DTU Compute
Department of Applied Mathematics and Computer Science

Exam Assignment

02612 Constrained Optimization

Nicolaj Hans Nielsen, s184335

Kongens Lyngby 2023

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Preface

This report is prepared at DTU Compute in fulfillment of the requirements for the course 02612 - Constrained Optimization 2022. The following student has participated in the creation of the code used to make this report.

- Magne Egede Rasmussen, s183963

Kongens Lyngby, October 20, 2023

Nicolaj Hans Nielsen, s184335

Contents

CHAPTER 1

Equality Constrained Convex Quadratic Programming

In the following, we will work with the equality constrained convex QP

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2}x^T H x + g^T x \\ \text{subject to} \quad & A^T x = b \end{aligned} \tag{1.1}$$

Where H is positive definite i.e. $H \succ 0$.

1.1 Exercise 1.1

We now introduce the Lagrange function $\mathcal{L}$ as a useful way to specify the constrained optimization problem. We construct the Lagrangian function according to the generic formulation on p. 44 [1].

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x) \tag{1.2}$$

In this generic case, $\mathcal{E}$ is the set of equality constraints and $\mathcal{I}$ is the set of inequality constraints. Each of the constraints will have an associated Lagrangian multiplier λ_i. The sign in equation 1.2 is debated. We will stick with the negative sign and be very explicit if we change that, both in the introduced theory and subsequent implementations.

For our specific problem of equation 1.1, we see that the Lagrangian is:

$$\mathcal{L}(x, \lambda) = \frac{1}{2}x^T H x + g^T x - \lambda^T (A^T x - b) \tag{1.3}$$

where in this case λ is a vector. If we use the notation introduce in [2], then we denote $A \in M_{n \times m}(\mathbb{R})$ and hence $\lambda \in \mathbb{R}^m$.

1.2 Exercise 1.2

When we have a feasible solution, we want to know if it is optimal. We assess this using a set of optimality conditions. The first order optimality conditions for an EQP are the Karush-Kuhn-Tucker, KKT, conditions, Proposition 2.10 [1]:

$$\begin{array}{ll} \nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) - \sum_{i\in\mathcal{E}\cup\mathcal{I}} \lambda_i \nabla c_i(x) = 0 & \\ c_i(x) = 0 & i \in \mathcal{E} \\ c_i(x) \geq 0 & i \in \mathcal{I} \\ \lambda_i \geq 0 & i \in \mathcal{I} \\ c_i(x) = 0 \quad \vee \quad \lambda_i = 0 & i \in \mathcal{I} \end{array} \tag{1.4}$$

For standard EQPs the first order conditions are only necessary but not sufficient. However, if the program is convex, then the first order conditions are also sufficient, Section 2.5 [1]. As H is positive definite, the problem is strictly convex and hence the KKT conditions are also sufficient.

1.2.1 Understanding the Defined Constraints

The following is a derivation of the optimality constraints. It can be of interest to consider how these conditions could be derived. Therefore, we rewrite the problem slightly and cover the more general case. Let $\phi(x) : \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$ be the objective function such that in this case

$$\phi(x) = \frac{1}{2} x H x + g^\intercal x. \tag{1.5}$$

Then write $h : \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$ for the equality constraint such that

$$h(x) = A^\intercal x - b \tag{1.6}$$

Consider a small step $p \in \mathbb{R}^{n_x}$ that is small enough for the second order term in a Taylor expansion around x to be insignificant. Then

$$\phi(x) \approx \phi(x) + \nabla\phi(x)^\intercal p \tag{1.7}$$

A minimizer, x^*, of ϕ must be at a point where the immediate neighbourhood has a larger objective

$$\phi(x^* + p) \geq \phi(x^*) \tag{1.8}$$

With the Taylor expansion of 1.7, we see that $\phi(x^* + p) \geq \phi(x^*)$ if for that point p:

$$\nabla\phi(x^*)^\intercal p \geq 0 \tag{1.9}$$

Note, that in the unconstrained case, we would of cause require that $\nabla\phi(x^*)^\intercal p$=0, however, in the equality constrained problem, we must also insure that the point

$x + p$ is feasible. Therefore, consider likewise the Taylor expansion of the equality constraint. Let h_j denote the j'the equality constraint out of n_h constraints:

$$h_j\left(x+p\right) \approx h_j(x) + \nabla h_j(x)^\intercal p \tag{1.10}$$

we require $h_j(x + p) = 0$ for all constraints to be feasible. Therefore, we must require $\nabla h_j(x)^\intercal p = 0$. Now, we can formulate this conditions for all constraints by introducing the Jacobian $\nabla h_j(x) = J_h \in M_{n_m \times n_x}(\mathbb{R})$. Notice, that for the specific problem we have the Jacobian, $J_h = A^\intercal$ hence we have J_h directly. We require that:

$$J_h(x)p = 0. \tag{1.11}$$

This condition is interesting as $J_h(x)p = 0$ has a nice interpretation from linear algebra. $J_h(x)p = 0$ means that the direction p must reside in the null space J_h i.e. the Jacobian of the constraints. From the dimension theorem, corollary 2.5.4 [2], we know that

$$\text{dim null } J_h + \text{dim col } J_h = n_x. \tag{1.12}$$

Recall that dim col J_h = rank J_h. If rank $J_h = \min\{n_h, n_x\}$, then J_h has full rank. If rank $J_h = n_x$, then we have nothing to optimize as from equation 1.12 we have dim null $J_h = n_x - n_x = 0$ hence there would be no space to search in as we have just one feasible point. On the other hand, consider a problem with more design variable than constraints, $n_x > n_h$. Then rank $J_h \leq n_h$ hence from equation 1.12 we would have

$$\text{dim null } J_h \geq n_x - n_h. \tag{1.13}$$

As $n_x > n_h$, then $n_x - n_h > 0$ hence dim null $J_h > 0$ and we have a feasible space in which we can optimize the objective.

In the above we have stipulated two conditions $\phi(x^* + p) \geq \phi(x^*)$ and $J_h(x)p = 0$. However, because we work with equality constraints, then the feasible direction could be inverted and we could have; $\nabla\phi(x)^\intercal p \geq 0$ and $-\nabla\phi(x)^\intercal p \geq 0$ which only holds with equality hence we must require $\nabla\phi(x)^\intercal p = 0$. In essences, we require

$$\nabla\phi(x^*)^\intercal p = 0 \qquad \text{for all } p \text{ where} \qquad J_n(x^*)p = 0 \tag{1.14}$$

We can rewrite this requirement for the optimum in one equation [3]:

$$\nabla f(x^*) = -\sum_{j=1}^{n_h} \lambda_j \nabla h_j(x^*) \tag{1.15}$$

We now see that the gradient of the objective function must be a linear combination of the gradients of the constraints. These scalars λ_j are the Lagrangian multipliers.

Before we turn to the Lagrangian, we conclude the following from the analysis above:

- For x^* t be an optimum, we must have

$$\nabla f(x^*) = -\sum_{j=1}^{n_h} \lambda_j \nabla h_j(x^*) \tag{1.16}$$

  and of cause still

$$h(x^*) = 0 \tag{1.17}$$

- Before we start anything, we must insure that we have more design variables than constraints such that $n_x \geq n_h$.
- It would be beneficial to have consistent constraints which means that J_h has full rank, rank $J_h = n_h$

Let us turn back to the Lagrangian function

$$\mathcal{L}(x, \lambda) = f(x) + h(x)^\intercal \lambda \tag{1.18}$$

If we take the gradient w.r.t. x and λ and set the equations equal to 0, we obtain the following:

$$\begin{aligned} \nabla_x \mathcal{L}(x, \lambda) &= \nabla_x f(x) + J_h(x)^\intercal \lambda = 0 \\ \nabla_\lambda \mathcal{L}(x, \lambda) &= h(x) = 0 \end{aligned} \tag{1.19}$$

which indeed are the first order conditions derived just before. This is why the Lagrangian is so useful as it allows us to reformulate the problem.
We now consider the specific problem at hand and write the conditions in matrix notation:

$$\begin{aligned} \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda) \\ \nabla_\lambda (x, \lambda) \end{bmatrix} &= \begin{bmatrix} \nabla_x f(x) + J_h(x)^\intercal \lambda \\ h(x) \end{bmatrix} \\ &= \begin{bmatrix} x^\intercal H + g - A\lambda \\ -A^\intercal + b \end{bmatrix} \\ &= \begin{bmatrix} H & -A \\ -A^\intercal & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} + \begin{bmatrix} g \\ b \end{bmatrix} \end{aligned} \tag{1.20}$$

We know that we have to set the above equation equal to zero. When we do that, we observed that we now have a system of equations to solve

$$\begin{bmatrix} H & A \\ A^\intercal & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} g \\ b \end{bmatrix} \tag{1.21}$$

Thus we have now written the KKT conditions in matrix vector format. The matrix of the system is the KKT matrix which we will denote $K \in M_{(n_x+n_h)\times(n_x+n_h)}(\mathbb{R})$. As the KKT conditions are necessary and sufficient for the problem, we now just have to find a way to solve a system of linear equations. In the following, we will present different ways to do that efficiently.

Before we introduce anything, notice that as soon as we introduce a constraint, n_h, then we would have a eigenvalue of 0 hence K would be indefinite [4]. Therefore, we have to consider methods that allow a factorization of such a matrix.

1.3 Exercise 1.3

In the following, we will solve these problems directly using various methods based on different matrix factorization. The driver for this exercise and subsequent exercise can be found in appendix A.3.

1.3.1 LU factorization

To solve a linear system, we have to find the inverse of K. First, we consider an LU factorization of K into a lower triangular matrix L and an upper triangular matrix U. To ensure numerical stability of the procedure, we will use LU factorization with partial pivoting, PLU, as suggested in the lecture slides, 05B_EqualityConstrainedQP.

$$PK = LU \tag{1.22}$$

where P is a permutation matrix. As K could be sparse due to constraints not involving all design variables and of cause because the lower right corner of K is zeros, it would be beneficial to make a sparse representation of the matrices. Here we only save the non-zero values and indices of the matrices involved. Below, you will find the the Matlab code and pseudo code of the two implementations.

Algorithm 1 EQP solver with LU-factorization

procedure EQUALITYQPSOLVERLU(H, g, A, b)
  K $\leftarrow \begin{bmatrix} H & -A \\ -A^\top & 0 \end{bmatrix}$
  $L, U \leftarrow$ lu(K), ▷ LU-factorization of K
  $\begin{bmatrix} x \\ \lambda \end{bmatrix} = U^{-1}\left(L^{-1}\left(-\begin{bmatrix} g \\ b \end{bmatrix}\right)\right)$ ▷ Forward & backward substitution
  return x, λ
end procedure

```
function [x, lambda] = EqualityQPSolverLUdense(H, g, A, b)
% EqualityQPSolverLUdense  dense LU solver
%
%           min   x'*H*x+g'x
%            x
%           s.t. A x  = b
%
```

```
%
% Syntax: [x, lambda] = EqualityQPSolverLUdense(H,g,A,b)
%
%         x                 : Solution
%         lambda            : Lagrange multipier

% Created: 02.05.2022
% Author: Nicolaj Hans Nielsen

%%
    % Construct KKT Matrix
    KKT = [H -A;-A', zeros(size(A,2), size(A,2))];

    % Fatorize the KKT matrix
    [L,U,p] = lu(KKT,'vector');

    % Solve
    rhs = -[g;b];
    clear sol;
    sol = U \ (L \ ( rhs(p)));
    x = sol(1:size(H,1));
    lambda = sol(size(H,1)+1:size(H,1)+size(b,1));
end
```

Listing 1.1: Dense LU-factorization solver for EQP

```
function [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
% EqualityQPSolverLUsparse   Sparse LU solver
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b
%
%
% Syntax: [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
%
%         x                 : Solution
%         lambda            : Lagrange multipier

% Created: 02.05.2022
% Author: Nicolaj Hans Nielsen

%%
    % Construct the sparse KKT matrix
    KKT = sparse([H -A;-A', zeros(size(A,2), size(A,2))]);

    % Fatorize the sparse KKT matrix
    [L,U,p] = lu(KKT,'vector');

    % Solve for x and lambda
    rhs = [g;b];
    sol = U \ (L \ ( -rhs(p)));
    x = sol(1:size(H,1));
```

```
    lambda = sol(size(H,1)+1:size(H,1)+size(b,1));
end
```

Listing 1.2: Sparse LU-factorization solver for EQP

1.3.2 LDL factorization

In the following, we will consider the LDL decomposition. This method is slightly related to the Cholesky decompostion, however, it allows for decomposition of matrices that are not positive-definite. The factorization is:

$$K = LDL^*, \tag{1.23}$$

where L is a lower triangular matrix, and D a diagonal matrix. The LDL factorization requires only half the computation of LU factorization [5], hence we would expect this implementation to be faster. The implementation is both implemented with a sparse and dense matrix representation. In the Matlab implementation it should be noted that we need to permute the solution vector accordingly.

Algorithm 2 EQP Solver Constructed with LDL-factorization

procedure EQUALITYQPSOLVERLDL(H, g, A, b)
  K $\leftarrow \begin{bmatrix} H & -A \\ -A^\top & 0 \end{bmatrix}$
  $L, D \leftarrow$ ldl(K), ▷ LDL-factorization of K
  $\begin{bmatrix} x \\ \lambda \end{bmatrix} = L^{-\top}\left(D^{-1}\left(L^{-1}\left(-\begin{bmatrix} g \\ b \end{bmatrix}\right)\right)\right)$ ▷ Forward & backward substitution
  return x, λ
end procedure

```
function [x, lambda] = EqualityQPSolverLDLdense(H,g,A,b)
% EqualityQPSolverLDLdense    Dense LDL solver
%
%            min   x'*H*x+g'x
%             x
%            s.t. A x  = b
%
%
% Syntax: [x, lambda] = EqualityQPSolverLDLdense(H,g,A,b)
%
%          x                  : Solution
%          lambda             : Lagrange multipier

% Created: 02.05.2022
% Author: Nicolaj Hans Nielsen

```

```
%%
    % Construct KKT Matrix
    KKT = [H -A;-A', zeros(size(A,2), size(A,2))];

    % Fatorize the KKT matrix
    [L,D,p] = ldl(KKT,'lower','vector');

    % Solve
    clear sol;
    rhs =  -[g;b];
    % note: the permutation vector p of the solution:
    sol(p) = L' \ (D \ (L \rhs(p)));
    x = sol(1:size(H,1));
    lambda = sol(size(H,1)+1:size(H,1)+size(b,1));
end
```

Listing 1.3: Dense LDL-factorization solver for EQP

```
function [x, lambda] = EqualityQPSolverLDLsparse(H, g, A, b)
% EqualityQPSolverLDLsparse   Sparse LDL solver
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b
%
%
% Syntax: [x, lambda] = EqualityQPSolverLDLsparse(H,g,A,b)
%
%         x                 : Solution
%         lambda            : Lagrange multipier

% Created: 02.05.2022
% Author: Nicolaj Hans Nielsen

%%
    % Construct the sparse KKT matrix
    KKT = sparse([H -A;-A', zeros(size(A,2), size(A,2))]);

    % Fatorize the sparse KKT matrix
    [L,D,p] = ldl(KKT,'lower','vector');

    % Solve for x and lambda
    rhs =  -[g;b];
    % note: the permutation vector p of the solution:
    sol(p) = L' \ (D \ (L \rhs(p)));
    x = sol(1:size(H,1));
    lambda = sol(size(H,1)+1:size(H,1)+size(b,1));
```

Listing 1.4: Sparse LDL-factorization solver for EQP

1.3.3 Range-Space factorization

In this method, we use the cholesky factorization of H. This is possible as we know H is positive definite. The method works well when there is only a limited set of constraints compared to the number of design variables. We refer to section 16.2, [4] where it is described under the other commonly used name *Schur-Complement method*.

The pseudo code is provided matches closely that of the provided example in, 05B_EqualityConstrainedQP:

Algorithm 3 EQP solver constructed with range-space factorization

procedure EQUALITYQPSOLVERRANGESPACE(H, g, A, b)
  $L = \text{chol}(H)$ ▷ Cholesky-factorization of A
  $h_g = L^{-1}\left(L^{-T} g\right)$
  $h_a = L^{-1}\left(L^{-T} A\right)$
  $\lambda = \left(A^\intercal h_a\right)^{-1}\left(A^\intercal h_g + b\right)$
  $x = h_a \lambda - h_g$
  return x, λ
end procedure

Below is the code used to make the implementations:

```
function [x, lambda] = EqualityQPSolverRangeSpace(H,g,A,b)
% EqualityQPSolverRangeSpace   Range Space solver
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b
%
%
% Syntax: [x, lambda, time_R] = EqualityQPSolverRangeSpace(H,g,A,b)
%
%          x                 : Solution
%          lambda            : Lagrange multipier

% Created: 03.05.2022
% Author : Nicolaj Hans Nielsen, Technical University of Denmark

%%
    % Factorize H using cholesky decomposition
    L=chol(H);

    % form intermediate qunatities
    h_g = L \(L'\g);
    h_a = L \(L'\A);

    % compute lambda and x
    lambda = (A'*h_a)\(b+A'*h_g);
```

```
    x = h_a*lambda-h_g;
end
```

Listing 1.5: Range-space factorization solver for EQP

1.3.4 Null Space Method

The null space method builds on the idea that we can rephrase our equality constraint problem into that of an unconstrained problem in the null space of A. We refer to exercise 2 where we go through the natural connection to the null space. For now we list the results related to the implementation. Note that the algorithm is built on a QR factorization of A, see e.g. section 6.5, [2]. Explicitly, the QR-factorization is:

$$\begin{aligned} A &= V \begin{bmatrix} R \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} Q_r & Q_n \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} \end{aligned} \tag{1.24}$$

Where Q_R has orthonormal columns that constitutes a basis for the range space of A. We require A to be full rank and $Q_n^\intercal H Q_n$ be positive definite where Q_n is a matrix with columns that spans the null space of A. The methods is useful when the number of constraints is close to the number of design variables, chapter 16.2 [4].

The pseudo code is provided below where we have used the a notation that follows quire closely that of lecture slides, 05B_EqualityConstrainedQP:

Algorithm 4 EQP solver constructed with null-space factorization

procedure EQUALITYQPSOLVERNULLSPACE(H, g, A, b)
  $\begin{bmatrix} Q_R & Q_N \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = \mathrm{qr}(A)$ ▷ QR-factorization of A
  $x_y = (R^\intercal)^{-1} b$
  $x_z = -\left(Q_n^\intercal H Q_n\right)^{-1} \left[Q_n^\intercal \left(H Q_r x_y + g\right)\right]$
  $x = Q_r x_y + Q_n x_z$
  $\lambda = (R^\intercal)^{-1} Q_r \left(g + H_x\right)$
  return x, λ
end procedure

The Matlab code is provided below:

```
function [x,lambda] = EqualityQPSolverNullSpace(H,g,A,b)
% EqualityQPSolverNullSpace    Null Space solver
%
%            min   x'*H*x+g'x
%             x
```

```
%           s.t. A x  = b
%
%
% Syntax: [x,lambda,time_N] = EqualityQPSolverNullSpace(H,g,A,b)
%
%         x                 : Solution
%         lambda            : Lagrange multipier

% Created: 03.05.2022
% Author : Nicolaj Hans Nielsen, Technical University of Denmark

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

    % Make QR factorization
    [Q,R] = qr(A,'vector');

    % devide into q_r and q_n
    Q_r = Q(:,1:m);
    Q_n = Q(:,m+1:n);
    R = R(1:m,1:m);

    % calculate x_r
    x_y = (R'\b);

    % create help variable
    intermidiate_inv = Q_n'*H*Q_n;
    L = chol(intermidiate_inv);

    x_z = -L \ (L'\(Q_n'*(H*Q_r*x_y+g)));
    x = Q_r*x_y+Q_n*x_z;
    lambda = R\Q_r'*(g+H*x);
end
```

Listing 1.6: Null-space factorization solver for EQP

1.3.5 The Interface for the Set of Solvers

The interface for the set of solvers has been included in appendix A.1.

1.4 Exercise 1.4

In the following, we will test the obtained result from each of the presented solver and in their sparse and dense forms. We will make the check on the following problem:

```
H =
 5.0000    1.8600    1.2400    1.4800   -0.4600
 1.8600    3.0000    0.4400    1.1200    0.5200
 1.2400    0.4400    3.8000    1.5600   -0.5400
 1.4800    1.1200    1.5600    7.2000   -1.1200
-0.4600    0.5200   -0.5400   -1.1200    7.8000
```

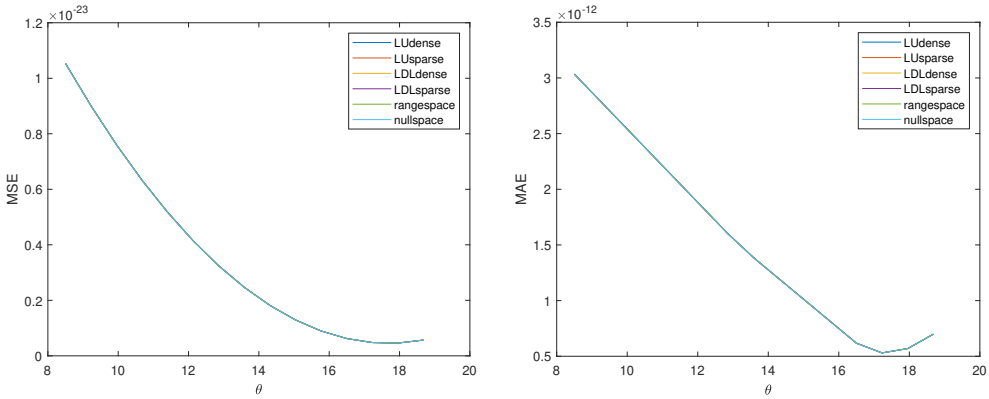
```
g =
-16.1000
 -8.5000
-15.7000
-10.0200
-18.6800
```

```
A =
16.1000    1.0000
 8.5000    1.0000
15.7000    1.0000
10.0200    1.0000
18.6800    1.0000
```

```
b =
   θ
   1
```

The value θ in b we will vary to test the implementations for different values. In our case, we will use 15 equidistant points in the interval [8.50:18.68]. The driver for this exercise can be seen in appendix A.3

Below, we see that for all of our implementations, the mean absolute error is below 10^{-10} while for mean squared error, we are down to 10^{-23} hence we conclude that the solvers all seem to be correct. Below we have depicted the errors as a function θ.



(a) Mean Squared Error **(b)** Mean absolute errors.

Figure 1.1: The errors between the implemented solvers and quadprog solutions. As the scale of the y-axis indicated, all solvers are correct.

In the table below we provide a set of the computed solutions:

$\theta =$	8.5000	11.0450	13.5900	16.1350	18.6800
$\phi =$	-7.2816	-10.2037	-12.8724	-15.2877	-17.4497
$x =$	-0.2250	-0.0681	0.0888	0.2458	0.4027
	0.8575	0.5809	0.3043	0.0277	-0.2489
	0.1712	0.2368	0.3024	0.3680	0.4336
	0.1756	0.1376	0.0995	0.0615	0.0234
	0.0207	0.1128	0.2049	0.2971	0.3892

Table 1.1: Objetive, ϕ, and minimizer, x, as a function of θ

1.5 Exercise 1.5

Given that the solvers seems correct, we will now test the implementations on a new problem. Specifically, we will focus on the problem from week 5, denoted the recycle problem. In the following, we will define the problem.

1.5.1 Recycle Problem

$$
\begin{aligned}
\min_{x} \quad & \phi = \frac{1}{2}\sum_{i=1}^{n+1}(x_i - \bar{x})^2 \\
\text{s.t.} \quad & -x_1 + x_n = -d_0 \\
& x_i - x_{i+1} = 0, \qquad\qquad i = 1, 2, \ldots, n-2 \\
& x_{n-1} - x_n - x_{n+1} = 0
\end{aligned}
\tag{1.25}
$$

where $\bar{x}$ and d_0 are parameters of the problem. We see that the number of constraints indeed scales with n in equation 1.25 but still the problems remain an EQP. The objective also only contains terms of at most second order hence we should be able to solve this problem with all the solvers introduced. We refer to week5 for thorough introduction to the problem with graphical representation etc. For now, we phrase the matrix form of the problem. To make the make the system match that of equation 1.1, we locate H and g:

$$
H = I_{((n+1)\times(n+1))}, \quad g = -\bar{x}_{(n+1)} \tag{1.26}
$$

where I is the identity matrix. Consider now the constraints which we can write can write with $A \in M_{((n+1)\times n)}(\mathbb{R})$ and $b \in \mathbb{R}^n$

$$
A = \begin{bmatrix}
-1 & 1 & 0 & 0\cdots & 0 & 0 & 0 \\
0 & -1 & 1 & 0\cdots & 0 & 0 & 0 \\
0 & 0 & -1 & 1\cdots & 0 & 0 & 0 \\
0 & 0 & 0 & 0\ddots & 0 & 0 & 0 \\
0 & 0 & 0 & 0\cdots & 1 & 0 & 0 \\
0 & 0 & 0 & 0\cdots & -1 & 1 & 0 \\
0 & 0 & 0 & 0\cdots & 0 & -1 & 1 \\
1 & 0 & 0 & 0\cdots & 0 & 0 & -1 \\
0 & 0 & 0 & 0\cdots & 0 & 0 & -1
\end{bmatrix}, \quad
b = \begin{bmatrix} -d_0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\tag{1.27}
$$

With the explicit formulation of the matrices, we can use the same framework from before and test the solvers. Specifically, we will now focus on the time it takes the solvers to solve this system as we introduce more design variables and constraints. We have chosen $\bar{x} = 0.2$ and $d_0 = 1$. We will test the problem for n from 100 to 3000. We will construct experiment with a vector n

$$
n = \begin{bmatrix} 100 & 200 & \ldots & 1000 & 1500 & 2000 & \ldots & 3000 \end{bmatrix}^{\intercal} \tag{1.28}
$$

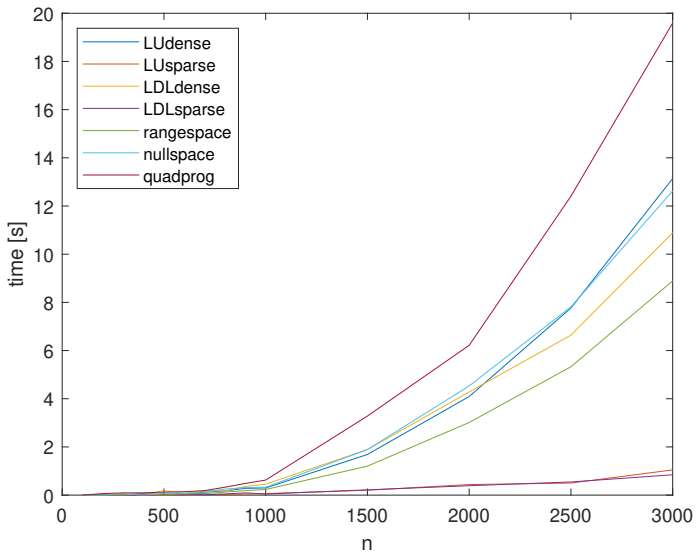


Figure 1.2: CPU-time as we increase n in the recycle problem

In figure 1.2, we see three different overall patterns. The built-in quadprog tends to take longer time to solve the problem. Especially, when it scales we see that the implementations with dense matrices becomes slow, and the solvers that utilized the sparse matrix structure are fastest.

We were initially surprised that the *quadprog* took longer than our solvers, however, *quadprog* is a very versatile solver hence mush have an additional overhead of test it runs to figure out how to solve the passed problem must efficiently. Our solvers are more informed and implemented directly to a problem where we know they could be applied.

1.5.2 Underlying Matrix Factorization

Each methods has its own specific way to factorize either the KKT or the H matrix. After that some subsequent calculations follows. We wondered how much of the increased time could be explained by the different factorization alone. This time will use $n \in [200, 400, 600, \ldots, 3000]^\intercal$.

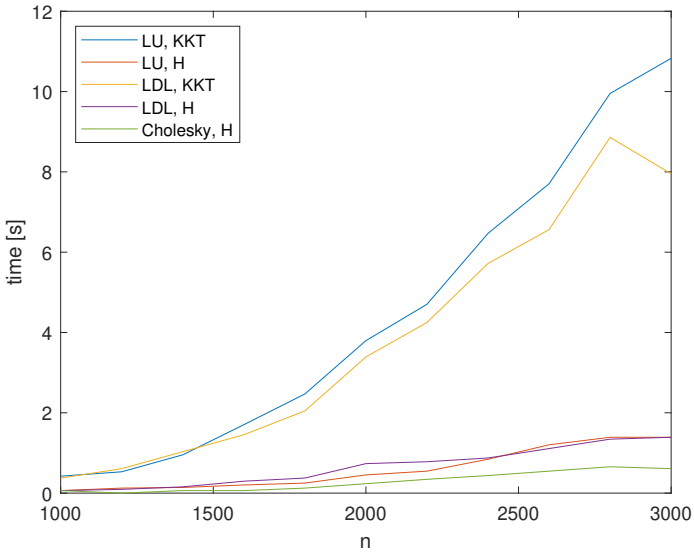


Figure 1.3: CPU-time as we increase n in the recycle problem with focus on decomposition method only.

In figure 1.3, we see that indeed it is the factorization of the KKT matrix that is most expensive which can partly explain why we see the patterns in figure 1.2.

1.5.3 Scalability in number of Constraints

We figured it would be interesting to test the implementation on problems with different design matrices and constraints. Therefore, we created a program that can generate new EQPs, see appendix section A.2. Here we simply have to ensure that we create a positive definite symmetric matrix H and a matrix A that has full rank. We used some inputs from Matlab fora to which we have left the references in the code. Here we introduced as many constraints as number of design variables and tested with n:

$$n = \begin{bmatrix} 100 & 200 & \dots & 1000 & 1500 & 2000 & \dots & 3000 \end{bmatrix}^\intercal \tag{1.29}$$

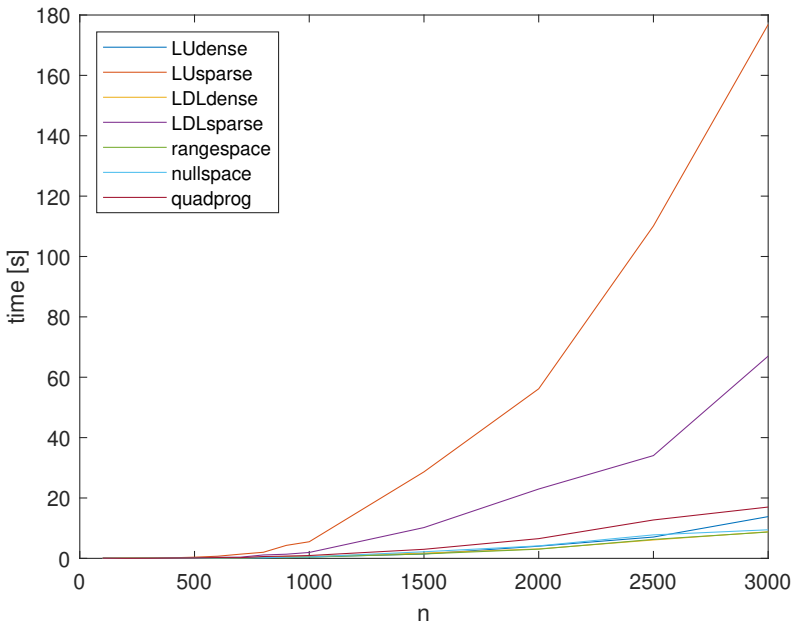


Figure 1.4: CPU-time as we increase n and generate randomly EQPs. We introduce as many constraints as design variables.

In figure 1.4, we see quite an interesting pattern. When the number of constraints is equal to the number of design variables, then the sparse matrix methods seems to take much longer. Indeed, we also see that the *quadprog* takes longer than the rest of the methods but this is on a completely different scale. If we consider only the rangespace and nullspace methods, it is quite interesting that they lie this close. We see that the nullspace methods lies slightly above the rangespace method. This is indeed in accordance with results given in [4] but not of the magnitude one could expect.

CHAPTER 2

Quadratic Program

In the following, we will consider a quadratic program of the following form:

$$\begin{array}{ll} \min_x & \phi = \frac{1}{2}x'Hx + g'x \\ \text{s.t.} & A'x = b \\ & l \leq x \leq u \end{array}$$

where A has full column rank. We will rewrite the problem formulation slightly such that is matches the form introduce in the notation introduce in *Convex Quadratic Programming Primal-Dual Interior Point Algorithm*, [6]. We want the inequality constraints to be of the form $C^\intercal x \geq d$. Therefore, let I be the identity matrix, then we can introduce:

$$C = \begin{bmatrix} I & -I \end{bmatrix}^\intercal, \qquad d = \begin{bmatrix} l, -u \end{bmatrix}^\intercal \tag{2.1}$$

such that we phrase the problem as follows:

$$\begin{array}{ll} \min\limits_x & \phi = \dfrac{1}{2}x^\intercal Hx + g^\intercal x \\ \text{s.t.} & A^\intercal x = b \\ & \begin{bmatrix} I & -I \end{bmatrix}^\intercal x \geq \begin{bmatrix} l \\ -u \end{bmatrix}. \end{array} \tag{2.2}$$

2.1 Exercise 2.1

We now let y be a vector of Lagrange multipliers for the equality constraints and z a vector of Lagrange multipliers for the inequality constraints. The Lagrangian matches quite closely the ones introduced in exercise 1.1. In short-hand notation we can write the Lagrangian as:

$$L(x, y, z) = \frac{1}{2}x^\intercal Hx + g^\intercal x - y^\intercal \left(A^\intercal x - b\right) - z^\intercal \left(C^\intercal x - d\right) \tag{2.3}$$

In the subsequent sections, we will cover the intuition and in-dept explanations of the meaning of each of the terms in the specified Lagrangian above.

2.2 Exercise 2.2

The necessary and sufficient conditions can be seen very explicitly as they are presented in section 2.4 [1].

$$\nabla_x \mathcal{L}(x,y,z) = \nabla f(x) - \sum_{i\in\mathcal{E}} y_i \nabla c_i(x) - \sum_{i\in\mathcal{I}} z_i \nabla c_i(x)$$

$$\begin{aligned} \nabla_y \mathcal{L}(x,y,z) = c_i(x) &= 0, \quad i \in \mathcal{E} \\ \nabla_z \mathcal{L}(x,y,z) = c_i(x) &\geq 0, \quad i \in \mathcal{I} \\ z_i &\geq 0, \quad i \in \mathcal{I} \\ c_i(x) z_i &= 0, \quad i \in \mathcal{I} \end{aligned} \tag{2.4}$$

In the following, we will cover the intuition behind the optimality conditions for the QP-problems.

In the beginning of the chapter, we showed that we could write the problem in a more generic way to match that of the slides on Quadratic Optimisation Interior Point Algorithms. The next step is to extend the framework slightly and introduce slack variables such that for each inequality conditions we define

$$-C_j^\intercal s + d = 0 \qquad \text{where} \qquad s \geq 0 \tag{2.5}$$

With this additional parameter, the Lagrangian is slightly different, however, we are still able to construct a Lagrangian where optimal points would be stationary points of the Lagrangian. For each of the optimality conditions, we can now define a specific residual as in [6]:

$$\begin{aligned} r_L &= Hx + g - Ay - Cz = 0 \\ r_A &= -A'x + b = 0 \\ r_C &= -C'x + s + d = 0 \\ z &\geq 0 \\ s &\geq 0 \\ s_i z_i &= 0 \quad i = 1, 2, \ldots, n_c \end{aligned}$$

We will now specify why each of the conditions accounts for and why the are needed

- r_L: Can be seen as the conditions that ensures that the optimum x^* is a minimum of f in the feasible space of the constraints.
- r_A: Ensures that x^* is feasible with respect to the equality constraints.
- r_C: Ensures that x^* is feasible with respect to the inequality constraints after the introduction of the slack variables, s.

- $s \geq 0$: We introduce the slack variable to the constraints $c_j^\intercal x - d \geq 0$ and such that $-c_j^\intercal x + s + d = 0$ and for that to be true we must require, $s \geq 0$. If $s = 0$ such that $-c_j^\intercal x + d = 0$, the constraints is said to be active.

- $z \geq 0$ and $s_i z_i = 0 \quad i = 1, 2, \ldots, n_c$: These need an introduction to the way of handling inequality constraints.

In the following, we will use the notation and approach used in the derivation of the optimality conditions for the equality constrained programs. The derivations are inspired by [3].

Let $g_h(x)$ be the j'th inequality constraint before an introduction of slack variables. As in section 1.2, we will now consider a Taylor expansion around a point and a small step p:

$$g_j(x+p) \approx g_j(x) + \nabla g_j(x)^\intercal p \leq 0 \qquad \text{for} \qquad i = 1, \ldots, n_g \tag{2.6}$$

If $g_j(x) < 0$, then the constraint is inactive and p is assumed small enough that the constraint remains inactive after p. If $g_j(x) = 0$, then the constraint is active and we need to take the constraint into consideration. The argument is similar to that of the equality constraint though we are not as restrictive. We require $\nabla g_j(x)^\intercal p \leq 0$ for all j active constraints which in matrix notation would be $J_g(x)p \leq 0$.
$J_g(x)p \leq 0$ means that we require the direction p to be in the intersection of the feasible subspace of each active inequality constraint, see figure 2.1.

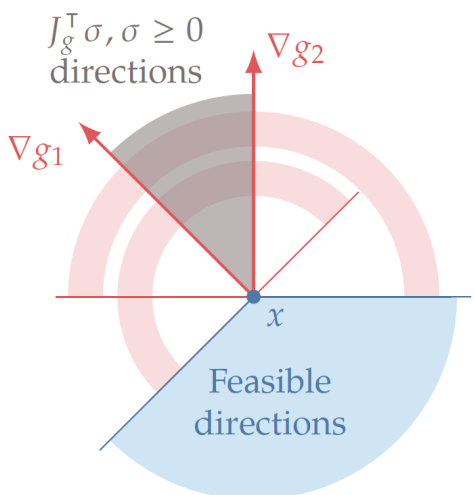


Figure 2.1: Intersection between the feasible subspace of active inequality constraints will span the space of possible decent directions. The figure is a replica of Fig. 5.17 [3].

A great and interesting discussion of the cone formed by the active constraints and the polar cone formed by the feasible direction can be found in section 5.3.2, [3]. We will instead devote our attain to the question of how we can determine if there exists a direction p in the intersection between the feasible subset of the active constraints and the subset of possible decent directions. To determine this, we use Farkas' lemma that states that only one of the following two possibilities can occur:

1. There exits a feasible decent direction. This happens when p is in the feasible space $J_g p \geq 0$ and the direction reduces the objective $\nabla f^\intercal p < 0$.

2. There exists a z where $J_g^\intercal = z - \nabla f$ with $z \geq 0$. This can only happen if the first possibility does not occur.

In the latter case, the we are at a minimum and require $\nabla f + J_g(x)^\intercal z = 0$ with $z \geq 0$. Notice the similarity with the optimality condition of the equality constraints program that

Remember, that J_g is only the Jacobian of the active constraints hence it requires a substantial amount of bookkeeping. This is where the slack variables comes in handy. We introduce the slack variables, s, and we know that if $s_j = 0$, then the constraint is active, and if $s_j > 0$ it is inactive. If it is inactive, we should consider it in J_g and the Lagrangian multiplier for that constraint should be 0.

We have formulated all of this in the optimality conditions. In the conditions $s \geq 0$, $z \geq 0$, and the *complimentary slack condition* $s_i z_i = 0$ for all i. If $s_j = 0$ the constaint is active and $\lambda_j > 0$. If $s_j > 0$, then the constraint is not active and $\lambda_j = 0$.

To write the complimentary slack condition in a matrix-vector notation, we introduce two diagonal matrices $S = \operatorname{diag}(s_1, s_2, \ldots, s_{n_h})$, and $Z = \operatorname{diag}(z_1, z_2, \ldots, z_{n_h})$ and a vector of ones $e = \underbrace{[1, 1, \ldots, 1]^\intercal}_{n_h}$ such that we can express $s_i z_i = 0$ for all i as

$$r_{SZ} = SZe = 0 \tag{2.7}$$

Thus, we have now explained why this extra optimality condition is included.

Necessary and sufficient conditions If H is positive definite, then the problem is convex and it then follows from Section 2.5 [1] that the first order conditions are necessary and sufficient. If H is not positive definite, then we need to consider second order conditions.

2.3 Exercise 2.3

In the following, we will introduce Mehrotra's Predictor-Corrector Algorithm which builds heavily on the first order conditions introduced in the section 2.2 above.

First, we need to see how Newton's method can be applied to find a point that satisfies the optimality condition. Therefore, we introduce the optimality conditions

in term of $F : M_{(2n_x+2n_m)}(\mathbb{R}) \mapsto M_{(2n_x+2n_m)}(\mathbb{R})$:

$$\begin{aligned} F(x,y,z,s) &= \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} Hx + g - Ay - Cz \\ -A'x + b \\ -C'x + s + d \\ SZe \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, (s,z) \geq 0. \end{aligned} \tag{2.8}$$

We see that we can solve $F = 0$ with $(s,z) \geq 0$ using a Newton like method. Let $J(x,y,z,s)$ be the Jacobian of F and introduce $\Delta_F = [\Delta x, \Delta y, \Delta z, \Delta s]$. Then we can formulate:

$$J(x,y,z,s)\Delta_F = -F(x,y,z,s) \tag{2.9}$$

which can be formulated explicitly in terms of the problem at hand as:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A' & 0 & 0 & 0 \\ -C' & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L = Hx + g - Ay - Cz \\ r_A = -A'x + b \\ r_C = -C'x + s + d \\ r_{SZ} = SZe \end{bmatrix} \tag{2.10}$$

in the Newton step, we would solve for Δ_F and update

$$\begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix} + \alpha \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix}, \quad (z,s) \geq 0$$

where we should note that $(z,s) \geq 0$ and we have introduced a line search parameter $\alpha \in]0,1[$. This is a pure Newton update sometimes denoted the affine scaling direction, section 14.1 [4]. Now, we want fast convergence which would imply $\alpha >> 0$, however, this would often lead to a violation of $(z,s) \geq 0$, see section 14.1 [4]. To avoid this, we will use path-following primal-dual interior-point methods. We will briefly introduce key concepts and refer to [4] and [7] for a thorough treatment of the topic. First, introduce the central path $\mathcal{C}$ of strictly feasible points, [4]. We parameterize this path with a scale parameter $\tau > 0$:

$$\mathcal{C} = \{(x_\tau, y_\tau, z_\tau, s_\tau) : \tau > 0\}. \tag{2.11}$$

This τ replaces the complementary slack condition such that

$$F\left(x_\tau, y_\tau, z_\tau, s_\tau\right) = \begin{bmatrix} Hx + g - Ay_\tau - Cz_\tau \\ -A'x_\tau + b \\ -C'x_\tau + s_\tau + d \\ S_\tau Z_\tau e^2 \end{bmatrix} = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \tau e \end{bmatrix}, \quad (z, s) \geq 0$$

For each $\tau > 0$, there exits a uniquely defined set $(x_\tau, y_\tau, z_\tau, s_\tau)$, as stipulated in section 14.1 [4]. This means that the central path is uniquely defined for all $\tau > 0$ and is always strictly feasible. If we let $\tau \to 0_+$, then $\mathcal{C}$ converges towards a solution. We will not follow the central path directly. Instead, we will let it guide us towards the solutions where we maintain $(s, z) > 0$ i.e. we let it impose some bias on our step. To control the amount of bias towards the central path, we introduce a new measure. This is the duality measure

$$\mu = \frac{1}{n_c} \sum_{i=1}^{n} z_i s_i = \frac{s^\intercal z}{n_c}. \tag{2.12}$$

Next, we introduce a controlling parameter, $\sigma \in [0, 1]$, which is denoted the *centering parameter* whic is defined as

$$\tau = \sigma\mu. \tag{2.13}$$

We now see that the value of σ affects heavily the step we take. If $\sigma = 1$, then we take the centering direction which would bias the step heavily towards the central path. This step will not heavily reduce the duality measure, however, it would insure that we are not close to violating (s, z). This would allow for a great reduction in the subsequent step, [4]. Conversely, if $\sigma = 0$, we take a full affine scale step that will reduce the duality measure greatly but would violate $(s, z) > 0$ and force us to choose $\alpha << 1$. A way to find a balance between the two extremes is by use of predictior-corrector algorithms. The key concepts it that we alternate between a step in a heavily centered direction and a step heavily towards the affine-scaling diction. The steps are called the corrector and predictor step respectively.

2.3.1 Mehrota's algorithm

This framework can be modifies to obtain better performance. We will introduce two modifications that are vital for Mehrota's algorithm [6].

1. A better way to adaptive set σ
2. an extra corrector step to ensure the solutions follows more closely the primal-dual solution set

To be explicit about the modifications, we will introduce each step. The section is heavily inspired by [7, Ch. 10] and [6]. First consider each step and modification explicitly. Consider first the affine step where $\sigma = 0$:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A' & 0 & 0 & 0 \\ -C' & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \\ \Delta z^{aff} \\ \Delta s^{aff} \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} \end{bmatrix} \tag{2.14}$$

We now find the maximal step length that we can take which would still make the problem feasible for both the primal and dual variables:

$$\begin{aligned} \alpha_{\text{aff}}^{\text{pri}} &= \arg\max \left\{ \alpha \in [0,1] \mid z + \alpha \Delta z^{\text{aff}} \geq 0 \right\} \\ \alpha_{\text{aff}}^{\text{dual}} &= \arg\max \left\{ \alpha \in [0,1] \mid s + \alpha \Delta s^{\text{aff}} \geq 0 \right\} \end{aligned} \tag{2.15}$$

We we were to take this full step, we could calculate the duality measure:

$$\mu_{aff} = \frac{\left(z + \alpha_{\text{aff}}^{\text{pri}} \Delta z^{\text{aff}}\right) \intercal \left(s + \alpha_{\text{aff}}^{\text{dual}} \Delta s^{\text{aff}}\right)}{n_c}. \tag{2.16}$$

We can now compare this duality measure to the duality measure without this step, μ. Now if, we see a substantial reduction $\mu_{aff} << \mu$, then it would be beneficial to move towards the full affine direction. If the improvement is small, we should take a more centered step and hopefully the next step in the affine direction will dramatically reduce the duality measure. To decide how much to favor each step type, we introduce the following heuristics for our controlling parameter:

$$\sigma = \left(\frac{\mu_{aff}}{\mu} \right)^3 \tag{2.17}$$

Now that we have found an adequate σ, we can calculate the centering step as:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A' & 0 & 0 & 0 \\ -C' & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{cen} \\ \Delta y^{cen} \\ \Delta z^{cen} \\ \Delta s^{cen} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \sigma \mu e \end{bmatrix} \tag{2.18}$$

we could now just update

$$\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \\ \Delta z^{aff} \\ \Delta s^{aff} \end{bmatrix} + \begin{bmatrix} \Delta x^{cen} \\ \Delta y^{cen} \\ \Delta z^{cen} \\ \Delta s^{cen} \end{bmatrix} \tag{2.19}$$

This is the first modification. However, there is a problem and room for improvement that the second modification mitigates and leverages. This is the corrector step. When we calculate the affine step, we linearize around F and hence introduce approximation errors. Explicitly, consider the errors introduce in the calculation of pairwise products:

$$\begin{aligned} &\left(x_i + \Delta x_i^{\text{aff}}\right)\left(s_i + \Delta s_i^{\text{aff}}\right) \\ &\quad = x_i s_i + x_i \Delta s_i^{\text{aff}} + s_i \Delta x_i^{\text{aff}} + \Delta x_i^{\text{aff}} \Delta s_i^{\text{aff}} = \Delta x_i^{\text{aff}} \Delta s_i^{\text{aff}} \end{aligned} \tag{2.20}$$

hence suddenly this product might no longer be 0. To compensate for that, we calculate the corrector step:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A' & 0 & 0 & 0 \\ -C' & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{cor} \\ \Delta y^{cor} \\ \Delta z^{cor} \\ \Delta s^{cor} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\Delta S^{aff} \Delta Z^{aff} e \end{bmatrix} \tag{2.21}$$

Where $\Delta S^{aff} = \operatorname{diag}\left(\Delta s_1^{\text{aff}}, \Delta s_2^{\text{aff}}, \ldots, \Delta s_n^{\text{aff}}\right)$ and likewise for ΔZ^{aff}. Now:

$$\begin{aligned} &\left(x_i + \Delta x_i^{\text{aff}} + \Delta x_i^{\text{cor}}\right)\left(s_i + \Delta s_i^{\text{aff}} + \Delta s_i^{\text{cor}}\right) \\ &\quad = \Delta x_i^{\text{aff}} \Delta s_i^{\text{cor}} + \Delta x_i^{\text{cor}} \Delta s_i^{\text{aff}} + \Delta x_i^{\text{cor}} \Delta s_i^{\text{cor}} \end{aligned} \tag{2.22}$$

It is stipulated in [7] that

$$\left\|\left(\Delta x^{\text{aff}}, \Delta s^{\text{aff}}\right)\right\| = O(\mu), \quad \|(\Delta x^{\text{cor}}, \Delta s^{\text{cor}})\| = O\left(\mu^2\right) \tag{2.23}$$

when the system matrix is approaching a nonsingular limit. This in turn means that with this correction step, we will reduce the error from order $O\left(\mu^2\right)$ to order $O\left(\mu^3\right)$ thus the correction will make our method a second order approximation [7]. We will not cover further details on this but state that though $\|(\Delta x^{\text{cor}}, \Delta s^{\text{cor}})\| = O\left(\mu^2\right)$ is not always of order $O\left(\mu^2\right)$ when we approach a singular matrix, the corrector step usually enhances the effectiveness of the algorithm in practice [7, p. 197].

In essence, by introducing this correction, we would now add $\mathbf{\Delta}_{cor}$ to the update equation. In combination, the Mehrotra step is the sum of the affine, centering and correction step. We notice, that we can combine this to the calculation of just one combined direction, the *Mehrotra's direction*:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A' & 0 & 0 & 0 \\ -C' & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} - \Delta X^{aff} \Delta \Lambda^{aff} e + \sigma \mu e \end{bmatrix}. \tag{2.24}$$

Of cause we will first have to do that affine step to find σ and $\Delta X^{aff}\Delta\Lambda^{aff}$ but notice that in equation 2.24, the system matrix is the same. Therefore, we only need to factorize the matrix ones and then we reuse the factorization which reduces the computational cost to that of one backward-substitution.

The theoretical background above showed how the important steps in the algorithm is derived. The pseudo code of the algorithm can be found in 6. In the following, we will provide some results and factorizations to improve the efficiency of the code.

2.3.2 Implementation Related Theory

As the KKT matrix is relatively sparse, it would be beneficial for numerical stabilty and computational wise to try to reduce the matrix dimensions. In [6, slide 24-26] a couple of augmentations are introduced. We will use the following augmented equations:

$$\begin{bmatrix} H + C\left(S^{-1}Z\right)C' & -A \\ -A' & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -r_L + C\left(S^{-1}Z\right)\left(r_C - Z^{-1}\bar{r}_{SZ}\right) \\ -r_A \end{bmatrix} \tag{2.25}$$

Here we have introduce this $\bar{r}_{SZ}$ as $\bar{r}_{sz} = r_{sz} + \Delta S S^{aff}\Delta Z^{aff}e - \sigma\mu e$. Hence the subsequent calculations can now be calculated as

$$\begin{aligned} \Delta z &= -\left(S^{-1}Z\right)C'\Delta x + \left(S^{-1}Z\right)\left(r_C - Z^{-1}\bar{r}_{sz}\right) \\ \Delta s &= -Z^{-1}\bar{r}_{sz} - Z^{-1}S\Delta z. \end{aligned} \tag{2.26}$$

Further details to the algorithm can be found in [6] which we will also rely heavily on.

To find a good initial point, we will use the heuristics introduced on slide 29 [6]. For this Inital Point Heuristics, we have created the pseudo code 5 below. The Matlab implementation of this code can be found in appendix B.1.

In the following, we will also provide the pseudo code for the implemented Mehrotra interior point methods which can be seen in 6. We have also implemented the solver in Matlab. This can be seen in appendix B.2.

Algorithm 5 Heuristics for Initial Point for Mehrotra Interior Point Method

procedure IntialPointHeuristics($\bar{x}, \bar{y}, \bar{z}, \bar{s}$)

$r_L = Hx + g - A\bar{y} - C\bar{z}$ ▷ Compute intial residuals

$r_A = b - A^\intercal \bar{x}$

$r_C = \bar{s} + d - C^\intercal \bar{x}$

$r_{\bar{s}\bar{z}} = \bar{z}^\intercal \bar{s}$

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

$\text{KKT} = \begin{bmatrix} \bar{H} & -A \\ -A^\intercal & 0 \end{bmatrix}$ ▷ Construct KKT matrix

$L, D = \text{ldl}(\text{KKT})$ ▷ Make ldl decomposition of KKT matrix

$\bar{r}_L \leftarrow r_L - C(S^{-1}Z)(r_C - Z^{-1}r_{\bar{s}\bar{z}})$

$\begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta \gamma^{\text{aff}} \end{bmatrix} \leftarrow L^{-\intercal}\left(D^{-1}\left(L^{-1}\left(-\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}\right)\right)\right)$ ▷ Compute affine directions

$\Delta \bar{z}^{\text{aff}} \leftarrow -(S^{-1}Z)C^\intercal \Delta x^{\text{aff}} + (S^{-1}Z)(r_C - Z^{-1}r_{\bar{s}\bar{z}})$

$\Delta s^{\text{aff}} \leftarrow -Z^{-1}r_{s\xi} - Z^{-1}S\Delta \bar{z}^{\text{aff}}$

$x_{init} = \bar{x}$ ▷ Stipulate Initial Points

$y_{init} = \bar{y}$

$z_{init} = \max\left\{1, |\bar{z} + \Delta \bar{z}^{\text{aff}}\right\}$ ▷ Compute Initial Points

$s_{init} = \max\left\{1, |\bar{s} + \Delta \bar{s}^{\text{aff}}\right\}$

return $x_{init}, y_{init}, z_{init}, s_{init}$

end procedure

Algorithm 6 Box Constriant QP solver using Mehrotra interior point method

procedure MPCMethodSolvingQP$(H, g, A, b, C, d, x_0, y_0, z_0 > 0, s_0 > 0, \eta)$
  $(x, y, z, s) = \text{IntialPointHeuristics}\ (x_0, y_0, z_0, s_0)$ ▷ See algorithm 5.
  $n_{design} = \text{len}(z)$

  $r_L = Hx + g - Ay - Cz$ ▷ Compute the residuals
  $r_A = b - A^\intercal x$
  $r_C = s + d - C^\intercal x$
  $r_{sz} = z^\intercal s$
  $\mu, \mu_0 = (z^\intercal s)/n_{design}$ ▷ Computing dual gap

  while not Stop **do**
    $\bar{H} = H + C(S^{-1}Z)C^\intercal$
    $\text{KKT} = \begin{bmatrix} \bar{H} & -A \\ -A^\intercal & 0 \end{bmatrix}$
    $L, D = \text{ldl}(\text{KKT})$ ▷ LDL-factorization of K

    $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}r_{sz})$ ▷ Compute affine scaling direction
    $\begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \end{bmatrix} = L^{-\intercal}\left(D^{-1}\left(L^{-1}\left(-\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}\right)\right)\right)$
    $\Delta z^{\text{aff}} = -(S^{-1}Z)C^\intercal \Delta x^{\text{aff}} + (S^{-1}Z)(r_C - Z^{-1}r_{sz})$
    $\Delta s^{\text{aff}} = -Z^{-1}r_{sz} - Z^{-1}S\Delta z^{\text{aff}}$
    $\Delta\alpha^{\text{aff}} = \max_\alpha \alpha, \quad \text{s.t.} \ \left(z + \alpha\Delta z^{\text{aff}}, s + \alpha\Delta s^{\text{aff}}\right) \geq 0$

    $\mu^{\text{aff}} = \left(z + \alpha^{\text{aff}}\Delta z^{\text{aff}}\right)^\intercal \left(s + \alpha^{\text{aff}}\Delta s^{\text{aff}}\right)/n_{design}$ ▷ Calculate cent. parm.
    $\sigma = \left(\mu^{\text{aff}}/\mu\right)^3$

    $\bar{r}_{sz} = r_{sz} + \Delta S^{\text{aff}} + \Delta Z^{\text{aff}}e - \sigma\mu e$ ▷ Construct Mehrotra's direction
    $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$
    $\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = L^{-\intercal}\left(D^{-1}\left(L^{-1}\left(-\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}\right)\right)\right)$
    $\Delta z = -(S^{-1}Z)C^\intercal \Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{sz})$
    $\Delta s = -Z^{-1}r_{sz} - Z^{-1}S\Delta z$
    $\Delta\alpha = \max_\alpha \alpha, \quad \text{s.t.} \ (z + \alpha\Delta z, s + \alpha\Delta s) \geq 0$

    $(x, y, z, s) = (x, y, z, s) + \eta\alpha(\Delta x, \Delta y, \Delta z, \Delta s)$ ▷ Update Variables
    $r_L = Hx + g - Ay - Cz$ ▷ Update Residuals
    $r_A = b - A^\intercal x$
    $r_C = s + d - C^\intercal x$
    $r_{sz} = z^\intercal s$
    $\mu, \mu_0 = (z^\intercal s)/n_{design}$ ▷ Compute Duality Gap

    if $\mu \leq \epsilon\, 0.01\mu_0$ **then** ▷ Check Convergence
      Stop
    end if
  end while
  return x, z, y, s

end procedure

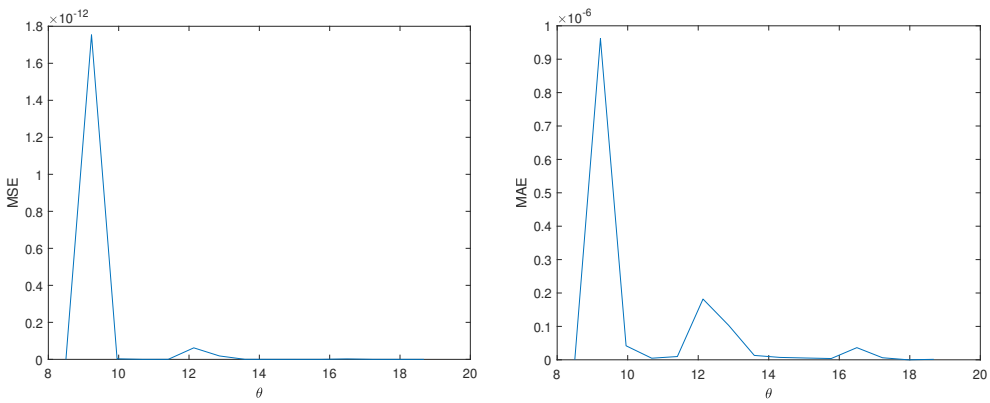
2.4 Consolidated section for Exercises 2.4-2.6

We have consolidated the the exercises from 2.4-2.6 as they are all related to the validation of the implemented solver and tests for performance.

Explicitly we state, that the implemented solver is to be found in appendix B.2. The driver for the test and presented figures is in appendix B.3. In the following, we will first test our solver on the problem introduced in section 1.4. Then we will consider the performance of our solver when we introduce larger problems with more constraints.

2.4.1 Correctness of the Implemented Solver

In the figures below, we will consider if the solver is correct. We will compare our implementation with that of the Matlab implemented *quadprog*.



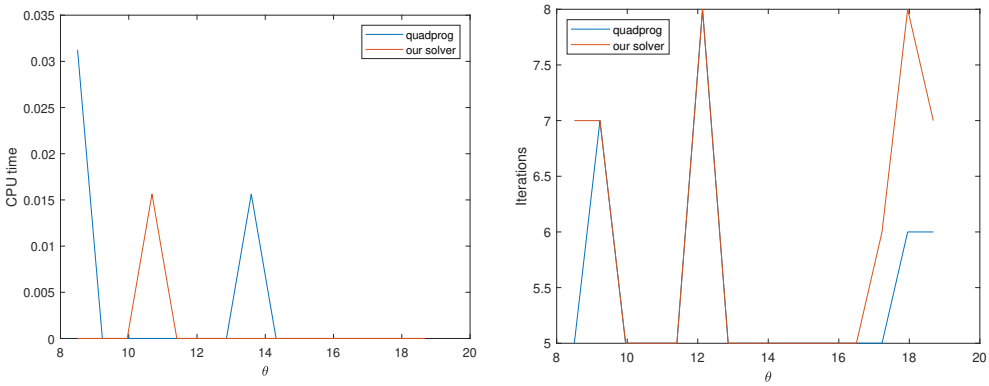
(a) Mean Squared Error **(b)** Mean absolute errors.

Figure 2.2: The errors between the implemented solvers and quadprog solutions on the problem introduced in section 1.4. As the scale of the y-axis indicated, all solvers are correct.

In figure 2.2, we see that the difference in the solution computed using our implementation and the *quadprog* is negligible. We see some spikes in the correctness, however, when we look at the scale on the y-axis we see that the difference is still very small.

2.4.2 Performance

Below, we will consider the performance of our solvers for the introduces problem both in terms of spent CPU time and number of iterations.



(a) CPU Time.

(b) Number of Iterations.

Figure 2.3: Performance statistics for *quadprog* our solver when tested on the problem introduced in section 1.4. There is no huge performance gain using one solver over the other.

In figure 2.3, we see that there is no immediate advantage of using our solver or the generic Matlab *quadprog* implementation as the CPU-time and used number of iterations are almost identical.

2.4.2.1 Performance Check on Problem of Scale

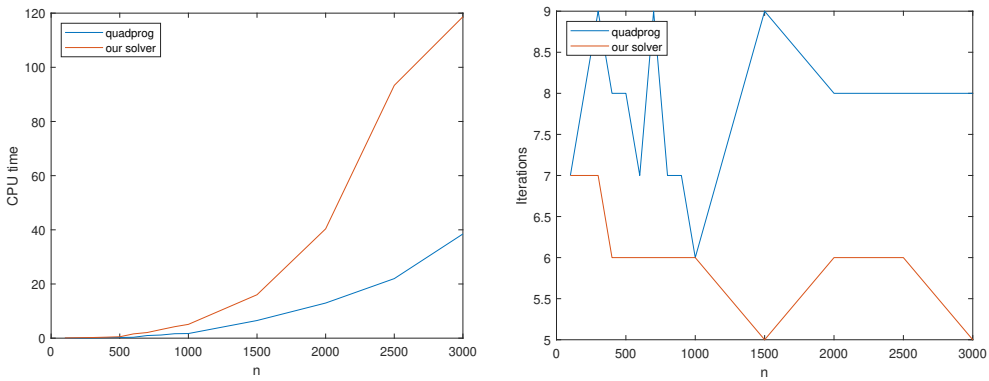
We will now test if the performance of our solver when we test on a problem of scale. In our case, we will randomly generate a equality constraint problem and then increase the number of design variables and constraints. The code to generate the EQP is the same as in Exercise 1. It can be found in appendix A.2. In the experiments, we tested

$$n = \begin{bmatrix} 100 & 200 & \dots & 1000 & 1500 & 2000 & \dots & 3000 \end{bmatrix}^{\intercal} \tag{2.27}$$

Where we set number of constraints equal to half of that of design variables. We also just used the same box constaints such that:

$$\begin{bmatrix} I & -I \end{bmatrix}^{\top} x \geq \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{2.28}$$

In this case, we saw that our solver is not efficient for large problems:



(a) CPU Time. **(b)** Number of Iterations.

Figure 2.4: Performance statistics for *quadprog* our solver when tested on generated EQPs with box-constraints. We see that our performs quite bad when we try to solve problems at scale.

In figure 2.4, we see that the *quadprog* solver outperforms our solver significantly when we increase n in terms of spent CPU time. However, on the flip side, out solver seems to keep run only a limited number of iterations. Here we really see the strong performance of the versatile *quadprog* as it is able to formulate an execution plan much more efficient than our solver. We speculate that it might be able to utilize some structure in the matrices or is better able to run computations in parallel.

Here we see that the additional overhead that seemed to hinder performance of the *quadprog* in the section 1 now turns out very useful as it greatly outperforms our solver.

CHAPTER 3

Linear Program

In this chapter we consider a linear program:

$$\begin{aligned} \min_{x} \quad & \phi = g^\top x \\ \text{s.t.} \quad & A^\top x = b \\ & l \leq x \leq u \end{aligned} \tag{3.1}$$

where we assume A has full column rank. As with the problems in the first two chapters, we will now rewrite the problem such that we obtain only inequality constraints of the form $C^\tau x \geq d$. For the specific problem of interest we obtain

$$\begin{aligned} \min_{x} \quad & \phi = g^\top x \\ \text{s.t.} \quad & A^\top x = b \\ & \begin{bmatrix} \mathbb{I} & -\mathbb{I} \end{bmatrix}^\top x \geq \begin{bmatrix} l \\ -u \end{bmatrix} \end{aligned} \tag{3.2}$$

3.1 Exercise 3.1

The Lagrangian of this problem is quite similar to that of EQP, equation 2.3. In this case, the first term is simply removed because we now deal with a linear objective hence the Lagrangian is:

$$L(x, y, z) = g^\intercal x - y^\intercal \left(A^\intercal x - b\right) - z^\intercal \left(C^\intercal x - d\right) \tag{3.3}$$

3.2 Exercise 3.2

In this case, we have a linear problem with inequality and equality constraints hence we can again state the first order optimality conditions:

$$
\begin{aligned}
\nabla_x \mathcal{L}(x,\gamma,\xi) &= \nabla f(x) - \sum_{i\in\mathcal{E}} \gamma_i \nabla c_i(x) - \sum_{i\in\mathcal{I}} \xi_i \nabla c_i(x) \\
\nabla_\gamma \mathcal{L}(x,\gamma,\xi) &= c_i(x) = 0, \quad i \in \mathcal{E} \\
\nabla_\xi \mathcal{L}(x,\gamma,\xi) &= c_i(x) \geq 0, \quad i \in \mathcal{I} \\
\xi_i &\geq 0, \quad i \in \mathcal{I} \\
c_i(x)\xi_i &= 0, \quad i \in \mathcal{I}
\end{aligned}
\tag{3.4}
$$

We work with a linear problem and these are always convex. It, therefore, follows directly from section 2.5 [1] that the listed first order conditions are necessary and sufficient.

3.3 Exercise 3.3

In terms of the derivation necessary to explain the mechanisms for the primal-dual algorithm, we will not cover the interior point method extensively. Many of the step resembles that of interior point method for the quadratic program hence we refer to section 2.2. The main difference for a linear program is of cause that the r_L for the linear program:

$$
r_L = b - A^\intercal s - C^\intercal z \tag{3.5}
$$

If we compare this with the one we found for the quadratic program, we notice that the term with H is not there. Consult [8] for each explicit steps for the linear case but in turn we will obtain the Mehrotra's direction:

$$
\begin{bmatrix} 0 & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix}
\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix}
= -
\begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} - \Delta Z^{aff} \Delta S^{aff} e + \sigma\mu e \end{bmatrix}
\tag{3.6}
$$

In [8] great suggestions on how to augment this problem to speed up calculation are presented. We have again used the presented initial point heuristics with the necessary changes to make it work in this LP setting.

Algorithm 7 Box Constriant LP solver using Mehrotra interior point method

procedure MPCMethodSolvingQP$(H, g, A, b, C, d, x_0, y_0, z_0 > 0, s_0 > 0, \eta)$
  $(x, y, z, s) = \text{IntialPointHeuristics}\ (x_0, y_0, z_0, s_0)$ ▷ See algorithm 5.
  $n_{design} = \text{len}(z)$

  $r_L = g - Ay - Cz$ ▷ Compute the residuals
  $r_A = b - A^\intercal x$
  $r_C = s + d - C^\intercal x$
  $r_{sz} = z^\intercal s$
  $\mu, \mu_0 = (z^\intercal s)/n_{design}$ ▷ Computing dual gap

  while Not Stop **do**
    $\bar{H} = H + C(S^{-1}Z)C^\intercal$
    $\text{KKT} = \begin{bmatrix} \bar{H} & -A \\ -A^\intercal & 0 \end{bmatrix}$
    $R = \text{chol}(\text{KKT})$ ▷ Cholesky-factorization of KKT

    $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}r_{sz})$ ▷ Compute affine scaling direction
    $\Delta y^{\text{aff}} = R^{-1}\left[R^{-\intercal}\left(\bar{r}_L\right)\right]$
    $\Delta x^{\text{aff}} = \left(S^{-1}Z\right)C^\intercal\left(-\bar{r}_L + A\Delta y^{\text{aff}}\right)$
    $\Delta z^{\text{aff}} = -(S^{-1}Z)C^\intercal\Delta x^{\text{aff}} + (S^{-1}Z)(r_C - Z^{-1}r_{sz})$
    $\Delta s^{\text{aff}} = -Z^{-1}r_{sz} - Z^{-1}S\Delta z^{\text{aff}}$
    $\Delta\alpha^{\text{aff}} = \max_\alpha \alpha, \quad \text{s.t.}\ \left(z + \alpha\Delta z^{\text{aff}}, s + \alpha\Delta s^{\text{aff}}\right) \geq 0$

    $\mu^{\text{aff}} = \left(z + \alpha^{\text{aff}}\Delta z^{\text{aff}}\right)^\intercal\left(s + \alpha^{\text{aff}}\Delta s^{\text{aff}}\right)/n_{design}$ ▷ Calculate cent. parm.
    $\sigma = \left(\mu^{\text{aff}}/\mu\right)^3$

    $\bar{r}_{sz} = r_{sz} + \Delta S^{\text{aff}} + \Delta Z^{\text{aff}}e - \sigma\mu e$ ▷ Construct Mehrotra's direction
    $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$
    $\Delta y^{\text{aff}} = R^{-1}\left[R^{-\intercal}\left(\bar{r}_L\right)\right]$
    $\Delta x^{\text{aff}} = \left(S^{-1}Z\right)C^\intercal\left(-\bar{r}_L + A\Delta y^{\text{aff}}\right)$
    $\Delta z = -(S^{-1}Z)C^\intercal\Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{sz})$
    $\Delta s = -Z^{-1}r_{sz} - Z^{-1}S\Delta z$
    $\Delta\alpha = \max_\alpha \alpha, \quad \text{s.t.}\ (z + \alpha\Delta z, s + \alpha\Delta s) \geq 0$

    $(x, y, z, s) = (x, y, z, s) + \eta\alpha(\Delta x, \Delta y, \Delta z, \Delta s)$ ▷ Update Variables
    $r_L = g - Ay - Cz$ ▷ Update Residuals
    $r_A = b - A^\intercal x$
    $r_C = s + d - C^\intercal x$
    $r_{sz} = z^\intercal s$
    $\mu, \mu_0 = (z^\intercal s)/n_{design}$ ▷ Compute Duality Gap

    if $\mu \leq \epsilon\, 0.01\mu_0$ **then** ▷ Check Convergence
      Stop
    end if
  end while
  return x, z, y, s

end procedure

3.4 Exercise 3.4

In appendix C.4 the driver for this exercise is presented. This has been created with collaboration with Magne Egede Rasmussen while the implementations of the primal-dual interior-point method with box constraints can be found in C.2 and the initial point heuristic algorithm can be found in appendix C.1.

3.5 Consolidated section for Exercises 3.5-3.6

In the following, we will test our solver on the following problem:

$$\begin{aligned} g &= \begin{bmatrix} -16.1000 & -8.5000 & -15.7000 & -10.0200 & -18.6800 \end{bmatrix}^\intercal \\ A &= \begin{bmatrix} 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \end{bmatrix}^\intercal \\ b &= \begin{bmatrix} & 1 & 1 \end{bmatrix}^\intercal \\ l &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}^\intercal \\ u &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}^\intercal \end{aligned} \tag{3.7}$$

In the following, we will test our solver with built-in solvers in Matlab. Specifically, we will compare with the LP solver *linprog* and simplex solver.

We will first test the correctness of our solver which we will do by comparing the found minima for the three solver.

	Simplex	IP	Our Solver
$\phi =$	-18.6800	-18.6800	-18.6800
$\Delta\phi =$		0	$-4.5173\ 10^{-1}$
x =	0	$1.8488\ 10^{-17}$	$7.3925\ 10^{-12}$
	0	$5.7270\ 10^{-17}$	$-3.4713\ 10^{-12}$
	0	$5.2980\ 10^{-18}$	$-3.4713\ 10^{-12}$
	0	$1.5358\ 10^{-17}$	$-4.2366\ 10^{-12}$
	1	$1.0000\ 10^{0}$	$1.0000\ 10^{0}$
MSE of x =		0	$3.8879e-12$

Table 3.1: Comparison of our implementation with *linprog* simplex, *linprog* interior-point and our LP interior-point. The computed errors assumes the simplex algorithm is correct

In table 3.1, we see that they all find the same minimum hence the solver seems to work correctly.

We will now consider the performance in terms of the need CPU time and number of iterations needed for this problem. These can be seen in table below:

	Simplex	IP	Our Solver
Time =	$1.1703\ 10^{-2}$	$1.8\ 10^{-2}$	$6.7022\ 10^{-3}$
Iterations =	1	4	10

Table 3.2: Performance in terms of CPU time and number of iteration used

In table 3.2, it seems that the time spent by the solvers to solve this small problem is quite much alike. The simplex uses only 1 iteration, the IP uses 4 and ours uses 10, however, our solver is still the fastest. As the built-in solvers probably have some overhead with input checks, feasibility checks etc. we thought it would be interesting to investigate the performance of the solvers for larger problem sizes.

3.5.1 Solver Performance for a Problem of Scale

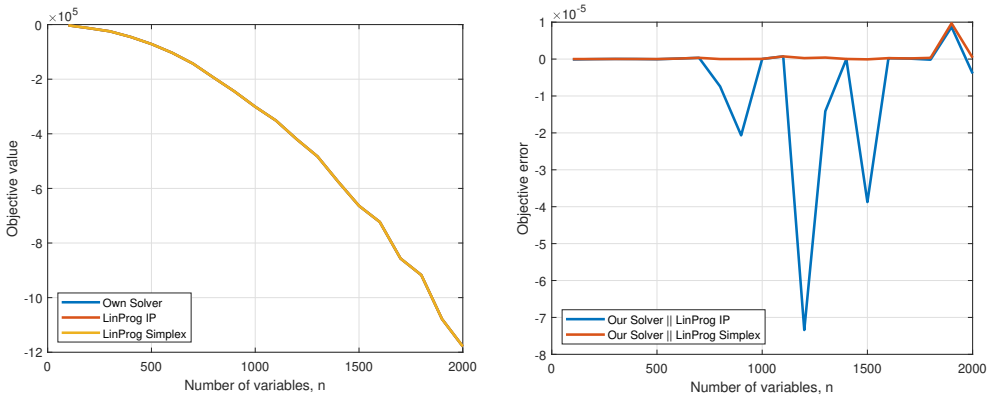
In the following, we will do as in exercise 1.5 and generate a random LP with n design variables and $n/2$. Specifically, we will generate 20 LPs of n design variables equidistantly sampled between 100 and 2000. We also introduce half the number of constraints for each of the constructed problems. For each of the generated LPs we also introduce the box constraints such the problems are more like the one stated above.

The code to generate the LPs can be seen in appendix C.3.

In the following, we will study if our solver retains is correctness when we work with problems at scale. Subsequently we will study the performance in terms of CPU time and number of iterations used.

3.5.1.1 Correctness for problem at scale

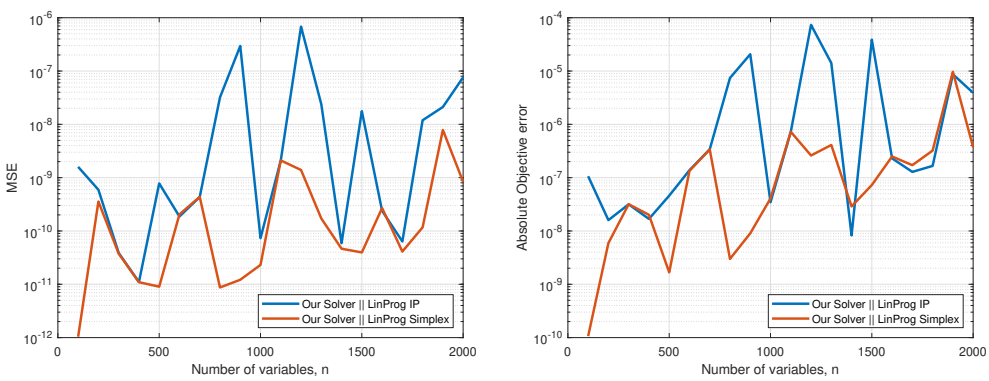
To understand the subsequent figures with the errors, it is important to understand how the objective changes with the n. In figure 3.1, we see that as we increase n, then the value of the objective decreases. As all lines are on top of each, we already have a good sign in the left figure. In the figure to the right we compare the error in the found objective between our solver and *fmincon* simplex and interior point methods. We see already that the difference is very small, $\approx 10^{-5}$, which really is noting compared to the scale of the objective at 10^5.



(a) The objective as a function of n **(b)** Comparison of found Objective

Figure 3.1: Found objective and difference between the implemented solver and quadprog solutions.

In the figures below, we consider the MSE and MAE for the found objective for this problem at scale in semi-log figures.



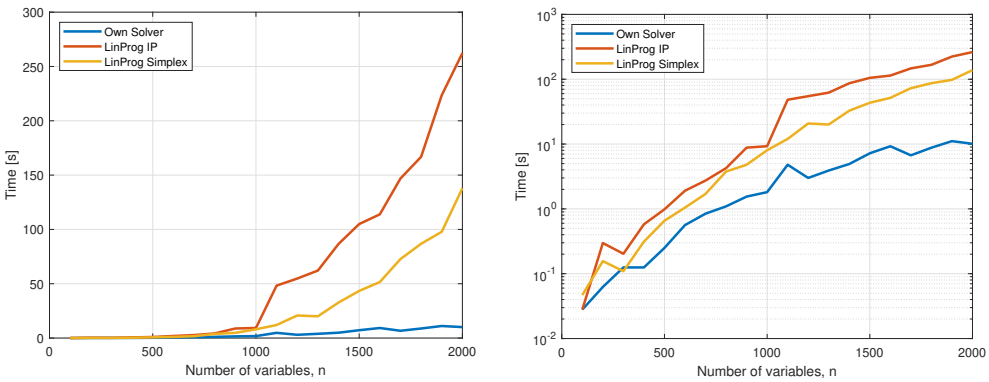
(a) Mean Squared Error in semi-log plot. **(b)** Mean absolute errors in semi-log plot.

Figure 3.2: The errors in the found objective when we compare our solver with *quadprog* solutions.

In figure 3.2, we see that that the difference in the found objectives are very close with errors on a scale that we will consider negligible. With that we conclude that our solver seems correct also when we work with problems at scale. We will now consider the performance statistics in terms of spent CPU-time and iterations.

3.5.1.2 Performance of Solver

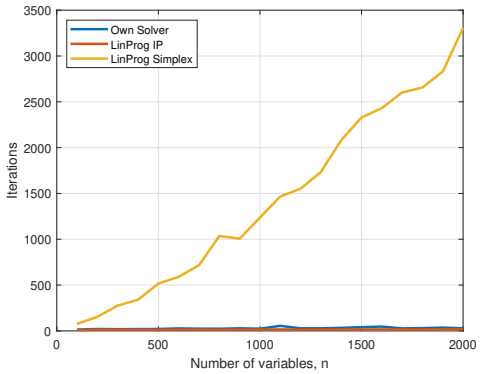
We will focus first on the CPU time spent by each solver and see how this changes as we start to scale-up the problem. Then we will consider the number of iterations used.



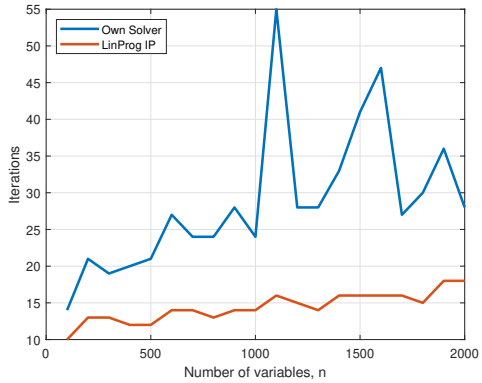
(a) Time spent as a function of n **(b)** Time spent as a function of n semi-log.

Figure 3.3: The CPU-time spent for the implemented solver and the built-in*quadprog* solver.

In figure 3.3, we see that as we increase n, then the *linprog* solvers still remains slower than the solver we have implemented. Especially, when n exceeds 1000 where our solver seems to reach a plateau in comparison to the other solvers that still scales exponentially with n. It seems that the interior-point method from the built-in *quadprog* quite much slower than the simplex as well.
When we consider the number of iterations used, we would expect that simplex could take a substantial amount of iterations as it can only go from vertex to vertex. Therefore, we have depicted this in one figure and excluded the simplex in the second figure to asses the performance of each of these interior-point methods.



(a) The number of iterations for all solvers.



(b) The number of iterations for the interior point methods only.

Figure 3.4: The number of iterations used as a function of n.

In figure 3.4, we see that the simplex algorithm is well above 3000 iterations for the last set of problems whereas the interior points methods have below 100 iterations. We find it quite interesting that though the simplex uses a multitude of iterations, it is still faster than the *quadprog* interior-point method. We conclude that our solver works well for these kinds of problems both in terms of correctness, spent CPU time and number of iterations used.

CHAPTER 4

Nonlinear Program

In this exercise, we will consider a non-linear problem of the form:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g_l \leq g(x) \leq g_u \\ & x_l \leq x \leq x_u \end{aligned} \tag{4.1}$$

we assume that the functions are sufficiently smooth for the algorithms to work and that $\nabla g(x)$ has full column rank.

In this case, we will also rewrite this problem. Initially, we write the constraints as:

$$\begin{aligned} \begin{bmatrix} I & -I \end{bmatrix}^\top x &\geq \begin{bmatrix} x_l \\ -x_u \end{bmatrix} \\ \begin{bmatrix} I & -I \end{bmatrix}^\top g(x) &\geq \begin{bmatrix} g_l \\ -g_u \end{bmatrix} \end{aligned} \tag{4.2}$$

Here we see directly that a very abbreviated version of the problem would be:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \begin{bmatrix} x \\ -x \\ g(x) \\ -g(x) \end{bmatrix} \geq \begin{bmatrix} x_l \\ -x_u \\ g_l \\ -g_u \end{bmatrix} \end{aligned} \tag{4.3}$$

In the subsequent sections, we will use this notation for the constraints and problem formulation.

4.1 Exercise 4.1

With the problem on the form as in equation 4.3, we can now see directly how we should formulate the Lagrangian of the problem. It will be directly as in the previous sections and as described on [p.44, [1]]

$$\mathcal{L}(x,\lambda) = f(x) - z^{\intercal} \begin{bmatrix} x - x_l \\ x_u - x \\ g(x) - g_l \\ g_u - g(x) \end{bmatrix} \tag{4.4}$$

4.2 Exercise 4.2

The first order necessary condition for the non-linear program is a direct extension of section 1.2 and section 2.2.

$$\begin{aligned} \nabla_x \mathcal{L}(x,z) = \nabla f(x) - z^{\intercal} \begin{bmatrix} I \\ -I \\ \nabla g(x) \\ -\nabla g(x) \end{bmatrix} &= 0 \\ \nabla_z \mathcal{L}(x,z) = \begin{bmatrix} x - x_l \\ x_u - x \\ g(x) - g_l \\ g_u - g(x) \end{bmatrix} &\geq 0, \\ z_i \geq 0, \quad & i \in \mathcal{I} \\ g_i(x) z_i = 0, \quad & i \in \mathcal{I} \end{aligned} \tag{4.5}$$

Where we in this case use the notation from [4] and let $\mathcal{I}$ be index for the set of inequality constraints. The ability to formulate the above follows directly from theorem 18.1, [4]

The formulated first order optimallity condition are always necessary but not sufficient as the problem does not longer have guaranteed convexity. Here we must include second order information around the stationary point. We will cover this further in the next section.

4.3 Exercise 4.3

To be sure that we have found a minimum, we must include second order information. In proposition 2.14 in [1], they let $F(x)$ be the space of all feasible directions, and introduce the second order condition for a non-linear program as:

$$h' \nabla^2_{xx} \mathcal{L}(x,z) h > 0, \quad \forall h \in \mathcal{F}(x) \tag{4.6}$$

This in turn means that we require the curvature of the Lagrangian around a stationary point to be positive i.e. x would be a strictly local minimizer. This is equal to test if $\nabla^2_{xx} \mathcal{L}(x,z)$ positive definite.

Indeed if x satisfies equation 4.5, then we have a stationary point but for the stationary point to be a strictly local minimizer, we require the condition in equation 4.6 to also hold.

4.4 Exercise 4.4

In the following, we will work with Himmelblau's test problem:

$$\begin{aligned} \min_x \quad & f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ \text{s.t.} \quad & g_1(x) = (x_1 + 2)^2 - x_2 \geq 0 \\ & g_2(x) = -4x_1 + 10x_2 \geq 0 \end{aligned} \tag{4.7}$$

This problem, we will now rewrite to match that of the generic formulation in equation 4.1. We note immediately that the there are no bounds on the design variables. Consider the objective f and we see that all the stationary points are contained in a box defined by the boundaries:

$$5 \geq x_1 \geq -5, \quad 5 \geq x_2 \geq -5 \tag{4.8}$$

We can calculate directly the boundaries we must impose on $g_1(x)$ and $g_2(x)$. In effect, we rephrase the problem as:

$$\begin{aligned} \min_x \quad & f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \\ \text{s.t.} \quad & 54 \geq g_1(x) \geq 0 \\ & 70 \geq g_2(x) \geq 0 \\ & 5 \geq x_1 \geq -5 \\ & 5 \geq x_2 \geq -5 \end{aligned} \tag{4.9}$$

We will now plot the contours of the problem we formulated in 4.9 along with the stationary points in figure 4.1.

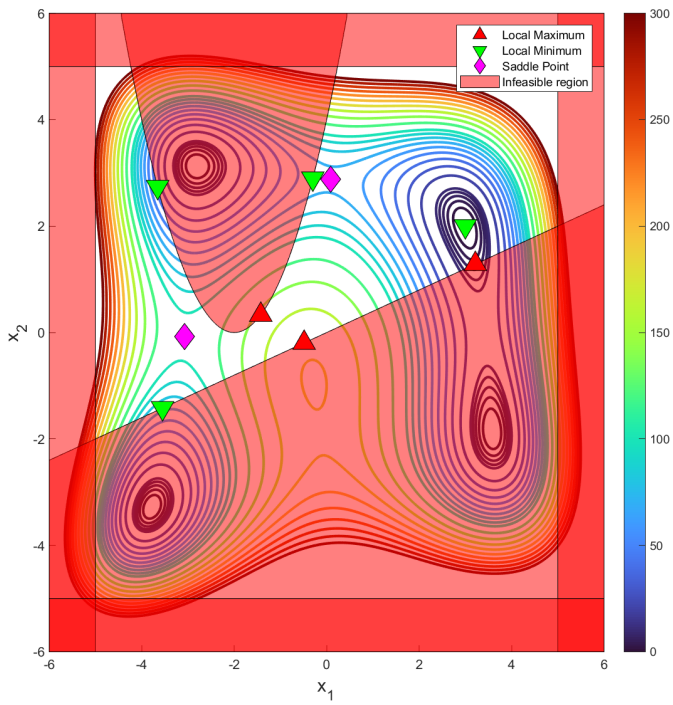


Figure 4.1: Contours of the Himmelblau problem along with contours for the constraints.

These figures have been created in collaboration with Magne Egede Rasmussen and the code can be seen in appendix D.1 and D.3. In figure 4.1, we see that we have the four main minima that our solver could converge towards. In the following, we will initially consider some built-in solver to solver the problem. Then we will describe and implement our own solver to solve this class of non-linear problems.

4.5 Exercise 4.5

In this section, we will use the library functions **fmincon** and **CasADi** to solve non-linear programs. The driver for this exercise can be seen in section D.8, developed with a fellow student Magne Egede Ramussen.

fmincon is a build-in NLP solver in Matlab. If one only passes the objective to this solver, it would use finite difference to compute the gradients. If the gradient is passed as close form, it would of cause use that. On the other hand, **CasADi** is an open-source solver for nonlinear optimization. It is highly efficient and built on C++

code. It differs significantly in its ability to compute the gradient in absence of a closed form solution for the gradient. **CasADi** uses automatic differentiation instead of finite difference. This is better if one favors accuracy, however, a drawback is that it can take longer to evaluate and a substantial number of gradients would have to be stored hence it will take up more memory.

4.5.1 Himmelblau's testproblem

In the following, we will solve Himmelblau's test problem using the solvers presented. As we see in figure 4.1, there are multiple minimizers in the feasible region hence we will initialize the solvers at different point and see report where they end up.

Below is a couple of the found minima for different initial values.

	$x_0 = [0.0, 0.0]^\intercal$	$x_0 = [-4.0, 0.0]^\intercal$	$x_0 = [-4.0, 1.0]^\intercal$
fmincom			
$f(x) =$	0.00000	72.85555	35.92985
$x_1 =$	3.00000	-3.54854	-3.65461
$x_2 =$	2.00000	-1.41941	2.73772
$t[s] =$	0.01562	0.01562	0.00000
CasADi			
$f(x) =$	0.00000	35.92985	35.92985
$x_1 =$	3.00000	-3.65461	-3.65461
$x_2 =$	2.00000	2.73772	2.73772
$t[s] =$	0.01562	0.01562	0.01562

Table 4.1: Found solutions and spent CPU time for different initial points for the Himmelblau's test problem.

In table 4.1, we see that for all solvers converge towards the same minimum when we pass the initial values $x_0 = [0.0, 0.0]^\intercal$ and $x_0 = [-4.0, 0.0]^\intercal$. The interesting instance is the one for $x_0 = \begin{bmatrix}-4 & 0\end{bmatrix}^\intercal$. Here we see that the solvers find a different minimum. This is depicted in figure 4.1.

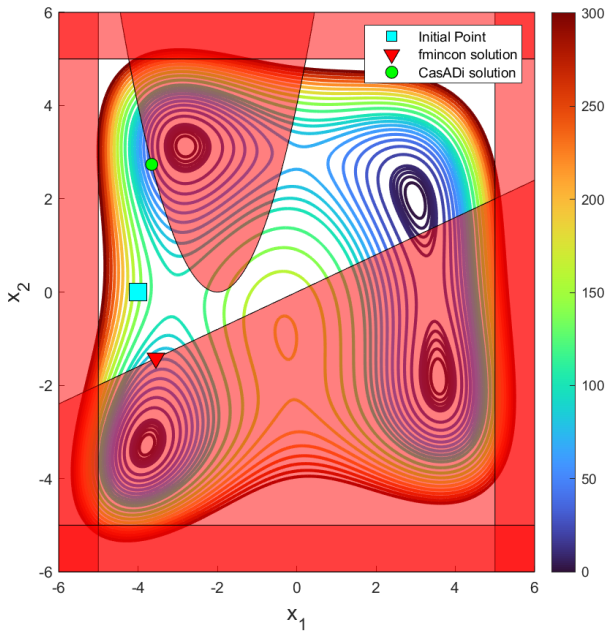


Figure 4.2: Himmelblau problem with found solutions for the initial point $x_0 = \begin{bmatrix} -4 & 0 \end{bmatrix}^\intercal$.

In figure 4.2, we see that the **CasADi** solver finds the a minimum different from the one found by **fmincon**. This is interesting as it we explicitly see that the solvers uses different algorithms. In the following, it will be interesting to see which minimum our solvers will go towards.

4.5.2 Rosenbrock with Unit circle constraints.

To test the solvers, we will also pose a more challenging problem. It is a specific instance of the Rosenbrock problem where we introduce the unit circle constraints. This means that we essentially will solve the following problem

$$\begin{aligned} \min_{x} \quad & f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{s.t.} \quad & 1 \geq g_1(x) \geq 0 \\ & 1 \geq x_1 \geq -1 \\ & 1 \geq x_2 \geq -1 \end{aligned} \tag{4.10}$$

In equation 4.10, we have $g_1(x) = x_1^2 + x_2^2$ and explicitly f and g_1 are twice differentiable. The Rosenbrock function is interesting as it can be hard to locate the

minimum for NLP solver due to the curvature around the minimum. We could have multiple possible minima depending on the initial conditions.Therefore, we consider multiple initial conditions and when we do this, we get the following minimum:

NLP Solver	$x_0 = [0.0, 0.0]^\intercal$	$x_0 = [-0.5, 0.0]^\intercal$	$x_0 = [-0.5, -0.5]^\intercal$
fmincom			
$f(x) =$	0.0457	0.0457	0.0457
$x_1 =$	0.7864	0.7864	0.7864
$x_2 =$	0.6177	0.6177	0.6177
$t[s] =$	0.0312	0.1719	0.0781
CasADi			
$f(x) =$	0.0457	0.0457	0.0457
$x_1 =$	0.7864	0.7864	0.7864
$x_2 =$	0.6177	0.6177	0.6177
time $[s] =$	0.0469	0.1094	0.0469

Table 4.2: Minimizer x and CPU time spent for different initial points for the program equation 4.10. In all cases, the solvers find the same minimum $[x_1, x_2] = [0.7864, 0.6177]$.

In table 4.2, we see that all the solvers find the same minimum and that for different initial conditions, then solvers still converges towards the same minimum. In figure 4.3, one can see an instance where we initialize at $x_0 = [0, 0]^\intercal$ and that we end up one the boundary of the unit circle which we tend to end up at each time.

4.5.3 Rosenbrock with Box Constraints

In the following, we will try to change the constaints to formulate a new NLP. In this case, we will introduce box-constraints such that we have:

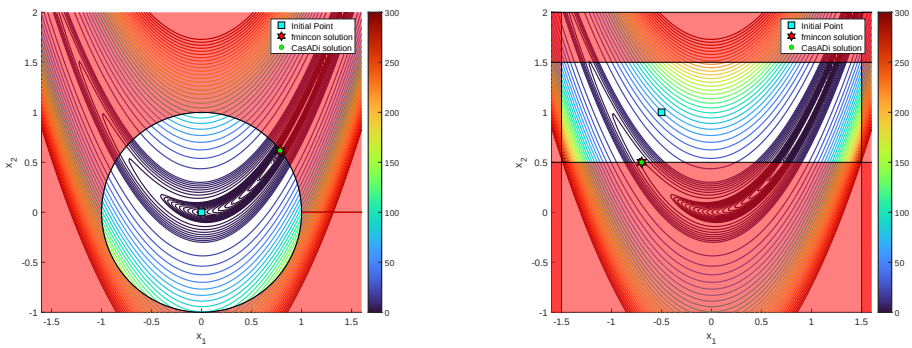
$$\begin{aligned} \min_x \quad & f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ & 1.5 \geq x_1 \geq -1.5 \\ & 1.5 \geq x_2 \geq 0.5. \end{aligned} \tag{4.11}$$

We will now again consider the problem for different initial conditions to see if we end up at different minima. Below we show the table of found minima:

NLP Solver	$x_0 = [0.0, 1.0]^\intercal$	$x_0 = [-1.2, 0.5]^\intercal$	$x_0 = [-0.5, 1.0]^\intercal$
fmincom			
$f(x) =$	0.0000	2.8995	2.8995
$x_1 =$	1.0000	-0.6985	-0.6985
$x_2 =$	1.0000	0.5000	0.5000
$t[s] =$	0.0781	0.0469	0.0938
CasADi			
$f(x) =$	0.0000	2.8995	2.8995
$x_1 =$	1.0000	-0.6985	-0.6985
$x_2 =$	1.0000	0.5000	0.5000
$t[s] =$	0.0469	0.0312	0.0469

Table 4.3: Minimizer x for different initial points for the program 4.11.

In table 4.3, we see that we converge towards different minima when we initialize different places. When we consider figure 4.3, we understand why this is the case as it seems more likely to find the local minimum with a positive x_1 if the initial conditions for x_1 is also positive.



(a) Rosenbrock with unitcicle constraints **(b)** Rosenbrock with box-constraints

Figure 4.3: Instances of found minima for different constraints for the Rosenbrock problem.

4.6 Exercise 4.6

In section 2.2, we saw the ease with which we could solve quadratic programs. The strong performance is leveraged when we go to non-linear programs. The ideas is to make a local QP approximation of the non-linear program and do this iteratively i.e. solve a sequence of quadratic programs; hence the name SQP. In figure 4.4 we see how we form local quadratic approximations of the non-linear program. The figure is a direct replica of Fig. 5.42 [3]

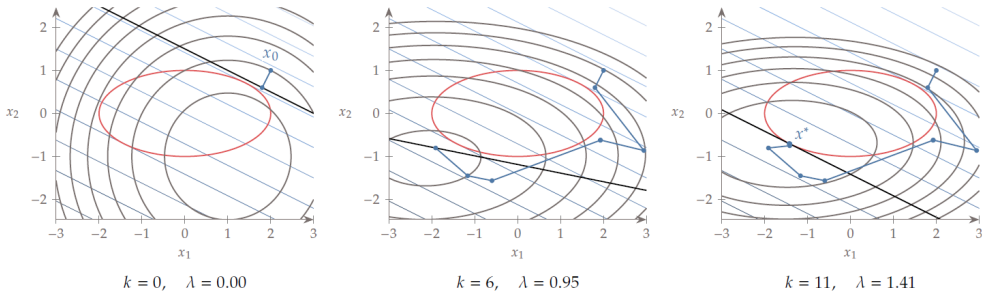


Figure 4.4: Depiction of how we sequentially form local QP approximation of our non-linear problem. The figure is a replica of Fig. 5.42 [3].

4.6.1 Derivation of the SQP algorithm

In the following, we will move forward as in 1.2. Explicitly, we will start only with equality constraints and then move to inequality constraints. The section is heavily inspried by section 5.5 in [3]. We know that for e.g. the Himmelblau problem, we are only interested in problems with inequality constraints, however, we find the derivation more intuitive when we start from equality constraints. Consider, therefore, an NLP with equality constraints only:

$$\begin{array}{ll} \min_{\boldsymbol{x}\in\mathbb{R}^n} & f(\boldsymbol{x}) \\ \text{s.t.} & h(\boldsymbol{x}) = \mathbf{0}. \end{array} \tag{4.12}$$

Where the Lagrangian is $\mathcal{L}(x, y) = f(x) - y^\intercal h(x)$. In the following, we let $\nabla^2_{x,x}\mathcal{L}(x, y)$ be the Hessian of the Lagrangian. We make a local quadratic approximation of the Lagrangian and a linear approximation of the constraints to find the next step Δx near the current point:

$$\begin{array}{ll} \min_{\Delta\boldsymbol{x}\in\mathbb{R}^n} & \frac{1}{2}\Delta\boldsymbol{x}^T\left(\nabla^2_{xx}L(\boldsymbol{x},\boldsymbol{y})\right)\Delta\boldsymbol{x} + \left(\nabla_x L(\boldsymbol{x},\boldsymbol{y})\right)^T\Delta\boldsymbol{x} \\ \text{s.t.} & \nabla h(\boldsymbol{x})^T\Delta\boldsymbol{x} = -h(\boldsymbol{x}). \end{array} \tag{4.13}$$

Note that we have removed the constant term in the objective as it will not change the solution when we calculate the next step to take.

Now remember that $\mathcal{L}_x = \nabla f^\intercal + \nabla h(x)y$ hence from the above, we see that we can substitute $\nabla h(x)y = -h(x)$ to obtain the objective:

$$\min_{\Delta \boldsymbol{x} \in \mathbb{R}^n} \frac{1}{2} \Delta \boldsymbol{x}^T \left(\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) \right) \Delta \boldsymbol{x} + \left(\nabla_x f(\boldsymbol{x}) \right)^T \Delta \boldsymbol{x} - \boldsymbol{y}^\intercal h(x). \tag{4.14}$$

As Δx is not present in the last term, we can for our part ignore it and hence we will end up with the reduced objective:

$$\begin{array}{ll} \min_{\Delta \boldsymbol{x} \in \mathbb{R}^n} & \frac{1}{2} \Delta \boldsymbol{x}^T \left(\nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) \right) \Delta \boldsymbol{x} + \left(\nabla_x f(\boldsymbol{x}) \right)^T \Delta \boldsymbol{x} - \boldsymbol{y}^\intercal h(x) \\ \text{s.t.} & \nabla h(\boldsymbol{x})^T \Delta \boldsymbol{x} = -h(\boldsymbol{x}). \end{array} \tag{4.15}$$

We now recognize this as a QP that we can solve. Consider, therefore the above formulated with a system of linear equations:

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \boldsymbol{y_{k+1}} \end{pmatrix} = - \begin{pmatrix} \nabla f(\boldsymbol{x}) \\ -h(\boldsymbol{x}) \end{pmatrix} \tag{4.16}$$

Now because $y_{k+1} = \Delta y + y_k$, we can rewrite the above to obtain:

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \end{pmatrix} + \begin{pmatrix} \nabla h(\boldsymbol{x}) y_k \\ \mathbf{0} \end{pmatrix} = - \begin{pmatrix} \nabla f(\boldsymbol{x}) \\ -h(\boldsymbol{x}) \end{pmatrix}. \tag{4.17}$$

This we know that we can equivalently write as:

$$\begin{pmatrix} \nabla^2_{xx} L(\boldsymbol{x}, \boldsymbol{y}) & -\nabla h(\boldsymbol{x}) \\ -\nabla h(\boldsymbol{x})^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} -\nabla_x \mathcal{L} \\ -h(x) \end{pmatrix} \tag{4.18}$$

We recognize the above directly as the linear system we solve when we used Newton's method on the KKT system.

For inequality constraints, we would use the same derivation, however, we would focus only on the the inequality constraints that are active as we did in the derivation of the interior point method, section 2.2. In section 5.5 [3], they explicitly go through the inequality case as well.

4.6.2 Introduction of damped BFGS

In the sections above, we have assumed that we can compute the Hessian of the Lagrangian function and that we can calculate it quickly to make the sequential algorithm work efficiently. However, this might be prohibitively expensive. In the following, we let $\tilde{H}_{\mathcal{L}_k}$ denote the Hessian of the Lagrangian at iteration k and introduce the following:

$$\begin{aligned} p_k &= x_{k+1} - x_k \\ q_k &= \nabla_x L \left(x^{k+1}, y^{k+1}, z^{k+1} \right) - \nabla_x L \left(x^k, y^{k+1}, z^{k+1} \right) \end{aligned} \tag{4.19}$$

Therefore, we use a BFGS update:

$$\tilde{H}_{\mathcal{L}_{k+1}} = \tilde{H}_{\mathcal{L}_k} - \frac{\tilde{H}_{\mathcal{L}_k} p_k p_k^\intercal \tilde{H}_{\mathcal{L}_k}}{p_k^\intercal \tilde{H}_{\mathcal{L}_k} p_k} + \frac{q_k q_k^\intercal}{q_k^\intercal p_k}, \tag{4.20}$$

Note that as y^{k+1}, z^{k+1} are used in both cases for calculation of q_k as we are interested in the curvature of the design variables space in this update. This is more clear after we have introduced a step more. In the SQP algorithm, we make an approximation, and we of cause want this quadratic approximation to have a unique solution. From previous sections, we know that a strictly convex problem will only have one unique solution i.e. when $\tilde{H}_{\mathcal{L}_k}$ is positive definite. We can ensure this by introducing:

$$r = \theta_k q_k + (1 - \theta_k)(\tilde{H}_{\mathcal{L}_k} p_k)$$

where we have introduced θ_k which we define as

$$\theta_k = \begin{cases} 1 & p^\intercal q \geq 0.2 p^\intercal (\tilde{H}_{\mathcal{L}_k} p) \\ \frac{0.8 p^\intercal (\tilde{H}_{\mathcal{L}_k} p)}{p^\intercal (\tilde{H}_{\mathcal{L}_k} p) - p^\intercal q} & p^\intercal q < 0.2 p^\intercal (\tilde{H}_{\mathcal{L}_k} p) \end{cases}, \qquad \theta_k \in [0, 1] \tag{4.21}$$

Consider now the following 3 scenarios for different values of θ_3:

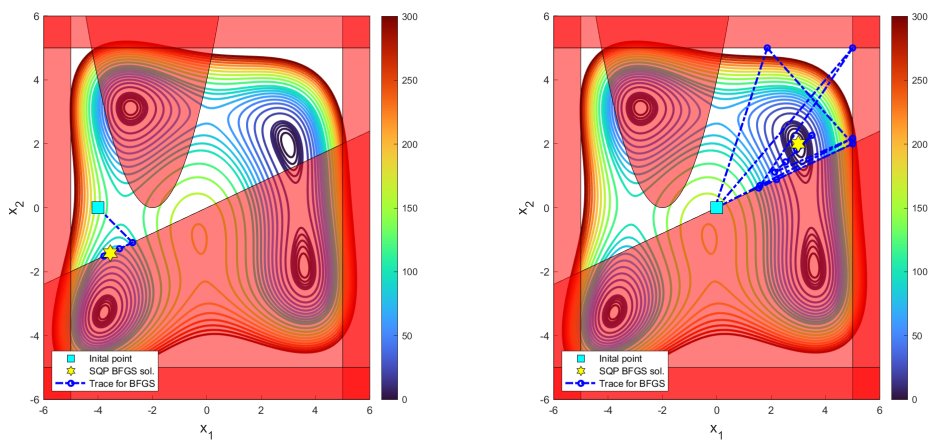
- $\theta_k = 0$: we do not update the Hessian.
- $\theta_k = 1$: We make a full BFGS update.
- $\theta_k \in]0, 1[$: This would happen if $p^\intercal q < 0.2 p^\intercal (\tilde{H}_{\mathcal{L}_k} p)$. Consider the $p^\intercal q = (\Delta x_{k+1} - \Delta x_k)^\intercal \left(\nabla_x L\left(x^{k+1}, y^{k+1}, z^{k+1}\right) - \nabla_x L\left(x^k, y^{k+1}, z^{k+1}\right)\right)$ which can be understood as the predicted contour in the next step. $0.2 p^\intercal (\tilde{H}_{\mathcal{L}_k} p)$ can be understood as the contour of the last step. The condition states that if the predicted contour is below 1/5 of the latest contour, then we should be careful as it indicates that f is flattening and we should dampen the update.

We now have a introduced a framework that allows us to formulate and solve sequences of quadratic programs. We have created an interface for selecting the SQP solver which can be found in appendix D.4. The created dampened BFGS algorithm can be seen in appendix D.5.

We tested our solver on the Himmelblau's test problem. In the table below the results are presented and the iterations can be seen in figure 4.5 for two of the initial conditions.

	$x_0 = [0.0, 0.0]$	$x_0 = [-4.0, 0.0]$	$x_0 = [-4.0, 1.0]$
fmincom			
$f(x) =$	0.00000	72.85555	35.92985
$x_1 =$	3.00000	-3.54854	-3.65461
$x_2 =$	2.00000	-1.41941	2.73772
$t[s] =$	0.04688	0.06250	0.06250
BFGS			
$f(x) =$	0.00000	72.85554	35.92985
$x_1 =$	3.00000	-3.54854	-3.65461
$x_2 =$	2.00000	-1.41941	2.73772
$t[s] =$	0.04688	0.06250	0.09375
Interations =	21	8	15
Function Calls =	44	18	32
MSE =	0	0	0

Table 4.4: Comparison of found solution of SQP BFGS and *fmincon* for three different solutions to the Himmelblau problem



(a) The iterations in the found solution using BFGS with initial point $x_0 = \begin{bmatrix} -4 & 0 \end{bmatrix}^\intercal$.

(b) The iterations in the found solution using BFGS with initial point $x_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^\intercal$.

Figure 4.5: The BFGS implementation tested on the Himmelblau's test problem.

4.7 Exercise 4.7

To ensure that the stepsize is adequate and feasible with respect to the constraints, we introduce a linear search in the following section. Consider a situation where we have just calculated the step Δx. The next step is to find an α that is optimal and ensures that the next step $x_{k+1} = x_k + \Delta x$ is feasible. Recall, we have y as the Lagrangian multiplier for the equality constraint and z as the Lagrangian for the inequality constraint. We now introduce two penalty parameters:

$$\lambda \geq |y|y, \qquad \mu \geq |z| \tag{4.22}$$

Both are used in Powell's exact ℓ_1-merit function which we will use as merit function. This is defined as:

$$P(x, \lambda, \mu) = f(x) + \lambda^\intercal |h(x)| + \mu^\intercal |\min\{0, g(x)\}| \tag{4.23}$$

The implications of this is that we subsequently update the penalty parameters according to the rule:

$$\begin{aligned} \lambda_{k+1} &= \max\left\{|y_k|, \frac{1}{2}(\lambda_k + |y_k|)\right\} \\ \mu_{k+1} &= \max\left\{|z_k|, \frac{1}{2}(\mu_k + |z_k|)\right\} \end{aligned} \tag{4.24}$$

Let $\phi(\alpha, k)$ be the merit function evaluated at time k such that:

$$\begin{aligned} \phi(\alpha, k) &= P(x^k + \alpha\Delta x, \lambda_k, \mu_k) \\ &= f(x^k + \alpha\Delta x) + \lambda_k^\intercal |h(x^k + \alpha\Delta x)| + \mu_k^\intercal |\min\{0, g(x^k + \alpha\Delta x)\}| \end{aligned} \tag{4.25}$$

We will now accept a step only if the Armijo condition is satisfied. Recall the generic Armijo condition [4, p.33]:

$$\phi(\alpha, k) \leq \phi(0, k) + c_1 \alpha \frac{\mathrm{d}\phi}{\mathrm{d}\alpha}(0), \qquad c_1 \in]0, 1[\tag{4.26}$$

An illustration of the Armijo condition can be seen in figure 4.6:

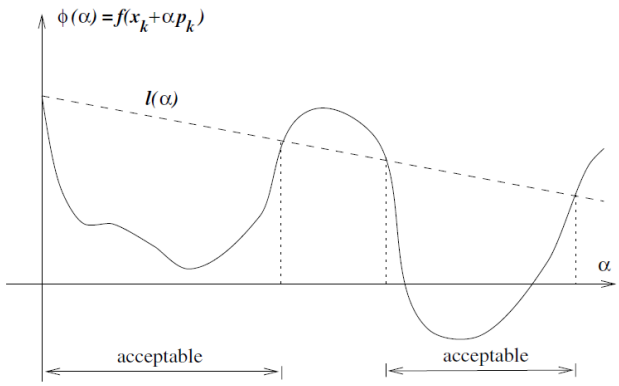


Figure 4.6: Depiction of the Armijo condition. The figure is a replica of Fig. 3.3 [4].

In our case we explicitly have from direct calculations that [9]:

$$\begin{aligned} \phi(0) &= f\left(x^k\right) + \lambda^\intercal \left|h\left(x^k\right)\right| + \mu^\intercal \left|\min\left\{0, g\left(x^k\right)\right\}\right| \\ \phi'(0) = \frac{\mathrm{d}\phi}{\mathrm{d}\alpha}(0) &= \nabla f\left(x^k\right)^\intercal \Delta x^k - \lambda^\intercal \left|h\left(x^k\right)\right| - \mu^\intercal \left|\min\left\{0, g\left(x^k\right)\right\}\right| \end{aligned} \tag{4.27}$$

We would evaluate the above expression and set them into the expression for the Armijo condition in 4.6. If the Armijo condition is met, then we stop our search and use $\alpha = 1$. If not, we will modify α according to the heuristics introduce on the slides in [9, p. 25]. Here we compute

$$\begin{aligned} a &= \frac{\phi(\alpha) - [\phi(0) + \alpha\phi'(0)]}{\alpha^2} \\ \alpha_{min} &= \frac{-\phi'(0)}{2a}. \end{aligned} \tag{4.28}$$

With these quantity we now use the update the α values according to the rule:

$$\alpha = \min\left\{0.9\alpha, \max\left\{\alpha_{\min}, 0.1\alpha\right\}\right\}. \tag{4.29}$$

The presented algorithm will sometimes accept very small step sizes. To alleviate this problem, we introduce a non-monotone strategy. This means that we will allow the algorithm to take a full step size if the step size calculated under the line search is below some tolerance ϵ. In appendix D.6, the code for the implemented algorithm can be seen. In table 4.5, we see that the solvers seems correctly implemented as it finds exactly the same minima as *fmincom*. In figure 4.7, the iterations are depicted for the three initial conditions with linear search.

	$x_0 = [0.0, 0.0]$	$x_0 = [-4.0, 0.0]$	$x_0 = [-4.0, 1.0]$
fmincom			
$f(x) =$	0.00000	72.85555	35.92985
$x =$	3.00000	-3.54854	-3.65461
	2.00000	-1.41941	2.73772
$t[s] =$	0.00000	0.04688	0.04688
SQP LS			
$f(x) =$	0.00000	72.85554	35.92985
$x =$	3.00000	-3.54854	-3.65461
	2.00000	-1.41941	2.73772
$t[s] =$	0.03125	0.01562	0.18750
Interations =	18	8	17
Function Calls =	152	50	146
MSE =	5.23347e-10	2.15061e-08	6.05941e-09

Table 4.5: SQP Line Search implementation and *fmincon* for different initial points for the Himmelblau test problem.

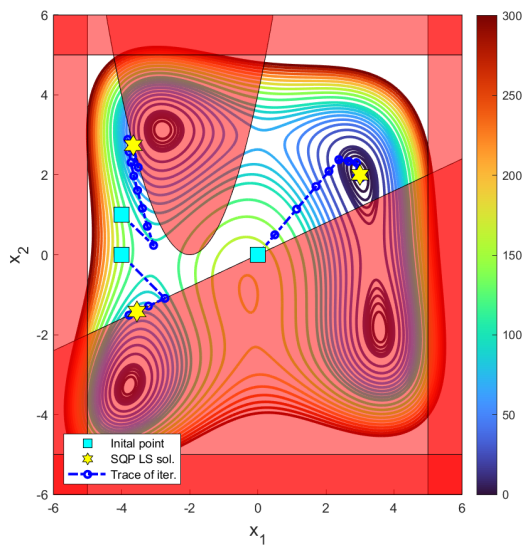


Figure 4.7: The three initial conditions depicted with their iterations for the linear search algorithm for the Himmelblau problem.

4.8 Exercise 4.8

The problem with the linear search and in combination with the methods described earlier is that we require the Hessian matrix to be positive definite. A way to alleviate this and introduce a new optimization possibility, is to use a trust region. The ideas is to define a trust region in which we search for a solution, and limit out attention to this region only. We do not strictly require the solution to be feasible but we aim to find the solution that is the most feasible and reduces the Lagrangian mostly. In the litterateur there exists relaxation, penalty and filter methods for trust region based algorithms [4, Ch. 18]. We will consider a relaxation method denoted Sℓ_1QP [4, p. 50] and base the algorithm on the slides from [10].

For trust region algorithms, we talk about solving sequences of sub-problems. For Sℓ_1QP, we move a ℓ_1 penalty into the objective. It can be shown, that this creates the following sub problem [4, Ch 18]:

$$\begin{aligned}
&\min_{p_k,v,w,t} \frac{1}{2} p_k^\intercal \nabla_{xx}^2 \mathcal{L}_k p_k + \nabla f_k^\intercal p_k + \mu \left(e_v^\intercal v + e_w^\intercal w + e_t^\intercal t\right) \\
&\text{s.t.} \quad c_i\left(x_k\right) + \nabla c_i\left(x_k\right)^\intercal p_k = v_i - w_i \quad i \in \mathcal{E} \\
&c_i\left(x_k\right) + \nabla c_i\left(x_k\right)^\intercal p_k \geq -t_i \quad i \in \mathcal{I} \\
&v, w, t \geq 0 \\
&\|p\|_\infty \leq \Delta_k
\end{aligned} \tag{4.30}$$

where we have introduce the v, w, t as slack variables, μ is a penalty parameter, and e is a vector of ones. $\|p\|_\infty \leq \Delta_k$ is constrain our step to be within the defined region. To have an idea about how well our trust region is approximating the actual problem, we introduce a merit function. In this case, the ℓ_1 merit function:

$$\phi_1(x;\mu) = f(x) + \mu \sum_{i \in \mathcal{E}} |c_i(x)| + \mu \sum_{i \in \mathcal{I}} [c_i(x)]^- \tag{4.31}$$

Where we have used the notation $[y]^- = \max\{0, -y\}$

We can now find the acceptance ratio ρ_k as

$$\rho_k = \frac{\text{actual}}{\text{predicted}} = \frac{\phi_1(x_k;\mu) - \phi_1(x_k + p_k;\mu)}{q_\mu(0) - q_\mu(p_k)} \tag{4.32}$$

We can now either reject or accept the trust region and adjust region accordingly for the next step to be adequate. In both cases, introduce a region adjusting function γ, [11]:

$$\gamma(\rho) = \min\left\{\max\left\{(2\rho - 1)^3 + 1, 0.25\right\}, 2\right\} \tag{4.33}$$

We will accept the trust region if $\rho > 0$ and reject if $\rho < 0$ in each case we will update the region according to:

$$\Delta_{k+1} = \begin{cases} \gamma(\rho)\Delta_k, & \rho > 0 \\ \gamma(\rho)\left\|\Delta x_k\right\|_\infty & \rho < 0 \end{cases} \tag{4.34}$$

The last bid to complete this algorithm is to introduce the penalty parameter as in [12] according to the rule:

$$\mu_{k+1} = \max\left(\frac{1}{2}(\mu_k + ||\gamma_k||_\infty), \frac{1}{2}(u_k + ||\xi_k||_\infty), ||\gamma_k||_\infty, ||\xi_k||_\infty\right) \tag{4.35}$$

With the theory above, we have implemented this trust region extension which can be seen in appendix D.7. We will now consider again Himmelblau's testproblem and below are the obtained results in table 4.6.

	$x_0 = [0.0, 0.0]^\intercal$	$x_0 = [-4.0, 0.0]^\intercal$	$x_0 = [-4.0, 1.0]^\intercal$
SQP TR			
$f(x) =$	0.0000	72.8555	35.9299
$x_1 =$	3.0000	-3.5485	-3.6546
$x_2 =$	2.0000	-1.4194	2.7377
time, $[s] =$	0.2031	0.0781	0.0938
Iterations =	15	12	10
'func' calls =	75	62	52

Table 4.6: SQP TR, trust region, tested on the Himmelblau' testproblem. Compare the results to that of table 4.4, to see that the solver seems correct

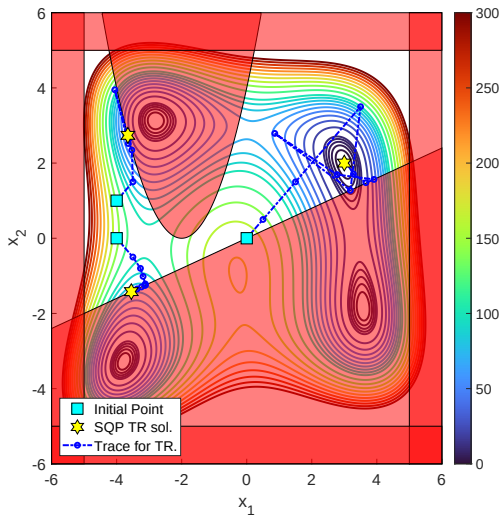


Figure 4.8: The three initial conditions depicted with their iterations for the trust region algorithm for the Himmelblau problem.

4.9 Exercise 4.9

In this exercise, we will make a summary of the results obtained above and test our solvers on the more difficult Rosenbrock problem and assess the performance.

4.9.1 Himmelblau's testproblem

In the section above, we saw that solvers we implemented always seems to find the same minima as the *fmincon* solver for the Himmelblau's testproblem. Sometimes it seemed that *fmincon* was slightly faster.

One thing we find interesting is that our trust region solver seem to follow the *fmincon* solution more than the CasADi solution. We see this very explicitly in figure 4.9. Here our solver find the same minimum as *fmincon* while *CasADi* finds another one.

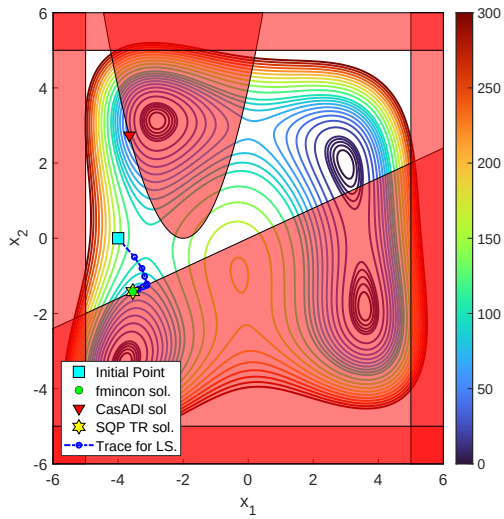


Figure 4.9: The found solutions for $x_0 = [-4, 0]^\intercal$ with iterations for the tryst region implementation. Notice that our solver finds the same solution as *fmincon* while CasADI favors the other.

4.9.2 Rosenbrock with Unit Cycle Constraints

To challenge the implemented solvers, we will make them solve the Rosenbrock problem. Initially, we will test it where we impose the unit cycle constraints. These results can of cause be compared directly with the results obtained in table 4.2. In table 4.7 we see that both the SQP dampened BFGS and the line search seems to find the correct minimum. On the other hand, it seems that the trust region solver finds a solution outside the feasible area defined by the unit circle hence it does not pass the Rosenbrock problem test.

When we compare the solvers in terms of used iterations and time spent, we see that SQP dampened BFGS uses fewest function calls and it is also the fastest for the first initial condition $x_0 = [0.0, 0.0]^\intercal$. However, for the two other initial conditions, we see that the linear search algorithm seems faster than the other two. When we compare this to the spent time of the built-in solvers in table 4.2, we see that built-in solvers are faster or as fast as ours.

NLP Solver	$x_0 = [0.0, 0.0]$	$x_0 = [-0.5, 0.0]$	$x_0 = [-0.5, -0.5]$
SQP BFGS			
$f(x) =$	0.0457	0.0457	0.0457
$x_1 =$	0.7864	0.7864	0.7864
$x_2 =$	0.6177	0.6177	0.6177
$t[s] =$	0.0625	0.0781	0.1719
iterations =	33	34	35
function calls =	68	70	72
SQP LS			
$f(x) =$	0.0457	0.0457	0.0457
$x_1 =$	0.7864	0.7864	0.7864
$x_2 =$	0.6177	0.6177	0.6177
$t[s] =$	0.0938	0.0625	0.0469
iterations =	15	31	30
function calls =	134	236	220
SQP TR			
$f(x) =$	0.0132	0.0000	0.0000
$x_1 =$	0.8850	0.9950	0.9986
$x_2 =$	0.7828	0.9900	0.9972
$t[s] =$	0.1094	0.0938	0.1719
iterations =	31	33	37
function calls =	149	161	177

Table 4.7: The 3 SQP solvers tested on the Rosenbrock problem with unit circle constraints. Notice, that the SQP trust region solver seems to find a solution outside the feasible area. SQP BFGS and SQP LS find the correct solution.

4.9.3 Rosenbrock with Box Constraints

We will now proceed and test our solvers on the Rosenbrock problem with box constraints as we initially did with the *fmincon* and CasADi solver in section 4.5.3. In that section one can also see results. These we can compare directly with the results we obtain in table 4.8.

In terms of the found minima, they all find feasible local minima with these box constraints. One should notice that the SQP dampened BFGS finds the minimum with $x^* = [-0.6985, 0.5000]^\intercal$ instead of $x^* = [1.0000, 1.0000]^\intercal$ which the other solvers, *fmincon* and CasADi finds.

In terms function calls, we again see that the BFGS solver uses fewer function calls, however, it is not drastically faster than the other solvers. When we compare

our spent CPU time with the built-in solvers, we see that our solvers tends to be somewhat slower for the Rosenbrock problem with box constraints.

NLP Solver	$x_0 = [0.0, 1.0]$	$x_0 = [-1.2, 0.5]$	$x_0 = [-0.5, 1.0]$
SQP BFGS			
$f(x) =$	2.8995	2.8995	2.8995
$x_1 =$	-0.6985	-0.6985	-0.6985
$x_2 =$	0.5000	0.5000	0.5000
$t[s] =$	0.1563	0.1094	0.0312
iterations =	11	22	9
function calls =	24	46	20
SQP LS			
$f(x) =$	0.0000	2.8995	2.8995
$x_1 =$	1.0000	-0.6985	-0.6985
$x_2 =$	1.0000	0.5000	0.5000
$t[s] =$	0.2812	0.0938	0.0469
iterations =	100	10	9
function calls =	1472	66	58
SQP TR			
$f(x) =$	0.0000	2.8995	2.8995
$x_1 =$	1.0000	-0.6985	-0.6985
$x_2 =$	1.0000	0.5000	0.5000
$t[s] =$	0.2969	0.0312	0.0156
iterations =	34	6	7
function calls =	160	32	37

Table 4.8: We see that SQP TR now outputs feasible solutions for all three initial points. Note, however, that the dampened BFGS solver finds a different minimum when we use the initial condition $x_0 = [0.0, 1.0]^\intercal$.

We conclude that all solvers seems to work adequately when we test them on the Himmelblau's testproblem, however, when we move to the Rosenbrock problem, one should choose the solver with care.

CHAPTER 5

Markowitz Portfolio Optimization

In the following, we will demonstrate an application of quadratic programming in the domain of finance. It is well-known that a diverse portfolio leads to higher return on average as the exposure to individual risk factors and securities is reduced. It is quite intuitive that a concentration risk mitigation would lead to higher returns on average. Consider a bank with exposure to only one sector; a sector breakdown would lead to a substantial loss on all asset and equity holdings. A diversified portfolio with exposure to different sectors with low degree of covariance would take a minor loss in case of one sector breakdown. In the following, we will formulate this as a Markowitz Portfolio Optimization problem and show how it can be solved with the methods introduced in this course.

Suppose we are given the following financial market with 5 securities:

Security	Covariance					Return
1	2.50	0.93	0.62	0.74	−0.23	16.10
2	0.93	1.50	0.22	0.56	0.26	8.50
3	0.62	0.22	1.90	0.78	−0.27	15.70
4	0.74	0.56	0.78	3.60	−0.56	10.02
5	−0.23	0.26	−0.27	−0.56	3.90	18.68

5.1 Exercise 5.1

Consider a construction of a portfolio of n securities where and we want find an optimal way to distributed all the of available capital. Let $x_i \in [0, 1]$ be the fractional amount invested in security i. As we require the portfolio be a convex combination of the securities, we explicitly have the constraint:

$$\sum_{i=1}^{n} x_i = 1 \tag{5.1}$$

Introduce r_i as the return on the i'th security over a specified period and introduce R as the portfolio return over the same period. It follows that R must be the investment weighted returns on the securities in the portfolio:

$$R = \sum_{i=1}^{n} x_i r_i \tag{5.2}$$

In the following, we assume $r_i \sim \mathcal{N}\left(\mu_i, \sigma_i^2\right)$ i.e. that the returns of the i'th security follows a normal distribution with mean μ_i and variance σ_i^2. We will not assume independence between the returns of securities in our portfolio hence we can write the joint distribution of the returns as a multivariate normal. Let $\mathbf{r} \in \mathbb{R}^n$ be the vector of returns, then $\mathbf{r} \sim \mathcal{N}_n\left(\mu, \Sigma\right)$ where $\mu \in \mathbb{R}^n$ is the vector of mean values of returns and $\Sigma \in M_{n\times n}\left(\mathbb{R}_+\right)$ is the covariance matrix of the returns of the securities which is of cause positive definite. If we introduce ρ_{ij} as the covariance of security i and j, then we can write Σ as:

$$\Sigma_{ij} = \rho_{ij}\sigma_i\sigma_j. \tag{5.3}$$

As portfolio managers, we are interested in the moments of the returns on a portfolio level. As R is convex combination of the assets in our portfolio and due to the linearity of the expectation operator, we obtain the expected portfolio return as:

$$\begin{aligned}\mathbb{E}[R] &= \mathbb{E}\left[\sum_{i=1}^{n} x_i r_i\right] \\ &= \sum_{i=1}^{n} x_i \mathbb{E}[r_i] \\ &= x^{\intercal}\mu.\end{aligned} \tag{5.4}$$

As managers, we would also like to know the volatilities of the expected returns to determine if it matches that of our risk appetite. As presented in e.g. section 22.4 [13], we can construct the portfolio variation using:

$$\begin{aligned}\mathrm{Var}[R] &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j \sigma_i \sigma_j \rho_{ij} \\ &= x^{\intercal}\Sigma x\end{aligned} \tag{5.5}$$

In any portfolio construction, we would like to invest in securities that maximizes the expectation of the portfolio returns and minimizes the variance. However, there is no free lunch on the financial markets and high returns would often be associated with high risk. Therefore, introduce $\kappa \in [0, \infty[$ as a risk tolerance parameter and then we

can formulate our objective as:

$$\begin{aligned} \max_x \quad & x^\intercal \mu - \kappa x^\intercal \Sigma x, \\ \text{s.t.} & \sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0 \end{aligned} \tag{5.6}$$

We now see that this takes more and more a form as that of a quadratic program from the earlier chapters. We would see this as a quadratic program immediately if we formulate it as a minimization problem and let the quadratic term be first such that:

$$\begin{aligned} \min_x \quad & -x^\intercal \mu + \kappa x^\intercal \Sigma x, \\ \text{s.t.} & \sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0 \end{aligned} \tag{5.7}$$

In the section above, we have shown how we can pose Markowitz' Portfolio optimization problem as a quadratic optimization problem.

5.2 Exercise 5.2

There is no immediate answer to which minimum and maximum value the returns of a financial market can attain. In the introduced financial market and under the normal distribution the assumption, the returns could be any real number. Indeed one could under the distributional assumption find percentiles to quantify the value of possible returns with some certainty. On the flip side, we know that returns on financial markets tends to have heavier tails than what is described by the Gaussian distribution section 13, [14] hence they might not even be good measures.

If we focus only on minimum and maximum expected returns and forget about volatility, then we see directly from the market statistics table, that the minimum is 8.5 and the maximum is 18.68.

5.3 Exercise 5.3

To formulate this as a optimization problem, we rephrase slightly Markowitz' Portfolio optimization problem as we know what the return of the our portfolio should be and we only want to minimize volatility i.e. minimize the risk on our return. We, therefore, introduce $\sum_{i=1}^{n} \mu_i x_i = 12$ as a constraint, take our $x^\intercal \mu$ from the objective and remove

κ as it would be irrelevant under a predetermined return:

$$\begin{aligned} \min_x \quad & x^\intercal \Sigma x \\ \text{s.t.} \quad & \sum_{i=1}^n x_i = 1 \\ & \sum_{i=1}^n \mu_i x_i = 12 \\ & x_i \geq 0 \end{aligned} \tag{5.8}$$

In E.1, we provide the code used to solve this problem. We have used *quadprog* to solve the problem. In the found solution, we of cause retained $\mathbb{E}[R] = 12$ and found the portfolio variance and optimal portfolio allocation to be:

$$\text{Var}[R] = 0.7654, \quad x = \begin{bmatrix} 0.0000 & 0.4765 & 0.2551 & 0.1234 & 0.1449 \end{bmatrix}^\intercal \tag{5.9}$$

5.4 Exercise 5.4

Different investors will have different risk appetites and desires for a high return. For our artificial market of 5 securities, it would be interesting to see the risk taken as a function of the expected return of our portfolio. There of cause exists a multitude of possible portfolio allocations for each desired return. We will focus on the efficient frontier which is the allocation that gives the minimal risk as a function of portfolio returns. We therefore formulate a sequence of problem as in equation 5.8:

$$\begin{aligned} \min_x \quad & x^\intercal \Sigma x \\ \text{s.t.} \quad & \sum_{i=1}^n x_i = 1 \\ & \sum_{i=1}^n \mu_i x_i = R \\ & x_i \geq 0. \end{aligned} \tag{5.10}$$

In the sequence of problems in equation 5.10, we change R and in this case we found an interesting interval, $R \in [8.5, 18.68]$ and sampled 1000 equidistant points in that interval. The found solutions are depicted below:

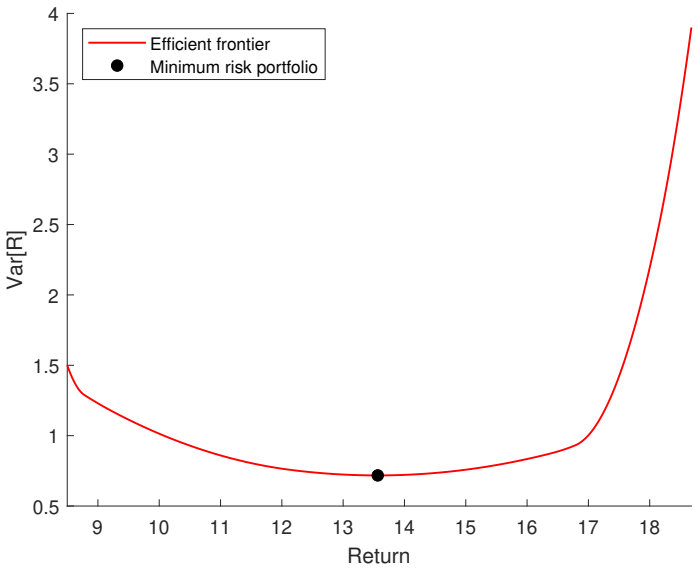


Figure 5.1: The efficient frontier as a function of the return on the portfolio

In figure 5.1, we made a dot. This is the tipping point at which we would get the highest return with minimal risk. For portfolio manager this points is often used to asses how to make the portfolio allocation. Explicitly, the points is attain at:

$$\mathrm{Var}[R] = 0.7176, \quad R \approx 13.56, \quad x = \begin{bmatrix} 0.0873 & 0.3071 & 0.3017 & 0.0999 & 0.2040 \end{bmatrix}^{\intercal} \tag{5.11}$$

To analyse how the allocations changes with R and the efficient frontier, consider the figure below:

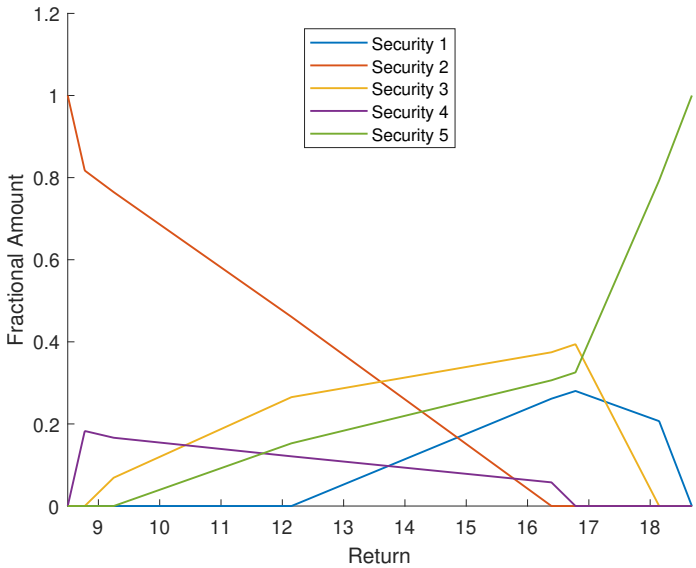


Figure 5.2: The fractional amount invested in each of the securities.

We will comment on three major shifts in portfolio allocation:

- Security 2 is decreasing as we increase the expected return R. If we consult the summary statistics table, we see that it has the lowest return but also the lowest variance of only 1.50. This means that its is the safest hold. Indeed, when $R = 8.5$, then we allocate all our available capital to security 1.
- Security 5 is increasing as we increase R. This also makes sense intuitively as it is the one that gives the highest return on average, however it also has the largest variance of 3.90.
- The minimal risk is seen when we allocate capital to all securities which we motivated as the benefit of diversification. To spot this directly, we see that some of the covariance are negative for security 5 and the other securities. We can hence *hedge* our exposures in asset 5 by also allocating capital to asset 1-4 which reduces the overall variance of our portfolio.

5.5 Bi-criterion optimization, Exercise 1

In the following, we will work with the bi-criterion optimization problem which can be formulated as

$$\min_x \ \alpha x^\intercal \Sigma x - (1-\alpha) x^\tau \mu, \quad \text{s.t.} \ \sum_{i=1}^{n} x_i = 1, \quad x_i \geq 0 \tag{5.12}$$

In equation 5.12, we have introduced $\alpha \in [0,1]$ which we can interpret directly as a risk appetite parameter. If $\alpha << 1$ then the portfolio manager has a high risk appetite and if $\alpha >> 0$, then the portfolio manager is very risk adverse.

5.6 Bi-criterion optimization, Exercise 2

In the following, we will solve the bi-criterion problem using the interior point methods we developed in chapter 2 for EQPs. The driver for this exercise can be found in appendix E.1. The curve in figure 5.3 is created sampling α for 1000 equidistant values.

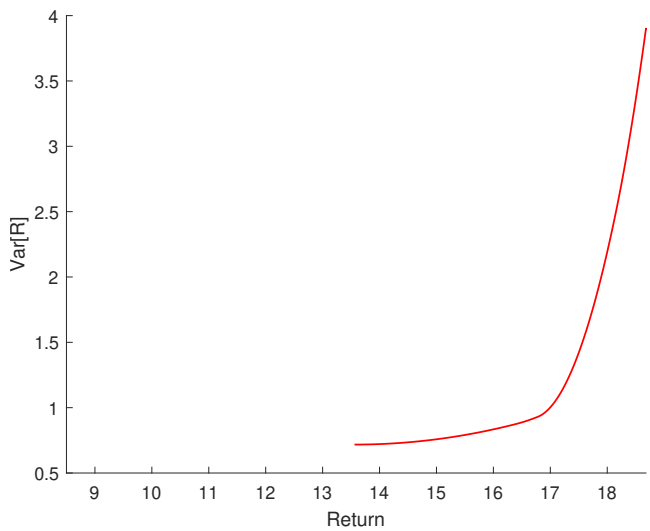


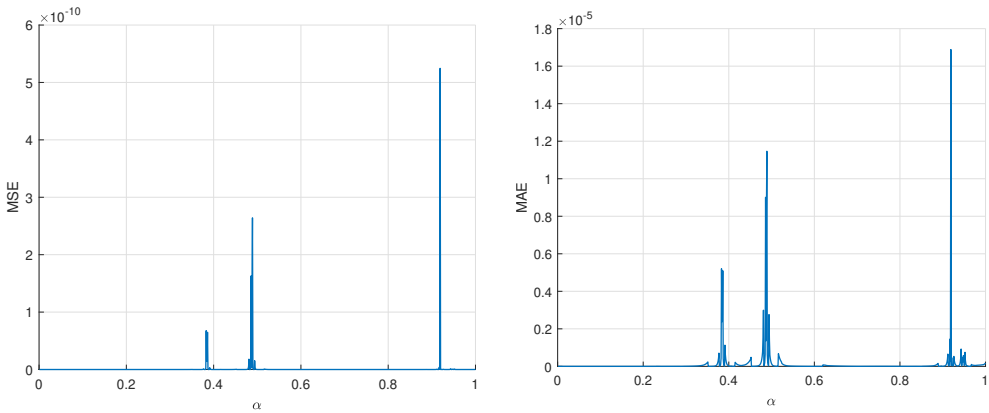
Figure 5.3: Solutions for the bi-criterion problem 1000 equidistant sampled α values in the interval from 0 to 1.

In figure 5.3, we see that the solution is like figure 5.1 beyond the mentioned tipping point which also intuitively makes sense as we also have the return included in the objective; after all the manager will not accept a lower expected return and higher risk.

5.7 Bi-criterion optimization, Exercise 3

We will now test the computed results from our solver from exercise 2, with the built-in *quadprog* solver.

In the figures below we will first compare the difference in the found solutions in terms of MSE and MAE:



(a) Mean Squared Error

(b) Mean absolute errors.

Figure 5.4: The errors between the implemented solvers and quadprog solutions. As the scale of the y-axis indicated, all solvers are correct.

In figure 5.4, we see that our solver seems correct with only minor, negligible differences. Instead, it would be of interest to test the performance in terms of time efficiency. In the figures below, we print the time it takes to solve the system for 1000 values of α where we repeat each value of alpha 5 times. In figure 5.5, we see that our solver seems to perform slightly better, however, it is nothing hugely different from the *quadprog* solver.

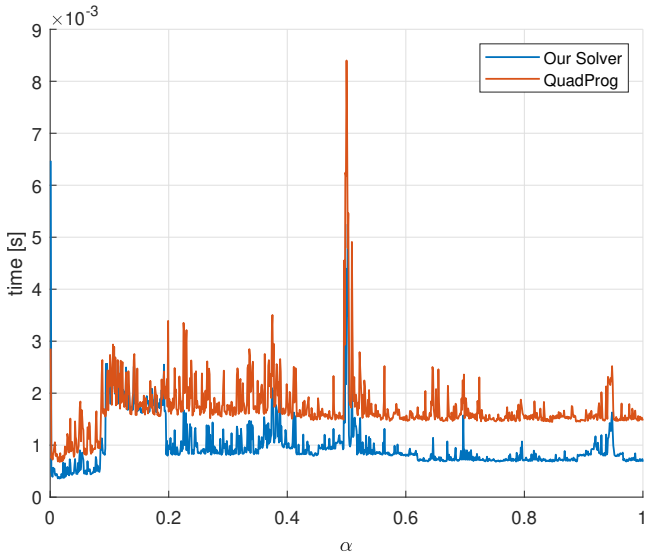


Figure 5.5: Time-efficiency for our QP solver and quadprog for bi-criterion problem for each of the sampled 1000 equidistantly values of α. The we repeat the experiment for each α 5 times.

5.8 Risk-free Asset, Exercise 1

In the following, we will consider the impacts of including a risk-free security to our artificial financial market. This means that it would have 0 variance and we further assume that it would have 0 covariance with the other securities in our portfolio. We are further asked to assume that the return on the risk free security is 0. This now means that we will have the following summary table:

Security	Covariance						Return
1	2.50	0.93	0.62	0.74	−0.23	0	16.10
2	0.93	1.50	0.22	0.56	0.26	0	8.50
3	0.62	0.22	1.90	0.78	−0.27	0	15.70
4	0.74	0.56	0.78	3.60	−0.56	0	10.02
5	−0.23	0.26	−0.27	−0.56	3.90	0	18.68
6	0	0	0	0	0	0	0

In the table above we have included the risk free security as security no. 6.

5.9 Risk-free Asset, Exercise 2

With the introduce security, we can now plot the efficient frontier as a function of the expected return. In the following, we have also included each of the securities as crosses.

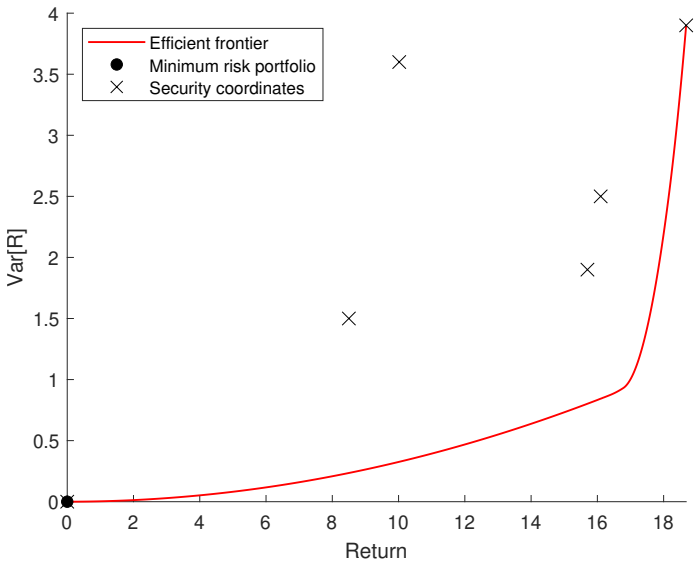


Figure 5.6: The efficient frontier as a function of the return on the portfolio with risk-free security included.

In figure 5.6, we now see that we the return is 0 if invest only in the risk-free security. However, the risk is also 0, if invest only in the risk-free security. We will now consider the portfolio allocation as a function of the returns and see how the introduction of this risk-free security changes the allocation. We see this in figure 5.7.

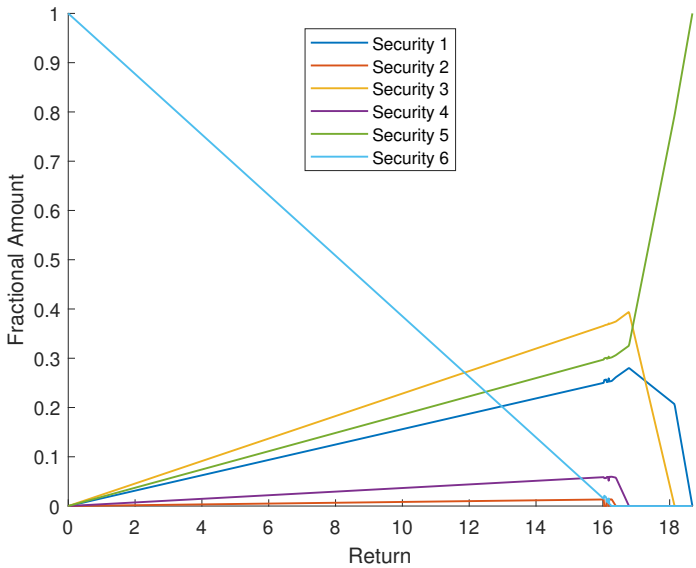


Figure 5.7: The fractional amount invested in each of the securities with the risk-free security included.

We will now comment on the major changes in the portfolio allocation with the introduced risk-free security.

- Security 6 takes on the role of security 2. If we want lower risk and accept lower return, we should just allocate more capital to the risk-free security, security 6. This was the role that security 2 had before and now we see that we hardly allocate any capital to security 2.
- Security 5 still take on the role of the high-risk, high-reward security to which we allocate great amount of capital for a high return portfolio.
- Beyond a expected return of 16, the two markets are very alike hence the risk-free security is not relevant for relevant portfolios that accept a high risk which is quite intuitive.
- The risk-free security introduces a high decree of linearity in the allocation diagram which we did not see in the financial market in the exercise before.

5.10 Risk-free Asset, Exercise 3

We will now find the optimal portfolio if we want a return of 14.00, $R = 14.00$. This we compare directly with the values found in the market without the risk-free security. We will comment on the obtained results in the subsequent exercise.

- **Prior** to risk-free security and $R = 14.00$

$$\mathrm{Var}[R] = 0.7214, \qquad x = \left[0.1141, 0.2597, 0.3130, 0.0934, 0.2198\right]^{\intercal}$$

- **With** to risk-free security and $R = 14.00$

$$\mathrm{Var}[R] = 0.6377, \qquad x = \left[0.2185, 0.0116, 0.3196, 0.0512, 0.2598, 0.1393\right]^{\intercal}$$

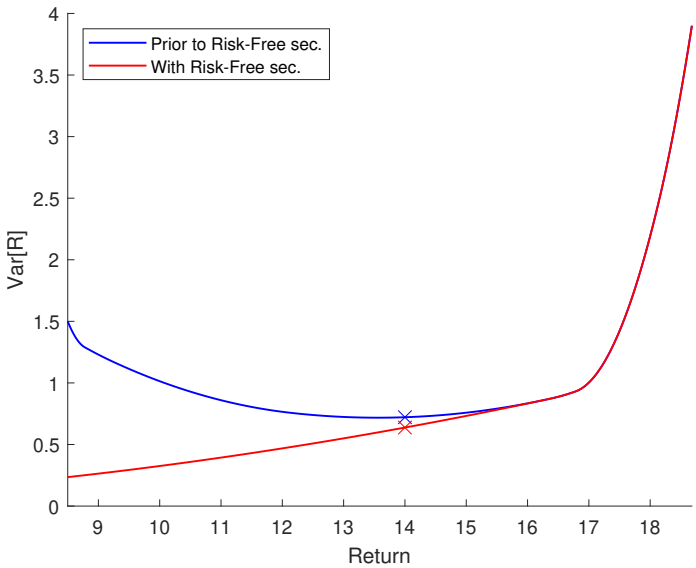


Figure 5.8: The Efficient Frontier with crosses for the return when $R = 14$ for the market prior to the introduction of the risk-free security and with the risk-free security as a valid investment opportunity.

5.11 Risk-free Asset, Exercise 4

We find that the most interesting results of the above is the implications of the risk-free asset. In figure 5.8, we see that the the risk free security offers a much more adequate risk picture as we would expect risk to be reduced when we accept a lower expected return [14].

In terms of risk allocation for e.g. $R = 14$, we see that security 2 get a diminishing size of the allocation after the introduction of the risk-free security. Before the risk-free opportunity, it had a fractional amount of 0.2597 which is reduced to only 0.0116. We also obtain a sizeable reduction in the variance of the return which goes from 0.7214 to 0.6377 after we have introduce the risk-free security to the market.

APPENDIX A

Exercise 1

A.1 EqualityQPSolver Interface

```
function [x, lambda,time] = EqualityQPSolver(H, g, A, b, solver)
% EqualityQPSolver      The solver interface for the equality EQP solvers
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b
%
%
% Syntax: [x, lambda,time] = EqualityQPSolver(H, g, A, b, solver)
%
%         x              : Solution
%         z              : Lagrange multipliers
%         time           : Time used on factorization in some of the
%                          algorithms

% Created: 03.05.2022
% Author : Nicolaj Hans Nielsen, Technical University of Denmark

%%
time = 0;
if solver == "LDLdense"
    [x, lambda] = EqualityQPSolverLDLdense(H,g,A,b);
elseif solver == "LDLsparse"
    [x, lambda] = EqualityQPSolverLDLsparse(H,g,A,b);
elseif solver == "LUdense"
    [x, lambda] = EqualityQPSolverLUdense(H,g,A,b);
elseif solver == "LUsparse"
    [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b);
elseif solver == "rangespace"
    [x, lambda] = EqualityQPSolverRangeSpace(H,g,A,b);
elseif solver == "nullspace"
    [x, lambda] = EqualityQPSolverNullSpace(H,g,A,b);
elseif solver == "quadprog"
    options = optimset('Display', 'off');
    [x, lambda] = quadprog(H,g,[],[],A',b,[],[],[],options);
else
    error("the required solver is not implemented")
end
```

Listing A.1: Interface to select solver

A.2 Generate EQP

```
function [H,g,A,b,x,lambda] = generateEQP(n_var,n_con)
% generateEQP    Generate a random EQP with n_var variables and n_con
%                        constraints
```

```
%
%
% Syntax: [H,g,A,b,x,lambda] = generateEQP(n,m)
%
%         x                 : Solution
%         lambda            : Lagrange multipier
%         H                 : Hessian
%         g                 : Linear term of the objective
%         A                 : Matrix of the constraints
%         b                 : lhs of the constraints

% Created: 15.05.2022
% Author: Nicolaj Hans Nielsen

%%
    % Create positive symmetric matrix
    % using input from
    % https://se.mathworks.com/matlabcentral/answers/424565-how-to-generate-a-symmetric-positive-definite-matrix
    H = rand(n_var);
    H = H*H.';

    % Create a matrix of full rank using
    % https://se.mathworks.com/matlabcentral/answers/490698-random-matrix-full-rank
    rand_matrix = rand(n_var);
    rand_matrix_tril = tril(rand_matrix,-1);
    full_rank_matrix = rand_matrix_tril + rand_matrix_tril'+eye(n_var).*rand(n_var);
    A = full_rank_matrix(:,1:n_con);

    % Create a random solution to the system
    x = rand(n_var,1);
    lambda = rand(n_con,1);

    % Create the matching g and b
    KKT = [H -A;-A' zeros(n_con)];
    sol = [x; lambda];
    rhs = KKT*sol;

    g = -rhs(1:n_var);
    b = -rhs(n_var+1:n_var+n_con);
end
```

Listing A.2: Generate an EQP with the necessary properties

A.3 Driver for Exercise 1

```
% This is the Driver for Exercise 1 where we document that we have tested
% the implemented methhods.
```

```
%% Testing correctness of the solvers
% define list of solvers
names = ["LUdense", "LUsparse", "LDLdense", "LDLsparse", "rangespace", "
    nullspace", "quadprog"];

% define the number of test as equidistant pts between 8.5 and 18.68
tests = 15;
bs = linspace(8.5, 18.68, tests);

% initialize output vectors
times = zeros(7,tests);
output_solution = zeros(5,tests,7);

% loop over each test
for test_i = 1:tests
    b_i = bs(test_i);
    n = 5;
    H = [5.0 1.86 1.24 1.48 -0.46; ...
        1.86 3.0 0.44 1.12 0.52; ...
        1.24 0.44 3.8 1.56 -0.54; ...
        1.48 1.12 1.56 7.2 -1.12; ...
        -0.46 0.52 -0.54 -1.12 7.8];
    g = [-16.1; -8.5; -15.7; -10.02; -18.68];
    A = [16.1 8.5 15.7 10.02 18.68; 1.0 1.0 1.0 1.0 1.0]';
    b = [b_i;1];

    % We test all solvers and save the computed outputs
    for solver_j = 1:length(names)
        start = cputime;
        [x, lambda] = EqualityQPSolver(H,g,A,b, names(solver_j));
        output_solution(:, test_i, solver_j) = x;
        times(solver_j, test_i) = cputime-start;
    end
end

%% Compute errors to compare correctness of solvers

% the mean squared error
output_solution_mean_sq_diff = zeros(6,tests);
for i=1:6
    output_solution_mean_sq_diff(i,:) = mean((output_solution(:,:,i)-
        output_solution(:,:,7)).^2);
end


% Mean absolute error
output_solution_MAE = zeros(6,tests);
for i=1:6
    output_solution_MAE(i,:) = mean(abs(output_solution(:,:,i)-
        output_solution(:,:,7)));
end

%% Plot MSE

```

```
figure
for i=1:6
    plot(bs, (output_solution_mean_sq_diff(i,:)))
    hold on
end
legend(names(1:6))
xlabel("\theta")
ylabel("MSE")
saveas(gcf,'./figures/Ex1/fig14MSE','epsc')

%% Plot MAE
figure
for i=1:6
    plot(bs, (output_solution_MAE(i,:)))
    hold on
end
legend(names(1:6))
xlabel("\theta")
ylabel("MAE")
saveas(gcf,'./figures/Ex1/fig14MAE','epsc')

%% Results
latex(output_solution(:,[1,5,10,15],7))

%% Plot the time it took the solver to solve the problem

hold off
for i=1:size(times,1)
    plot(bs, times(i,:))
    hold on
end
legend(names)
xlabel("b(1)")
ylabel("time [log s]")


%% Introduce the Recycling problem

ns = [100:100:900 1000:500:3000];
tests = length(ns);
times = ones(7,tests)*tests;

% test each solver with different values of n
for i = 1:tests
    disp("running on n="+ ns(i))

    [H, g, A, b] = ProblemEQPRecycling(ns(i), 0.2, 1);

    for solver_j = 1:length(names)
        start = cputime;
        [x, lambda] = EqualityQPSolver(H,g,A,b, names(solver_j));
        times(solver_j, i) = cputime-start;
    end
end

```

```
%% Plot the CPU time for recycle problem of different size.
figure
hold off
for i=1:size(times,1)
    plot(ns, times(i,:))
    hold on
end
legend(names, 'Location','northwest')
xlabel("n")
ylabel("time [s]")
saveas(gcf,'./figures/Ex1/recycleProblem','epsc')

%% Factorization comparison with random EQP

ns = [1000:200:3000];
tests = length(ns);

times = ones(5,tests)*100;
%We bench LU vs LDL versus Cholesky of varying sizes
for i = 1:tests
    disp("running on n="+ ns(i))
    j=1;

    [H,g,A,b,x,lambda] = generateRandomEQP(ns(i),ns(i));
    KKT = [H -A;-A', zeros(size(A,2), size(A,2))];
    KKT_sparse = sparse(KKT);

    start = cputime;
    x = lu(KKT,'vector');
    times(j, i) = min(cputime-start, times(j,i));
    j = j+1;


    start = cputime;
    x = lu(KKT_sparse,'vector');
    times(j, i) = min(cputime-start, times(j,i));
    j = j+1;

    start = cputime;
    [x, , ] = ldl(KKT,'vector');
    times(j, i) = min(cputime-start, times(j,i));
    j = j+1;

    start = cputime;
    [x, , ] = ldl(KKT_sparse,'vector');
    times(j, i) = min(cputime-start, times(j,i));
    j = j+1;

    start = cputime;
    x = chol(H);
    times(j, i) = min(cputime-start, times(j,i));
    j = j+1;

end
```

```
%% Plot the CPU times for the factorizations

hold off
for test_i=1:5
    plot(ns, times(test_i,:))
    hold on
end
legend(["LU KKT", "LU KKT Sparse", "LDL KKT", "LDL KKT Sparse", "Cholesky H
    matrix"], 'Location','northwest')
xlabel("n")
ylabel("time [s]")

saveas(gcf,'./figures/Ex1/matrixFactorization','epsc')


%% Generate Numorous EQP and to test performance

ns = [100:100:900 1000:500:3000];
tests = length(ns);
times = ones(7,tests)*tests;

% For each generated EQP, we test the solvers
for i = 1:tests
    disp("running on n="+ ns(i))

    [H,g,A,b,x,lambda] = generateEQP(ns(i), ns(i));

    for solver_j = 1:length(names)
        start = cputime;
        [x, lambda] = EqualityQPSolver(H,g,A,b, names(solver_j));
        times(solver_j, i) = cputime-start;
    end
end

%% Show Performance on Randomly Generated EQPs

figure
hold off
for i=1:size(times,1)
    plot(ns, times(i,:))
    hold on
end
legend(names, 'Location','northwest')
xlabel("n")
ylabel("time [s]")
saveas(gcf,'./figures/Ex1/random_eqp_large','epsc')
```

Listing A.3: Driver for Exercise 1

APPENDIX B

Exercise 2

B.1 Initial Point Heuristics Algorithm

```
function [x,y,z,s] = IntialPointHeuristics(H,g,A,b,l,u,x0,y0,z0,s0)
% IntialPointHeuristics   This function finds the inital point using the
%                           heuristics introduced in reppart section 2.3
%
%
% Syntax: [x,y,z,s, iter, ldltime] = IntialPointHeuristics(H,g,A,b,l,u,x0,y0
    ,z0,s0)
%
%         x              : Design variables
%         y              : Equality Lagrange multipliers
%         z              : Inequality Lagrange multipliers
%         s              : Slack variables

% Created: 15.05.2022
% Author: Nicolaj Hans Nielsen

    % Set problem specific constants
    n_design = length(x0);
    n_eq_con = length(y0);

    % Lagrange multipliers
    sl = s0(1:n_design);
    su = s0(n_design+1:n_design*2);
    zl = z0(1:n_design);
    zu = z0(n_design+1:n_design*2);

    %initial residuals
    r_l = H*x0+g-A*y0-(z0(1:n_design)-z0(n_design+1:n_design*2));
    r_a = b-A'*x0;
    r_c = s0+[l; -u] - [x0; -x0];

    % Start point heuristic
    zsl = zl./sl;
    zsu = zu./su;
    Hbar = H + diag(zsl + zsu);
    KKT = [Hbar -A; -A' zeros(n_eq_con)];
    KKT = sparse(KKT);

    % make ldl factorization of the KKT matrix
```

```
    [L,D,p] = ldl(KKT,'vector');


    %
    r_cs = (r_c-s0);
    r_l_bar = r_l - zsl.*r_cs(1:n_design) +zsu.*r_cs(1+n_design:2*n_design);
    rhs = -[r_l_bar; r_a];

    solution(p) = L' \ (D \ (L \rhs(p)));

    dxAff = solution(1:length(x0))';
    dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z0./s0).*r_cs;
    dsAff = -s0-(s0./z0).*dzAff;

    %Update of starting point
    x = x0;
    y = y0;
    z = max(1,abs(z0+dzAff));
    s = max(1,abs(s0+dsAff));
end
```

Listing B.1: Initial Point Heuristics Algorithm

B.2 Mehrota's Interior Point methods Algorithm

```
function [x,y,z,s, iter] = quadraticPrimalDualIM_box(H,g,A,b,l,u,x0,y0,z0,s0
    )
% quadraticPrimalDualIM_box          An interior point solver based on
    Mehrota's predictor-corrector
%                                    primal-dual interior point algorithm. It
     takes
%                                    problems of the form
%
%             n_design     x'Hx+g'x
%              x
%             s.t      Ax  = b
%                 u>=  x >= l
%
%
% Syntax: [x,y,z,s, iter, ldltime] = quadraticPrimalDualIM_box(H,g,A,b,l,u,
    x0,y0,z0,s0)
%
%         x              : Solution
%         y              : Equality lagrange multipliers
%         z              : Inequality lagrange multipliers
%         s              : Slack variables
%         iter           : Iterations used
%         ldltime        : Time used on ldl factorization

% Created: 15.05.2022
```

```
% Author: Nicolaj Hans Nielsen

%%
    % Sets constants for the algorithm

    % Set problem specific constants
    n_design = length(x0);
    n_eq_con = length(y0);
    epsilon = 0.000001;
    ldltime = 0;
    max_iter = 100;
    eta = 0.995;
    iter = 0;

    % Using Intial Point Heuristics
    [x, y, z, s] = IntialPointHeuristics(H,g,A,b,l,u,x0,y0,z0,s0);

    % intialize identity matrix
    e = ones(n_design*2,1);

    % Update constraints
    sl = s(1:n_design);
    su = s(n_design+1:n_design*2);
    zl = z(1:n_design);
    zu = z(n_design+1:n_design*2);

    % Update of initial residuals
    rL = H*x+g-A*y-(zl-zu);
    rA = b-A'*x;
    rC = s+[l; -u] - [x; -x];

    % Initial dual gap
    dualGap = (z'*s)/(2*n_design);
    dualGap0 = dualGap;


    for i = 1:max_iter
        iter = iter + 1;
        zsl = zl./sl;
        zsu = zu./su;
        Hbar = H + diag(zsl + zsu);
        KKT = [Hbar -A; -A' zeros(n_eq_con)];
        KKT = sparse(KKT);
        start = cputime;
        [L,D,p] = ldl(KKT,'vector');
        ldltime = ldltime + cputime-start;

        % Compute the direct affince step
        rCs = (rC-s);
        rLbar = rL - zsl.*rCs(1:n_design) +zsu.*rCs(1+n_design:2*n_design);

        rhs = -[rLbar ; rA];
        solution(p) = L' \ (D \ (L \rhs(p)));

        dxAff = solution(1:length(x))';
```

```
        dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z./s).*rCs;
        dsAff = -s-(s./z).*dzAff;

        % Compute the maximum affine step
        dZS = [dzAff; dsAff];
        alphas = (- [z;s]./dZS);
        alphaAff = min([1;alphas(dZS<0)]);

        dualGapAff = ((z + alphaAff * dzAff)'*(s + alphaAff * dsAff))/(2 *
            n_design);
        sigma = (dualGapAff/dualGap)^3;


        % Affine-Centering-Correction Direction
        rSZz =  s + dsAff.*dzAff./z-dualGap * sigma * e./z;
        rCs = (rC-rSZz);
        rLbar = rL - zsl.*rCs(1:n_design) +zsu.*rCs(1+n_design:2 * n_design)
            ;

        rhs = -[rLbar ; rA];
        solution(p) = L' \ (D \ (L \rhs(p)));

        dx = solution(1:length(x))';
        dy = solution(length(x)+1:length(x)+length(y))';

        dz = - [zsl.*dx; -zsu.*dx] + (z./s).*rCs;
        ds = -rSZz-(s./z).*dz;

        % Compute the max alpha
        dZS = [dz; ds];
        alphas = (-[z;s]./dZS);
        alpha = min([1;alphas(dZS<0)]);

        alphaBar = eta*alpha;

        % We can now update the position
        x = x + alphaBar * dx;
        y = y + alphaBar * dy;
        z = z + alphaBar * dz;
        s = s + alphaBar * ds;

        % Make the correct subset of s and z to subsequently calculate
        % the residuals.
        sl = s(1:n_design);
        su = s(n_design+1:n_design*2);
        zl = z(1:n_design);
        zu = z(n_design+1:n_design*2);

        % Update the residuals
        rL = H*x+g-A*y-(z(1:n_design)-z(n_design+1:n_design*2));
        rA = b-A'*x;
        rC = s+[l; -u] - [x; -x];

        % Compute the dual gap
```

```
        dualGap = (z'*s)/(2*n_design);

        % Check for convergence
        if(dualGap <= epsilon*0.01*dualGap0)
            return
        end
    end
end
```

Listing B.2: Mehrota's Interior Point methods Algorithm

B.3 Driver for Exercise 2

```
%% Testing correctness of the solvers for the given problem

% define the number of test as equidistant pts between 8.5 and 18.68
tests = 15;
bs = linspace(8.5, 18.68, tests);

% initialize output vectors
times = zeros(tests,2);
output_solution = zeros(5,tests,2);
iterations = zeros(tests, 2);

% Matrices and vectors for the defined problem
H = [5.0 1.86 1.24 1.48 -0.46; ...
    1.86 3.0 0.44 1.12 0.52; ...
    1.24 0.44 3.8 1.56 -0.54; ...
    1.48 1.12 1.56 7.2 -1.12; ...
    -0.46 0.52 -0.54 -1.12 7.8];
g = [-16.1; -8.5; -15.7; -10.02; -18.68];
A = [16.1 8.5 15.7 10.02 18.68; 1.0 1.0 1.0 1.0 1.0]';
l = zeros(5,1);
u = ones(5,1);


% loop over each of the 15 test
for test_i = 1:tests
    b_i = bs(test_i);
    n = 5;
    b = [b_i;1];

    % set options for quad prog
    options = optimset('Display', 'off', 'Algorithm', 'interior-point-convex
        ');
    start = cputime;
    [x_quadProg, optval, exitflag,output] = quadprog(H, g, [],[], A', b,l,u,
        0, options);
    times(test_i,1) = cputime-start;

```

```
    % save results
    output_solution(:,test_i,1) = x_quadProg;
    iterations(test_i,1) = output.iterations;

    % Setup vectors etc. to input to our solver
    x0 = zeros(n,1);
    s0 = ones(2*n,1);
    y0 = ones(length(b),1);
    z0 = ones(2*n,1);

    % solve with our solver
    start = cputime;
    [x_own,y,z,s, iter] = quadraticPrimalDualIM_box(H,g,A,b,l,u,x0,y0,z0,s0)
        ;
    times(test_i,2) = cputime-start;

    % save results
    output_solution(:,test_i,2) = x_own;
    iterations(test_i, 2) = iter;

    if exitflag  = 1
        disp("Ran Into Problems")
    end
end

%% Compute the difference in the found Solution

% the mean squared error
output_solution_mean_sq_diff = mean((output_solution(:,:,2)-output_solution
    (:,:,1)).^2);

% mean absolute error
output_solution_MAE = mean(abs(output_solution(:,:,2)-output_solution(:,:,1)
    ));

%% Plot MSE Correctness

figure
plot(bs, (output_solution_mean_sq_diff))
xlabel("\theta")
ylabel("MSE")
saveas(gcf,'./figures/Ex2/ProbEx1_OwnSolverTestMSE','epsc')

%% Plot MAE Correctness
figure
plot(bs, output_solution_MAE)
xlabel("\theta")
ylabel("MAE")
saveas(gcf,'./figures/Ex2/ProbEx1_OwnSolverTestMAE','epsc')

%% Test Time Performance of Simple Problem
figure
plot(bs, times)
legend(["quadprog";"our solver"], 'Location','northeast')
xlabel("\theta")
```

```
ylabel("CPU time")
saveas(gcf,'./figures/Ex2/ProbEx1_CPU_time','epsc')

%% Test Iteration Performance of Simple Problem
figure
plot(bs, iterations)
legend(["quadprog";"our solver"], 'Location','northwest')
xlabel("\theta")
ylabel("Iterations")
saveas(gcf,'./figures/Ex2/ProbEx1_iteration','epsc')


%% Generate a set of larger problems to be solved

ns = [100:100:900 1000:500:3000];
tests = length(ns);

% we only care about time and iteration performance
times = ones(tests,2)*tests;
iterations = zeros(tests, 2);

% Run the algorithms for each of the tests
for i = 1:tests
    disp("running on n="+ ns(i))

    % generate the eqp
    [H,g,A,b,x,lambda] = generateEQP(ns(i), ns(i)/2);

    % box constraints
    l = zeros(ns(i),1);
    u = ones(ns(i),1);

    % solve using quad prog
    options = optimset('Display', 'off', 'Algorithm', 'interior-point-convex');
    start = cputime;
    [x_quadProg, optval, exitflag,output] = quadprog(H, g, [],[], A', b,l,u, 0, options);
    times(i,1) = cputime-start;

    % save results
    iterations(i,1) = output.iterations;

    % solve using own solver
    x0 = zeros(ns(i),1);
    s0 = ones(2*ns(i),1);
    y0 = ones(length(b),1);
    z0 = ones(2*ns(i),1);

    start = cputime;
    [x_own,y,z,s, iter] = quadraticPrimalDualIM_box(H,g,A,b,l,u,x0,y0,z0,s0);
    times(i,2) = cputime-start;

    % save results
```

```
    %output_solution(:,test_i,2) = x_own;
    iterations(i, 2) = iter;

    if exitflag  = 1
        disp("Ran Into Problems")
    end
end

%% Test Time Performance on the Larger Problem
figure
plot(ns, times)
legend(["quadprog";"our solver"], 'Location','northwest')
xlabel("n")
ylabel("CPU time")
saveas(gcf,'./figures/Ex2/Prob_large_CPU_time','epsc')

%% Assess Number of Iterations for the Large Problem
figure
plot(ns, iterations)
legend(["quadprog";"our solver"], 'Location','northwest')
xlabel("n")
ylabel("Iterations")
saveas(gcf,'./figures/Ex2/Prob_large_iteration','epsc')
```

Listing B.3: Driver for Exercise 2

APPENDIX C

Exercise 3

C.1 Initial Point Heuristics Algorithm for LPs

```
function [x,y,z,s] = IntialPointHeuristicsLP(g,A,b,l,u,x0,y0,z0,s0)
% IntialPointHeuristics   This function finds the inital point using the
%                           heuristics introduced in rapport section 233
%
%
% Syntax: [x,y,z,s, iter, ldltime] = IntialPointHeuristics(H,g,A,b,l,u,x0,y0
    ,z0,s0)
%
%         x              : Design variables
%         y              : Equality Lagrange multipliers
%         z              : Inequality Lagrange multipliers
%         s              : Slack variables

% Created: 15.05.2022
% Author: Nicolaj Hans Nielsen

    % Set problem specific constants
    n_design = length(x0);
    n_eq_con = length(y0);

    % Lagrange multipliers
    sl = s0(1:n_design);
    su = s0(n_design+1:n_design*2);
    zl = z0(1:n_design);
    zu = z0(n_design+1:n_design*2);

    % initial residuals
    r_l = g-A*y0-(z0(1:n_design)-z0(n_design+1:n_design*2));
    r_a = b-A'*x0;
    r_c = s0+[l; -u] - [x0; -x0];

    % Start point heuristic
    zsl = zl./sl;
    zsu = zu./su;

    r_cs = (r_c-s0);
    r_l_bar = r_l - zsl.*r_cs(1:n_design) +zsu.*r_cs(1+n_design:2*n_design);

    % compute diagonal h inv easily
```

```
    diag_h_inv = 1./(zsl+zsu);

    % make the Cholesky-factorization
    normalfactor = A' * (diag_h_inv .*  A);
    R = chol(normalfactor);

    % caluclate the step
    dy = R \ (R' \ (r_a + A' * (diag_h_inv .* r_l_bar)));
    dx = diag_h_inv .* (-r_l_bar + A*dy);
    dz = -(z0./s0).*[dx; -dx] + (z0./s0).*r_cs;
    ds = -s0-(s0./z0).*dz;

    %Update of starting point
    z = max(1,abs(z0+dz));
    s = max(1,abs(s0+ds));


    %Update of starting point
    x = x0;
    y = y0;
    z = max(1,abs(z+dz));
    s = max(1,abs(s+ds));
end
```

Listing C.1: Initial Point Heuristics Algorithm for LPs

C.2 Mehrota's Interior Point methods Algorithm for LPs

```
function [x,y,z,s, iter] = LinearPrimalDualIM_box(g,A,b,l,u,x0,y0,z0,s0)
% LinearPrimalDualIM_box   An interior point solver for LP. It is based on
%                          Mehrota's predictor-corrector primal-dual
%                          Interior point algorithm. It takes problems of
%                          the form
%
%            min     g'x
%             x
%            s.t      Ax  = b
%                u>=  x >= l
%
%
% Syntax: [x,y,z,s, iter] = LinearPrimalDualIM_box(g,A,b,l,u,x0,y0,z0,s0)
%
%         x              : Solution
%         y              : Equality lagrange multipliers
%         z              : Inequality lagrange multipliers
%         s              : Slack variables
%         iter           : Iterations used

% Created: 15.05.2022
% Author: Nicolaj Hans Nielsen, DTU
```

```
%%
    % Sets constants for the algorithm
    n_design = length(u);
    epsilon = 0.000000001;
    max_iter = 100;
    eta = 0.995;
    iter = 0;

    % identity matrix
    e = ones(n_design*2,1);

    % check for singularity:

    while (any(s0==0))
        x0 = x0 + 1e-6;
        s0 = [x0-l; -x0+u];
    end


    [x,y,z,s] = IntialPointHeuristicsLP(g,A,b,l,u,x0,y0,z0,s0);

    % Update constraints
    sl = s(1:n_design);
    su = s(n_design+1:n_design*2);
    zl = z(1:n_design);
    zu = z(n_design+1:n_design*2);

    % Update of initial residuals
    r_l = g-A*y-(z(1:n_design)-z(n_design+1:n_design*2));
    r_a = b-A'*x;
    r_c = s+[l; -u] - [x; -x];

    % Initial dual gap
    dual_gap = (z'*s)/(n_design);
    dual_gap_0 = dual_gap;

    for i = 1:max_iter
        iter = iter + 1;
        zsl = zl./sl;
        zsu = zu./su;

        % Affine step
        r_cs = (r_c-s);
        r_l_bar = r_l - zsl.*r_cs(1:n_design) +zsu.*r_cs(1+n_design:2*
            n_design);


        % compute diagonal h inv easily
        diag_h_inv = 1./(zsl+zsu);

        % make the Cholesky-factorization
        normalfactor = A' * (diag_h_inv .*  A);
        R = chol(normalfactor);

```

```
% caluclate the affine step
dy_aff = R \ (R' \ (r_a + A' * (diag_h_inv .* r_l_bar)));
dx_aff = diag_h_inv .* (-r_l_bar + A*dy_aff);
dz_aff = -[zsl.*dx_aff; -zsu.*dx_aff] + (z./s).*r_cs;
ds_aff = -s-(s./z).*dz_aff;


%compute max alpha affine
d_zs = [dz_aff; ds_aff];
alphas = (-[z;s]./d_zs);
alpha_aff = min([1;alphas(d_zs<0)]);

dual_gap_aff = ((z+alpha_aff*dz_aff)'*(s+alpha_aff*ds_aff))/(
    n_design);
sigma = (dual_gap_aff/dual_gap)^3;


% Affine-Centering-Correction Direction
r_sz = s + ds_aff.*dz_aff./z-dual_gap*sigma*e./z;
r_cs = (r_c-r_sz);
r_l_bar = r_l - zsl.*r_cs(1:n_design) +zsu.*r_cs(1+n_design:2*
    n_design);



%This is normal equation stuff as well
dy = R \ (R' \ (r_a + A' * (diag_h_inv .* r_l_bar)));
dx = diag_h_inv .* (-r_l_bar + A*dy);

dz = - [zsl.*dx; -zsu.*dx] + (z./s).*r_cs;
ds = -r_sz-(s./z).*dz;

%compute max alpha
d_zs = [dz; ds];
alphas = (-[z;s]./d_zs);
alpha = min([1;alphas(d_zs<0)]);

alpha_bar = eta*alpha;

% Update of position
x = x + alpha_bar * dx;
y = y + alpha_bar * dy;
z = z + alpha_bar * dz;
s = s + alpha_bar * ds;

% Update of residuals
sl = s(1:n_design);
su = s(n_design+1:n_design*2);
zl = z(1:n_design);
zu = z(n_design+1:n_design*2);


r_l = g-A*y-(z(1:n_design)-z(n_design+1:n_design*2));
r_a = b-A'*x;
r_c = s+[l; -u] - [x; -x];
```

```


        % Compute the dual gap
        dual_gap = (z'*s)/(2*n_design);

        % Check for convergence
        if(dual_gap <= epsilon*0.01*dual_gap_0)
            return
        end
    end
end
```

Listing C.2: Mehrota's Interior Point methods Algorithm for LPs

C.3 Generation of Random LP

```
function [g,A,b,x,lam] = randomLP(n,m,scale)
% randEQP    Generate a random convex EQP
%
% Inputs:
%   n       : a positive integer giving number of variabels
%   m       : a positive integer giving number of constraints
%   scale   : a float which simply scales the values of the problem
%             (optional). Default value: 1
%
% Outputs:
%   H       : a symmetric positive semi-definite Hessian matrix of dimension
     (n,n) defining the quadratic terms of the objective function
%   g       : a (n,) dimensional vector defining the linear part of the
     objective function
%   A       : a (n,m) dimensional matrix giving the rhs of the equality
     constrains
%   b       : a (m,) dimensional vector defining the rhs of the equality
     constrains
%   x       : a (n,) dimensional vector giving the solution
%   lam     : a (m,) dimensional vector giving the lagrange multiplier
%
%%

if nargin < 3
    scale = 1;
end

H = n*eye(n);

% Generate A matrix with full rank
% Check if it is, retry if not
% (While loops and check is probably not nessesary, just 'in case')
maxrank = m;
rankA = 0;
```

```
triesA = 0;
while rankA < maxrank
    triesA = triesA + 1;
    if (triesA > 100)
        error("RuntimeError: Failed to generate valid A matrix in 100 tries
            ")
    end
    A = scale*rand(n,m);
    rankA = rank(A);
end

% Generate a random solution to the system
x = rand(n,1);
lam = rand(m,1);

% Generate matching g and b vectors from using KKT system
KKT = [H -A; -A' zeros(m)];
sol = [x; lam];
rhs = -(KKT*sol);
g = rhs(1:n);
b = rhs(n+1:end);

end
```

Listing C.3: Generation of Random LP

C.4 Driver for Exercise 3

```
%% Problem 3.4 - 3.6 Driver
%% Clean up
clear
close all

% Options for linprog
default_options = optimset('Display', 'off');

%% Test on toy problem for correctness

Nsamples = 10; % Run test multiple times for small problems
% Defining the data
g = [-16.1000,  -8.5000,  -15.7000, -10.0200, -18.6800]';

A = ones(5,1);
b = ones(1,1);

[n_var,m_eqcon] = size(A);

l = zeros(n_var,1);
u = ones(n_var,1);

```

```
% Starting point
x0 = zeros(n_var,1);
s0 = ones(2*n_var,1);
y0 = ones(m_eqcon,1);
z0 = ones(2*n_var,1);

% Test on our implementation with linprog interior point and simplex
tic;
for i=1:Nsamples
    [x,y,z,s,iter] = LinearPrimalDualIM_box(g,A,b,l,u,x0,y0,z0,s0);
end
time_own = toc/Nsamples;
obj = g'*x;

% Test linprog interior-point
options = optimset(default_options, 'Algorithm', 'interior-point');
tic;
for i=1:Nsamples
    [x_linprog, obj_linprog, , output] = linprog(g,[],[],A',b,l,u, options)
        ;
end
time_linprog = toc/Nsamples;
iter_linprog = output.iterations;

% Test linprog Dual Simplex
options = optimset(default_options, 'Algorithm', 'dual-simplex');
tic;
for i=1:Nsamples
    [x_simplex, obj_simplex, , output] = linprog(g,[],[], A', b,l,u,
        options);
end
time_simplex= toc/Nsamples;
iter_simplex = output.iterations;


%% Compute the errors
err = mean(sqrt((x - x_simplex).^2));
err_linprog = mean(sqrt((x_simplex - x_linprog).^2));

%% Make Table

data = [obj_linprog, obj];
data(2,:) = [nan, obj-obj_linprog];
data(3:7,:) = [x_linprog, x];
data(8,:) = [nan, err];
data(9,:) = [nan, nan];
data(10,:) = [time_linprog, time_own];
data(11,:) = [iter_linprog, iter];
input.data = data;

% Set column labels (use empty string for no label):
input.tableColLabels = {'Linprog IP', 'Own Solver'};
% Set row labels (use empty string for no label):
input.tableRowLabels = {'$\phi =$', ...
                        '$\Delta \phi =$', ...
```

```
                        'x =', '', '', '', '', ...
                        'MSE =', ...
                        '', ...
                        'TimeIterations =','Iterations ='};
% Set the row format of the data values
% Set the row format of the data values
input.dataFormatMode = 'row';
input.dataFormat = {'%.4e',10,'%d',1};
% Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
input.tableColumnAlignment = 'r';
% Switch table borders on/off:
input.booktabs = 1;
% LaTex table caption:
input.tableCaption = sprintf('Comparison between LinProg interior-point and
    Own LP interior-point alogorithm. Error is computed relative to LinProg
    IP solver.');
% LaTex table label:
input.tableLabel = 'ex3_objective_comp';
input.makeCompleteLatexDocument = 0;
input.dataNanString = '';
input.tablePlacement = 'ht';
% Now call the function to generate LaTex code:
latex = latexTable(input);
savelatexTable(latex, input.tableLabel, 3);

% Make table with simplex also

data = [obj_simplex, obj_linprog, obj];
data(2,:) = [nan, obj_linprog-obj_simplex, obj-obj_simplex];
data(3:7,:) = [x_simplex, x_linprog, x];
data(8,:) = [nan err_linprog, err];
data(9,:) = [nan, nan, nan];
data(10,:) = [time_simplex, time_linprog, time_own];
data(11,:) = [iter_simplex,iter_linprog, iter];
input.data = data;


% Set column labels (use empty string for no label):
input.tableColLabels = {'Linprog Simplex', 'Linprog IP', 'Own Solver'};
% Set row labels (use empty string for no label):
input.tableRowLabels = {'$\phi =$', ...
                        '$\Delta \phi =$', ...
                        'x =', '', '', '', '', ...
                        'MSE =', ...
                        '', ...
                        'Time =','Iterations ='};
% Set the row format of the data values
input.dataFormatMode = 'row';
input.dataFormat = {'%.4e',10,'%d',1};
% Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
input.tableColumnAlignment = 'r';
% Switch table borders on/off:
input.booktabs = 1;
% LaTex table caption:
```

```
input.tableCaption = sprintf('Comparison between LinProg interior-point, Own
      LP interior-point and LinProg Simplex alogorithm. Error is computed
     relative to LinProg simplex solver.');
% LaTex table label:
input.tableLabel = 'ex3_objective_comp_simplex';
input.makeCompleteLatexDocument = 0;
input.dataNanString = '';
input.tablePlacement = 'ht';
% Now call the function to generate LaTex code:
latex = latexTable(input);
savelatexTable(latex, input.tableLabel, 3);

%% Size dependent problem 1    %%

% Number of tests and test sizes and arrays for storing results
Ntests = 20;

Nsizes = round(linspace(100,2000, Ntests));

solvers = ["Own Solver", "LinProg IP", "LinProg Simplex"];
Nsolvers = length(solvers);

objs = zeros(Nsolvers,Ntests);
iterations = zeros(Nsolvers,Ntests);
times = zeros(Nsolvers, Ntests);
MSE_errs = zeros(Nsolvers-1,Ntests);
obj_errs = zeros(Nsolvers-1,Ntests);

x_true = nan*zeros(Ntests,max(Nsizes));

fprintf("\nTest 2 : Size dendendt problem, %d solvers, for each %d value(s)"
     + ...
        " between [%d, %d].\n\n", Nsolvers, Ntests,Nsizes(1),Nsizes(end));

rng(200) % Set seed
poorMansProgressBar(Ntests);
for j=1:Ntests
    poorMansProgressBar(0);  % Update progress for iteration


    n_val = Nsizes(j);  % Set value of b

    % Generate problem
    % Defining the data
    [g,A,b,x,lam] = randomLP(n_val,floor(n_val/2), 1);
    [n_var,m_eqcon] = size(A);
    l = zeros(n_val,1);
    u = ones(n_val,1);

    % Starting point
    x0 = zeros(n_var,1);
    s0 = ones(2*n_var,1);
    y0 = ones(m_eqcon,1);
    z0 = ones(2*n_var,1);

```

```
    % Run Own
    tstart = cputime;
    [x,y,z,s,iter] = LinearPrimalDualIM_box(g,A,b,l,u,x0,y0,z0,s0);
    time = cputime-tstart;
    obj = g'*x;

    % Run Linprog IP
    options = optimset(default_options, 'Algorithm', 'interior-point');
    tstart = cputime;
    [x_ip, obj_ip,  , outtmp] = linprog(g,[],[],A',b,l,u, options);
    time_ip = cputime-tstart;
    iter_ip = outtmp.iterations;

    % Run Linprog Simplex
    options = optimset(default_options, 'Algorithm', 'dual-simplex');
    tstart = cputime;
    [x_sim, obj_sim,  , outtmp] = linprog(g,[],[],A',b,l,u, options);
    time_sim = cputime-tstart;
    iter_sim = outtmp.iterations;

    % Save and compute stuff
    objs(1,j) = obj;
    objs(2,j) = obj_ip;
    objs(3,j) = obj_sim;

    obj_errs(1,j) = obj - obj_ip;
    obj_errs(2,j) = obj - obj_sim;

    MSE_errs(1,j) = mean(sqrt((x - x_ip).^2));
    MSE_errs(2,j) = mean(sqrt((x - x_sim).^2));

    iterations(1,j) = iter;
    iterations(2,j) = iter_ip;
    iterations(3,j) = iter_sim;
    times(1,j) = time;
    times(2,j) = time_ip;
    times(3,j) = time_sim;

end
poorMansProgressBar(-1);

%% Plot Objective as Function of n

f = figure('Name','Objective');
for i=1:Nsolvers
    plot(Nsizes, objs(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers, 'Location','southwest')
xlabel("Number of variables, n")
ylabel("Objective value")
saveas(f, "./figures/Ex3/ex3_obj_large",'epsc');

f = figure('Name','Objective semilogy');
```

```
for i=1:Nsolvers
    semilogy(Nsizes, objs(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers, 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Objective value")
saveas(f, "./figures/Ex3/ex3_obj_semilogy_large",'epsc');

%% Objective Error Directly


f = figure('Name','Objective Error');
for i=1:Nsolvers-1
    plot(Nsizes, obj_errs(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(sprintfc("Our Solver || %s",solvers(2:end)), 'Location','southwest')
xlabel("Number of variables, n")
ylabel("Objective error")
saveas(f, "./figures/Ex3/ex3_obj_err_large",'epsc');

%% Objective Relative Error
f = figure('Name','Relative Objective Error');
for i=1:Nsolvers-1
    plot(Nsizes, abs(obj_errs(i,:)/obj(1,:)), 'LineWidth',2);
    hold on
end
grid on
legend(sprintfc("Our Solver || %s",solvers(2:end)), 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Relative objective error")
saveas(f, "./figures/Ex3/ex3_rel_obj_err_large",'epsc');

%% Absolute Error in semilog plot

f = figure('Name','Abs Objective Error semilog');
for i=1:Nsolvers-1
    semilogy(Nsizes, abs(obj_errs(i,:)), 'LineWidth',2);
    hold on
end
grid on
legend(sprintfc("Our Solver || %s",solvers(2:end)), 'Location','southeast')
xlabel("Number of variables, n")
ylabel("Absolute Objective error")
saveas(f, "./figures/Ex3/ex3_obj_MAE_semilogy_large",'epsc');

%% MSE

f = figure('Name','MSE Error');
for i=1:Nsolvers-1
    plot(Nsizes, MSE_errs(i,:), 'LineWidth',2);
    hold on
```

```
end
grid on
legend(sprintfc("Our Solver || %s",solvers(2:end)), 'Location','northwest')
xlabel("Number of variables, n")
ylabel("MSE")
saveas(f, "./figures/Ex3/ex3_mse_err_large",'epsc');

%% MSE in semi-log Plot

f = figure('Name','MSE Error log');
for i=1:Nsolvers-1
    semilogy(Nsizes, MSE_errs(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(sprintfc("Our Solver || %s",solvers(2:end)), 'Location','southeast')
xlabel("Number of variables, n")
ylabel("MSE")
saveas(f, "./figures/Ex3/ex3_mse_err_semilogy_large",'epsc');



%% Check the Time Spent
f = figure('Name','Time');
for i=1:Nsolvers
    plot(Nsizes, times(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers, 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Time [s]")
saveas(f, "./figures/Ex3/ex3_time_large",'epsc');

%% Time Spent Semi-log
f = figure('Name','Time log');
for i=1:Nsolvers
    semilogy(Nsizes, times(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers, 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Time [s]")
saveas(f, "./figures/Ex3/ex3_time_semilogy_large",'epsc');

%% No. Iterations
f = figure('Name','Iterations');
for i=1:Nsolvers
    plot(Nsizes, iterations(i,:), 'LineWidth',2);
    hold on
end
grid on
    legend(solvers, 'Location','northwest')
xlabel("Number of variables, n")
```

```
ylabel("Iterations")
saveas(f, "./figures/Ex3/ex3_iterations_large",'epsc');
%% No iteration semi-log
f = figure('Name','Iterations semilogy');
for i=1:Nsolvers
    semilogy(Nsizes, iterations(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers, 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Iterations")
saveas(f, "./figures/Ex3/ex3_iterations_semilogy_large",'epsc');

%% No iterations without simplex
f = figure('Name','Iterations No simplex');
for i=1:Nsolvers-1
    plot(Nsizes, iterations(i,:), 'LineWidth',2);
    hold on
end
grid on
legend(solvers([1,2]), 'Location','northwest')
xlabel("Number of variables, n")
ylabel("Iterations")
saveas(f, "./figures/Ex3/ex3_iterations_no_simplex_large",'epsc');


%% Implementation of a process bar to save the current iteration

function [msg] = poorMansProgressBar(state)
% state > 0 : Begin progress bar, state gives expected number of iterations
% state = 0 : Update progress bar, call this max the expected number of
% times
% state = -1 : Display done
if state > 0
    msg = sprintf("Start |");
    for i=1:state
        msg = msg + sprintf(" -");
    end
    msg = msg + sprintf(" | Finish\n        ");
elseif state == 0
    msg = sprintf(" *");
else
    msg = fprintf("   DONE!!\n");
end
fprintf("%s",msg);
end
```

Listing C.4: Driver Exercise 3

APPENDIX D

Exercise 4

D.1 Generate Contours for the Himmelblau's test problem

```
function [c, con] = contourHimmel(add_con, view, xminmax, yminmax, res, f,
    varargin)
% contourplot Generates a contour plot of the given function
%
% Output:
%   figure     : The figure object
%%

if nargin < 5
    f = @(x1,x2) (x1.^2+x2-11).^2 + (x1+x2.^2-7).^2;
end

if nargin < 1
    add_con = false;
end

if nargin < 2
    view = 6;
end

if nargin < 3
    xminmax = 5;
    yminmax = 5;
    res = 50;
end

x = -view:1/res:view;
y = -view:1/res:view;
[X,Y] = meshgrid(x,y);
v = [0:2:10 10:10:100 100:20:300];
F = f(X,Y);
[m,c]=contour(X,Y,F,v,"linewidth",2);
colormap("turbo")

% Add constraints
con = [];
if add_con
```

```
    % Con 1
    p1 = (x+2).^2;
    idx = abs(p1) <= view;
    p1 = p1(idx);
    xp = x(idx);
    xp = [xp,xp(end),xp(1),xp(1)];
    p1 = [p1,view, view, p1(1)];
    con1 = patch("XData", xp,"YData", p1,'FaceColor', 'red', 'FaceAlpha', 0
        .5, 'EdgeColor', 'k');

    % Con 2
    p2 = 4/10*x;
    idx2 = abs(x) <= view;
    p2 = p2(idx2);
    xp2 = x(idx2);
    xp2 = [xp2,xp2(end),xp2(1),xp2(1)];
    p2 = [p2,-view,-view,xp2(1)];
    con2 = patch("XData", xp2,"YData", p2,'FaceColor', 'red', 'FaceAlpha', 0
        .5, 'EdgeColor', 'k');

    % X con
    con3 = patch("XData", [-view, -xminmax, -xminmax,-view, -view],...
           "YData", [-view, -view, view, view, -view],...
           'FaceColor', 'red', 'FaceAlpha', 0.5, 'EdgeColor', 'k');
    con4 = patch("XData", [xminmax, view, view, xminmax, xminmax],...
           "YData", [-view, -view, view, view, -view],...
           'FaceColor', 'red', 'FaceAlpha', 0.5, 'EdgeColor', 'k');

    % Y con
    con5 = patch("XData", [-view, -view, view, view, -view],...
           "YData", [view, yminmax, yminmax, view, view],...
           'FaceColor', 'red', 'FaceAlpha', 0.5, 'EdgeColor', 'k');
    con6 = patch("XData", [-view, -view, view, view, -view],...
             "YData", [-yminmax, -view, -view, -yminmax, -yminmax],...
             'FaceColor', 'red', 'FaceAlpha', 0.5, 'EdgeColor', 'k');

    con = [con1, con2, con3, con4, con5, con6];

end

% Turns off the legend for the contour
set(get(get(c,'Annotation'),'LegendInformation' ),'IconDisplayStyle', 'off'
    );

% Appropiate labels for the plot
xlabel('x_1','Fontsize',14)
ylabel('x_2','Fontsize',14)
colorbar

axis([-view view -view view])

end
```

Listing D.1: Code Generate Contours Himmelblau

D.2 Generate Contour Points for the Himmelblau's Testproblem

```
function [plt] = plotPoint(x, y, type, color, sz)
%plotPoint Plots a point on a countur plot
%   Type gives the marker

if nargin < 3
    type = "gen"; % generic
end

if type == "max" % Maximum
    a = '^'; b='MarkerFaceColor'; c='r'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "min" % Minimum
    a = 'v'; b='MarkerFaceColor'; c='g'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "sad" % Saddle
    a = 'd'; b='MarkerFaceColor'; c='m'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "gen" % Just some point
    a = 'o'; b='MarkerFaceColor'; c='b'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "int" % A inital starting point
    a = 's'; b='MarkerFaceColor'; c='c'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "sol" % Found solution point
    a = 'h'; b='MarkerFaceColor'; c='y'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
else

    error("Unknown type of point: %s", type)
end

if nargin >= 4
    c = color;
end


if nargin >= 5
    g = sz;
end

plt = plot(x,y,a,b,c,d,e,f,g);

end
```

Listing D.2: Generate Contour Points

D.3 Generate Contour Points for the Rosenbrock

```
function [plt] = plotPoint(x, y, type, color, sz)
%plotPoint Plots a point on a countur plot
%   Type gives the marker

if nargin < 3
    type = "gen"; % generic
end

if type == "max" % Maximum
    a = '^'; b='MarkerFaceColor'; c='r'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "min" % Minimum
    a = 'v'; b='MarkerFaceColor'; c='g'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "sad" % Saddle
    a = 'd'; b='MarkerFaceColor'; c='m'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "gen" % Just some point
    a = 'o'; b='MarkerFaceColor'; c='b'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "int" % A inital starting point
    a = 's'; b='MarkerFaceColor'; c='c'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
elseif type == "sol" % Found solution point
    a = 'h'; b='MarkerFaceColor'; c='y'; d='MarkerEdgeColor'; e='k'; f='
        markersize';g=15;
else

    error("Unknown type of point: %s", type)
end

if nargin >= 4
    c = color;
end


if nargin >= 5
    g = sz;
end

plt = plot(x,y,a,b,c,d,e,f,g);

end
```

Listing D.3: Generate Contour Points

D.4 Interface for SQP solvers

```
function [x, obj, lambda, output] = SQPsolver(objfun,confun,xlower,xupper,
    clower,cupper,x0,solver)
% SQPSolver  Interface for variuos SQP solvers.
%
% Solves NLP's on the form
%
%   \min_{x} f(x)
%    s.t.     cl <= c(x) <= cu
%             xl <= x <= xu
%
% With options for using solvers with different facotrizations.
%
% Inputs:
%   objfun  : Objective function, should take a vector x of size size(n),
%           : and return [function value at x, gradients at x]
%   confun  : Constraint function, should take a vector x of size size(n)
%           : and reutrn [constraint function value at x, constraint
    gradients at x]
%   xlower  : (n,)-dim vector of lower bounds for x
%   xupper  : (n,)-dim vector of upper bounds for x
%   clower  : (m,)-dim vector of lower bounds for x
%   cupper  : (m,)-dim vector of lower bounds for x
%   x0      : (n,)-dim vector of inital point for x
%   solver  : string with the solver of choice, options are "bfgs", "line",
%             and "trust"
%
% Outputs:
%   x       : a (n,) dimensional vector of found solution
%   obj     : a float giving the objective value
%   lambda  : a (m,) dimensional vector of lagrange multipliers
%   output  : a object with performance and iteration information
%%

solver_match = lower(solver);
switch solver_match
    case "bfgs"
        [x, lambda, output] = SQPsolverBFGS(objfun,confun,xlower,xupper,
            clower,cupper,x0);
    case "line"
        [x, lambda, output] = SQPsolverLS(objfun,confun,xlower,xupper,clower
            ,cupper,x0);
    case "trust"
        [x, lambda, output] = SQPsolverTR(objfun,confun,xlower,xupper,clower
            ,cupper,x0);
    otherwise
        error("Unknown solver choice: '" + solver + "'. Availible options
            are: 'bfgs', 'line', and 'trust'.")
end
obj = objfun(x);
```

Listing D.4: Interface for SQP solvers

D.5 SQP BFGS Solver

```
function [x, lambda, output] = SQPsolverBFGS(objfun,confun,xlower,xupper,
    clower,cupper,x0)
% SQPsolverBFGS SQP solvers using dampend BFGS approximation.
%
% Solves NLP's on the form
%
%   \min_{x} f(x)
%     s.t.     cl <= c(x) <= cu
%              xl <= x <= xu
%
% Inputs:
%   objfun  : Objective function, should take a vector x of size size(n),
%           : and return [function value at x, gradients at x]
%   confun  : Constraint function, should take a vector x of size size(n)
%           : and reutrn [constraint function value at x, constraint
    gradients at x]
%   xlower  : (n,)-dim vector of lower bounds for x
%   xupper  : (n,)-dim vector of upper bounds for x
%   clower  : (m,)-dim vector of lower bounds for x
%   cupper  : (m,)-dim vector of lower bounds for x
%   x0      : (n,)-dim vector of inital point for x
%
% Outputs:
%   x       : a (n,) dimensional vector of found solution
%   lambda  : a (m,) dimensional vector of lagrange multipliers
%   output  : a object with performance and iteration information
%
%%

% Define constants
maxiter = 100;
epsilon = 1e-9;

% Allocate storage
n = length(x0);
x = reshape(x0, n, 1);        % Make sure x is a collumn vector
[ ,df] = objfun(x);           % Compute obj and con function for x0
[c, ,dc, ] = confun(x);
c = -1*c;
dc = -1*dc;
m = size(c,1);
B = eye(n);
z = ones(2*m+2*n,1);
lid = 1:m;
uid = (m+1):(2*m);
clid = (2*m+1):(2*m+n);
cuid = (2*m+n+1):(2*(n+m));


% Create outputs struct
output.iterations = 0;
```

```
output.converged = false;
output.xk = x;
output.time_qp = 0;
output.function_calls = 2;

% Define options for quadprog
options = optimset('Display', 'off');

% Run until convergence or max iter
while (output.iterations < maxiter) &&  output.converged

    output.iterations = output.iterations + 1;

    % Set lower and upper bounds iteration k
    xlowerk = -x + xlower;
    xupperk = -x + xupper;
    clowerk = -c + clower;
    cupperk = -c + cupper;

    start = cputime;
    [Deltax, ,flag,out,lambda] = quadprog(B,df, ...
                                    -[dc'; -dc'],-[clowerk;-cupperk], ...
                                    [],[], ...
                                    xlowerk,xupperk,[], ...
                                    options);
    output.time_qp = output.time_qp + (cputime-start);

    if flag < 0
        error("Quadprog error: %s", out.message)
    end
    zhat = [lambda.lower; lambda.upper; lambda.ineqlin];

    % Update the current point
    z = z + (zhat-z);
    x = x + Deltax;

    % Save step
    output.xk(:, output.iterations+1) = x;

    % For the quasi Newton update
    dL = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);

    % Compute function values for next iteration
    [ ,df] = objfun(x);
    [c, ,dc, ] = confun(x);
    c = -1*c;
    dc = -1*dc;
    output.function_calls = output.function_calls + 2;
    %% Dampend BFGS Update

    % Compute Quasi Newton update of the hessian
    dL2 = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);


    % Compute values used multiple times for for BFGS update
```

```
    q = dL2-dL;
    Bp = (B*Deltax);
    pBp = Deltax'*Bp;

    % Compute appropiate theta value
    if Deltax'*q < 0.2*pBp
        theta = (0.8*pBp)/(pBp-Deltax'*q);
    else
        theta = 1;
    end

    r = theta*q+(1-theta)*(Bp);
    B = B + r*r'/(Deltax'*r) - Bp*Bp'/pBp;

    % Check for convergence
    if norm(dL2, 'inf') < epsilon
        output.converged = true;
    end
end

% Warn if not converged
if  output.converged
    warning("Max number of iterations reached before convergence")

end
```

Listing D.5: SQP BFGS Solver

D.6 SQP Line Search Solver

```
function [x, lambda, output] = SQPsolverLS(objfun,confun,xlower,xupper,
    clower,cupper,x0)
% SQPsolverLS SQP solvers using dampend BFGS approximation and line seach.
%
% Solves NLP's on the form
%
%   \min_{x} f(x)
%     s.t.     cl <= c(x) <= cu
%              xl <= x <= xu
%
% Inputs:
%   objfun   : Objective function, should take a vector x of size size(n),
%            : and return [function value at x, gradients at x]
%   confun   : Constraint function, should take a vector x of size size(n)
%            : and reutrn [constraint function value at x, constraint
    gradients at x]
%   xlower   : (n,)-dim vector of lower bounds for x
%   xupper   : (n,)-dim vector of upper bounds for x
%   clower   : (m,)-dim vector of lower bounds for x
%   cupper   : (m,)-dim vector of lower bounds for x
```

```
%   x0       : (n,)-dim vector of inital point for x
%
% Outputs:
%   x        : a (n,) dimensional vector of found solution
%   lambda   : a (m,) dimensional vector of lagrange multipliers
%   output   : a object with performance and iteration information
%
%%

% Define constants
maxiter = 100;
epsilon = 1e-9;
tol_c1 = 0.1;   % Tolerance for line search
non_monotone = true;

% Allocate storage
n = length(x0);
x = reshape(x0, n, 1);       % Make sure x is a collumn vector
[f,df] = objfun(x);          % Compute obj and con function for x0
[c, ,dc, ] = confun(x);
c = -1*c;
dc = -1*dc;
m = size(c,1);
B = eye(n);
z = ones(2*m+2*n,1);
lid = 1:m;
uid = (m+1):(2*m);
clid = (2*m+1):(2*m+n);
cuid = (2*m+n+1):(2*(n+m));
d = [xlower; -xupper; clower; -cupper];
mu = 0;

% Create outputs struct
output.iterations = 0;
output.converged = false;
output.xk = x;
output.stepLengths = zeros(0,1);
output.time_qp = 0;
output.function_calls = 2;

% Define options for quadprog
options = optimset('Display', 'off');

% Run until convergence or max iter
while (output.iterations < maxiter) &&  output.converged

    output.iterations = output.iterations + 1;

    % Set lower and upper bounds iteration k
    xlowerk = -x + xlower;
    xupperk = -x + xupper;
    clowerk = -c + clower;
    cupperk = -c + cupper;

    start = cputime;
```

```
    [Deltax, ,flag,out,lambda] = quadprog(B,df, ...
                                    -[dc'; -dc'],-[clowerk;-cupperk], ...
                                    [],[], ...
                                    xlowerk,xupperk,[], ...
                                    options);
    output.time_qp = output.time_qp + (cputime-start);

    if flag < 0
        error("Quadprog error: %s", out.message)
    end
    zhat = [lambda.lower; lambda.upper; lambda.ineqlin];

    %% Do line search for step length
    % Set starting alpha at max
    alpha = 1.0;
    % Compute reference merit
    c_alpha = -1*confun(x);
    output.function_calls = output.function_calls + 2;
    c_alpha = [x; -x; c_alpha; -c_alpha]-d;
    absz = abs(z);
    mu = max(absz, 0.5*(mu+absz));
    phi0 = phi(f,mu,c_alpha);
    dphi0 = dphi(df,Deltax,mu,c_alpha);
    % Run til convergence
    searchingForAlpha = true;
    while searchingForAlpha
        % Compute step canditate
        x_alpha = x + alpha*Deltax;
        % Compute values for step candidate
        f_alpha = objfun(x_alpha);
        c_alpha = -1*confun(x_alpha);
        output.function_calls = output.function_calls + 2;
        c_alpha = [x_alpha; -x_alpha; c_alpha; -c_alpha]-d;
        % Compute merit of step
        phi_alpha = phi(f_alpha,mu,c_alpha);

        if phi_alpha <= phi0 + tol_c1*dphi0*alpha
            % Accept alpha value
            searchingForAlpha = false;
        else
            % Update alpha value
            alpha_tmp = (phi_alpha-(phi0+alpha*dphi0))/(alpha*alpha);
            alpha_min = -dphi0/(2*alpha_tmp);
            alpha = min(0.9*alpha, max(alpha_min, 0.1*alpha));
        end
    end

    % Use non monotone update strategy if alpha is very small, and it is
    % set
    if all(alpha*Deltax<epsilon) && non_monotone
        alpha = 1.0;
    end

    % Save used step length
    output.stepLengths(output.iterations) = alpha;
```

```
    % Update the current point using alpha
    z = z + alpha*(zhat-z);
    x = x + alpha*Deltax;

    % Save step
    output.xk(:, output.iterations+1) = x;

    % For the quasi Newton update
    dL = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);

    % Compute function values for next iteration
    [f,df] = objfun(x);
    [c, ,dc, ] = confun(x);
    c = -1*c;
    dc = -1*dc;
    output.function_calls = output.function_calls + 2;
    %% Dampend BFGS Update

    % Compute Quasi Newton update of the hessian
    dL2 = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);


    % Compute values used multiple times for for BFGS update
    q = dL2-dL;
    Bp = (B*Deltax);
    pBp = Deltax'*Bp;

    % Compute appropiate theta value
    if Deltax'*q < 0.2*pBp
        theta = (0.8*pBp)/(pBp-Deltax'*q);
    else
        theta = 1;
    end

    r = theta*q+(1-theta)*(Bp);
    B = B + r*r'/(Deltax'*r) - Bp*Bp'/pBp;

    % Check for convergence
    if norm(dL2, 'inf') < epsilon
        output.converged = true;
    end
end

% Warn if not converged
if  output.converged
    warning("Max number of iterations reached before convergence")
end

% end of function
end

% Below is function definitions

% Define merit function
```

```
function [res] = phi(f, mu, c)
    res = f + mu'*abs(min(0,c));
end
% Define derivative of merit function
function [res] = dphi(df,dx,mu,c)
    res = df'*dx-mu'*abs(min(0,c));
end
```

Listing D.6: SQP Line Search Solver

D.7 SQP Trust Region Solver

```
function [x, lambda, output] = SQPsolverTR(objfun,confun,xlower,xupper,
    clower,cupper,x0)
% SQPsolverBFGS SQP solvers using dampend BFGS approximation and Trust
    Region.
%
% Solves NLP's on the form
%
%   \min_{x} f(x)
%    s.t.     cl <= c(x) <= cu
%             xl <= x <= xu
%
% Inputs:
%   objfun  : Objective function, should take a vector x of size size(n),
%           : and return [function value at x, gradients at x]
%   confun  : Constraint function, should take a vector x of size size(n)
%           : and reutrn [constraint function value at x, constraint
    gradients at x]
%   xlower  : (n,)-dim vector of lower bounds for x
%   xupper  : (n,)-dim vector of upper bounds for x
%   clower  : (m,)-dim vector of lower bounds for x
%   cupper  : (m,)-dim vector of lower bounds for x
%   x0      : (n,)-dim vector of inital point for x
%
% Outputs:
%   x       : a (n,) dimensional vector of found solution
%   lambda  : a (m,) dimensional vector of lagrange multipliers
%   output  : a object with performance and iteration information
%
%%

% Define constants
maxiter = 100;
epsilon = 1e-4;

% Allocate storage
n = length(x0);
x = reshape(x0, n, 1);        % Make sure x is a collumn vector
[f,df] = objfun(x);           % Compute obj and con function for x0
```

```
[c, ,dc, ] = confun(x);
c = -1*c;      % Fix sign
dc = -1*dc;    % Fix sign
m = size(c,1);
B = eye(n);
z = ones(2*m+2*n,1);
lid = 1:m;
uid = (m+1):(2*m);
clid = (2*m+1):(2*m+n);
cuid = (2*m+n+1):(2*(n+m));

% Trust region specific stuff
tr0 = 0.5;       % Initial trust region size
tr = tr0;        % Iterative trust region size
mu_val = 1000;     % Initial penalty value
mu = mu_val * ones(2*m+2*n,1);  % Make a vector for stuff
dL2 = Inf;

% Allocate storange for penalty program
penaltyH = zeros(3*n+2*m);
nm2 = 2*n+2*m;

% Create outputs struct
output.iterations = 0;
output.converged = false;
output.xk = x;
output.time_qp = 0;
output.function_calls = 2;

% Define options for quadprog
options = optimset('Display', 'off');

% Run until convergence or max iter
while (output.iterations < maxiter) &&  output.converged

    output.iterations = output.iterations + 1;

    % Set lower and upper bounds iteration k
    xlowerk = -x + xlower;
    xupperk = -x + xupper;
    clowerk = -c + clower;
    cupperk = -c + cupper;

    % Define and solve program for with penalty and tr constraints
    penaltyH(1:n,1:n) = 0;
    penaltyH(1:n,1:n) = B;
    penaltyC = [eye(n), -eye(n), dc, -dc, zeros(n,nm2), eye(n), -eye(n); ...
                    eye(nm2), eye(nm2), zeros(nm2,n*2)]';
    penaltyg = [df; mu];
    penaltyd = [xlowerk; -xupperk; ...
                clowerk; -cupperk; ...
                zeros(nm2,1);  ...
                -tr*ones(2*m,1)];
    start = cputime;
    [Deltax, ,flag,out,lambda] = quadprog(penaltyH,penaltyg, ...
```

```
                                                -penaltyC,-penaltyd, ...
                                                [],[],[],[],[],options);
    output.time_qp = output.time_qp + (cputime-start);

    if flag < 0
        error("Quadprog error: %s", out.message)
    end

    % Get only the relevant parts
    zhat = lambda.ineqlin(1:nm2);
    Deltax = Deltax(1:n);

    % Compute new penalty
    z_inf = norm(z, "inf");
    mu_val = max(0.5*(mu_val+z_inf), z_inf);
    mu = mu_val * ones(nm2, 1);

    % Compute actual / predicted ratio
    [c_full, ,dc_full, ] = confun(x);  % Current con
    c_full = [x; -x; c_full; -c_full] - penaltyd(1:nm2);
    dc_full = [eye(n), -eye(n), dc_full, -dc_full];
    c_full = -1*c_full;      % Fix sign
    dc_full = -1*dc_full;    % Fix sign
    [c_pred_full] = confun(x+Deltax);  % predicted con
    c_pred_full = [x; -x; c_pred_full; -c_pred_full] - penaltyd(1:nm2);
    c_pred_full = -1*c_pred_full;      % Fix sign

    % Compute predicted
    qmu_pred = f + df'*Deltax + 0.5*Deltax'*B*Deltax + mu'*max(0,-(c_full+
        dc_full'*Deltax));
    qmu = f+ mu'*max(0,-(c_full));
    phi1 = qmu;

    [f_pred, ] = objfun(x+Deltax); % predicted obj
    phi1_pred = f_pred+mu'*max(0,-c_pred_full);

    rho = (phi1-phi1_pred)/(qmu-qmu_pred);
    gamma = min(max( (2*rho-1)^3 + 1, 0.25), 2);

    output.function_calls = output.function_calls + 3;
    if any( isfinite(Deltax))
        disp("Detla x is inappropriate")
    end
    if  isfinite(rho)
        disp("rho is inappropriate")
    end
    %fprintf("Before: Deltax = [%.6f,%.6f],  rho : %f  , mu = %f, tr = %f,
        gamma = %f, x = [%.5f, %.5f], dl2 = %f\n",Deltax(1), Deltax(2), rho,
         mu_val, tr, gamma, x(1), x(2),norm(dL2, 'inf'));
    % Adjust trust region accordingly
    if rho > 0
        % If accepted
        % Update the current point
        z = zhat;
        x = x + Deltax;
```

```
        % Save step
        output.xk = [output.xk, x];

        % For the quasi Newton update
        dL = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);

        % Compute function values for next iteration
        [f,df] = objfun(x);
        [c, ,dc, ] = confun(x);
        c = -1*c;
        dc = -1*dc;
        output.function_calls = output.function_calls + 2;
        %% Dampend BFGS Update

        % Compute Quasi Newton update of the hessian
        dL2 = df - z(lid)-z(uid)-dc*z(clid)-dc*z(cuid);

        % Compute values used multiple times for for BFGS update
        q = dL2-dL;
        Bp = (B*Deltax);
        pBp = Deltax'*Bp;

        % Compute appropiate theta value
        if Deltax'*q < 0.2*pBp
            theta = (0.8*pBp)/(pBp-Deltax'*q);
        else
            theta = 1;
        end

        r = theta*q+(1-theta)*Bp;
        B = B + r*r'/(Deltax'*r) - Bp*Bp'/pBp;

        % Update trust region
        tr = gamma*tr;
    else
        % Not accepted
        % Update trust region
        tr = gamma*norm(Deltax,"inf");
    end
    %fprintf("After: Deltax = [%.6f,%.6f], rho : %f , mu = %f, tr = %f,
        gamma = %f, x = [%.5f, %.5f], dl2 = %f\n",Deltax(1), Deltax(2), rho,
         mu_val, tr, gamma, x(1), x(2),norm(dL2, 'inf'));

    % Check for convergence
    if norm(dL2, 'inf') < epsilon
        output.converged = true;
    end
end

% Warn if not converged
if output.converged
    warning("Max number of iterations reached before convergence")

end
```

Listing D.7: SQP Trust Region Solver

D.8 The Driver for Exercise 4

```
%% Problem 4 Driver
%% Clean up
clear
close all

runContourPlot_44 = false;
runSolveTest_45= false;
runSolveTest_452= false;
runBFGS_46 = false;
runBFGS_LS_47 = false;
runBFGS_TR_48 = false;
himmelblau49 = false;
rosenbrock49 = true;
rosen_con_case = 1;

addpath('../../casadi-windows-matlabR2016a-v3.5.5')
import casadi.*
disp("Casadi import succesfull")


%% Problem 4.4 - Plot Himmelblau
if runContourPlot_44
%%
disp("Generating and saving contour plot");
% Define points of interest
min_points = [ 3.0, 2.0; ...
              -0.2983476136,2.895620844; ...
              -3.654605171,2.737718273; ...
              -3.548537388,-1.419414955;];

max_points = [-0.4869360343, -0.1947744137; ...
               3.216440661,  1.286576264; ...
              -1.424243078,  0.3314960331];

saddle_points = [0.08667750456, 2.884254701; ...
                 -3.073025751, -0.08135304429];

fig = figure("Name", "Himmelblau - POI", 'Position', [100, 100,
    800,800]);
hold on
% Create contour plot and constraints
[cfig, conFigs] = contourHimmel(true);

% Add points of interest
mx = plotPoint(max_points(:,1),max_points(:,2), "max");
```

```
    mn = plotPoint(min_points(:,1),min_points(:,2), "min");
    sd = plotPoint(saddle_points(:,1),saddle_points(:,2), "sad");
    if isempty(conFigs)
        legend([mx,mn,sd],{'Local Maximum', 'Local Minimum', 'Saddle Point'
            })
    else
        legend([mx,mn,sd, conFigs(1)],{'Local Maximum', 'Local Minimum', '
            Saddle Point', 'Infeasible region'})
    end

    hold off
    savefigpdf(fig, "ex4_himmelblau_poi", 4);

    end
    %% Problem 4.5 - Solve Himmelblau
    if runSolveTest_45
    %%
    disp("Solving Test problem using fmincon and Casadi");

    x0s = [[0.0; 0.0], [-4.0; 0], [-4; 1]];

    for j=1:length(x0s)
        %sympref('FloatingPointOutput',1);
        x0 = x0s(:,j);     % Initial point
        xl = [-5; -5];       % Lower bound for x
        xu = [5; 5];         % Upper bound for x
        cl = [0; 0];         % Lower bound for constraints
        cu = [54; 70];       % Upper bound for constraints

        % Solve problem using fmincon
        options = optimoptions('fmincon',...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin,fval,exitflag,output] = fmincon(@objfunHimmelblau, ...
                                                  x0, ...
                                                  [], [], [], [], ...
                                                  xl, xu, ...
                                                  @confunHimmelblau, ...
                                                  options);
        time_fmincon = cputime - tstart;

         % Call fmincon
        options = optimoptions('fmincon',...
                               'SpecifyObjectiveGradient',true,...
                               'SpecifyConstraintGradient',true,...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin_grad,fval_grad,exitflag_grad,output_grad]=fmincon( ...
                                                    @objfungradHimmelblau, x0,
                                                        ...
                                                    [], [], [], [], ...
                                                    xl, xu, ...
                                                    @confungradHimmelblau1, ...
```

```
                                                    options);
        time_fmincon_grad = cputime - tstart;

        % Define problem for casadi
        x1 = SX.sym('x1');  % Define variables
        x2 = SX.sym('x2');
        % Define problem
        hbprob = struct('x',[x1; x2], ...
                        'f',(x1^2+x2-11)^2+(x1+x2^2-7)^2, ...
                        'g',[(x1+2)^2-x2; -4*x1+10*x2] ...
                        );

        % Solve the problem with casadi
        options = struct;
        options.ipopt.print_level = 0;
        options.print_time = 0;
        tstart = cputime;
        S = nlpsol('S', 'ipopt', hbprob, options);
        r = S('x0', x0, 'lbg',0,'ubg',inf);
        time_cas = cputime - tstart;
        sol_cas = full(r.x);
        obj_cas = full(r.f);

        % Print results
        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t fmincon solution: [%.5e, %.5e], objective: %.5e, time: %
            .5e\n', sol_fmin(1), sol_fmin(2), fval, time_fmincon);
        fprintf('\t fmincon grad solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e\n', sol_fmin_grad(1), sol_fmin_grad(2), fval_grad,
            time_fmincon_grad);
        fprintf('\t Casadi solution: [%.5e, %.5e], objective: %.5e, time: %
            .5e\n', sol_cas(1), sol_cas(2), obj_cas, time_cas);

        fig = figure("Name", sprintf("Himmelblau - Solution for x0=[%+.1f,
            %+.1f]",x0(1),x0(2)), 'Position', [150, 150, 600, 600]);
        hold on
        % Create contour plot and constraints
        [cfig, conFigs] = contourHimmel(true);

        % Add points of interest
        intp = plotPoint(x0(1),x0(2), "int");
        %solp = plotPoint(sol_fmin(1),sol_fmin(2), "sol", 'y', 18);
        solp_grad = plotPoint(sol_fmin_grad(1),sol_fmin_grad(2), "min", 'r',
             10);
        solp_cas = plotPoint(sol_cas(1),sol_cas(2), "gen", 'g', 8);
        legend([intp,solp_grad,solp_cas],{'Initial Point', 'fmincon solution
            ','CasADi solution'})
        hold off
        savefigpdf(fig, sprintf("ex4_5_himmelblau_x0=%+.0f_%+.0f",x0(1),x0
            (2)), 4);

        data(j,:) = [fval, sol_fmin', time_fmincon, nan...
                     fval_grad, sol_fmin_grad', time_fmincon_grad, nan ...
                     obj_cas, sol_cas', time_cas];

```

```
    end

    input.data = data';
    % Set column labels (use empty string for no label):
    input.tableColLabels = sprintfc("$x_0=[%.1f, %.1f]$", x0s')';
    % Set row labels (use empty string for no label):
    input.tableRowLabels = {'$f(x)_{\textit{fmincom}} =$', ...
                            '$x_{\textit{fmincom}} = $', '', ...
                            '$\text{time}_{\textit{fmincom}}\, [s] =$', ...
                            '', ...
                            '$f(x)_{\textit{fmincom grad}} =$', ...
                            '$x_{\textit{fmincom  grad}} =$', '', ...
                            '$\text{time}_{\textit{fmincom  grad}}\, [s] =$'
                                , ...
                            '', ...
                            '$f(x)_{\textit{CasADi}} =$', ...
                            '$x_{\textit{CasADi}} = $', '', ...
                            '$\text{time}_{\textit{CasADi}}\, [s] =$'};
    % Set the row format of the data values
    %input.dataFormatMode = 'row';
    input.dataFormat = {'%.5f'};
    % Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified
        ):
    input.tableColumnAlignment = 'r';
    % Switch table borders on/off:
    input.booktabs = 1;
    % LaTex table caption:
    input.tableCaption = sprintf('Found solution for different inital points
         for the Himmelblau test problem.');
    % LaTex table label:
    input.tableLabel = 'ex4_solve_test_himmel';
    input.makeCompleteLatexDocument = 0;
    input.dataNanString = '';
    input.tablePlacement = '!ht';
    % Now call the function to generate LaTex code:
    latex = latexTable(input);
    savelatexTable(latex, input.tableLabel, 4);


    end

    %% Problem 4.5 - Solve Rosenbrock Part 1
    if runSolveTest_452
    %%
    disp("Solving Rosenbrock Test problem using fmincon and Casadi");

    % Select constraint for rosenbrok, 1 = circle, 2 = box
    con_case = 1;

    switch con_case
        case 1
            con_type_name = "Circle";
            x0s = [[0.0; 0.0]];
        case 2
            con_type_name = "Box";
```

```
            x0s = [[-0.5; 1]];
        otherwise
            error("Unknown case for rosenbrock")
    end

    fig = figure("Name", sprintf("Rosenbrock - solutions for x0 %s constraints", con_type_name), 'Position', [100, 100, 800,800]);
    hold on
    clear data
    % Create contour plot and constraints
    [cfig, conFigs] = contourRosen(con_case);

    for j=1:1
        % Define variables for casadi
        x1 = SX.sym('x1');  % Define variables
        x2 = SX.sym('x2');

        % Solve problem using fmincon
        options = optimoptions('fmincon',...
                               'Display','none',...
                               'Algorithm','interior-point');

        switch con_case
            case 1
                x0 = x0s(:,j);    % Initial point
                xl = [-1; -1];      % Lower bound for x
                xu = [1; 1];        % Upper bound for x
                % For fmincon
                tstart = cputime;
                [sol_fmin,fval,exitflag,output] = fmincon(@objfunRosenbrock,...
                                                          x0, ...
                                                          [], [], [], [], ...
                                                          xl, xu, ...
                                                          @confunRosenbrock, ...
                                                          options);
                time_fmincon = cputime - tstart;

                % For casadi
                g = [x1^2+x2^2];
            case 2
                x0 = x0s(:,j);    % Initial point
                xl = [-1.5; 0.5];       % Lower bound for x
                xu = [1.5; 1.5];         % Upper bound for x
                % fmincon
                tstart = cputime;
                [sol_fmin,fval,exitflag,output] = fmincon(@objfunRosenbrock,...
                                                          x0, ...
                                                          [], [], [], [], ...
                                                          xl, xu, ...
```

```
                                                    @confunRosenbrock_part2, ...
                                                    options);
                time_fmincon = cputime - tstart;
                % For casadi
                g = [];
            otherwise
                error("Unknown case for rosenbrock")
        end

        % Define problem
        hbprob = struct('x',[x1; x2], ...
                        'f',100 * (x2-x1^2)^2 + (1-x1)^2, ...
                        'g',g);

        % Solve the problem with casadi
        options = struct;
        options.ipopt.print_level = 0;
        options.print_time = 0;
        tstart = cputime;
        S = nlpsol('S', 'ipopt', hbprob, options);
        switch con_case
            case 1
                r = S('x0', x0, 'lbg',0,'ubg',1, 'lbx',-1,'ubx',1);
            case 2
                r = S('x0', x0, 'lbx',[-1.5, 0.5],'ubx',[1.5, 1.5]);
        end
        time_cas = cputime - tstart;
        sol_cas = full(r.x);
        obj_cas = full(r.f);

        % Print results
        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t fmincon solution: [%.5e, %.5e], objective: %.5e, time: %.5e\n', sol_fmin(1), sol_fmin(2), fval, time_fmincon);
        fprintf('\t Casadi solution: [%.5e, %.5e], objective: %.5e, time: %.5e\n', sol_cas(1), sol_cas(2), obj_cas, time_cas);

        data(j,:) = [fval, sol_fmin', time_fmincon, nan, ...
                     obj_cas, sol_cas', time_cas, nan, mean(sqrt((sol_fmin-sol_cas).^2))];


        % Add points of interest
        %fm = traceIterations([x0, sol_fmin], "r", '-');
        %cas = traceIterations([x0, sol_cas], "g");
        intp = plotPoint(x0(1),x0(2), "int");
        solp_fm = plotPoint(sol_fmin(1),sol_fmin(2), "sol", "r", 18);
        solp_cas = plotPoint(sol_cas(1),sol_cas(2), "gen", "g", 8);
    end

    legend([intp,solp_fm,solp_cas, conFigs(1)],{'Initial Point', 'fmincon solution', 'CasADi solution',})
    hold off
```

```
    savefigpdf(fig, sprintf('ex4_solve_x0s_test_rosen_%s', lower(con_type_name)), 4);

    input.data = data';
    % Set column labels (use empty string for no label):
    input.tableColLabels = sprintfc("$x_0=[%.1f, %.1f]$", x0s)';
    % Set row labels (use empty string for no label):
    input.tableRowLabels = {'$f(x)_{\textit{fmincom}} =$', ...
                            '$x_{\textit{fmincom}} = $', '', ...
                            '$\text{time}_{\textit{fmincom}}\, [s] =$', ...
                            '', ...
                            '$f(x)_{\textit{CasADi}} =$', ...
                            '$x_{\textit{CasADi}} = $', '', ...
                            '$\text{time}_{\textit{CasADi}}\, [s] =$', ...
                            '', 'Mean Squared Difference = '};
    % Set the row format of the data values
    %input.dataFormatMode = 'row';
    input.dataFormat = {'%.4f'};
    % Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
    input.tableColumnAlignment = 'r';
    % Switch table borders on/off:
    input.booktabs = 1;
    % LaTex table caption:
    input.tableCaption = sprintf('Found solution for different inital points for the Rosenbrock test problem with %s constraint .', lower(con_type_name));
    % LaTex table label:
    input.tableLabel = sprintf('ex4_solve_test_rosen_%s', lower(con_type_name));
    input.makeCompleteLatexDocument = 0;
    input.dataNanString = '';
    input.tablePlacement = '!ht';
    % Now call the function to generate LaTex code:
    latex = latexTable(input);
    savelatexTable(latex, input.tableLabel, 4);

    %% Contour plot - Rosenbrock
    f = @(x,y) (1-x).^2 + 100*(y-x.^2).^2;
    view = 6;
    res = 50;
    x = -view:1/res:view;
    y = -view:1/res:view;
    [X,Y] = meshgrid(x,y);
    v = [0:2:10 10:10:100 100:20:300];
    F = f(X,Y);
    fig = figure;
    hold on
    [m,c]=contour(X,Y,F,v,"linewidth",2);
    colormap("turbo")
    xlabel('x_1','Fontsize',14)
    ylabel('x_2','Fontsize',14)
    colorbar
    xlim([-2, 2])
    ylim([-2, 4])
```

```
    hold off
    savefigpdf(fig, "ex4_5_rosenbrock", 4);
    end

    %% Problem 4.6 - Dampend BFGS
    if runBFGS_46

    clear data
    x0s = [[0.0; 0.0],[1.0; 2.0], [-4.0; 0], [-4; 1]];

    for j=1:length(x0s)
        %sympref('FloatingPointOutput',1);
        x0 = x0s(:,j);    % Initial point

        xl = [-5; -5];       % Lower bound for x
        xu = [5; 5];         % Upper bound for x
        cl = [0; 0];         % Lower bound for constraints
        cu = [47; 70];       % Upper bound for constraints

        tstart = cputime;
        [sol_bfgs, obj, lambda, output] = SQPsolver(@objfungradHimmelblau, ...
                                                    @confungradHimmelblau1, ...
                                                    xl, xu, ...
                                                    cl, cu, ...
                                                    x0, 'bfgs');
        t_bfgs_total = cputime - tstart;


        % Compare with fmin con
        options = optimoptions('fmincon',...
                               'SpecifyObjectiveGradient',true,...
                               'SpecifyConstraintGradient',true,...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin_grad,fval_grad,exitflag_grad,output_grad]=fmincon( ...
                                                @objfungradHimmelblau, x0, ...
                                                [], [], [], [], ...
                                                xl, xu, ...
                                                @confungradHimmelblau1, ...
                                                options);
        time_fmincon_grad = cputime - tstart;

        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t BFGS solution: [%.5e, %.5e], objective: %.5e, time: %.5e, iter: %d\n', sol_bfgs(1), sol_bfgs(2), obj, t_bfgs_total, output.iterations);
        fprintf('\t fmincon grad solution: [%.5e, %.5e], objective: %.5e, time: %.5e\n', sol_fmin_grad(1), sol_fmin_grad(2), fval_grad, time_fmincon_grad);
        fprintf('\t MSE: %.5e\n', mean(sqrt((sol_fmin_grad-sol_bfgs).^2)));

```

```
        fig = figure("Name", sprintf("SQP - BFGS - Himmelblau - Solution for x0=[%+.1f, %+.1f]",x0(1),x0(2)), 'Position', [150, 150, 600, 600]);
        hold on
        % Create contour plot and constraints
        [cfig, conFigs] = contourHimmel(true);

        % Add points of interest
        h = traceIterations(output.xk, "b");
        intp = plotPoint(x0(1),x0(2), "int");
        solp_bfgs = plotPoint(sol_bfgs(1),sol_bfgs(2), "sol", "y", 16);
        %solp_grad = plotPoint(sol_fmin_grad(1),sol_fmin_grad(2), "gen", "g", 8);
        legend([intp,solp_bfgs, h],{'Initial Point', 'SQP BFGS sol.', "Trace for BFGS"},'Location','southwest')
        hold off
        savefigpdf(fig, sprintf("ex4_6_bfgs_himmelblau_x0=%+.0f_%+.0f",x0(1),x0(2)), 4);

        data(j,:) = [fval_grad, sol_fmin_grad', time_fmincon_grad, nan ...
                     obj, sol_bfgs', t_bfgs_total, output.iterations, ...
                     nan, mean(sqrt((sol_fmin_grad-sol_bfgs).^2))];

    end

    input.data = data';
    % Set column labels (use empty string for no label):
    input.tableColLabels = sprintfc("$x_0=[%.1f, %.1f]$", x0s)';
    % Set row labels (use empty string for no label):
    input.tableRowLabels = {'$f(x)_{\textit{fmincom}} =$', ...
                            '$x_{\textit{fmincom}} = $', ...
                            '', ...
                            '$\text{time}_{\textit{fmincom}}\, [s] =$', ...
                            '', ...
                            '$f(x)_{\textit{SQP BFGS}} =$', ...
                            '$x_{\textit{SQP BFGS}} =$', ...
                            '', ...
                            '$\text{time}_{\textit{SQP BFGS}}\, [s] =$', ...
                            '$\text{Interations}_{\textit{SQP BFGS}}\, =$', ...
                            '', ...
                            'MSE = '};

    % Set the row format of the data values
    input.dataFormatMode = 'row';
    input.dataFormat = {'%.4f', 9, "%d", 1, "%.4e",2};
    % Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
    input.tableColumnAlignment = 'r';
    % Switch table borders on/off:
    input.booktabs = 1;
    % LaTex table caption:
    input.tableCaption = sprintf('Comparison of found solution of SQP BFGS and $\textit{fmincon}$ for different inital points for the Himmelblau test problem.');
```

```
    % LaTex table label:
    input.tableLabel = 'ex4_bfgs_himmel';
    input.makeCompleteLatexDocument = 0;
    input.dataNanString = '';
    input.tablePlacement = '!ht';
    % Now call the function to generate LaTex code:
    latex = latexTable(input);
    savelatexTable(latex, input.tableLabel, 4);

    end
    %% Problem 4.7 - Dampend BFGS with line search
    if runBFGS_LS_47

    clear data
    x0s = [[0.0; 0.0], [-4.0; 0], [-4; 1]];

    fig = figure("Name", "SQP - Line Search - Himmelblau - Solution for x0s
        ", 'Position', [150, 150, 600, 600]);
    hold on
    % Create contour plot and constraints
    [cfig, conFigs] = contourHimmel(true);

    traces = [];
    legens = [];
    for j=1:length(x0s)
        %sympref('FloatingPointOutput',1);
        x0 = x0s(:,j);    % Initial point

        xl = [-5; -5];       % Lower bound for x
        xu = [5; 5];         % Upper bound for x
        cl = [0; 0];         % Lower bound for constraints
        cu = [47; 70];       % Upper bound for constraints

        tstart = cputime;
        [sol_ls, obj, lambda, output] = SQPsolver(@objfungradHimmelblau, ...
                                                  @confungradHimmelblau1,
                                                      ...
                                                  xl, xu, ...
                                                  cl, cu, ...
                                                  x0, 'line');
        t_ls_total = cputime - tstart;


        % Compare with fmin con
        options = optimoptions('fmincon',...
                               'SpecifyObjectiveGradient',true,...
                               'SpecifyConstraintGradient',true,...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin_grad,fval_grad,exitflag_grad,output_grad]=fmincon( ...
                                                 @objfungradHimmelblau, x0,
                                                     ...
                                                 [], [], [], [], ...
                                                 xl, xu, ...
```

```
                                                @confungradHimmelblau1, ...
                                                options);
        time_fmincon_grad = cputime - tstart;

        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t Line Search solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e, iter: %d\n', sol_ls(1), sol_ls(2), obj, t_ls_total,
            output.iterations);
        fprintf('\t fmincon grad solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e\n', sol_fmin_grad(1), sol_fmin_grad(2), fval_grad,
            time_fmincon_grad);
        fprintf('\t MSE: %.5e\n', mean(sqrt((sol_fmin_grad-sol_ls).^2)));


        % Add points of interest
        h = traceIterations(output.xk, "b");
        intp = plotPoint(x0(1),x0(2), "int");
        solp_ls = plotPoint(sol_ls(1),sol_ls(2), "sol", "y", 16);
        % solp_grad = plotPoint(sol_fmin_grad(1),sol_fmin_grad(2), "gen", "g
            ", 8);

        data(j,:) = [fval_grad, sol_fmin_grad', time_fmincon_grad, nan ...
                    obj, sol_ls', t_ls_total, output.iterations,
                        output.function_calls ...
                     nan, mean(sqrt((sol_fmin_grad-sol_ls).^2))];

    end

    legend([intp,solp_ls, h],{'Initial Point' 'SQP LS sol.', 'Trace for LS.'
        },'Location','southwest')
    hold off
    savefigpdf(fig, "ex4_6_ls_himmelblau_x0s", 4);


    input.data = data';
    % Set column labels (use empty string for no label):
    input.tableColLabels = sprintfc("$x_0=[%.1f, %.1f]$", x0s')';
    % Set row labels (use empty string for no label):
    input.tableRowLabels = {'$f(x)_{\textit{fmincom}} =$', ...
                            '$x_{\textit{fmincom}} = $', ...
                            '', ...
                            '$\text{time}_{\textit{fmincom}}\, [s] =$', ...
                            '', ...
                            '$f(x)_{\textit{SQP LS}} =$', ...
                            '$x_{\textit{SQP LS}} =$', ...
                            '', ...
                            '$\text{time}_{\textit{SQP LS}}\, [s] =$', ...
                            '$\text{Interations}_{\textit{SQP LS}}\, =$',
                                ...
                            '$\text{Function Calls}_{\textit{SQP LS}}\, =$',
                                ...
                            '', ...
                            'MSE = '};

    % Set the row format of the data values
```

```
    input.dataFormatMode = 'row';
    input.dataFormat = {'%.5f', 9, "%d", 2, "%.5e",2};
    % Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
    input.tableColumnAlignment = 'r';
    % Switch table borders on/off:
    input.booktabs = 1;
    % LaTex table caption:
    input.tableCaption = sprintf('Comparison of found solution of SQP Line Search and \\textit{fmincon} for different inital points for the Himmelblau test problem.');
    % LaTex table label:
    input.tableLabel = 'ex4_ls_himmel';
    input.makeCompleteLatexDocument = 0;
    input.dataNanString = '';
    input.tablePlacement = '!ht';
    % Now call the function to generate LaTex code:
    latex = latexTable(input);
    savelatexTable(latex, input.tableLabel, 4);

    end

    %% Problem 4.8 - Dampend BFGS with line search
    if runBFGS_TR_48
    clear data
    x0s = [[0.0; 0.0],[-4.0; 0], [-4; 1]];

    fig = figure("Name", "SQP - Trust Region - Himmelblau - Solution for x0s", 'Position', [150, 150, 600, 600]);
    hold on
    % Create contour plot and constraints
    [cfig, conFigs] = contourHimmel(true);

    traces = [];
    legens = [];
    for j=1:length(x0s)
        %sympref('FloatingPointOutput',1);
        x0 = x0s(:,j);    % Initial point

        xl = [-5; -5];       % Lower bound for x
        xu = [5; 5];         % Upper bound for x
        cl = [0; 0];         % Lower bound for constraints
        cu = [54; 70];       % Upper bound for constraints

        tstart = cputime;
        [sol_tr, obj, lambda, output] = SQPsolver(@objfungradHimmelblau, ...
                                                  @confungradHimmelblau1, ...
                                                  xl, xu, ...
                                                  cl, cu, ...
                                                  x0, 'trust');

        %[sol_tr,z,Hist, output] = SQP_trust_playground(x0,@objHimmel,@consHimmel,xl,xu,cl,cu,true,9,0.5, 1000);
        t_tr_total = cputime - tstart;
```

```
        %disp(Hist.Iterations)
        %obj = objfungradHimmelblau(sol_tr);

        % Compare with fmin con
        options = optimoptions('fmincon',...
                               'SpecifyObjectiveGradient',true,...
                               'SpecifyConstraintGradient',true,...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin_grad,fval_grad,exitflag_grad,output_grad]=fmincon( ...
                                                @objfungradHimmelblau, x0,
                                                    ...
                                                [], [], [], [], ...
                                                xl, xu, ...
                                                @confungradHimmelblau1, ...
                                                options);

        time_fmincon_grad = cputime - tstart;

        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t Trust regionsolution: [%.5e, %.5e], objective: %.5e,
            time: %.5e, iter: %d\n', sol_tr(1), sol_tr(2), obj, t_tr_total,
            output.iterations);
        fprintf('\t fmincon grad solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e\n', sol_fmin_grad(1), sol_fmin_grad(2), fval_grad,
            time_fmincon_grad);
        fprintf('\t MSE: %.5e\n', mean(sqrt((sol_fmin_grad-sol_tr).^2)));


        % Add points of interest
        h = traceIterations(output.xk, "b");
        intp = plotPoint(x0(1),x0(2), "int");
        solp_ls = plotPoint(sol_tr(1),sol_tr(2), "sol", "y", 16);
        % solp_grad = plotPoint(sol_fmin_grad(1),sol_fmin_grad(2), "gen", "g
            ", 8);

        %data(j,:) = [fval_grad, sol_fmin_grad', time_fmincon_grad, nan ...
        data(j,:) = [ obj, sol_tr', t_tr_total, output.iterations,
            output.function_calls ...
                    nan, mean(sqrt((sol_fmin_grad-sol_tr).^2))];

    end

    legend([intp,solp_ls, h],{'Initial Point' 'SQP TR sol.', 'Trace for TR.'
        },'Location','southwest')
    hold off
    savefigpdf(fig, "ex4_6_tr_himmelblau_x0s", 4);

    input.data = data';
    % Set column labels (use empty string for no label):
    input.tableColLabels = sprintfc("$x_0=[%.1f, %.1f]$", x0s)';
    % Set row labels (use empty string for no label):
    input.tableRowLabels ={'$f(x)=$', ...
                           '$x =$', ...
```

```
                                '', ...
                                'time [s] $=$', ...
                                '$\text{Interations}\; =$', ...
                                '$\text{Function Calls}\, =$', ...
                                '', ...
                                'MSE $=$'};


    % Set the row format of the data values
    input.dataFormatMode = 'row';
    input.dataFormat = {'%.4f', 4, "%d", 2, "%.4e",2};
    % Column alignment ('l'=left-justified, 'c'=centered,'r'=right-justified):
    input.tableColumnAlignment = 'r';
    % Switch table borders on/off:
    input.booktabs = 1;
    % LaTex table caption:
    input.tableCaption = sprintf('Comparison of found solution of SQP using Trust Region and \\textit{fmincon} for different initial points for the Himmelblau test problem.');
    % LaTex table label:
    input.tableLabel = 'ex4_tr_himmel';
    input.makeCompleteLatexDocument = 0;
    input.dataNanString = '';
    input.tablePlacement = '!ht';
    % Now call the function to generate LaTex code:
    latex = latexTable(input);
    savelatexTable(latex, input.tableLabel, 4);
    end

    %% 4.9 Himmelblau
    if himmelblau49
    clear data
    x0s = [[-4.0; 0]];

    fig = figure("Name", "SQP - Line Search - Himmelblau - Solution for x0s", 'Position', [150, 150, 600, 600]);
    hold on
    % Create contour plot and constraints
    [cfig, conFigs] = contourHimmel(true);

    traces = [];
    legens = [];
    for j=1:1
        %sympref('FloatingPointOutput',1);
        x0 = x0s(:,j);    % Initial point

        xl = [-5; -5];       % Lower bound for x
        xu = [5; 5];         % Upper bound for x
        cl = [0; 0];         % Lower bound for constraints
        cu = [47; 70];       % Upper bound for constraints

        tstart = cputime;
        [sol_tr, obj, lambda, output] = SQPsolver(@objfungradHimmelblau, ...
```

```
                                                    @confungradHimmelblau1, ...
                                                    xl, xu, ...
                                                    cl, cu, ...
                                                    x0, 'trust');
        t_tr_total = cputime - tstart;


        % Compare with fmin con
        options = optimoptions('fmincon',...
                               'SpecifyObjectiveGradient',true,...
                               'SpecifyConstraintGradient',true,...
                               'Display','none',...
                               'Algorithm','interior-point');
        tstart = cputime;
        [sol_fmin_grad,fval_grad,exitflag_grad,output_grad]=fmincon( ...
                                            @objfungradHimmelblau, x0, ...
                                            [], [], [], [], ...
                                            xl, xu, ...
                                            @confungradHimmelblau1, ...
                                            options);
        time_fmincon_grad = cputime - tstart;

        % Define problem for casadi
        x1 = SX.sym('x1');  % Define variables
        x2 = SX.sym('x2');
        % Define problem
        hbprob = struct('x',[x1; x2], ...
                        'f',(x1^2+x2-11)^2+(x1+x2^2-7)^2, ...
                        'g',[(x1+2)^2-x2; -4*x1+10*x2] ...
                        );

        % Solve the problem with casadi
        options = struct;
        options.ipopt.print_level = 0;
        options.print_time = 0;
        tstart = cputime;
        S = nlpsol('S', 'ipopt', hbprob, options);
        r = S('x0', x0, 'lbg',0,'ubg',inf);
        time_cas = cputime - tstart;
        sol_cas = full(r.x);
        obj_cas = full(r.f);

        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t Trust Search solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e, iter: %d\n', sol_tr(1), sol_tr(2), obj, t_tr_total,
            output.iterations);
        fprintf('\t fmincon grad solution: [%.5e, %.5e], objective: %.5e,
            time: %.5e\n', sol_fmin_grad(1), sol_fmin_grad(2), fval_grad,
            time_fmincon_grad);
        fprintf('\t MSE: %.5e\n', mean(sqrt((sol_fmin_grad-sol_tr).^2)));


        % Add points of interest
```

```
        h = traceIterations(output.xk, "b");
        intp = plotPoint(x0(1),x0(2), "int");
        solp_tr = plotPoint(sol_tr(1),sol_tr(2), "sol", "y", 16);
        solp_grad = plotPoint(sol_fmin_grad(1),sol_fmin_grad(2), "gen", "g",
            8);
        solp_cas = plotPoint(sol_cas(1),sol_cas(2), "min", "r", 10);

        data(j,:) = [fval_grad, sol_fmin_grad', time_fmincon_grad, nan ...
                    obj, sol_tr', t_tr_total, output.iterations,
                        output.function_calls ...
                     nan, mean(sqrt((sol_fmin_grad-sol_tr).^2))];

    end

    legend([intp, solp_grad, solp_cas, solp_tr, h],{'Initial Point', '
        fmincon sol.', 'CasADI sol', 'SQP TR sol.', 'Trace for LS.'},'
        Location','southwest')
    hold off
    savefigpdf(fig, "ex4_9_ls_himmelblau_single_x0", 4);

    end

    %% Problem 4.9 - Solve Rosenbrock
    if rosenbrock49
    %%
    disp("Solving Rosenbrock Test problem all own solvers");

    % Select constraint for rosenbrok, 1 = circle, 2 = box
    switch rosen_con_case
        case 1
            con_type_name = "Circle";
            x0s = [[0.0; 0.0], [-0.5; 0], [-0.5; -0.5]];
        case 2
            con_type_name = "Box";
            x0s = [[0.0; 1.0], [-1.25; 0.5], [-0.5; 1]];
        otherwise
            error("Unknown case for rosenbrock")
    end

    own_solvers = ["trust"];

    for j=1:length(x0s)

        fig = figure("Name", sprintf("Rosenbrock - solutions for x0 %s
            constraints", con_type_name), 'Position', [100, 100, 800,800]);
        hold on
        %Create contour plot and constraints
        [cfig, conFigs] = contourRosen(rosen_con_case);

        legs = [];
        tits = [];
        for i=1:length(own_solvers)
            solv = own_solvers(i);
            fprintf("Using solver: %s\n", solv)
            switch rosen_con_case
```

```
                case 1
                    x0 = x0s(:,j);    % Initial point
                    xl = [-1; -1];      % Lower bound for x
                    xu = [1; 1];        % Upper bound for x
                    cu = [1;100];
                    cl = [0;-100];
                    % For fmincon
                    tstart = cputime;
                    [sol_own,obj_own, ,output_own] = SQPsolver(
                        @objfungradRosenbrock, ...
                                                                @confungradRosenbrock
                                                                    ,...
                                                                xl, xu, ...
                                                                cl, cu,...
                                                                x0, solv);

                    time_own= cputime - tstart;
                case 2
                    x0 = x0s(:,j);    % Initial point
                    xl = [-1.5; 0.5];      % Lower bound for x
                    xu = [1.5; 1.5];        % Upper bound for x
                    cl = [-100; -100];
                    cu = [100; 100];
                    % fmincon
                    tstart = cputime;
                    [sol_own,obj_own, ,output_own] = SQPsolver(
                        @objfungradRosenbrock, ...
                                                                @confungradRosenbrock_part2
                                                                    ,...
                                                                xl, xu, ...
                                                                cl, cu,...
                                                                x0, solv);

                    time_own = cputime - tstart;
                otherwise
                    error("Unknown case for rosenbrock")
            end
            fprintf('\t %s solution: [%.4f, %.4f], objective: %.4f, Iter: %d
                \n', solv, sol_own(1), sol_own(2), obj_own,
                output_own.iterations);
            fprintf('%d function calls\n', output_own.function_calls)
            fprintf('%.4f time used\n', time_own)

            colors = ['r', 'g', 'y'];
            col = colors(i);
            % Add points of interest
            traceIterations(output_own.xk, col, '-');
            ow = plotPoint(sol_own(1),sol_own(2), "sol", col, 18);
            legs = [legs, ow];
            tits = [tits, solv];
        end

        % Solve problem using fmincon
        options = optimoptions('fmincon',...
                               'Display','none',...
```

```
                                'Algorithm','interior-point');

        switch rosen_con_case
            case 1
                x0 = x0s(:,j);    % Initial point
                xl = [-1; -1];      % Lower bound for x
                xu = [1; 1];        % Upper bound for x
                % For fmincon
                tstart = cputime;
                [sol_fmin,fval,exitflag,output] = fmincon(@objfunRosenbrock,
                    ...
                                                        x0, ...
                                                        [], [], [], [],
                                                            ...
                                                        xl, xu, ...
                                                        @confunRosenbrock
                                                            , ...
                                                        options);
                time_fmincon = cputime - tstart;

            case 2
                x0 = x0s(:,j);    % Initial point
                xl = [-1.5; 0.5];      % Lower bound for x
                xu = [1.5; 1.5];        % Upper bound for x
                % fmincon
                tstart = cputime;
                [sol_fmin,fval,exitflag,output] = fmincon(@objfunRosenbrock,
                    ...
                                                        x0, ...
                                                        [], [], [], [],
                                                            ...
                                                        xl, xu, ...
                                                        @confunRosenbrock_part2
                                                            , ...
                                                        options);
                time_fmincon = cputime - tstart;

            otherwise
                error("Unknown case for rosenbrock")
        end


        % Print results
        fprintf('Found solutions for x0 = [%.2f, %.2f] ::\n', x0(1), x0(2));
        fprintf('\t fmincon solution: [%.5e, %.5e], objective: %.5e, time: %
            .5e\n', sol_fmin(1), sol_fmin(2), fval, time_fmincon);


        %data(j,:) = [fval, sol_fmin', time_fmincon, nan, ...
        %             obj_cas, sol_cas', time_cas, nan, mean(sqrt((sol_fmin-
            sol_cas).^2))];


        % Add points of interest
        %fm = traceIterations([x0, sol_fmin], "r", '-');
```

```
        intp = plotPoint(x0(1),x0(2), "int");
        solp_fm = plotPoint(sol_fmin(1),sol_fmin(2), "sol", "b", 10);
        legs = [legs, intp, solp_fm];
        tits = [tits, 'Initial point', 'Fmincon'];
    end

    legend(legs,tits)
    hold off
    end

    %% Function definition

    function f = objfunHimmelblau(x,p)
        % Function giving the objective of the Himmelblau problem
        % Function taken from: Slide 8, Lecture 01B
        tmp1 = x(1)*x(1)+x(2)-11;
        tmp2 = x(1)+x(2)*x(2)-7;
        f = tmp1*tmp1 + tmp2*tmp2;
    end

    function [c,ceq] = confunHimmelblau(x,p)
        % Function giving the constraints for the Himmelblau problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(2,1);
        ceq = zeros(0,1);

        % Inequality constraints c(x) <= 0
        tmp = x(1)+2;
        c(1,1) = -(tmp*tmp - x(2));
        c(2,1) = -(-4*x(1) + 10*x(2));
    end

    function [f,dfdx] = objfungradHimmelblau(x,p)
        % Function giving the objective and gradients of the Himmelblau
            problem
        % Function taken from: Slide 8, Lecture 01B
        tmp1 = x(1)*x(1)+x(2)-11;
        tmp2 = x(1)+x(2)*x(2)-7;
        f = tmp1*tmp1 + tmp2*tmp2;
        % compute the gradient of f
        if nargout > 1
            dfdx = zeros(2,1);
            dfdx(1,1) = 4*tmp1*x(1) + 2*tmp2;
            dfdx(2,1) = 2*tmp1 + 4*tmp2*x(2);
        end
    end

    function [c,ceq,dcdx,dceqdx] = confungradHimmelblau1(x,p)
        % Function giving the constraints and gradients of the Himmelblau
            problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(2,1);
        ceq = zeros(0,1);
        % Inequality constraints c(x) <= 0
```

```
        tmp = x(1)+2;
        c(1,1) = -(tmp*tmp - x(2));
        c(2,1) = -(-4*x(1) + 10*x(2));
        % Compute constraint gradients
        if nargout > 2
            dcdx = zeros(2,2);
            dceqdx = zeros(2,0);
            dcdx(1,1) = -2*tmp; % dc1dx1
            dcdx(2,1) = 1.0; % dc1dx2
            dcdx(1,2) = 4; % dc1dx1
            dcdx(2,2) = -10; % dc1dx2
        end
    end

    function [h] = traceIterations(xks, color, linetype)
    % Display the iterations for a SQP algortihm for two variables
        if nargin<2
            color = 'r';
        end
        if nargin<3
            linetype = '-.';
        end
        spec = sprintf("%so%s", linetype, color);
        Nvals = size(xks, 2);
        plot(xks(1,[1, Nvals]),xks(2,[1,Nvals]),"MarkerFaceColor", color,'
            markersize',8,'LineStyle', 'none' );
        h = plot(xks(1,:),xks(2,:),spec,'linewidth',2,'MarkerSize',5);
    end

    function f = objfunRosenbrock(x,p)
        % Function giving the objective of the Rosenbrock problem
        f = 100 * (x(2)-x(1)^2)^2 + (1-x(1))^2;
    end

    function [c,ceq] = confunRosenbrock(x,p)
        % Function giving the constraints for the Himmelblau problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(1,1);
        ceq = zeros(0,1);

        % Inequality constraints c(x) <= 0
        c(1,1) = x(1)^2 + x(2)^2 - 1;
    end

    function [c,ceq] = confunRosenbrock_part2(x,p)
        % Function giving the constraints for the Himmelblau problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(0,1);
        ceq = zeros(0,1);
    end

    function [f,dfdx] = objfungradRosenbrock(x,p)
        % Function giving the objective and gradients of the Himmelblau
            problem
        % Function taken from: Slide 8, Lecture 01B
```

```
        f = 100 * (x(2)-x(1)^2)^2 + (1-x(1))^2;
        % compute the gradient of f
        if nargout > 1
            dfdx = zeros(2,1);
            dfdx(1,1) = -400*(-x(1)^2 + x(2))*x(1) - 2 + 2*x(1);
            dfdx(2,1) = -200*x(1)^2 + 200*x(2);
        end
    end

    function [c,ceq,dcdx,dceqdx] = confungradRosenbrock(x,p)
        % Function giving the constraints and gradients of the Himmelblau
            problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(1,1);
        ceq = zeros(0,1);
        % Inequality constraints c(x) <= 0
        c(1,1) = x(1)^2 + x(2)^2 - 1;
        % Compute constraint gradients
        if nargout > 2
            dcdx = zeros(2,1);
            dceqdx = zeros(2,0);
            dcdx(1,1) = 2*x(1); % dc1dx1
            dcdx(2,1) = 2*x(2); % dc1dx2
        end
    end

    function [c,ceq,dcdx,dceqdx] = confungradRosenbrock_part2(x,p)
        % Function giving the constraints and gradients of the Himmelblau
            problem
        % Function taken from: Slide 8, Lecture 01B
        c = zeros(0,1);
        ceq = zeros(0,1);
        % Inequality constraints c(x) <= 0
        % Compute constraint gradients
        if nargout > 2
            dcdx = zeros(2,1);
            dceqdx = zeros(2,0);
        end
    end
```

Listing D.8: SQP Trust Region Solver

APPENDIX E

Exercise 5

E.1 Driver Exercise 5

```
%% Problem 5 Driver
%% Clean up
clear
close all

%% Section 5.3
% Setup the artificial financial markets with the securities
returns = [16.1, 8.5, 15.7, 10.02, 18.68];

Sigma = [2.5 0.93 0.62 0.74 -0.23;
         0.93 1.5 0.22 0.56 0.26;
         0.62  0.22 1.9  0.78  -0.27;
         0.74 0.56 0.78 3.6 -0.56;
         -0.23 0.26 -0.27 -0.56 3.9];
%% Formulate as solverable problem

R_val = 12;
Aeq = [returns; [1,1,1,1,1]];
beq = [R_val; 1];
Aineq = -eye(5);
bineq = zeros(5,1);

% Use quadprog to solve the problme
x_min = quadprog(Sigma, [], Aineq, bineq, Aeq, beq);

% value to report
risk_12 = x_min'*Sigma*x_min;

%% Section 5.4

% Make the 1000 return values to be solved
n = 1000;
R = linspace(8.5, 18.68, n);

% allocate memory to store output
risk_R = zeros(n,1);
port_R = zeros(n,5);

% options for solver
```

```
options = optimset('Display', 'off');
for i = 1:length(R)
    beq = [R(i); 1];
    x = quadprog(Sigma, [], Aineq, bineq, Aeq, beq,[],[],[],options);
    risk_R(i) = x'*Sigma*x;
    port_R(i,:) = x;
end

%% Plot Efficient Frontier
% illustrating the ramping
f = figure("Name","Efficientcy frontier");
hold on

opt_return = R(risk_R==min(risk_R));
opt_risk = min(risk_R);

% Plot efficient frontier for each return value
plot(R,risk_R,'r', "LineWidth", 1)
plot(opt_return, opt_risk ,'ko','MarkerFaceColor', 'black');
xlabel('Return')
ylabel('Var[R]')
legend('Efficient frontier','Minimum risk portfolio', 'Location','northwest'
    )
xlim([8.5, 18.68])
hold off

saveas(f, "./figures/Ex5/efficient_frontier",'epsc');

%% Plot the portfolio with minimum variance as a function of return values
f = figure("Name","Portfolio Distribution");
hold on
plot(R,port_R(:,1),"LineWidth",1);
plot(R,port_R(:,2),"LineWidth",1);
plot(R,port_R(:,3),"LineWidth",1);
plot(R,port_R(:,4),"LineWidth",1);
plot(R,port_R(:,5),"LineWidth",1);
legend('Security 1','Security 2','Security 3','Security 4','Security 5','
    Location','north')
xlabel('Return')
ylabel('Fractional Amount')
xlim([8.5, 18.68])
hold off
saveas(f, "./figures/Ex5/ex5_4_portfolio_distribution",'epsc');

%% Tipping point to be reported
beq = [opt_return; 1];
x_min = quadprog(Sigma, [], Aineq, bineq, Aeq, beq);
risk_min = x_min'*Sigma*x_min;

%% Section 5.5-5.7,
% Setup Bi-criterion optimization program for the artificial market.
f = -returns;

Aeq = [1,1,1,1,1];
beq = 1;
```

```
Aineq = -eye(5);
bineq = zeros(5,1);

% The number of trials we will consider
alpha_trials = 1000;
alphas = linspace(0.001, 1, alpha_trials);

x = zeros(alpha_trials,5,3);
port_risk= zeros(alpha_trials,1,3);
port_return= zeros(alpha_trials,1,3);

x0 = zeros(5,1);
s0 = ones(2*5,1);
y0 = ones(length(beq),1);
z0 = ones(2*5,1);

quadtime = zeros(alpha_trials,1);
qpsolvertime = zeros(alpha_trials,1);

% we repeat the experiment 5 times for each alpha
Nsamples = 5;

for i = 1:alpha_trials

    % solve with built-in quadprog solver
    tic
    for sample=1:Nsamples
        x(i,:,1) = quadprog( alphas(i).*Sigma, (1-alphas(i)).*f', Aineq,
            bineq, Aeq, beq,[],[],[],options);
    end
    quadtime(i) = toc/5;

    % solve with own solver
    tic
    for sample=1:Nsamples
        x(i,:,2) = quadraticPrimalDualIM_box(alphas(i).*Sigma, (1-alphas(i))
            .*f',Aeq',beq,zeros(5,1),ones(5,1),x0,y0,z0,s0);
    end
    qpsolvertime(i) = toc/5;

    % compute the portfolio risk and returns
    port_risk(i,2) = x(i,:,2)*Sigma*x(i,:,2)';
    port_return(i,2) = -f*x(i,:,2)';
end

%% Computing MSE and and MAE

% the mean squared error
output_solution_mean_sq_diff = mean((x(:,:,2)-x(:,:,1)).^2,2);
% the mean absolute error
output_solution_MAE = mean(abs(x(:,:,2)-x(:,:,1)),2);
%% MSE figure

hold on
```

```
grid on
plot(alphas, output_solution_mean_sq_diff,'LineWidth',1);
xlabel('\alpha')
ylabel('MSE')
hold off

saveas(gcf,'./figures/Ex5/ProbOwnSolverMSE','epsc')
%% The Mean Absolute Error

figure
hold on
grid on
plot(alphas, output_solution_MAE,'LineWidth',1);
xlabel('\alpha')
ylabel('MAE')
hold off

saveas(gcf,'./figures/Ex5/ProbOwnSolverMAE','epsc')

%% Time comparison between our and built-in solver
f = figure("Name","Time comparrison");
hold on
grid on
plot(alphas, qpsolvertime,'LineWidth',1);
plot(alphas, quadtime,'LineWidth',1);
legend("Our Solver", "QuadProg")
xlabel('\alpha')
ylabel('time [s]')
hold off
saveas(gcf,'./figures/Ex5/ex5_5_time_comparisson','epsc')

%% Efficient Frontier for Bi-Criterion problem
hold on
plot(port_return(:,2),port_risk(:,2),'r', 'LineWidth', 1)
xlim([8.5, 18.68])
xlabel('Return')
ylabel('Var[R]')
hold off
saveas(gcf,'./figures/Ex5/ex5_5_bi_criterion_for_alpha','epsc');


%% Section 5.8-5.11
% add the risk-free asset to the financial system
returns_ext = [16.1, 8.5, 15.7, 10.02, 18.68, 0];
R_val = 14;
Sigma_ext = [2.5 0.93 0.62 0.74 -0.23 0;
         0.93 1.5 0.22 0.56 0.26 0;
         0.62  0.22 1.9  0.78  -0.27 0;
         0.74 0.56 0.78 3.6 -0.56 0;
         -0.23 0.26 -0.27 -0.56 3.9 0;
         0 0 0 0 0 0];

Aeq_ext = [returns_ext; [1,1,1,1,1,1]];
beq = [R_val; 1];
Aineq_ext = -eye(6);
```

```
bineq_ext = zeros(6,1);

x_min_ext = quadprog(Sigma_ext, [], Aineq_ext, bineq_ext, Aeq_ext, beq);

risk_14_ext = x_min_ext'*Sigma_ext*x_min_ext;

%% Run the experiment with 1000 different return values
n = 1000;

R_ext = linspace(0, 18.68, n);
risk_R_ext = zeros(n,1);
port_R_ext = zeros(n,6);

for i = 1:length(R_ext)
    beq = [R_ext(i); 1];
    x_ext = quadprog(Sigma_ext, [], Aineq_ext, bineq_ext, Aeq_ext, beq);
    risk_R_ext(i) = x_ext'*Sigma_ext*x_ext;
    port_R_ext(i,:) = x_ext;
end

%% Plotting efficient frontier for extended market
% find tipping point to be reported and added to plot
opt_return_ext = R_ext(risk_R_ext==min(risk_R_ext));
opt_risk_ext = min(risk_R_ext);

hold on
plot(R_ext,risk_R_ext,'r', "LineWidth", 1)
plot(opt_return_ext, opt_risk_ext ,'ko','MarkerFaceColor', 'black');
plot(returns_ext, diag(Sigma_ext), 'x', 'Color', 'black','MarkerSize',10);
xlabel('Return')
ylabel('Var[R]')
legend('Efficient frontier','Minimum risk portfolio','Security coordinates',
    'Location','northwest')
xlim([0, 18.68])
hold off


saveas(gcf, "./figures/Ex5/ex5_8_efficient_frontier",'epsc');

%% Plot the Portfolio with minimum variance as a function of return
% This time we also plot the risk profile for each of the securities

hold on
plot(R_ext,port_R_ext(:,1),"LineWidth",1);
plot(R_ext,port_R_ext(:,2),"LineWidth",1);
plot(R_ext,port_R_ext(:,3),"LineWidth",1);
plot(R_ext,port_R_ext(:,4),"LineWidth",1);
plot(R_ext,port_R_ext(:,5),"LineWidth",1);
plot(R_ext,port_R_ext(:,6),"LineWidth",1);
legend('Security 1','Security 2','Security 3','Security 4','Security 5','
    Security 6','Location','north')
xlabel('Return')
ylabel('Fractional Amount')
xlim([0, 18.68])
hold off
```

```
saveas(gcf, "./figures/Ex5/ex5_8_portfolio_distribution",'epsc');

%% Based on Frontier, we find the minimum risk (Pareto point)
beq = [opt_return_ext; 1];
x_min_ext = quadprog(Sigma_ext, [], Aineq_ext, bineq_ext, Aeq_ext, beq);
risk_min_ext = x_min_ext'*Sigma_ext*x_min_ext;

%% Plot efficient frontiers

hold on
plot(R,risk_R,'b', "LineWidth", 1)
plot(R_ext,risk_R_ext,'r', "LineWidth", 1)
plot(14, 0.7214, 'x', 'Color','blue','MarkerSize',10)
plot(14, 0.6377, 'x', 'Color','red','MarkerSize',10)
xlabel('Return')
ylabel('Var[R]')
legend('Prior to Risk-Free sec.','With Risk-Free sec.', 'Location','
    northwest')
xlim([8.5, 18.68])

hold off

saveas(gcf, "./figures/Ex5/ex5_11_efficient_frontiers",'epsc');
```

Listing E.1: The Driver used to solver Exercise 5

Bibliography

[1] J. B. Jørgensen, *Numerical Methods for Constrained Optimization.* Springer, first ed., 2021.

[2] S. R. Garcia and R. A. Horn, *A second course in linear algebra.* Cambridge University Press, 2017.

[3] J. R. R. A. Martins and A. Ning, *Engineering Design Optimization.* first ed., 2021.

[4] J. Nocedal and S. J. Wright, *Numerical Optimization.* New York, NY, USA: Springer, second ed., 2006.

[5] "Matlab ldl factorization." `https://se.mathworks.com/help/dsp/ref/ldlfactorization.html"`.

[6] J. B. Jørgensen, "Convex quadratic programming - primal-dual interior point algorithm lecture slides for lecture 06," 2022.

[7] S. J. Wright, *Primal-Dual Interior-Points Methods.* Philadelphia, Pa, USA: SIAM, 1997.

[8] J. B. Jørgensen, "Linear programming lecture 07b - primal-dual interior-point algorithm lecture slides for lecture 06," 2022.

[9] J. B. Jørgensen, "Sequential quadratic programming (sqp) lecture 09a introduction to sqp algorithms lecture slides for lecture 09," 2022.

[10] J. B. Jørgensen, "Sequential quadratic programming (sqp) lecture 09b practical algorithms lecture slides for lecture 09," 2022.

[11] J. B. Jørgensen, "02612 constrained optimization lecture 09c box constrained optimization and parameter estimation lecture slides for lecture 09," 2022.

[12] D. Boirou and J. B. Jørgensen, "Sequential l_1 quadratic programming for nonlinear model predictive control," 2019.

[13] J. C. Hull, *Options, Futures, and Other Derivatives.* 2015.

[14] J. C. Hull, *Risk Management and Financial Institutions.* 2018.

$$\Theta^{\sqrt{17}} + \Omega \int \delta e^{i\pi} = -1$$

$$\{2.7182818284\} \mathring{\lambda}$$

ϑφερτυθιοπσδφγηξκλ

$$\chi^2 \sum_{!} \gg \approx$$