^3;
    tau = sigma*s;

    % ================================================================
    % Center-Corrector Step
    % ================================================================
    rC = rC + dx.*dlambda - tau;

    tmp = (x.*rL + rC)./lambda;
    rhs = -rA + A*tmp;

    dmu = L'\(L\rhs);
    dx = xdivlambda.*(A'*dmu) - tmp;
    dlambda = -(rC+lambda.*dx)./x;

    % Step length
    idx = find(dx < 0.0);
    alpha = min([1.0; -x(idx,1)./dx(idx,1)]);

    idx = find(dlambda < 0.0);
    beta = min([1.0; -lambda(idx,1)./dlambda(idx,1)]);

    % ================================================================
    % Take step
    % ================================================================
    x = x + (eta*alpha)*dx;
    mu = mu + (eta*beta)*dmu;
    lambda = lambda + (eta*beta)*dlambda;

    % ================================================================
    % Residuals and Convergence
    % ================================================================
    % Compute residuals
    rL = g - A'*mu - lambda;    % Lagrangian gradient
    rA = A*x - b;               % Equality Constraint
    rC = x.*lambda;             % Complementarity
    s = sum(rC)/n;              % Duality gap

    % Converged
    Converged = (norm(rL,inf) <= tolL) && ...
                (norm(rA,inf) <= tolA) && ...
                (abs(s) <= tols);
end
```

```
%%
% Return solution
info = Converged;
if ~Converged
    x=[];
    mu=[];
    lambda=[];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Simplex algorithm for solving LP (Part 1)
function [tbl,x_] = LPsimplex1(A,b,g)
%This function implments the simplex matrix algorithm.
% Inputs of A and b must be in equality standard form. Where A is matrix
% of constraints and b is a vector of constraints such that Ax=b.
% g is the objective function to be minimized in defined as a vector, such
% that J(x)=g'(x).
% The output is the optimal minimal solution [x_] and the final tableau
% of it. The final tableau has the form:
%  [ A | b  ]
%  [ g | J  ]

[A,b] = phase1(A,b);
[tbl,x_] = simplex1(A,b,g);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Simplex algorithm for solving LP (Part 2)
function [tbl,x_] = simplex1(A,b,g)
[m,n]        = size(A);
tbl          = zeros(m+1,n+1);
tbl(1:m,1:n) = A;
tbl(m+1,1:n) = g(:);
tbl(1:m,end) = b(:);

active = true;
while active
    if any(tbl(end,1:n)<0)%check if there is negative cost coeff.
        [~,J] = min(tbl(end,1:n)); %yes, find the most negative
        % now check if corresponding column is unbounded
        if all(tbl(1:m,J)<=0)
          error('problem unbounded. All entries <= 0 in column %d',J);
        %do row operations to make all entries in the column 0
        %except pivot
        else
            pivot_ = 0;
            current_min = 1.0e+50; % Start larger than any "real" input would.
            for i = 1:m
                if tbl(i,J)>0
                    tmp = tbl(i,end)/tbl(i,J);
                    if tmp < current_min
                        current_min = tmp;
                        pivot_ = i;
                    end
                end
            end
            tbl(pivot_,:) = tbl(pivot_,:)/tbl(pivot_,J);
            %now make all entries in J column zero.
            for i=1:m+1
```

```
                if i ~= pivot_
                    tbl(i,:)=tbl(i,:)-sign(tbl(i,J))*...
                                abs(tbl(i,J))*tbl(pivot_,:);
                end
            end
        end
            x_ = get_current_x();
    else
        active = false;
    end
end

    %internal function, finds current basis vector
    function current_x = get_current_x()
        current_x = zeros(n,1);
        for j=1:n
            if length(find(tbl(:,j)==0))==m
                idx= tbl(:,j)==1;
                current_x(j)=tbl(idx,end);
            end
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Simplex algorithm for solving LP (Part 3)
function [A,b] = phase1(A,b)
[m,n]              = size(A);
tbl                = zeros(m+1,n+m+1);
tbl(1:m,1:n)       = A;
tbl(end,n+1:end-1) = 1;
tbl(1:m,end)       = b(:);
tbl(1:m,n+1:n+m)   = eye(m);

for i = 1:m %now make all entries in bottom row zero
    tbl(end,:) = tbl(end,:)-tbl(i,:);
end

tbl = simplex1(tbl(1:m,1:n+m),tbl(1:m,end),tbl(end,1:n+m));

A = tbl(1:m,1:n);
b = tbl(1:m,end);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Driver files for testing (Interior-point)
%g = A'*mu + lambda;
% Ex.1
% A=[3,1;
%    1,2];
% b=[7;9];
% g=[-3,-5]';
% x=[1,1]';
% format short;
% Should give back x1=1, x2=4, g=23

% Ex. 2
A = [1 1 1 1 1];
```

```
b = [1];
g = [-15.10 -12.50 -14.70 -9.02 -17.68]'; % Expected security returns.
x = [1 1 1 1 1]'/5; % Starting values set so that all funds are allocated equally.
%b = A*x;
%g, A, b
[xlp,info,mulp,lambdalp,iter] = LPippd(g,A,b,x);
iter=iter
xlp
% Measure CPU time.
f = @() LPippd(g,A,b,x);
cpu_time = timeit(f)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Driver files for testing (Simplex)
% Ex.1
% A=[3,1;
%    1,2];
% b=[7;9];
% g=[-3,-5];
% format short;
% Should give back x1=1, x2=3, g=23

% Ex.2
% A=[1,0,1,0,0;
%    0,1,0,1,0;
%    1,1,0,0,1];
% b=[4,6,8];
% g=[-2,-5,0,0,0];

% Ex. 3
% %Inequality constraints. The should be in the form [A]{x}={b}.
% % All securities must add up to a total of 1.
g=[-15.10 -12.50 -14.70 -9.02 -17.68];
A=[1 1 1 1 1];
b=[1];

[t,x] = LPsimplex1(A,b,g)
%t(1:2,5)
f = @() LPsimplex1(A,b,g);
cpu_time = timeit(f)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Driver files for testing (linprog)
% Ex.1
% A=[3,1;
%    1,2];
% b=[7;9];
% g=[-3,-5]';
% x=[1,1]';
% Aeq=[];
% beq=[];
% lb=zeros(2,1);
% ub=ones(2,1)*1000;
% x0=[1,1]';
% Should give back x1=1.4, x2=3.8, g=24.6

% Ex.2
%objectve function: input coefficients. These are the returns of securities
g=[-15.10 -12.50 -14.70 -9.02 -17.68];
```

```
%Inequality constraints. The should be in the form [A]{x}={b}.
% All securities must add up to a total of 1.
A=[1 1 1 1 1];
b=[1];
%Lower and upper bounds of variables. Lowest allocation is nothing ie. =
%0, and highest allocation is everything ie. = 1.
lb=zeros(5,1);
ub=ones(5,1);
%Add empty matrices for coefficients of equality constraints and initial
%guess
Aeq=[];
beq=[];
x0=[1 1 1 1 1]'/5; % Starting values set so that all funds are allocated equally.
%Specify search options:
%Use the dual-simplex algorithm since simplex will be removed after this
version
%Display the results of all iterations
options=optimoptions('linprog','Algorithm','interior-point','Display','iter') %'simplex' can be used
[x,fval,exitflag,output] = linprog(g,A,b,Aeq,beq,lb,ub,x0,options)
f = @() linprog(g,A,b,Aeq,beq,lb,ub,x0,options);
cpu_time = timeit(f)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Question 5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 5.6, Implementation of a damped BFGS approximation of Hessian on a SQP
% procedure.

function [x,lambda,info] = SQP_BFGS(ObjFun,ConFun,x,lambda,tolerance,maxit)
% This function is meant to solve a non-linear program using a damped BFGS
% approximation to the Hessian of the lagrangian.

% Utilize the objective function and constraint function.
[~,g] = ObjFun2(x);
[c,A] = ConFun2(x);

% Lagrangian gradient.
dL = g-A*lambda;
% KKT condition.
F = [dL; c];

iter = 1;
n = length(x);
x(:,iter) = x;

% Start the Hessian being equal to identity matrix of appropriate size n.
B = eye(n);
while ((iter < maxit) && (norm(F,"inf") > tolerance))
    % Define KKT system and solve system.
    xlambda = [B -A; A' zeros(1)]\[-F]; % Hard coded for this problem.
    % Find next point in iteration.
    point = xlambda(1:n);
    % Extract x value.
    x(:,iter+1) = x(:,iter)+point;
    % Update initial lambda value.
    lambda = lambda + xlambda(n+1:end);
    iter = iter+1;
```

```
    % Evaluate function and constraints.
    [~,g] = ObjFun2(x(:,iter));
    [c,A] = ConFun2(x(:,iter));
    % Compute new Lagrangian gradient.
    dL2 = g-A*lambda;
    % KKT condition.
    F = [dL2; c];
    % Damped BFGS updating.
    p = x(:,iter) - x(:,iter-1);
    y = dL2 - dL;
    dL = dL2;
    if p'*y >= 0.1*p'*B*p
        theta = 1; % Set to arbitrary value.
    else
        theta = (0.1*p'*B*p)/(p'*B*p-p'*y);
    end
    r = theta*y+(1-theta)*B*p;
    B = B-(B*(p*p')*B)/(p'*B*p) + r*r'/(p'*r);
end

info.iteration = iter;
info.history = x;
% Final solution.
x = x(:,iter);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Driver file for SQP with damped BFGS
% Testing a SQP procedure with a damped BFGS approximation to the Hessian
% matrix for the problem.

x0 = [0;0];
lambda0 = [1];
tol = 0.01;
maxit = 200;

[x1,lambda1,info1] = SQP_BFGS(@ObjFun2,@ConFun2,x0,lambda0,tol,maxit)
% Should give optimal result x3 = [-3.654, 2.737]

%% Countour plot and iterations
x = -5:0.05:5;
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar
yc1 = (x+2).^2;
hold on
fill([x(yc1<=5.1)],[yc1(yc1<=5.1)],[0.7 0.7 0.7],'facealpha',0.2)
plot(info1.history(1,:),info1.history(2,:),':ms','linewidth',0.5);
axis([-5 5 -5 5])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Damped BFGS and line search
function [x,lambda,info] = SQP_BFGS_LS(ObjFun,ConFun,x,lambda,tolerance, maxit)
% This function solves a nonlinear program by using a damped BFGS
% approximation to the hessian.
```

```
% Define objective function and constraints.
[f,g] = ObjFun2(x);
[c,A] = ConFun2(x);
n = length(x);
% Gradient of lagrangian.
dL = g-A*lambda;
F = [dL; c];
iter = 1;
x(:,iter) = x;
% Start the Hessian being equal to identity matrix of appropriate size n.
B = eye(n);
while ((iter < maxit) && (norm(F,"inf") > tolerance))
    % Compute to find search direction.
    %B
    xlambda = [B -A; A' zeros(1)]\[-dL; -c];
    point = xlambda(1:n);
    plambda = xlambda(n+1:end);

    % Line search method.
    sigma = 1;
    rho = 0.1;
    mu = (g'*point + sigma/2*point'*B*point)/((1-rho)*norm(c,1));
    alpha = 1;
    tau = 0.8;
    % Check sufficient decrease condition
    fun = ObjFun2(x(:,iter)+alpha*point);
    con = ConFun2(x(:,iter)+alpha*point);
    eta = 0.4;
    while fun+mu*norm(con,1) > f+mu*norm(c,1)+eta*alpha *(g'*point-mu*norm(c,1))
        alpha = tau*alpha; % update alpha
        fun = ObjFun2(x(:,iter)+alpha*point);
        con = ConFun2(x(:,iter)+alpha*point);
    end

    %f = fun;
    alpha_value(iter) = alpha; % Alpha values for every sequence.
    % Next point in iteration sequence.
    x(:,iter+1) = x(:,iter) + alpha*point;
    % Update lambda value.
    lambda = lambda+(alpha*plambda);
    iter = iter+1;
    % Objective function and constraint on new point.
    [f,g] = ObjFun2(x(:,iter));
    [c,A] = ConFun2(x(:,iter));
    dL2 = g-A* lambda; % new gradient of the Lagrangian
    F = [dL2; c];
    % Update damped BFGS
    p = x(:,iter) - x(:,iter-1);
    y = dL2 - dL;
    dL = dL2;
    if p'*y >= 0.1*p'*B*p
        % changing theta changes iteration sequence significantly. 0.8
        % seems to give closest end result.
        theta = 0.8;
    else
        theta = (0.8*p'*B*p)/(p'*B*p-p'*y);
    end
```

```
    r = theta*y+(1- theta)*B*p; % ensure possitive definiteness
    B = B - B*(p*p')*B/(p'*B*p) + r*r'/(p'*r); % B updating (Hessian)

end
% store number of iterations and iteration sequence
info.iteration = iter;
info.history = x;
info.alpha = alpha_value; % return step lenght sequence
x = x(:,iter);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Testing a SQP procedure with a damped BFGS approximation to the Hessian
% matrix for the problem and line search.

x0 = [-2;3];
lambda0 = [1];
tol = 0.01;
maxit = 200;

[x2,lambda2,info2] = SQP_BFGS_LS(@ObjFun2,@ConFun2,x0,lambda0,tol,maxit)
%info2.history
% Gets stuck at [-3.6517; 2.7281] after 28 iterations and never fully
% converges.
% Should give optimal result x3 = [-3.654, 2.737]

%% Countour plot and iterations
x = -5:0.05:5;
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar
yc1 = (x+2).^2;
hold on
fill([x(yc1<=5.1)],[yc1(yc1<=5.1)],[0.7 0.7 0.7],'facealpha',0.2)
plot(info2.history(1,:),info2.history(2,:),':ms','linewidth',0.5);
axis([-5 5 -5 5])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Trust Region SQP
function [x,lambda,info] = SQP_TrustRegion(ObjFun,ConFun,x,lambda,tolerance, maxit)

tol = tolerance;
maxit = maxit;
mu = 100;
gamma = 1;
eta = 0.5;
tr = 1; % Trust region.
iter = 0;
conver = 0; % Start with no convergence.
lambda1(:,1) = [1;1];
x(:,1) = x;

while(iter < maxit && ~conver)
    iter = iter + 1;
    x
    % Get f, c, and gradient objective function f.
```

```
    [f,g,H] = ObjFun2(x(:,iter));
    [c,dc,d2c] = ConFun2(x(:,iter));
    %A = dc'
    A = dc; % Maybe not correct.
    lambda1(:,iter)
    % Check convergence.
    if(norm(g-A'*lambda1(:,iter),"inf") < tol) % Changed A'.
        conver = 1;
        x_opt = x(:,iter);
        lambda_opt = lambda1(:,iter);
        info.history = x;
        info.iteration = iter;
        info.lamda = lambda1;
        info.steps = p(1:2,:);
    end
    % System of equations 18.50
    A
    A = [A,eye(2,1)]'; % Might cause dimensionality issues.
    g(end+1:end+2) = mu;
%     H_k
%     lam
%     d2c
    H = H - lambda1(1,iter)*d2c(:,:,1) - lambda1(2,iter)*d2c(:,:,1);
    xx_L = H; % unneccessary
    H(end+1:end+2,end+1:end+2) = 0;
    H
    g
%     %-A_eq
     A
     c
    lb = [-tr, -tr, 0, 0];
    ub = [tr, tr, Inf, Inf];

    opt = optimoptions('quadprog','Display','off');
    [p(:,iter),~,~,~,lambda] = quadprog(H,g,[],[],[A'],[0,0],[],[],lb,ub,[],opt);
    p(:,iter)
%     [p(:,iter),lambda] = EqualityQPSolver(H_k,g_k,-A,c); % maybe just A
    lambda1(:,iter+1) = lambda.eqlin;
%     lam(:,iter+1) = lambda;
    % Calculate mu.
    [f,df_k,~] = ConFun2(x);
    if(p(:,iter)'*H*p(:,iter) > 0)
        sigma = 1;
    else
        sigma = 0;
    end

    tmp = [df_k]'*p(:,iter)+(sigma/2)*(p(:,iter)'*H*p(:,iter) )/(0.5*norm(c,1));
    if(mu >= tmp)
        mu = mu;
    else
        mu = tmp;
    end

    % Merit fucntions 18.51 N&W.
    phi_1 = f + mu * sum(max(0,-c));
    phi_12 = ObjFun2(x(:,iter)+p(1:2,iter)) + mu*sum(max(0,-ConFun2(x(:,iter)+p(1:2,iter))));
```

```
    phi_tot = (phi_1 - phi_12)

    % q_mu 18.49 NEW.
    q_mu = f + df_k'*p(1:2,iter) + 0.5*(p(1:2,iter)'*xx_L*p(1:2,iter)) + mu*sum(max(0,-(c+dc'*p(1:2,iter))
    q_0 = f + mu*sum(max(0,-ConFun2([0;0])));
    q_mu
    q_0
    q_tot = (q_0 - q_mu)

    % Compute rho_k.
    %rho_k = (phi_1 - phi_12) / (q_0 - q_mu);
    rho_k = phi_tot / q_tot

    % Check trust region size.
    if(rho_k < eta)
        x(:,iter+1) = x(:,iter) + p(1:2,iter);
        %x(:,iter+1) = p(1:2,iter);
        tr = tr+1; % Add arbitrary size.
        display("yes")
    else
        x(:,iter+1) = x(:,iter);
        tr = gamma * norm(p(1:2,iter));
        display("no")
    end
%     x(:,iter+1) = x(:,iter) + p(1:2,iter);
end
% store number of iterations and iteration sequence
info.iteration = iter;
info.history = x;
info.lambda = lambda; % return step lenght sequence
x = x(:,iter);
lambda = lambda1(:,iter);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test driver for trust region.
% x0 = [2;3];
% [f_k,g_k,H_k] = ObjFun2(x0(:,iter))
% [c,dc,d2c] = ConFun2(x0(:,iter))
% A = dc'
% [x,lambda] = EqualityQPSolver(d2c,g_k,dc,c)
x0 = [0;0];
lambda0 = [1];
tolerance = 0.01;
maxit = 100;
[x3,lambda3,info3] = SQP_TrustRegion(@ObjFun2,@ConFun2,x0,lambda0,tolerance,maxit)
info3.history
%% Countour plot and iterations
x = -5:0.05:5;
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar
yc1 = (x+2).^2;
hold on
fill([x(yc1<=5.1)],[yc1(yc1<=5.1)],[0.7 0.7 0.7],'facealpha',0.2)
```

```
plot(info3.history(1,:),info3.history(2,:),':ms','linewidth',0.5);
axis([-5 5 -5 5])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Interior point algorithm for nonlinear program
function [x_sol,info] = QP_IP_PrimalDual(ObjFun,ConFun,x,maxit)
% Implement the Primal-Dual Interior-Point Algorithm for this convex quadratic
% program
% min (1/2)x'Hx+g'x
% x
% s.t. A x = b
% C x >= d
[f,g,H] = ObjFun2(x);
[c,dc,d2c] = ConFun2(x);
A = dc;
b = -c;
x = x;
%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);

eta = 0.995;
% iteration and stopping criteria
k = 0;
x(:,1) = x;
maxit = maxit;
stopL = 1.0e-9;
stopA = 1.0e-9;
stopC = 1.0e-9;
stopmu = 1.0e-9;

% residuals

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

rL = H*x(:,1)+g-A*y-C*z;
rA = -A'*x(:,1)+b;
rC = -C'*x(:,1)+d+s;
rSZ = S*Z*e;
mu = z'*s/nC;

while (k<=maxit && norm(rL)>=stopL && norm(rA)>=stopA && norm(rC) >= stopC ...
        && abs(mu)>=stopmu)
    k = k+1;
    A = dc;
    Hbar = H+C*(S\Z)*C';
    % Solve for step using LDL
    eq1 = [Hbar,-A; -A',zeros(size(A,2))];
    [L,D,P,] = ldl(eq1,"lower","vector");
    % Find affine direction
    rLbar = rL-C*(S\Z)*(rC-Z\rSZ);
```

```
    eq2 = -[rLbar; rA];
    delta_xy_a(P,:) = L'\(D\(L\(eq2(P,:))));

    delta_x_a = delta_xy_a(1:length(x(:,k)));
    delta_y_a = delta_xy_a(length(x(:,k))+1:length(x(:,k))+length(y));
    delta_z_a = -(S\Z)*C'*delta_x_a+(S\Z)*(rC-Z\rSZ);
    delta_s_a = -(Z\rSZ)-(Z\(S*delta_z_a));
    % Calculate alpha
    alpha_a = 1;
    idelta_x_z = find(delta_z_a<0);
    if (isempty(idelta_x_z) == 0)
        alpha_a = min(alpha_a, min(-z(idelta_x_z)./delta_z_a(idelta_x_z)));
    end
    idelta_x_s = find(delta_s_a<0);
    if (isempty(idelta_x_s) == 0)
        alpha_a = min(alpha_a, min(-s(idelta_x_s)./delta_s_a(idelta_x_s)));
    end
    % Affine calculation for mu (duality gap)
    mu_a = ((z+alpha_a*delta_z_a)'*(s+alpha_a*delta_s_a))/nC;
    % centering
    sigma = (mu_a/mu)^3;
    % Correcting step
    rSZbar = rSZ+diag(delta_s_a)*diag(delta_z_a)*e-sigma*mu*e;
    rLbar = rL-C*(S\Z)*(rC-Z\rSZbar);
    % Solve for step using LDL as before
    eq1 = -[rLbar; rA];
    delta_xy(P,:) = (L'\(D\(L\(eq1(P,:)))));
    delta_x = delta_xy(1:length(x(:,k)));
    delta_y = delta_xy(length(x(:,k))+1:length(x(:,k))+length(y));

    delta_z = -(S\Z)*C'*delta_x+(S\Z)*(rC-Z\rSZbar);
    delta_s = -Z\rSZbar-Z\S*delta_z;

    % Update alpha
    alpha = 1;
    idelta_x_z = find(delta_z<0);
    if (isempty(idelta_x_z) == 0)
        alpha = min(alpha, min(-z(idelta_x_z)./delta_z(idelta_x_z)));
    end
    idelta_x_s = find(delta_s<0);
    if (isempty(idelta_x_s) == 0)
        alpha = min(alpha, min(-s(idelta_x_s)./delta_s(idelta_x_s)));
    end
    % Line search method to update real alpha.
    %point = x(:,k);
    %sigma = 1;
    %rho = 0.1;
    tau = 0.4;
    %mu = (g'*point + sigma/2*point'*Hbar*point)/((1-rho)*norm(c,1)); % Hbar might be wrong dimensions.
    %alpha = 1;
    % Check sufficient decrease condition
    %fun = ObjFun2(x(:,k)+alpha*point);
    %con = ConFun2(x(:,k)+alpha*point);
    eta = 0.65;
    alpha = tau*alpha; % update alpha
%     while fun+mu*norm(con,1) > f+mu*norm(c,1)+eta*alpha *(g'*point-mu*norm(c,1))
%         alpha = tau*alpha; % update alpha
```

```
%         display("YES")
% %         fun = ObjFun2(x(:,iter)+alpha*point);
% %         con = ConFun2(x(:,iter)+alpha*point);
%     end

    alpha_value(k) = alpha; % Alpha values for every sequence.

    % Update initial values
    x(:,k+1) = x(:,k)+eta*alpha*delta_x;
    %x = x+eta*alpha*delta_x;
    %y(:,k+1) = y(:,k)+eta*alpha*delta_y;
    y = y+eta*alpha*delta_y;
    z = z+eta*alpha*delta_z;
    s = s+eta*alpha*delta_s;

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

    [f,g,H] = ObjFun2(x(:,k+1));
    [c,dc,d2c] = ConFun2(x(:,k+1));
    A = dc;
    % Update residuals
    rL = H*x(:,k+1)+g-A*y-C*z;
    rA = -A'*x(:,k+1)+b;
    rC = -C'*x(:,k+1)+s+d;
    rSZ = S*Z*e;
    mu = z'*s/nC;
    mu

end
% Final outputs
x_sol = x(:,k);
% y_sol = y;
% z_sol = z;
% s_sol = s;
info.iteration = k;
info.history = x;
info.alpha = alpha_value;

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test driver for Interior-ponint algorithm on nonlinear program.
% x0 = [2;3];
% [f_k,g_k,H_k] = ObjFun2(x0(:,iter))
% [c,dc,d2c] = ConFun2(x0(:,iter))
% A = dc'
% [x,lambda] = EqualityQPSolver(d2c,g_k,dc,c)
x0 = [1;3];
lambda0 = [1];
tolerance = 0.01;
maxit = 300;
[x4,info4] = SQP_IP2(@ObjFun,@ConFun,x0,maxit);
info4.history
%info4.alpha
%% Countour plot and iterations
x = -5:0.05:5;
```

```
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar
yc1 = (x+2).^2;
hold on
fill([x(yc1<=5.1)],[yc1(yc1<=5.1)],[0.7 0.7 0.7],'facealpha',0.2)
plot(info4.history(1,:),info4.history(2,:),':ms','linewidth',0.5);
axis([-5 5 -5 5])
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% fmincon test driver for solving nonlinear program.
fun = @(x)(x(1)^2 + x(2) - 11)^2 + (x(1) + x(2)^2 - 7)^2;
% variable bounds
%lb = 1.0 * ones(4);
lb = [];
%ub = 5.0 * ones(4);
ub = [];

% show initial objective
disp(['Initial Objective: ' num2str(fun(x0))]);

% linear constraints
A = [];
b = [];
Aeq = [];
beq = [];
lb = [];
ub = [];

% nonlinear constraints
nonlincon = @nlcon;

function [x fval history] = Test_5_fmincon_plot1(x0)
    history = [];
    options = optimset('OutputFcn', @myoutput);
    [x, fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlincon);
      function stop = myoutput(x,optimvalues,state);
          stop = false;
          if isequal(state,'iter')
            history = [history; x];
          end
      end
      function z = objfun(x)
        z = (x)(x(1)^2 + x(2) - 11)^2 + (x(1) + x(2)^2 - 7)^2;
      end
end

[x,val,history] = Test_5_fmincon_plot1(x0)
```

References

[1] N. Lioudis. Understanding the sharpe ratio. [Online]. Available: https://www.investopedia.com/articles/07/sharpe_ratio.asp

[2] J. Nocedal and S. J. Wright, *Numerical Optimization*. Madison, WI: Springer, 2006.

[3] R. Vanderbei, *Linear Programming: Foundations and Extensions.* New York: Springer, 2014.

[4] A. Ruszczyński and R. Vanderbei, "Frontiers of Stochastically Nondominated Portfolios," *Econometrica*, vol. 71, no. 4, pp. 1287–1297, 2003.