DTU Compute
Department of Applied Mathematics and Computer Science

Exam hand-in

02612 Constrained Optimization

Anton Ruby Larse (s174356)

Kongens Lyngby 2021

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

Introduction

In this assignment we will cover different types of constrained optimization problems. The exercise 1 we cover equality constrained quadratic programming (EQP) where we will discuss four different solvers. One based on the LU factorization, one based on the LDL factorization, a range space method and a null space method. In exercise 2 we will cover EQPs with bounds and develop a specialized interior point algorithm based on Mehrota's predictor-corrector method. In exercise 3 we will discuss bound constrained linear programs (LP) for which we will also develop a specialized interior point algorithm based on Mehrota's predictor-corrector method. In exercise 4 we will extend the scope to non linear programming (NLP) and discuss the framework of sequential quadratic programming for which we will develop five different solvers. One solver with a damped BFGS update of the hessian, one with a damped BFGS update of the hessian and line search, the two previous but extended with infeasibility handling and lastly a trust region based solver. In exercise 5 we will apply quadratic programming to optimize portfolios of securities using Markowitz Portfolio Optimization.

The assignment formulation can be found in appendix A.6.

Contents

CHAPTER 1

Equality Constrained Convex Quadratic Programming

An equality constrained convex quadratic program(EQP) can be formulated as seen in 1.1

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2}x^T Hx + g^T x \\ \text{s.t.} \quad & A^T x = b \end{aligned} \tag{1.1}$$

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

1.1 Exercise 1.1

When we want to minimize a function subject to one or more constraints we can use a representation called the Lagrangian function. The function is given on page 44 in the lecture notes [Jør21] as

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

Where $\mathcal{E}$ is the set of all equality constraints, $\mathcal{I}$ is the set of all inequality constraints and λ are the Lagrange multipliers. The Lagrangian function can be defined with a + or - in front of the Lagrange multipliers. This is something one needs to be aware of when developing solvers or formulating problems to an already developed solver but this we will get back to if necessary.

The Lagrangian function for our problem has the form

$$\begin{aligned}\mathcal{L}(x,\lambda) &= f(x) - \sum_{i\in\mathcal{E}} \lambda_i c_i(x) \\ &= \frac{1}{2}x^T Hx + g^T x - \lambda^T(A^T x - b)\end{aligned} \tag{1.3}$$

1.2 Exercise 1.2

When developing algorithms to solve a convex EQP one must know when an optimal solution is achieved. Necessary conditions for such a solution are known as the first order conditions or the Karush-Kuhn-Tucker (KKT) conditions. These are stated in proposition 2.10 on page 44 in the lecture notes [Jør21] and are as follows

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) - \sum_{i\in\mathcal{E}\cup\mathcal{I}} \lambda_i \nabla c_i(x) = 0 \tag{1.4}$$

$$c_i(x) = 0, \quad i \in \mathcal{E} \tag{1.5}$$

$$c_i(x) \geq 0, \quad i \in \mathcal{I} \tag{1.6}$$

$$\lambda_i \geq 0, \quad i \in \mathcal{I} \tag{1.7}$$

$$c_i(x) = 0 \quad \vee \quad \lambda_i = 0, \quad i \in \mathcal{I} \tag{1.8}$$

We see that only 1.4 and 1.5 are relevant for problems only containing equality constraints and hence only these are relevant for our problem. To understand what the first order conditions means for an EQP we will look closer at 1.5.

$$\begin{aligned}c_i(x) &= \nabla_\lambda \mathcal{L}(x,\lambda) \\ &= -A^T x + b\end{aligned} \tag{1.9}$$

We see that 1.5 is the partial derivative of 1.3 w.r.t. λ. We can therefore further conclude that for an EQP the first order conditions are satisfied at all stationary points of the Lagrangian. This can geometrically be seen as feasible points where $\nabla f(x)$ is parallel with $\nabla c(x)$ as it can be seen in figure 1.1.

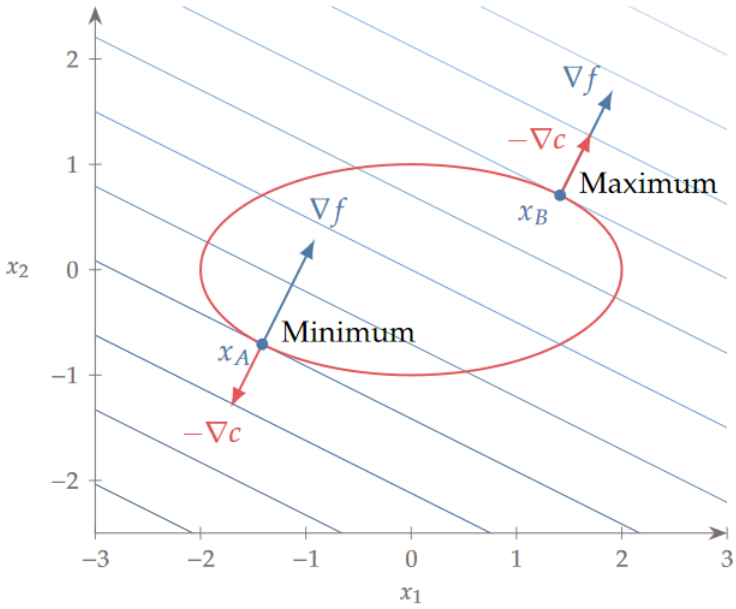


Figure 1.1: We see at both stationary points the gradient of the constraint $c(x)$ is parallel with the gradient of the objective function $f(x)$. The figure is a modification of figure 5.12 in [MN21]

Next we need to investigate if the necessary KKT conditions are sufficient. In the lecture notes [Jør21], section 2.5 states that if our optimization problem is convex the first order conditions are also sufficient. Some intuition for this statement is that convex functions can be understood as a bowl function with only one minimum and hence a stationary point cannot be a saddel point. Futhermore we also know that the obtained minimum is not just a local minimum but the global minimum.
We can therefore conclude that the first order conditions are both necessary and sufficient for a convex EQP.

1.3 Exercise 1.3

For an EQP we can write 1.4 and 1.5 as a linear system of equations.

$$\begin{aligned}
\begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ \nabla_\lambda \mathcal{L}(x,\lambda) \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Leftrightarrow \\
\begin{bmatrix} Hx - A\lambda \\ -A^T x \end{bmatrix} &= \begin{bmatrix} -g \\ -b \end{bmatrix} \Leftrightarrow \\
\begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} &= \begin{bmatrix} -g \\ -b \end{bmatrix}
\end{aligned} \tag{1.10}$$

1.10 we will call the KKT system and in the following develop four different methods to solve it. First we will solve the system directly by use of a LU and LDL factorization. Next we will develop a null space and a range space method which do not factorize the KKT system directly.

1.3.1 LU factorization

We know from section 16.2 in [NW06] that if we have one or more constraints then the matrix of the KKT system will be indefinite. We therefore need a matrix factorization which can handle indefinite matrices. One such matrix factorization is the LU factorization.
To solve for x and λ we follow algorithm 1.

Algorithm 1 An EQP solver based on the LU factorization

1: **procedure** LUSOLVER(H, g, A, b)
2: $KKT \leftarrow \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix}$
3: $[L, U] \leftarrow lu(KKT)$ ▷ LU factorize the KKT matrix
4: $\begin{bmatrix} x \\ \lambda \end{bmatrix} \leftarrow U^{-1}\left(L^{-1}\begin{bmatrix} -g \\ -b \end{bmatrix}\right)$
5: **return** x, λ
6: **end procedure**

We see from algorithm 1 that the matrix of the KKT-system has zeros in the bottom right corner hence a sparse LU-factorization can be beneficial when many constraints are present. Matlab code for both a dense and sparse LU solver are stated in listings 1.1 and 1.2.

```
1 function [x, lambda] = EqualityQPSolverLUdense(H, g, A, b)
2 % EqualityQPSolverLUdense  dense LU solver
3 %
4 %          min  x'*H*x+g'x
```

```
5  %            x
6  %          s.t. A x  = b       (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLUdense(H,g,A,b)
10 %
11 %         x                : Solution
12 %         lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19     % Create a KKT system
20     KKT = get_KKT(H,g,A,b);
21
22     % Fatorize the KKT matrix
23     [L,U,p] = lu(KKT,'vector');
24
25     % Solve for x and lambda
26     rhs = -[g;b];
27     solution(p) = U \ (L \ ( rhs(p)));
28     x = solution(1:size(H,1));
29     lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
30 end
```

Listing 1.1: A dense LU-solver for an EQP

```
1  function [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
2  % EqualityQPSolverLUsparse   Sparse LU solver
3  %
4  %          min  x'*H*x+g'x
5  %           x
6  %          s.t. A x  = b       (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
10 %
11 %         x                : Solution
12 %         lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19     % Create a sparse KKT system
20     KKT = get_KKT_sparse(H,g,A,b);
21
22     % Fatorize the KKT matrix
23     [L,U,p] = lu(KKT,'vector');
24
```

```
25      % Solve for x and lambda
26      rhs = [g;b];
27      solution(p) = U \ (L \ ( -rhs(p)));
28      x = solution(1:size(H,1));
29      lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
30  end
```

Listing 1.2: A sparse LU-solver for an EQP

The LU factorization though have some issues. One problem of the factorization is numeric instability. This we can see from the following example taken from this Stack Exchange post[1]. Consider the matrix

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$$

This has the exact LU decomposition,

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

However, suppose we make a small (relative) rounding error and end up representing $1 - 10^{20}$ as -10^{20}. Then,

$$\begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}$$

which is far from A.

The LU factorization does not utilize the symmetry of the KKT matrix. Hence we will look into the LDL factorization which utilizes the symmetry to enhance the factorization.

1.3.2 LDL factorization

On Matlabs webpage[2] they state that the LDL factorization requires half the computation of the LU decomposition, and is always stable. This agrees with page 455 in [NW06] so clearly a much better factorization. The LDL factorization only needs the matrix to be symmetric. like the matrix of our KKT system. It decomposes the matrix into a lower, upper, and diagonal component. The LDL algorithm is given in algorithm 2.

[1] https://math.stackexchange.com/questions/3052219/gauss-elimination-vs-lu-factorization
[2] https://se.mathworks.com/help/dsp/ref/ldlfactorization.html

Algorithm 2 An EQP solver based on the LDL factorization

procedure LDLSOLVER(H, g, A, b)
  $KKT \leftarrow \begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix}$
  $[L, D] \leftarrow ldl(KKT)$ ▷ LDL factorize the KKT matrix
  $\begin{bmatrix} x \\ \lambda \end{bmatrix} \leftarrow L^{-T}\left(D^{-1}\left(L^{-1}\begin{bmatrix} -g \\ -b \end{bmatrix}\right)\right)$
  return x, λ
end procedure

The LDL solver is implemented with a dense and a sparse version in listings 1.3 and 1.4.

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

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

%%
    % Create a KKT system
    KKT = get_KKT(H,g,A,b);

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

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

Listing 1.3: A dense LDL-solver for an EQP

```
function [x, lambda] = EqualityQPSolverLDLsparse(H, g, A, b)
% EqualityQPSolverLDLsparse    Sparse LDL solver
%
```

```
4  %           min  x'*H*x+g'x
5  %            x
6  %           s.t. A x  = b       (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLDLsparse(H,g,A,b)
10 %
11 %         x                : Solution
12 %         lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19     % Create a sparse KKT system
20     KKT = get_KKT_sparse(H,g,A,b);
21
22     % Fatorize the KKT matrix
23     [L,D,p] = ldl(KKT,'lower','vector');
24
25     % Solve for x and lambda
26     rhs =  -[g;b];
27     solution(p) = L' \ (D \ (L \rhs(p)));
28     x = solution(1:size(H,1));
29     lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
```

Listing 1.4: A sparse LDL-solver for an EQP

1.3.3 Null space method

The null space method is the first of two methods we will cover, which instead of factorizing the KKT system directly, transforms the problem. From page 457 in [NW06] we know that the null space method works as long A is full rank and $Z^T HZ$ is positive definite, where Z is the basis of the null space. The null space method is using the algorithmic idea constraint elimination, presented on page 428-429 in [NW06], where we transform our problem into an unconstrained problem.

The null space method is based on the null space of A. This means that when the number of constraints is almost the same as variables then the null space method should be fast. We have derived the null space method in appendix A.1.1 and summarize it here in algorithm 3. The pseudo code is implemented as matlab code in listing 1.5.

Algorithm 3 An EQP solver based on the null space method

procedure NULLSPACESOLVER(H, g, A, b)
  $\begin{bmatrix} Q_{range} & Q_{null} \end{bmatrix} \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \leftarrow qr(A)$ ▷ QR factorize A
  $\hat{x} \leftarrow Q_{range}(\hat{R}^{-1})^T b$
  $v^* \leftarrow -(Q_{null}^T H Q_{null})^{-1}(\hat{x} H Q_{null} + g^T Q_{null})$
  $x \leftarrow \hat{x} + Q_{null} v^* = Q_{range}(\hat{R}^{-1})^T b + Q_{null} v^*$
  $\lambda \leftarrow \hat{R}^{-1} Q_{range}^T (g + H x^*)$
  return x, λ
end procedure

```
function [x,lambda,time_N] = EqualityQPSolverNullSpace(H,g,A,b)
% EqualityQPSolverNullSpace    Null Space solver
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b       (Lagrange multiplier: lambda)
%
%
% Syntax: [x,lambda,time_N] = EqualityQPSolverNullSpace(H,g,A,b)
%
%          x                : Solution
%          lambda           : Lagrange multipier
%          time_N           : Time spend on qr factorization

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

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

    % Factorize A
    start = cputime;
    [Q,R] = qr(A,'vector');
    time_N = cputime-start;

    % Solve for x and lambda
    Qrange = Q(:,1:m);
    Qnull = Q(:,m+1:n);
    R = R(1:m,1:m);
    Y = (R'\b);
    Qnt = Qnull';
    lpre = Qnt*H*Qnull;
    L = chol(lpre);
    mu=L'\(-Qnt*(H*Qrange*Y+g));
    Z=L\mu;
    x = Qrange*Y+Qnull*Z;
    lambda = R\Qrange'*(g+H*x);
```

Listing 1.5: A Null Space solver for an EQP

1.3.4 Range space method

Lastly we have the range space method. It assumes H is positive definite and is described on page 455-456 in [NW06].

Contray to the null space method it is based on the range space of A. This means that when the number of constraints is much smaller than the number of variables then the range space method should be fast. We have derived the range space method in appendix A.1.2 and summarize it here in algorithm 4. The pseudo code is implemented as matlab code in listing 1.6.

Algorithm 4 An EQP solver based on the range space method

procedure RANGESPACESOLVER(H, g, A, b)
  $R \leftarrow chol(H)$ ▷ Cholesky factorize H
  $hg \leftarrow R^{-1}(R^{-T}g)$
  $ha \leftarrow R^{-1}(R^{-T}A)$
  $\lambda \leftarrow (A^T ha)^{-1}(b + A^T hg)$
  $x \leftarrow ha\lambda - hg$
  return x, λ
end procedure

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

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

%%
    % Factorize H
```

```
21        start = cputime;
22        R=chol(H);
23        time_R = cputime-start;
24
25        % Solve for x and lambda
26        mu=R'\g;
27        Hg=R\mu;
28        mu=R'\A;
29        HA=R\mu;
30        lambda = (A'*HA)\(b+A'*Hg);
31        x = HA*lambda-Hg;
```

Listing 1.6: A Range Space solver for an EQP

1.4 Exercise 1.4

We now test the correctness of our implemented EQP solvers with the problem given problem.

$$
\begin{aligned}
&\mathrm{H} = \\
&\begin{bmatrix}
5.0000 & 1.8600 & 1.2400 & 1.4800 & -0.4600 \\
1.8600 & 3.0000 & 0.4400 & 1.1200 & 0.5200 \\
1.2400 & 0.4400 & 3.8000 & 1.5600 & -0.5400 \\
1.4800 & 1.1200 & 1.5600 & 7.2000 & -1.1200 \\
-0.4600 & 0.5200 & -0.5400 & -1.1200 & 7.8000
\end{bmatrix} \\
&\mathrm{g} = \\
&\begin{bmatrix} -16.1000 \\ -8.5000 \\ -15.7000 \\ -10.0200 \\ -18.6800 \end{bmatrix} \\
&\mathrm{A} = \\
&\begin{bmatrix}
16.1000 & 1.0000 \\
8.5000 & 1.0000 \\
15.7000 & 1.0000 \\
10.0200 & 1.0000 \\
18.6800 & 1.0000
\end{bmatrix} \\
&b = \\
&\begin{bmatrix} \omega \\ 1 \end{bmatrix}
\end{aligned}
\tag{1.11}
$$

Where we sample ω as 20 equidistant points in the range $(8.5, 18.68)$, giving us 20 test problems. We will compare the solution for every solver to the solution

calculated with the Matlab solver quadprog. The matlab code for this test can be found in appendix A.4.1, under 'Given problem'. We see the result in figure 1.2 where all methods give the exact same answer and all with an error less than 10^{-12} which we will accept.

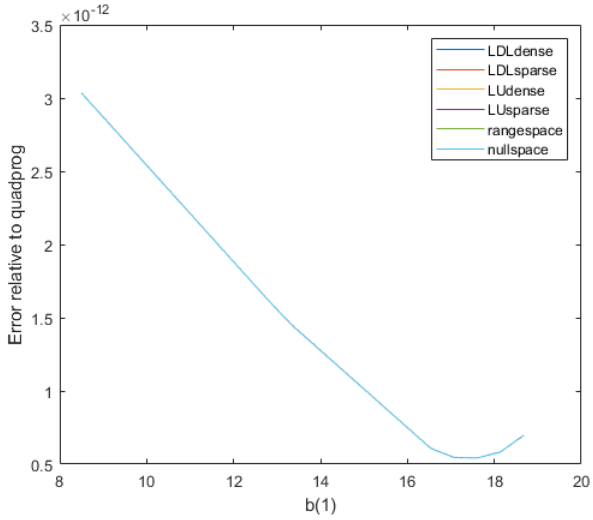


Figure 1.2: The relative error compared to the Matlab solver quadprog for every implemented solver

1.5 Exercise 1.5

Now, where we know our solvers provide a reliable answer, we want to test the efficiency. To do this, we will solve the recycle system 1.12, introduced in week 5.

$$\begin{aligned} \min_u \quad & \tfrac{1}{2}\sum_{i=1}^{n+1}(u_i-\bar{u})^2 \\ \text{s.t.} \quad & -u_1+u_n=-d_0 \\ & u_i-u_{i+1}=0 \quad i=1,2,\ldots,n-2 \\ & u_{n-1}-u_n-u_{n+1}=0 \end{aligned} \tag{1.12}$$

where $\bar{u}$ and d_0 are parameters of the problem. The problem size can be adjusted selecting $n > 3$. Before we are able to solve 1.12 with our solvers we need the system to be in the form 1.1.

First we look at the objective function of 1.12

$$
\begin{aligned}
f(u) &= \frac{1}{2}\sum_{i=1}^{n+1}(u_i-\bar{u})^2 \\
&= \frac{1}{2}\left((u_1-\bar{u})^2+(u_2-\bar{u})^2+\ldots+(u_{n+1}-\bar{u})^2\right) \\
&= \frac{1}{2}\left(u_1^2+\bar{u}^2-2u_1\bar{u}+u_2^2+\bar{u}^2-2u_2\bar{u}+\ldots+u_{n+1}^2+\bar{u}^2-2u_{n+1}\bar{u}\right) \\
&= \frac{1}{2}\left(\begin{bmatrix}u_1 & u_2 & \ldots & u_{n+1}\end{bmatrix} I^{[(n+1)\times(n+1)]}\begin{bmatrix}u_1\\u_2\\\vdots\\u_{n+1}\end{bmatrix} - 2\begin{bmatrix}\bar{u} & \bar{u} & \ldots & \bar{u}\end{bmatrix}\begin{bmatrix}u_1\\u_2\\\vdots\\u_{n+1}\end{bmatrix} + n+1\bar{u}^2\right) \\
&= \frac{1}{2}u^T I^{[(n+1)\times(n+1)]}u - \bar{u}^{[1\times(n+1)]}u + \frac{n+1}{2}\bar{u}^2 \qquad (1.13)
\end{aligned}
$$

So

$$x = \begin{bmatrix}u_1\\u_2\\\vdots\\u_{n+1}\end{bmatrix} \qquad (1.14)$$

$$H = I^{[(n+1)\times(n+1)]} \qquad (1.15)$$

$$g = -\bar{u}^{[1\times(n+1)]} \qquad (1.16)$$

Next we look at the constraints

$$-u_1 + u_n = -d_0 \qquad (1.17)$$

$$u_i - u_{i+1} = 0, \quad i = 1, 2, ..n-2 \qquad (1.18)$$

$$u_{n-1} - u_n - u_{n+1} = 0 \qquad (1.19)$$

giving

$$A = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \ddots & 1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 1 \\ 1 & 0 & 0 & \cdots & 0 & -1 \\ 0 & 0 & 0 & \cdots & 0 & -1 \end{bmatrix}^{[(n+1)\times n]} \tag{1.20}$$

$$b = \begin{bmatrix} -d_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^{[1\times n]} \tag{1.21}$$

Now where we have the problem in a suitable form we can test the solvers. The matlab code for this test can be found in appendix A.4.1, under 'Recycling problem'. In figure 1.3 we see the different solvers together with the matlab solver quadprog.

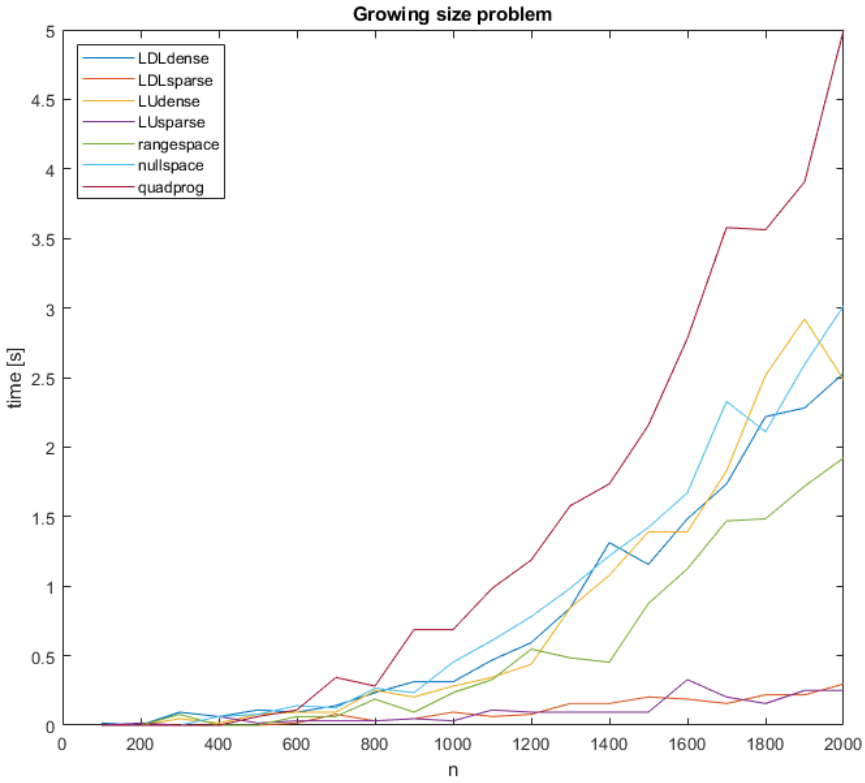


Figure 1.3: The different solvers are tested on 1.12 with n sampled as 20 equidistant points in the range $[100, 2000]$

We see that all our solvers do a better job than quadprog but quadprog is also capable of solving a wide range of different QP's where our solvers only specialize in EQP's. Further we see that the sparse solvers are much faster than the dense solvers which also makes sense due to the sparse nature of the KKT matrix.

More surprising we see that the LDL solver is not detectable faster than the LU solver even though it is stated both in [NW06] and on Matlabs webpage that the LDL solver should use half the computations. We will therefore try to benchmark the factorizations for n up to 5000. The benchmark of the factorizations is done on random EQPs generated by the program found in appendix A.5.1. The matlab code for this test can be found in appendix A.4.1, under 'Factorization benchmark'.

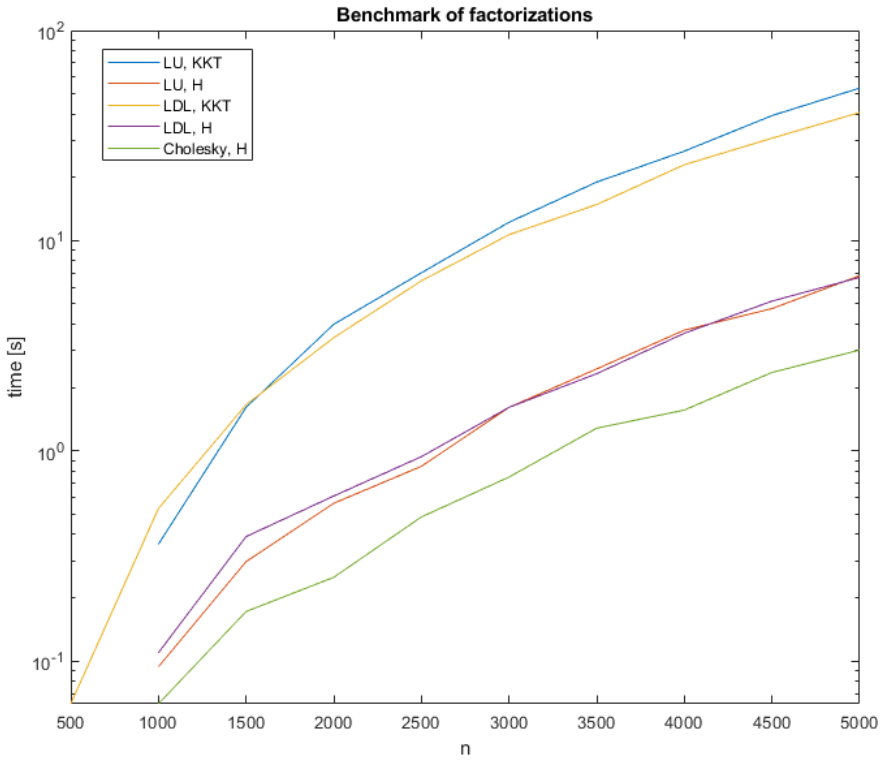


Figure 1.4: The different factorizations tested with n sampled as 20 equidistant points in the range $[250, 5000]$

We see that for indefinite matrices the LDL factorization gets faster than the LU around 1500 and then it only gets faster. We hence conclude that the LDL factorization is algorithmically a fast factorization but Matlabs slow implementation results in it being slower than the LU factorization for small n.

Lastly we will test how the range space method and the null space method, compare with a varying number of constraints, again testing on random EQPs generated appendix A.5.1. We know from section 16.2 in [NW06] that the null space method should be the better method when the degrees of freedom is low, i.e. $n - m$ is small, contraiy to the range space method which is best when the degrees of freedom is high. From section 2.3.3 in [HV07] we know that the tipping point theoretically should be $m \simeq 0.65n$.

We though see from figure 1.5 that the null space method does not get faster than the range space method at any point. We though also see that the QR factorization

is very slow which maybe the reason why the null space method does not get faster than the range space method.

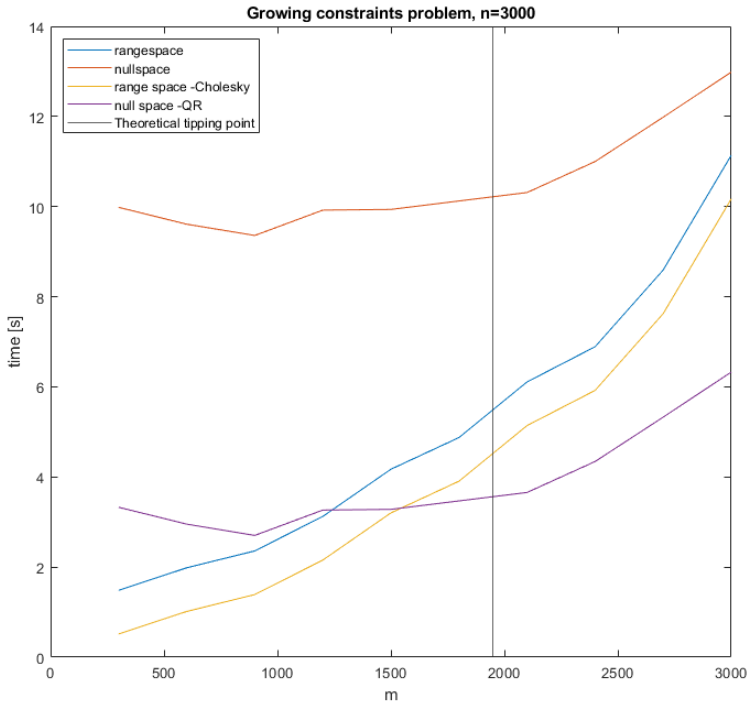


Figure 1.5: Real computational costs for the range space and the null space methods.

CHAPTER 2

Box Constrained Quadratic Programming

In this exercise we will work with a problem of the form

$$\begin{aligned} \min_x \quad & \phi = \tfrac{1}{2} x' H x + g' x \\ \text{s.t.} \quad & A' x = b \\ & l \leq x \leq u \end{aligned} \tag{2.1}$$

Before working with any of the exercises we want the constraints of the problem to be in the form $C^T x \geq d$. So we rewrite the problem.

$$\begin{aligned} \min_x \quad & \phi = \tfrac{1}{2} x' H x + g' x \\ \text{s.t.} \quad & A^T x = b \\ & \begin{bmatrix} I & -I \end{bmatrix}^T x \geq \begin{bmatrix} l \\ -u \end{bmatrix} \end{aligned} \tag{2.2}$$

2.1 Exercise 2.1

To be able to optimize 2.2 we again need the Lagrangian as in exercise 1. As stated in exercise 1 the Lagrangian is given on page 44 in the lecture notes [Jør21] as

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x) \tag{2.3}$$

There is though one major differences between 2.2 and 1.1. Here we have equality and inequality constraints compared to only having equality constraints in exercise 1. Because we now have two kinds of lagrange multipliers we will change notation by

referring to equality multipliers by y and inequality multipliers by z. This gives us the Lagrangian seen in 2.4

$$\begin{aligned}\mathcal{L}(x,y,z) &= f(x) - \sum_{i\in\mathcal{E}} y_i c_i(x) - \sum_{i\in\mathcal{I}} z_i c_i(x) \\ &= \frac{1}{2}x^T Hx + g^T x - y^T(A^T x - b) - z^T(C^T x - d) \\ &= \frac{1}{2}x^T Hx + g^T x - y^T(A^T x - b) - z^T \begin{bmatrix} x - l \\ -x + u \end{bmatrix} \end{aligned} \tag{2.4}$$

One thing to notice is the simple term for the inequality multipliers. This is due to the simple structure of the C matrix which will give rise to many optimizations.

2.2 Exercise 2.2

With the presence of inequalities the first order optimality conditions gets a bit more complicated. We now need all conditions of proposition 2.10 on page 44 in the lecture notes [Jør21] which we again state for good measure.

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) - \sum_{i\in\mathcal{E}} y_i \nabla c_i(x) - \sum_{i\in\mathcal{I}} z_i \nabla c_i(x) = 0 \tag{2.5}$$

$$c_i(x) = 0, \quad i \in \mathcal{E} \tag{2.6}$$

$$c_i(x) \geq 0, \quad i \in \mathcal{I} \tag{2.7}$$

$$z_i \geq 0, \quad i \in \mathcal{I} \tag{2.8}$$

$$c_i(x) z_i = 0, \quad i \in \mathcal{I} \tag{2.9}$$

To obtain an insight in how the three new conditions ensure an optimum in the presence of inequality constraints we will start with 2.7. 2.7 is the inequality counterpart to 2.6 and ensures primal feasibility.

Before we can move on to 2.9 we need to define what active and inactive constraints mean. For a general constraint $c_i(x) \geq 0$, constraint i is said to be active if $c_i(x) = 0$ and inactive if $c_i(x) > 0$. Condition 2.9 then says if c_i is active, the corresponding lagrange multiplier z_i can take values in all of the real numbers. On the other hand if c_i is inactive, z_i must equal zero.

We now only need to understand 2.8 which is also called the dual feasibility condition. This is the hardest to understand so we will use figure 2.1 to support our explanation. Because we have defined our inequalities in 2.2 with greater than or equal, the gradient of the constraints points in the feasible direction, and hence the negative gradient direction is the infeasible direction as it is shown in 2.1. This implies that if one wants to improve the objective while still remain feasible the following

must hold.

$$(\nabla f(x)h < 0) \cap (\cap_{i \in \mathcal{I}} \nabla c(x)h > 0)) \neq \emptyset, \quad \forall h \in \mathcal{D}(x) \tag{2.10}$$

Where $\mathcal{D}(x)$ denotes all direction from a point x. Such an example is shown to the left of 2.1 where this set of feasible descent directions, is shaded blue.
Because 2.5 must also hold at an optimum, the active constraints for which 2.10 hold, must have negative Lagrange multipliers. We hence see that if any Lagrange multipliers are negative we can not be at a minimum. This also implies the opposite, that if all Lagrange multipliers are non negative we must be at a minimum. Hence 2.9 must hold at a minimum.

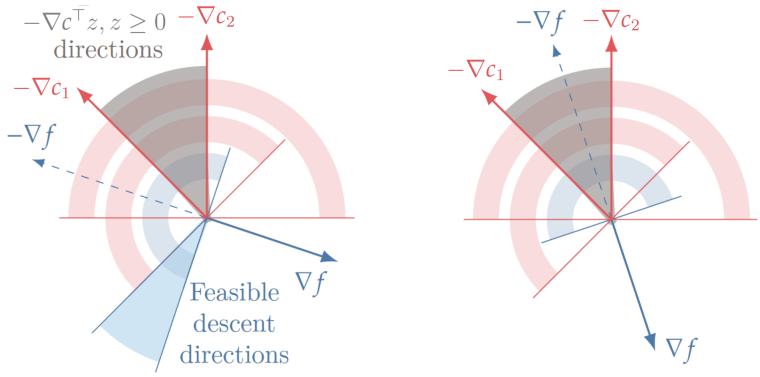


Figure 2.1: Here a general illustration of the first order conditions is given. The figure is a modification of figure 5.18 in [MN21] which can be accessed here

Lastly we need to investigate if the necessary KKT conditions are sufficient. If H is positive semi definite then the program is convex and as in exercise 1, the first order conditions are both necessary and sufficient. If the program is not convex we also need second order conditions to determine if the found stationary point is a minimum. We will assume convexity and therefore not dive deeper into the second order condition here. If one wants to read more about these we refer to section 4.2.

2.3 Exercise 2.3

When solving inequality constrained QPs or LPs two major families of methods exists. One is called active-set methods with algorithms such as the simplex algorithm and primal and dual active set methods. We will not dive further into the family of active set methods here but devote our attention to the other family called interior point methods. We have special focus on a subfamily called primal-dual interior methods from which we will derive an algorithm called the Mehrotra predictor-corrector al-

gorithm. The derivation is based on the slide show "QuadraticOptimization" from lecture 6 and the book [Wri97].

2.3.1 Interior point methods for QPs

We will first develop a general interior point algorithm for QPs and afterwards specialize it for our specific problem. We develop an algorithm for a problem of the form

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2}x^T Hx + g^T x \\ \text{s.t.} \quad & A^T x = b \\ & C^T x \geq d \end{aligned} \tag{2.11}$$

For the interior point method we do not want general inequalities as $C^T x \geq d$ but only inequalities of the form $x \geq 0$. We therefore introduce slack variables to rewrite 2.11. We define

$$s := C^T x - d \geq 0$$

so

$$\begin{aligned} -C^T x + s + d &= 0 \\ s &\geq 0 \end{aligned}$$

We now have 2.11 in a new form

$$\begin{aligned} \min_{x \in R^n} \quad & \phi = \tfrac{1}{2}x^T Hx + g^T x \\ \text{s.t.} \quad & A^T x = b \\ & C^T x + d + s = 0 \\ & s \geq 0 \end{aligned} \tag{2.12}$$

When having introduced slack variables, the KKT conditions of 2.12 are

$$\nabla_x \mathcal{L}(x, y, z) = Hx + g - Ay - Cz = 0 \tag{2.13}$$
$$\nabla_y \mathcal{L}(x, y, z) = -A^T x + b = 0 \tag{2.14}$$
$$\nabla_z \mathcal{L}(x, y, z) = -C^T x + s + d = 0 \tag{2.15}$$
$$SZe = 0 \tag{2.16}$$
$$z, s \geq 0 \tag{2.17}$$

where

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

The primal-dual framework apply a variant of Newton's method where we find a search direction based on 2.13, 2.14, 2.15 and 2.16. A step length is then computed to ensure 2.17.

$$\begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix}_{k+1} = \begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix}_{k} + \alpha \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix}, \quad s.t. \quad (z_{k+1}, s_{k+1}) \geq 0$$

To solve for the search direction we restate the modified KKT conditions 2.13-2.17.

$$F(x, y, z, s) = \begin{bmatrix} Hx + g - Ay - Cz \\ -A^T x + b \\ -C^T x + s + d \\ SZe \end{bmatrix} = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.18}$$
$$z, s \geq 0$$

We apply newtons on 2.18 to solve for the search direction.

$$J_F \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = -F \Rightarrow$$
$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} \tag{2.19}$$

The search direction could also have been formulated with $r_L, r_A, r_C = 0$, which means the starting point needs to be feasible. In stead we have modified the r.h.s. to contain the residuals of 2.13, 2.14 and 2.15. This means the only requirement to a starting point is 2.17. This gives us a framework which can deal with infeasible starting points. The theory on infeasible primal-dual algorithms is further explained in section 6 in [Wri97].

If we just apply the until explained framework we would often experience very poor convergence. This is due to only being able to take very small steps along the computed search direction before violating 2.17. To mitigate this, the primal-dual framework introduce a concept called the central path. The central path is obtained by

adding τ to the KKT conditions.

$$\nabla_x \mathcal{L}(x,y,z) = Hx + g - Ay - Cz = 0 \tag{2.20}$$
$$\nabla_y \mathcal{L}(x,y,z) = -A^T x + b = 0 \tag{2.21}$$
$$\nabla_z \mathcal{L}(x,y,z) = -C^T x + s + d = 0 \tag{2.22}$$
$$SZe = \tau \tag{2.23}$$
$$z, s \geq 0 \tag{2.24}$$

The introduced τ parameterize a set of points $\mathcal{C}$ which composes the central path.

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

Where $\tau > 0$ gives rise to a unique set $(x_\tau, y_\tau, z_\tau, s_\tau)$. This is shown in section 2 in [Wri97].
Having introduced this concept we can incorporate this centering bias in our search direction.

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} + \tau e \end{bmatrix} \tag{2.25}$$

To calculate the centering bias τ we introduce a centering parameter $\sigma \in [0,1]$ and a duality gap μ. The duality gap measures the distance between the current point and the optimal solution by

$$\mu = \frac{s^T z}{m_c}$$

So we can calculate τ in 2.25 by

$$\tau = \sigma\mu, \quad \sigma \in [0,1]$$

A natural question to ask is how to pick the centering parameter σ? Many different strategies have been developed and some are described in chapter 5 in [Wri97]. We will concentrate on a subclass of strategies called predictor-corrector methods.
The predictor-corrector framework combines in some way the two extreme values of σ, 0 and 1. When $\sigma = 1$ we call it the centering direction where we strongly bias the direction towards the central path. When doing so we do not gain any reduction in the duality measure.
When $\sigma = 0$ the direction is called the affine-scaling direction. This direction is the same direction as 2.19 and focuses only on improving the current point. As mentioned earlier this often entail moving to the boundary of the feasible area in (z,s)-space. This leaves subsequent step sizes to be very small if no correction is done.
Motivated by the two explained extremes a plain vanilla predictor-corrector algorithm

therefore alternates between a centering direction called the centering step and an affine-scaling direction called the predictor step. Such an algorithm can be seen in figure 2.2 where the x-space and the (z,s)-space is shown.

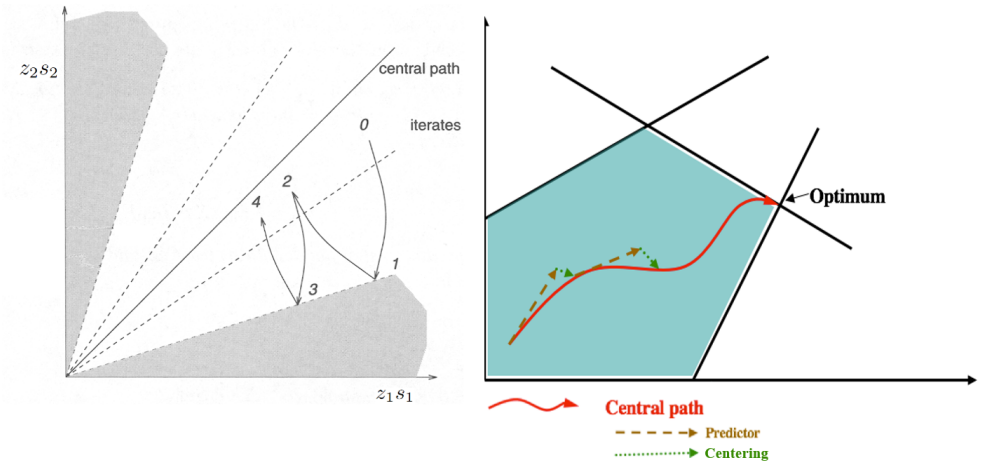


Figure 2.2: Illustration of the central path for a predictor-corrector algorithm with alternating $\sigma = 0$ and $\sigma = 1$ steps. On the right the x-space is shown and on the left (z,s)-space. The illustrations are heavily inspired by this toturial and figure 5.2 in chapter 5 in [Wri97]

If we take a look at the x-space to the right in figure 2.2, we see that the predictor step is tangent to the central path at its starting point. If we were able to make use of second order information in our approximation of the central path we would be able to converge faster.

Such a feature is implemented by the more advanced predictor-corrector method called Mehrota's predictor-corrector method. It adds two key features compared to the plain vanilla predictor-corrector method.

1. It adds second order information called the corrector step.

2. It implements an adaptive choice of σ

To see how the corrector step adds curvature information, we expand $(z_i + \Delta z_i)(s_i + \Delta s_i)$ which should correspond to the last equation of the affine scaling direction, 2.19.

$$(z_i + \Delta z_i)(s_i + \Delta s_i) = s_i z_i + z\Delta s_i + s\Delta z_i + \Delta s_i \Delta z_i = 0$$

We now expand the last equation of the affine scaling direction, 2.19, elementwise

$$z\Delta s_i + s\Delta z_i = -s_i z_i \Rightarrow$$
$$z\Delta s_i + s\Delta z_i + s_i z_i = 0$$

We see that the second order term $\Delta s_i \Delta z_i$ is missing. Therefore by subtracting this term from the r.h.s. of the last equation in 2.25 we obtain second order information to our approximation of the central path. This is called the corrector step and is illustrated in figure 2.3. If we set the centering step σ to 0 we see that the method has a lot in common with a second-order trajectory-following methods known from ODEs as also described in chapter 10 of [Wri97]. One crucial difference though is that we search for a step length along a linear direction rather than a quadratic path. This difference may be significant because the duality gap always decreases for small steps along the quadratic path, whereas it might increase along the linear approximation.

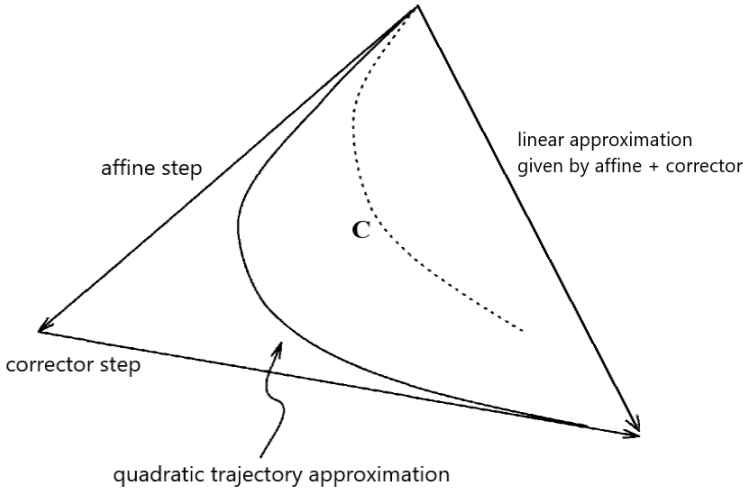


Figure 2.3: A comparison of the corrector-affine step and a step from a second-order trajectory-following method. We see that despite the close relationship between the two methods we cannot quite classify Mehrota's PC method as a second-order trajectory-following algorithm because it still searches along a linear direction rather than a quadratic path. The illustration is heavily inspired by figure 10.1 in chapter 10 in [Wri97]

The second new feature of Mehrota's predictor-corrector method is the adaptively chosen σ. At each iteration we first calculate the affine-scaling direction and assess its usefulness. If we obtain a large reduction in the duality gap we choose a small σ to get as much out of the affine step as possible. Oppositely, if the reduction is small, we want σ to be large. This will center our position and hence we will be able to

obtain a large reduction in the next step. We quantify this idea by selecting σ as

$$\sigma = \left(\frac{\mu^{aff}}{\mu}\right)^3 \tag{2.26}$$

To summarize the Mehrota's predictor-corrector method first calculates an affine direction:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \\ \Delta z^{aff} \\ \Delta s^{aff} \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} \tag{2.27}$$

Then it combines it with a centering direction and a corrector direction by:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} - \Delta Z^{aff} \Delta S^{aff} e + \sigma\mu e \end{bmatrix} \tag{2.28}$$

We combine the directions by use of the centering parameter 2.26. One huge advantage of the method is that the matrix of the affine direction and the full direction is the same. This means we can factorize it once and then reuse it which makes the method computational attractive.

2.3.1.1 Implementation related

The augmented equation is a reduction of the system 2.28. This reduction is motivated by the jacobian which has two problems. It is not symmetric which can cause numerical instability and the matrix is very large. Because the complexity of a matrix factorization grows cubicly with the size, we could gain a lot computational wise if we could reduce the size. On slide 24 and 25 of the slide show "QuadraticOptimization", a reductions of the system is shown which gives a smaller symmetric matrix. We will not go through the reduction here but just state the reduced system.

$$\begin{bmatrix} H + C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -r_L + C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{SZ}) \\ -r_A \end{bmatrix} \tag{2.29}$$

where $\bar{r}_{SZ} = r_{SZ} - \Delta Z^{aff} \Delta S^{aff} e + \sigma\mu e$. We can then compute Δz and Δs by

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

When picking a starting point for the algorithm we just need $(z, s) \geq 0$. However in chapter 5 and 6 in [Wri97] they strongly suggest that a good starting point should satisfy two other conditions. First all pairwise products $z_i s_i$ should be fairly similar and secondly the starting point should not be too infeasible. Meaning that the ratio $\frac{\|(r_L^0, r_A^0, r_C^0)\|}{\mu_0}$ should not be too large.
A heuristic which implements this is given on slide 29 of the slide show "QuadraticOptimization" and states:

Algorithm 5 A starting point heuristic

1: **procedure** StartingPoint($\bar{x}, \bar{y}, \bar{z} \geq 0, \bar{s} \geq 0$)
2: Compute the affine scaling direction 2.27
3: Update the starting point by
4: $x := \bar{x},\ y := \bar{y},\ z := \max\left\{1, \left|\bar{z} + \Delta z^{aff}\right|\right\},\ s := \max\left\{1, \left|\bar{s} + \Delta s^{aff}\right|\right\}$
5: **return** x, y, z, s
6: **end procedure**

2.3.2 Mehrota's PC method for a Box constrained QP

To summarize the above theory we have written a pseudo code in algorithm 6, which implements optimizations specific to a bound constraint EQP. We saw in 2.2, that the C matrix is two concatenated diagonal matrices. This gives rise to many general matrix multiplications can be simplified to element-wise products. Therefore we have made use of the Hadamard product, $\odot$, and the Hadamard division, $\oslash$, in the algorithm 6. Furthermore due the concatenation of two diagonal matrices another general trend in the optimization are two identical operations being performed on variables related to upper bound and lower bounds. We will therefore refer to variables related to upper bounds by subscript u, e.g. s_u, and to lower bounds by subscript l, e.g. s_l.

Algorithm 6 Mehrota's predictor-corrector method for a bound constraint EQP

procedure INTPOINTSOLVEREQPBOUND($H, g, A, b, l, u, x_0, y_0, z_0 > 0, s_0 > 0$)
  $(x, y, z, s) \leftarrow StartingPoint(x_0, y_0, z_0, s_0)$ ▷ Use algorithm 5
  $n \leftarrow length(x)$
  $r_L \leftarrow Hx + g - Ay - (z_l - z_u)$ ▷ Compute the initial residuals
  $r_A \leftarrow b - A^T x$
  $r_C \leftarrow s - \begin{bmatrix} l \\ -u \end{bmatrix} - \begin{bmatrix} x \\ -x \end{bmatrix}$
  $\mu, \mu_0 \leftarrow \frac{z^T s}{2n}$ ▷ Compute the initial dual gap
  while *Not Stop* **do**
    $\bar{H} \leftarrow H + I \odot (z_l \oslash s_l + z_u \oslash s_u)$
    $KKT \leftarrow \begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix}$
    $[L, D] \leftarrow ldl(KKT)$ ▷ LDL factorize the KKT matrix
    $\bar{r}_L \leftarrow r_L - (z_l \oslash s_l) \odot (r_C - s)_l + (z_u \oslash s_u) \odot (r_C - s)_u$
    Solve $KKT \begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$ ▷ Compute the affine direction
    $\Delta z^{aff} \leftarrow - \begin{bmatrix} (z_l \oslash s_l) \odot \Delta x^{aff} \\ -(z_u \oslash s_u) \odot \Delta x^{aff} \end{bmatrix} + z \oslash s \odot (r_C - s)$
    $\Delta s^{aff} \leftarrow -s - (s \oslash z) \odot \Delta z^{aff}$
    $\alpha^{aff} \leftarrow \max_\alpha \alpha, \quad \text{s.t.} \quad z + \alpha \Delta z^{aff} \geq 0, \quad s + \alpha \Delta s^{aff} \geq 0, \quad 1 \geq \alpha \geq 0$
    $\mu^{aff} \leftarrow \frac{(z + \alpha^{aff} \Delta z^{aff})^T (s + \alpha^{aff} \Delta s^{aff})}{2n}$ ▷ Affine dual gap and cetering step
    $\sigma = (\frac{\mu^{aff}}{\mu})^3$
    $\gamma \leftarrow s + \Delta s^{aff} \odot \Delta z^{aff} \oslash z - \mu \sigma \mathbf{e} \oslash z$
    $\bar{r}_L \leftarrow r_L - (z_l \oslash s_l) \odot \gamma_l + (z_u \oslash s_u) \odot \gamma_u$
    Solve $KKT \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$ ▷ Compute the full direction
    $\Delta z \leftarrow - \begin{bmatrix} (z_l \oslash s_l) \odot \Delta x \\ -(z_u \oslash s_u) \odot \Delta x \end{bmatrix} + z \oslash s \odot \gamma$
    $\Delta s \leftarrow -\gamma - (s \oslash z) \odot \Delta z$
    $\alpha \leftarrow \max_\alpha \alpha, \quad \text{s.t.} \quad z + \alpha \Delta z \geq 0, \quad s + \alpha \Delta s \geq 0, \quad 1 \geq \alpha \geq 0$
    $\bar{\alpha} \leftarrow 0.995 \alpha$
    $(x, y, z, s) \leftarrow (x, y, z, s) + \bar{\alpha}(\Delta x, \Delta y, \Delta z, \Delta s)$ ▷ Update current point
    $r_L \leftarrow Hx + g - Ay - (z_l - z_u)$ ▷ Update residuals
    $r_A \leftarrow b - A^T x$
    $r_C \leftarrow s - \begin{bmatrix} l \\ -u \end{bmatrix} - \begin{bmatrix} x \\ -x \end{bmatrix}$
    $\mu \leftarrow \frac{z^T s}{2n}$ ▷ Update the dual gap
    if $\mu \leq \varepsilon 0.01 \mu_0$ **then** ▷ Check convergence
      Stop
    end if
  end while
  return x, y, z, s
end procedure

2.4 Exercise 2.4-2.6

We have implemented the algorithm 6 in Matlab and the code can be seen in appendix A.3.2.1. To test the algorithm we extend the problem 1.11 given in exercise 1.4 with bounds, by restricting x to the interval $0 \leq x \leq 1$. The driver which we used to generate all the following plots can be seen in appendix A.4.3. We have tested our algorithm up against quadprog's interior point algorithm, and the open source library CVX. We first inspect if our algorithm produces correct solutions.

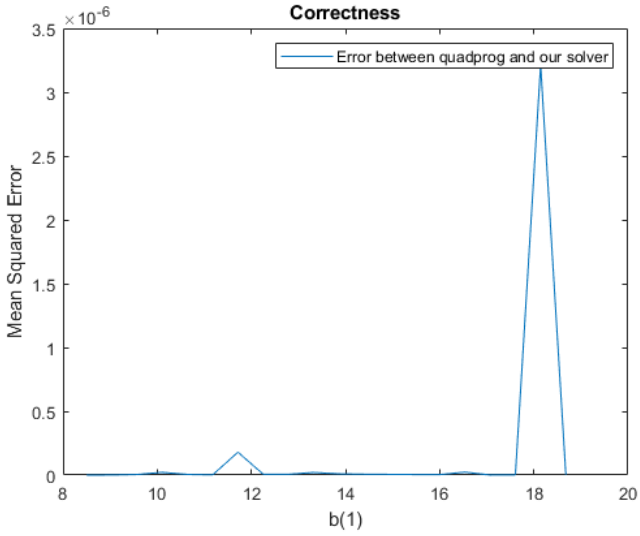
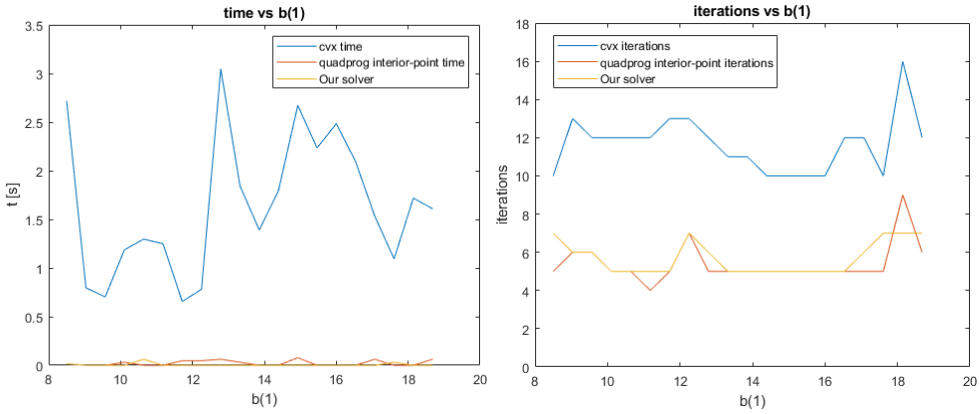


Figure 2.4: Mean squared error between our own solver and quadprog solving 1.11 with bounds

We see that the mean squared error between our solutions and the solutions provided by quadprog do not deviate with more than 10^{-5}. We therefore conclude that our algorithm provides correct answers to the given problems.
Next we test for number of iterations and CPU time.

We see in figure 2.5, that the CVX solver is much slower than the two other solvers. The reason for this is that CVX is a solver which takes a lot of different programs. Due to this versatility it losses out on efficiency. We will therefore try to plot only quadprog and our own algorithm.



(a) CPU time (b) Iterations

Figure 2.5: quadprog, CVX and our own algorithm tested on 1.11 with bounds

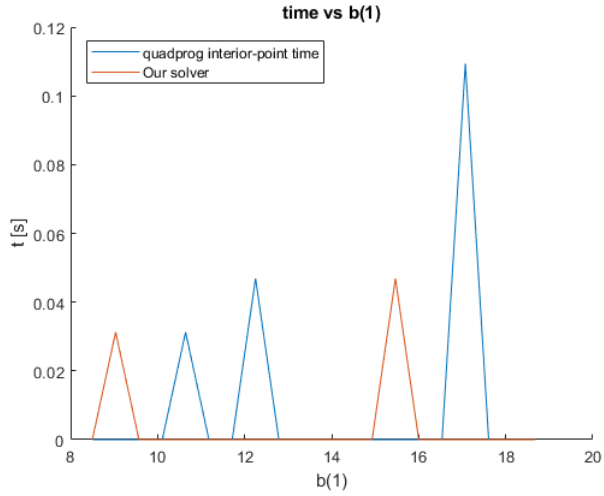
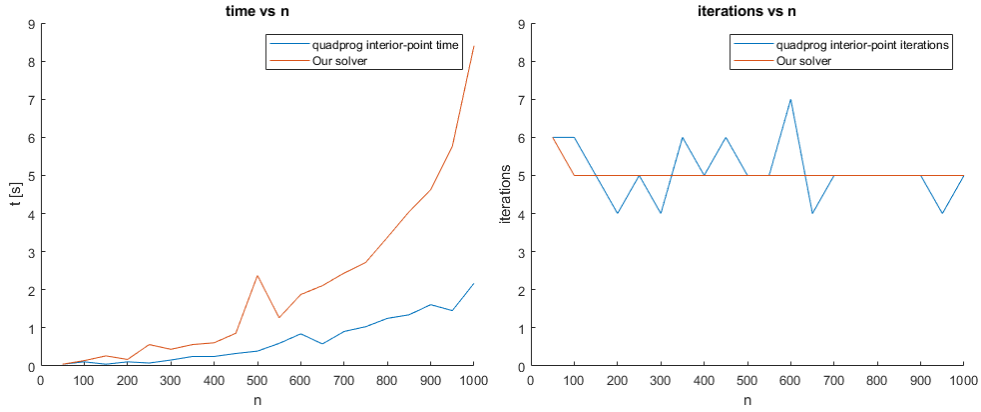


Figure 2.6: CPU time for quadprog and our own algorithm solving 1.11 with bounds

We see that the two solvers, solves the programs so fast that we dont really get any usefull information out of the plot.

We therefore turn to the random EQP generater we used to benchmark the factorizations with in exercise 1. We extend the generated EQP with 0,1-bounds as for 1.11.

We now clearly see from figure 2.7, that the quadprog solver is much faster than



(a) CPU time (b) Iterations

Figure 2.7: quadprog and our own algorithm tested on random EQPs with bounds

our own solver even though we have optimized it to solve EQPs with bounds. The reason for this is of course a blend of many things but we think, there are three dominating reasons. First of all quadprog is implemented in C which is a much faster language than matlab. Secondly we are of course not experts so we may lack some tricks in our code that are utilized in quadprog. Lastly we saw in exercise 1 that the LDL factorization was very slow for indefinite matrices, like the one we factorize in our solver. We cannot test the first two reasons but we can measure how much of the total time in our algorithm is used on the LDL factorization.

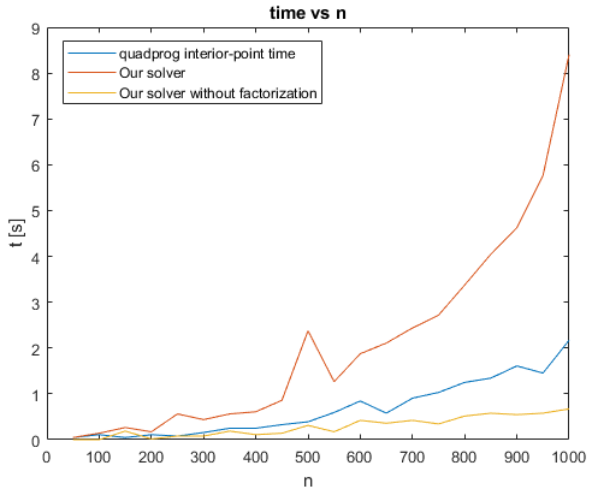


Figure 2.8: CPU time for quadprog, our own algorithm and our own algorithm without factorization, solving random EQPs with bounds

We see from figure 2.8, that without the factorization we are much faster than quadprog which indicates that with a faster factorization we could maybe be faster than quadprog. In later sections we are faster than quadprog with similar algorithms so there is definitely some indications of this could be the case.

CHAPTER 3

Box Constrained Linear Programming

In this exercise we will work with a linear program of the form

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

Before working with any of the exercises we want the constraints of the problem to be in the form $C^T x \geq d$ as in exercise 2. So we rewrite the problem.

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

3.1 Exercise 3.1

We almost have the same problem as in exercise 2. The only thing which has changed is that our objective is now linear instead of quadratic. This does not change the Lagrangian much, leaving us with almost the same Lagrangian as in Exercise 2.

$$\begin{aligned} \mathcal{L}(x, y, z) &= f(x) - \sum_{i \in \mathcal{E}} y_i c_i(x) - \sum_{i \in \mathcal{I}} z_i c_i(x) \\ &= g^T x - y^T (A^T x - b) - z^T (C^T x - d) \\ &= g^T x - y^T (A^T x - b) - z^T \begin{bmatrix} x - l \\ -x + u \end{bmatrix} \end{aligned} \tag{3.3}$$

3.2 Exercise 3.2

Our program 3.2 is still an inequality constrained program so we have the same first order conditions as in exercise 2. We state them again for good measure.

$$\nabla_x \mathcal{L}(x, \lambda) = \nabla f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i \nabla c_i(x) = 0 \tag{3.4}$$

$$c_i(x) = 0, \quad i \in \mathcal{E} \tag{3.5}$$

$$c_i(x) \geq 0, \quad i \in \mathcal{I} \tag{3.6}$$

$$z_i \geq 0, \quad i \in \mathcal{I} \tag{3.7}$$

$$c_i(x) z_i = 0, \quad i \in \mathcal{I} \tag{3.8}$$

Lastly we need to investigate if the necessary first order conditions are sufficient. Because we are working with a linear program the program will always be convex. Therefore as in exercise 1 and 2, and in section 2.5 of the lecture notes [Jør21], the first order conditions are both necessary and sufficient.

3.3 Exercise 3.3

We are again to implement Mehrota's predictor corrector method as in exercise 2. The only difference is that our objective is now linear instead of quadratic. This leaves many of the derivations exactly the same, and we will therefore only comment on parts which are different. If one wants to refresh how the whole interior point framework and the Mehrota's predictor corrector method is working we refer to section 2.3.

The only change to system 2.28 is that H is now 0 giving

$$\begin{bmatrix} 0 & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} - \Delta Z^{aff} \Delta S^{aff} e + \sigma \mu e \end{bmatrix} \tag{3.9}$$

This allows for a further reduction of the augmented system, 2.29, which for 3.9 looks like

$$\begin{bmatrix} C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -r_L + C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{SZ}) \\ -r_A \end{bmatrix} \Rightarrow$$
$$\begin{bmatrix} C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -\bar{r}_L \\ -r_A \end{bmatrix} \tag{3.10}$$

Because $H = 0$ the matrix of 3.10 allows for further reductions to the normal equation, described in section 14.2, [NW06].

$$A(CS^{-1}ZC^T)^{-1}A^T \Delta y = r_A + A^T (CS^{-1}ZC^T)^{-1} \bar{r}_L \tag{3.11}$$

Because we only have box constraints, $C = \begin{bmatrix} I & -I \end{bmatrix}$, $C(S^{-1}Z)C^T$ becomes

$$CS^{-1}ZC^T = S_u^{-1}Z_u + S_l^{-1}Z_l \tag{3.12}$$

This is an diagonal matrix which is very cheap to invert. We therefore for almost no computation, get a significant reduction of the matrix we have to factorize. The factorization is a $O(n^3)$ operation so this a huge computational advantage but further more we also get a positive semi definite matrix. This means we can use the cholesky factorization in stead of a LDL factorization which also gives a speed up in computation.
Besides that everything is the same and we will just state a pseudo code given in algorithm 7.

Algorithm 7 Mehrota's predictor-corrector method for a bound constraint LP

procedure INTPOINTSOLVERLPBOUND($g, A, b, l, u, x_0, y_0, z_0 > 0, s_0 > 0$)
  $(x, y, z, s) \leftarrow StartingPoint(x_0, y_0, z_0, s_0)$ ▷ Use algorithm 5
  $n \leftarrow length(x)$
  $r_L \leftarrow g - Ay - (z_l - z_u)$ ▷ Compute initial residuals
  $r_A \leftarrow b - A^T x$
  $r_C \leftarrow s - \begin{bmatrix} l \\ -u \end{bmatrix} - \begin{bmatrix} x \\ -x \end{bmatrix}$
  $\mu, \mu_0 \leftarrow \frac{z^T s}{2n}$ ▷ Compute the initial dual gap
  while *Not Stop* **do**
    $LHS \leftarrow A^T(\mathbf{e} \oslash (z_l \oslash s_l + z_u \oslash s_u) \odot A)$
    $R \leftarrow chol(LHS)$ ▷ Cholesky factorize LHS
    $\bar{r}_L \leftarrow r_L - (z_l \oslash s_l) \odot (r_C - s)_l + (z_u \oslash s_u) \odot (r_C - s)_u$
    Solve $LHS\ \Delta y^{aff} = -\bar{r}_L$ ▷ Compute the affine direction
    $\Delta x^{aff} \leftarrow \mathbf{e} \oslash (z_l \oslash s_l + z_u \oslash s_u) \odot (-\bar{r}_L + A\Delta y^{aff})$
    $\Delta z^{aff} \leftarrow -\begin{bmatrix} (z_l \oslash s_l) \odot \Delta x^{aff} \\ -(z_u \oslash s_u) \odot \Delta x^{aff} \end{bmatrix} + z \oslash s \odot (r_C - s)$
    $\Delta s^{aff} \leftarrow -s - (s \oslash z) \odot \Delta z^{aff}$
    $\alpha^{aff} \leftarrow \max_\alpha \alpha, \quad \text{s.t. } z + \alpha\Delta z^{aff} \geq 0, \quad s + \alpha\Delta s^{aff} \geq 0, \quad 1 \geq \alpha \geq 0$
    $\mu^{aff} \leftarrow \frac{(z+\alpha^{aff}\Delta z^{aff})^T(s+\alpha^{aff}\Delta s^{aff})}{2n}$ ▷ Affine dual gap and cetering step
    $\sigma = (\frac{\mu^{aff}}{\mu})^3$
    $\gamma \leftarrow s + \Delta s^{aff} \odot \Delta z^{aff} \oslash z - \mu\sigma\mathbf{e} \oslash z$
    $\bar{r}_L \leftarrow r_L - (z_l \oslash s_l) \odot \gamma_l + (z_u \oslash s_u) \odot \gamma_u$
    Solve $LHS\ \Delta y = -\bar{r}_L$ ▷ Compute the full direction
    $\Delta x \leftarrow \mathbf{e} \oslash (z_l \oslash s_l + z_u \oslash s_u) \odot (-\bar{r}_L + A\Delta y)$
    $\Delta z \leftarrow -\begin{bmatrix} (z_l \oslash s_l) \odot \Delta x \\ -(z_u \oslash s_u) \odot \Delta x \end{bmatrix} + z \oslash s \odot \gamma$
    $\Delta s \leftarrow -\gamma - (s \oslash z) \odot \Delta z$
    $\alpha \leftarrow \max_\alpha \alpha, \quad \text{s.t. } z + \alpha\Delta z \geq 0, \quad s + \alpha\Delta s \geq 0, \quad 1 \geq \alpha \geq 0$
    $\bar{\alpha} \leftarrow 0.995\alpha$
    $(x, y, z, s) \leftarrow (x, y, z, s) + \bar{\alpha}(\Delta x, \Delta y, \Delta z, \Delta s)$ ▷ Update current point
    $r_L \leftarrow g - Ay - (z_l - z_u)$ ▷ Update residuals
    $r_A \leftarrow b - A^T x$
    $r_C \leftarrow s - \begin{bmatrix} l \\ -u \end{bmatrix} - \begin{bmatrix} x \\ -x \end{bmatrix}$
    $\mu \leftarrow \frac{z^T s}{2n}$ ▷ Update the dual gap
    if $\mu \leq \varepsilon 0.01\mu_0$ **then** ▷ Check convergence
      Stop
    end if
  end while
  return x, y, z, s
end procedure

3.4 Exercise 3.4-3.5

We have implemented algorithm 7 in Matlab and the code can be seen in appendix A.3.3.1. To test the algorithm we have used the random EQP generater we have seen in exercise 1 and 2. We only use g, A and b and throw H away, to obtain a LP instead of a QP. To obtain the same form as 3.2 we add 0,1-bounds.

The driver which we used to generate all the following plots can be seen in appendix A.4.4. We have tested our algorithm up against Matlab's own linprog using the 'interior point' algorithm and the 'dual-simplex' algorithm. Furthermore we also test up against the open source library CVX. We first inspect if our algorithm produces correct solutions. To do this we compare with linprog's solvers.

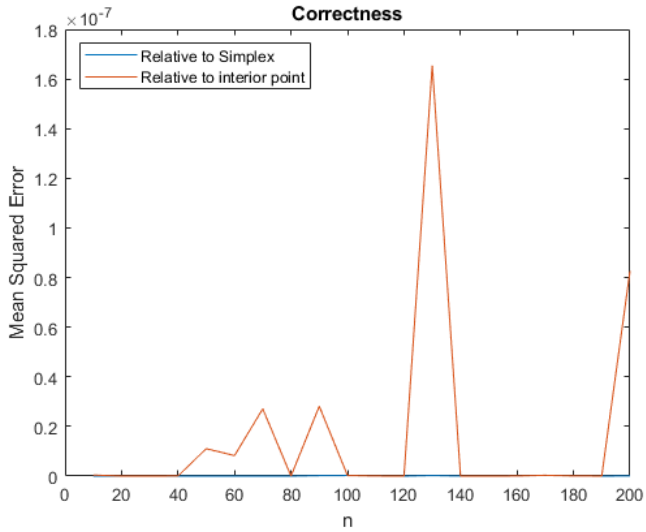


Figure 3.1: Mean squared error between our own solver and linprog's solvers solving a random LP with bounds

We see in figure 3.1, that our solver does not deviate more than 10^{-7} from linprog's solvers. Hence we conclude that our solver provides correct answers.

Next we test for number of iterations and CPU time.

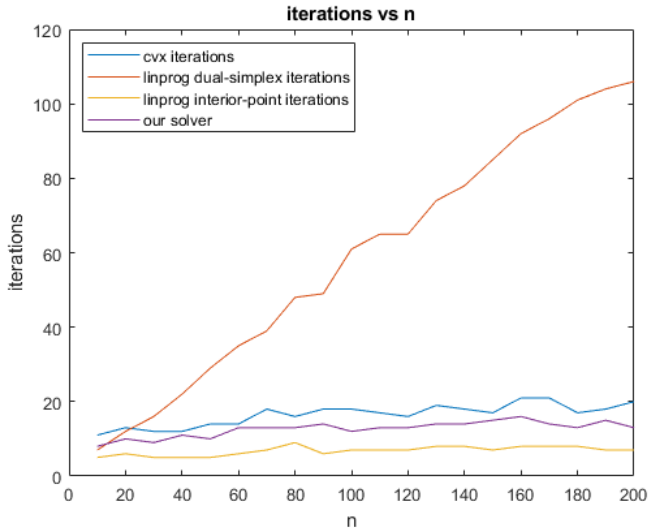


Figure 3.2: Iterations for linprog's solvers, CVX and our own algorithm tested on a random LP with bounds

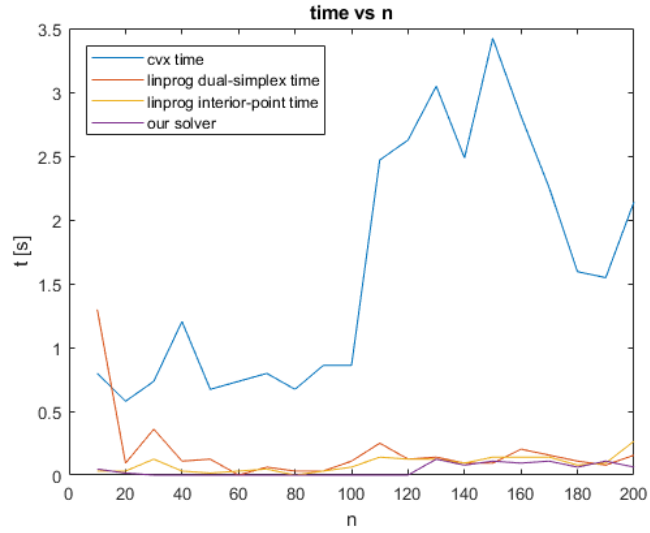


Figure 3.3: CPU time for linprog's solvers, CVX and our own algorithm tested on a random LP with bounds

We see in figure 3.3 that the CVX solver is much slower than the three other

solvers for the same reasons as in exercise 2. We will therefore try to plot only linprog's solvers and our own algorithm.

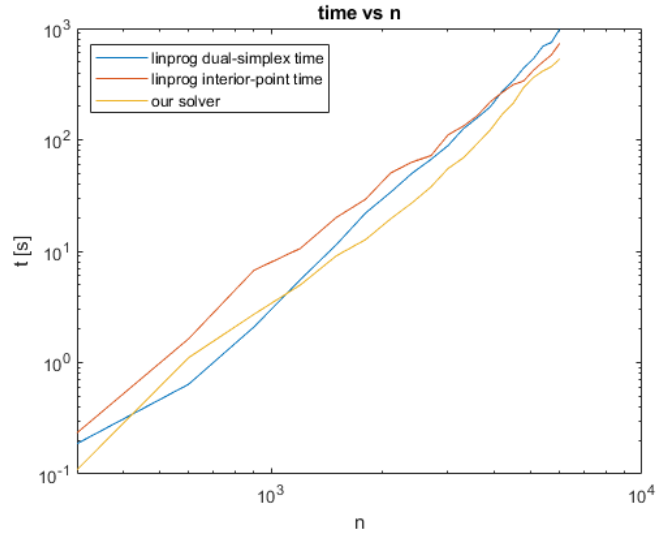


Figure 3.4: CPU time for only linprog's solvers and our own solver

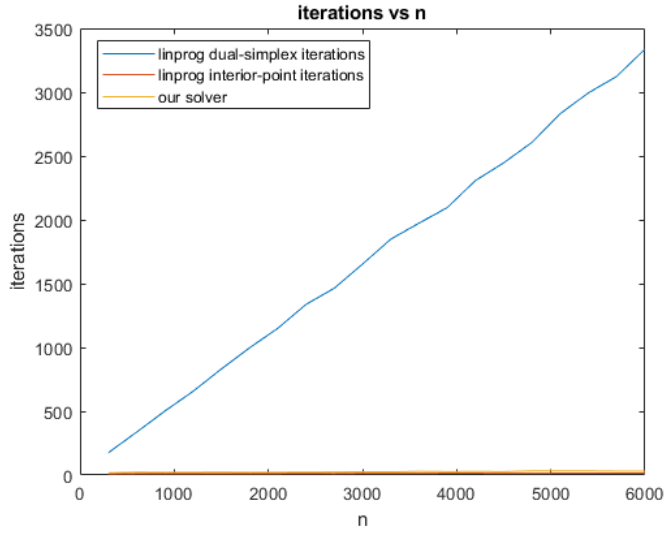


Figure 3.5: Iterations for only linprog's solvers and our own solver

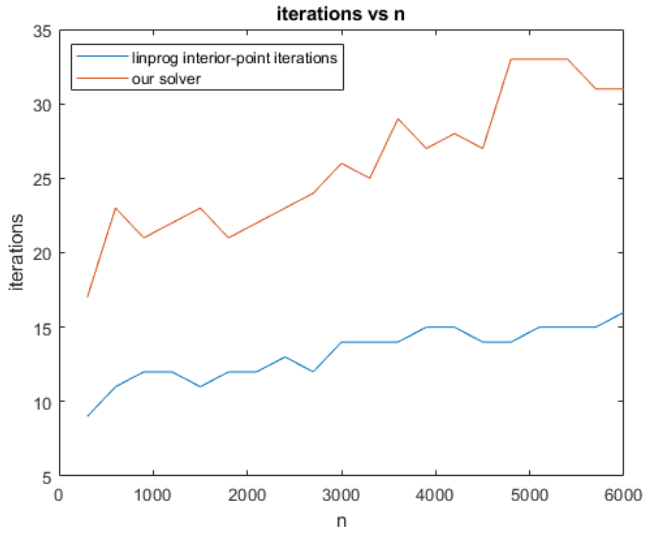


Figure 3.6: Iterations for only linprog's interior point algorithm and our own solver

We see that our own implementation of the interior point algorithm is actually faster then linprog's interior point algorithm. With this information we conclude that it was probably the slow LDL factorization in exercise 2, which where the main reason why the interior point algorithm for the QP was so slow. Here we have reduced the system to the normal equation and hence able to use a cholesky factorization which is fast in Matlab.

We further see from figure 3.4 that the simplex algorithm starts out a bit faster than both interior point algorithms but when n grows it becomes slower than both our and linprog's interior point algorithm. The reason for this can be seen in figure 3.5 where the iterations for the simplex method grows linearly with n. The simplex algorithm is an active set method that utilizes, that the solutions of the underlying LP will always lie on one of the corners of the high dimensional simplex which the problem span. It then solves sub-problems under different active sets jumping from corner to corner. This combinatoric nature is what leads to worse performance for large problems compared to the interior point methods. We see in figure 3.6 that the iterations of the interior point algorithms does also grow with n but much much slower. From this plot we see that linprog's interior point algorithm actually takes fewer iterations than ours. Nevertheless we obtain a fast algorithm because we have specialized it for the specific problem structure.

CHAPTER 4

Sequential Quadratic Programming

We now consider a nonlinear program of the form

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & x_l \leq x \leq x_u \\ & c_l \leq c(x) \leq c_u \end{aligned} \tag{4.1}$$

We assume that the involved functions are twice differentiable and $\nabla c(x)$ has full column rank. Before doing any analysis or development of solvers we will rewrite the program as previous.

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \begin{bmatrix} x \\ -x \\ c(x) \\ -c(x) \end{bmatrix} \geq \begin{bmatrix} l \\ -u \\ c_l \\ -c_u \end{bmatrix} \end{aligned} \tag{4.2}$$

4.1 Exercise 4.1

Even though we have expanded our problem domain significantly compared to the previous three exercises the Lagrangian for the problem still attains the same form described in section 2.3 of the [Jør21]. The Lagrangian function for the specific form 4.2 is given by

$$\mathcal{L}(x, z) = f(x) - z^T \begin{bmatrix} x - l \\ -x + u \\ c(x) - c_l \\ -c(x) + c_u \end{bmatrix}$$

4.2 Exercise 4.2-4.3

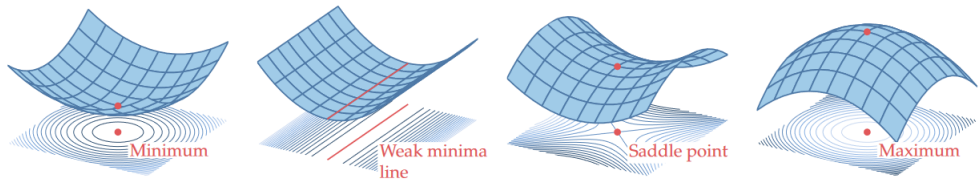
When expanding the scope to all twice differentiable functions we know from section 18.1 in [NW06] that the first order conditions still holds. For our specific problem the first order conditions become

$$\begin{aligned}
\nabla_x \mathcal{L}(x,z) &= f(x) - \begin{bmatrix} I & -I & \nabla c(x) & -\nabla c(x) \end{bmatrix} z = 0 \\
\nabla_z \mathcal{L}(x,z) &= - \begin{bmatrix} x-l \\ -x+u \\ c(x)-c_l \\ -c(x)+c_u \end{bmatrix} \geq 0 \\
z_i &\geq 0 \quad \forall i \in \mathcal{I} \\
z_i c_i(x) &= 0 \quad \forall i \in \mathcal{I}
\end{aligned}$$

These conditions are all necessary for optimality but not sufficient because we cannot guarantee convexity anymore. To ensure a stationary point is a minimum in a non-convex space, we need the necessary second order condition. It is given in proposition 2.14 in [Jør21] as

$$h' \nabla^2_{xx} \mathcal{L}(x,\lambda) h \leq 0 \quad \forall h \in \mathcal{F}(x)$$

where $\mathcal{F}(x)$ is all feasible directions from a feasible point x. By restricting $\nabla_{xx}\mathcal{L}(x,z)$ to be positive semi definite we only accept stationary points for which all feasible directions have non-negative curvature. Hereby we eliminate all saddle points and maxima as seen in figure 4.1.



(a) Positive definite (b) Positive semidefinite (c) Indefinite (d) Negative definite

Figure 4.1: We here illustrate a quadratic function with Hessians spanning from positive definite to negative definite. The illustration is taken from figure 4.11 in [MN21]

4.3 Exercise 4.4

To test the subsequent SQP methods we will use Himmelblau's test problem.

$$\begin{aligned} \min_x \quad & f(x) = \left(x_1^2 + x_2 - 11\right)^2 + (x_1 + x_2^2 - 7)^2 \\ \text{s.t.} \quad & c_1(x) = (x_1 + 2)^2 - x_2 \geq 0 \\ & c_2(x) = -4x_1 + 10x_2 \geq 0 \end{aligned} \tag{4.3}$$

4.2 is given with lower and upper bounds on both general constraints and variables. Therefore we will transform 4.3 to include bounds.

$$\begin{aligned} \min_x \quad & f(x) = \left(x_1^2 + x_2 - 11\right)^2 + (x_1 + x_2^2 - 7)^2 \\ \text{s.t.} \quad & 47 \geq c_1(x) \geq 0 \\ & 70 \geq c_2(x) \geq 0 \\ & 5 \geq x_1 \geq -5 \\ & 5 \geq x_2 \geq -5 \end{aligned} \tag{4.4}$$

4.4 is illustrated in figure 4.2 with all maxima, minima and saddle points marked. The code for plotting 4.4 can be found in appendix A.4.5.

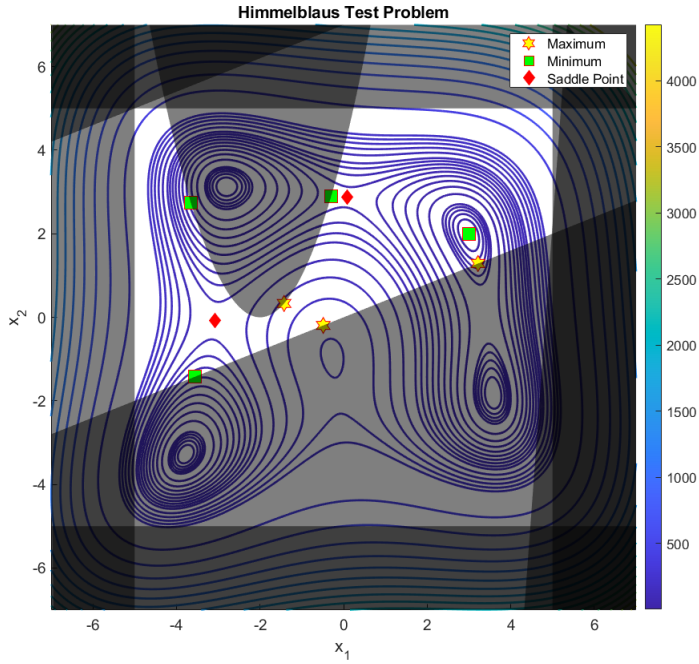


Figure 4.2: 4.4 illustrated with all maxima, minima and saddle points marked.

4.4 Exercise 4.5

Before we start developing any solvers we will try to solve 4.4 with fmincon and CasADi. CasADi is an open source framework for automatic differentation, optimization and optimal control. fmincon is Matlab's own nonlinear solver as quadprog were for quadratic problems and linprog for linear problems.
One significant difference between fmincon and CasADi is how they compute gradients. fmincon uses finite difference while CasADi uses automatic differentation. The two methods are approximatly equally fast but differ in precision. Automatic differentation obtain derivatives with machine precision while finite difference suffer from truncation and round-off error. Therefore if precision is the main concern and fmincon is the only solver available, one should provide the gradients analytically.

We will just stick with finite difference and solve 4.4 with $x_0 = (0, 0)$. A more detailed code than the one below is available in appendix A.4.5.

```
% fmincon
options = optimset('Display', 'off');
xfmin = fmincon(@objfminconHimmel, [0 0], [], [], [], [], l, u, ...
    @consfminconHimmel, options);

% Our own solver
x0 = [0;0];
options = struct('log',false, 'infesibility_handling', false, ...
    'method', 'SQP', 'subsolver', 'own solver');
[xown,¬] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);

% CasADi
S = nlpsol('S', 'ipopt', nlp,options);
r = S('x0', [0,0], 'lbg',0,'ubg',inf);
x_Cas =full(r.x);

% Solutions
>>  CasADi, solution: [3.000000,2.000000]
    fmincon, solution: [3.000000,2.000000]
    Own solver, solution: [2.999995,1.999985]
```

We see in the above that we have included our own solver, to compare it with fmincon and CasADi. How our own solver works will be explained in the rest of this chapter.

4.5 Exercise 4.6-4.8

The section is split up into three subsections. First we will go through the theory behind a basic SQP method with BFGS update, a line search extension, infeasibility

handling and lastly a trust region method. Next we will go through some optimizations specific to the form of 4.2 and lastly we will present results.

4.5.1 Theory

As mentioned we will develop solvers to solve 4.2 using the frame work sequential quadratic program(SQP). Without loss of generality, we will for the subsequent theory, consider the simpler program

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & c(x) \geq 0 \end{array} \tag{4.5}$$

where $x \in R^n$ and $c(x) = \{c_1(x), \ldots, c_m(x)\}^T$. The specific form of 4.2 can easily be introduced later with problem specific optimizations. In the following we will derive the general SQP framework, a quasi Newton extension, a line search extension, infeasibility handling and a trust region based SQP method. The derivation will be based on [SY10], chapter 18 in [NW06] and chapter 5 in [MN21].

4.5.1.1 The general SQP method

To derive the general SQP method we will not start working with 4.5 but a program only containing equality constraints.

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & a(x) = 0 \end{array} \tag{4.6}$$

The Lagrangian function for 4.6 is given by.

$$\mathcal{L}(x, y) = f(x) - y^T a(x)$$

For x to be a stationary point in 4.6 it must hold that

$$F(x, y) = \begin{bmatrix} \nabla_x \mathcal{L}(x, y) \\ \nabla_y \mathcal{L}(x, y) \end{bmatrix} = \begin{bmatrix} \nabla_x f(x) - \nabla_x a(x) y \\ -a(x) \end{bmatrix} = 0$$

We now let (x_k, y_k) be an approximation of the solution to 4.6, and apply Newtons method to improve (x_k, a_k) iteratively.

$$\begin{aligned} \nabla F(x_k, y_k) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} &= -F(x_k, y_k) \Rightarrow \\ \begin{bmatrix} \nabla_{xx} \mathcal{L}(x_k, y_k) & -\nabla_x a(x_k) \\ -\nabla_x a(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} &= - \begin{bmatrix} \nabla_x \mathcal{L}(x, y) \\ \nabla_y \mathcal{L}(x, y) \end{bmatrix} \end{aligned} \tag{4.7}$$

We can add $\begin{bmatrix} \nabla_x a(x_k) y \\ 0 \end{bmatrix}$ to 4.7 and rewrite to obtain

$$\begin{bmatrix} \nabla_{xx}\mathcal{L}(x_k, y_k) & -\nabla_x a(x_k) \\ -\nabla_x a(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} \nabla_x f(x_k) \\ -a(x_k) \end{bmatrix} \tag{4.8}$$

This formulation reminds us about the residual formulation of the primal-dual interior point method, 2.19. It had the ability to start from infeasible start points and the same property holds for the SQP.

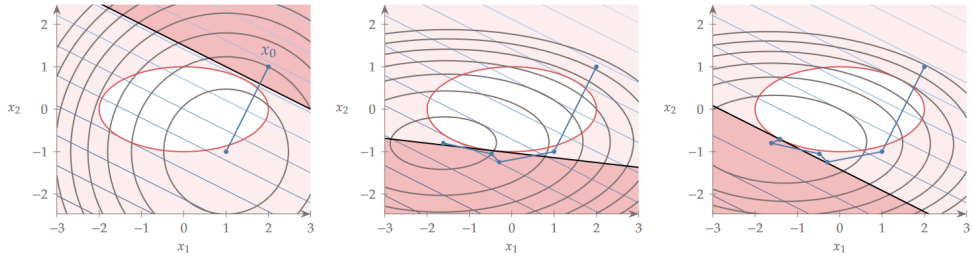
The system 4.8 can be formulated as a QP as

$$\begin{aligned} \min_{\Delta x \in R^n} \quad & \tfrac{1}{2}\Delta x^T \nabla_{xx}\mathcal{L}(x_k, y_k)\Delta x + \nabla_x f\left(x_k\right)^T \Delta x \\ \text{s.t.} \quad & \nabla_x a_i\left(x_k\right)^T \Delta x = -a_i\left(x_k\right) \quad i \in \mathcal{E} \end{aligned} \tag{4.9}$$

To obtain a framework which also accepts inequality constraints, we can just add them to the local EQP problem, 4.9.

$$\begin{aligned} \min_{\Delta x \in R^n} \quad & \tfrac{1}{2}\Delta x^T \nabla_{xx}\mathcal{L}(x_k, y_k, z_k)\Delta x + \nabla_x f\left(x_k\right)^T \Delta x \\ \text{s.t.} \quad & \nabla_x a_i\left(x_k\right)^T \Delta x = -a_i\left(x_k\right) \quad i \in \mathcal{E} \\ & \nabla_x c_i\left(x_k\right)^T \Delta x \geq -c_i\left(x_k\right) \quad i \in \mathcal{I} \end{aligned} \tag{4.10}$$

We can understand the SQP method as a constraint extension of Newton's method. This iterative local QP approximation of the underlying problem is illustrated in figure 4.3



(a) $k = 0$ (b) $k = 3$ (c) $k = 7$

Figure 4.3: In the illustration the blue lines are the contours of the underlying objective, the red ellipse is a non-linear inequality constraint and the black lines illustrate a local IQP approximation. The illustration is taken from figure 5.44 in [MN21]

4.5.1.2 A quasi Newton extension

If n is large it can be very expensive to compute the hessian of the Lagrangian function. We therefore utilize a quasi newton approximation. We will use the BFGS update

which is given as

$$B_{k+1} = B_k - \frac{B_k p_k (B_k p_k)^T}{p_k^T B_k p_k} + \frac{q_k q_k^T}{p_k^T q_k} \tag{4.11}$$

where $p_k = \Delta x_k = x_{k+1} - x_k$ and $q_k = \nabla_x \mathcal{L}(x_{k+1}, z_{k+1}) - \nabla_x \mathcal{L}(x_k, z_{k+1})$. We see that the Lagrangian multipliers are fixed to the updated value, z_{k+1}. The reason behind is that we are only interested in approximating the curvature of $\mathcal{L}$ in the primal space.

Recall from exercise 1 and 2 that for a QP to have an unique solution the problem must be strictly convex. Therefore we introduce a damped version of the BFGS update that ensures positive definiteness. The method replaces q in 4.11 with a vector r defined as

$$r_k = \theta_k q_k + (1 - \theta_k) B_k p_k$$

where $\theta_k \in [0, 1]$ and defined as

$$\theta_k = \begin{cases} 1 & \text{if} \quad p_k^T q_k \geq 0.2 p_k^T B_k p_k, \\ \frac{0.8 p_k^T B_k p_k^T}{p_k^T B_k p_k - p_k^T q_k} & \text{if} \quad p_k^T q_k < 0.2 p_k^T B_k p_k, \end{cases}$$

We see that the damping is activated when the approximated curvature in the new point is below one fifth of the current curvature. When the damping is activated we reuse some of the old hessian to compensate for the flattening curvature. This makes sure B_{k+1} stays positive definite even when f's curvature is flattening.

4.5.1.3 A line search extension

As in the unconstrained case we can also apply line search in the constrained case. We though not only have to consider sufficient decrease in the objective function but also violation of our constraints. To measure this we use what is called a merit function. The specific merit function we apply is called Powell's exact l_1-merit function.

$$P(x, z) = f(x) + \mu^T |min\{0, c(x))\}|$$

Where $\mu \geq |z|$ and updated by $\mu = max\{|z|, \frac{1}{2}(\mu + |z|\}$. The function f is our local objective, given by $\frac{1}{2}\Delta x^T \nabla_{xx} \mathcal{L}(x_k, y_k, z_k) \Delta x + \nabla_x f\,(x_k)^T \Delta x$.

To ensure sufficient decrease we apply the Armijo rule.

$$f(x_k + \alpha_k \Delta x_k) \leq f(x_k) + c_1 \alpha_k \Delta x_k^T \nabla f(x_k)$$

where $c_1 \in (0, 1)$. This condition is illustrated in figure 4.4

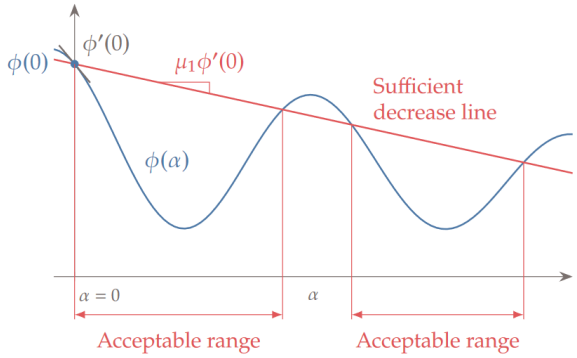


Figure 4.4: The Armijo rule is illustrated in the figure. The illustration is taken from figure 4.22 in [MN21]

To calculate the step size, α, we apply the line search algorithm from slide 25 of the slideshow "Lecture 09A". The line search is given in algorithm 8.

Algorithm 8 A line search algorithm

procedure LINESEARCH($f(x_k), \nabla_x f(x_k), c(x_k), \mu, \Delta x_k$)
  $\alpha \leftarrow 1$
  $\phi_0 \leftarrow f(x_k) + \mu |min\{0, c(x_k)\}|$
  $\nabla_\alpha \phi_0 \leftarrow \nabla_x f(x_k)^T \Delta x_k - \mu^T |min\{0, c(x_k)\}|$
  while not Stop **do**
    $x \leftarrow x_k + \alpha \Delta x_k$
    $f_\alpha \leftarrow f(x)$
    $c_\alpha \leftarrow c(x)$
    $\phi_\alpha \leftarrow f_\alpha + \mu^T |min\{0, c_\alpha\}|$
    if $\phi_\alpha \leq \phi_0 + 0.1\alpha\nabla_\alpha\phi_0$ **then**
      Stop
    else
      $a \leftarrow \frac{\phi_\alpha - (\phi_0 + \alpha\nabla_\alpha\phi_0)}{\alpha^2}$
      $\alpha_{min} \leftarrow \frac{-\nabla_\alpha\phi_0}{a}$
      $\alpha \leftarrow min\{0.9\alpha, max\{\alpha_{min}, 0.1\alpha\}\}$
    end if
  end while
  return α
end procedure

Sometimes one can experience that the above algorithm only accepts very small step sizes. This is called the Marato's effect which can be solved by use of an aug-

mented Lagragian merit function, a second order correction or a non-monotone strategy. We have applied the last option which allows the line search to increase in value for some iterations. More specifically we apply a strategy that takes a full step if the current step is rounded to zero under the given precision. In the below we will represent rounding with a given precision, ε, as $\overset{\varepsilon}{\approx}$.

$$\alpha_k = \begin{cases} 1 & , \|\alpha_k \Delta x\|_2 \overset{\varepsilon}{\approx} 0 \\ \alpha_k & \text{otherwise} \end{cases}$$

4.5.1.4 Handling inconsistent linearization

At every iteration of our SQP method we linearize our constraints to create the local QP, 4.10. This linearization can lead to infeasible programs. We can solve this problem by allowing infeasible solutions as long the solution is the "most" feasible one. This formulation is called the elastic mode formulation and for 4.5 is given as

$$\begin{aligned} \min_{\Delta x, t} \quad & \tfrac{1}{2} \Delta x^T \nabla_{xx} \mathcal{L}(x_k, y_k, z_k) \Delta x + \nabla_x f(x_k)^T \Delta x + \mu \textstyle\sum_{i \in \mathcal{I}} t_i \\ \text{s.t.} \quad & \nabla_x c_i(x_k)^T \Delta x + c_i(x_k) \geq -t_i \qquad i \in \mathcal{I} \\ & t \geq 0 \end{aligned} \tag{4.12}$$

where μ is a non-negative constant. This new formulation allows for infeasible solutions but at a cost. We see from the objective that for every t we are infeasible we need to pay μt, and hence the reason for why μ must be non-negative.

4.5.1.5 A trust region extension

The last SQP method we will look into is a trust region based method. In these methods we define a so called region of trust for which we solve the sub problem. The sub problem is defined as follows:

$$\begin{aligned} \min_{\Delta x, t} \quad & \tfrac{1}{2} \Delta x^T \nabla_{xx} \mathcal{L}(x_k, y_k, z_k) \Delta x + \nabla_x f(x_k)^T \Delta x \\ \text{s.t.} \quad & \nabla_x c_i(x_k)^T \Delta x + c_i(x_k) \geq 0, \quad i \in \mathcal{I} \\ & \|\Delta x\|_\infty \leq \Delta_k \end{aligned} \tag{4.13}$$

One problem with the trust region is that we are not guaranteed feasibility (see figure 18.1 in [NW06]). To handle this, many methods exists and some are described in [NW06]. We have chosen the sequential l_1 quadratic programming (Sl_1QP) method which handles the infeasibility problem in the same way as 4.5.1.4. This gives the

new sub problem:

$$\begin{aligned} \min_{\Delta x,t} \quad & \tfrac{1}{2}\Delta x^T \nabla_{xx}\mathcal{L}(x_k, y_k, z_k)\Delta x + \nabla_x f\left(x_k\right)^T \Delta x + \mu \textstyle\sum_{i\in\mathcal{I}} t_i \\ \text{s.t.} \quad & \nabla_x c_i\left(x_k\right)^T \Delta x + c_i\left(x_k\right) \geq -t_i \qquad i \in \mathcal{I} \\ & t \geq 0 \\ & \|\Delta x\|_\infty \leq \Delta_k \end{aligned} \tag{4.14}$$

To determine if we accept or reject the current trust region we must define an estimate of the actual reduction and predicted reduction. To estimate the actual reduction we use the l_1 merit function.

$$\phi_1(x;\mu) = f(x) + \mu \sum_{i\in\mathcal{I}} \left[c_i(x)\right]^- \tag{4.15}$$

Where $[y]^- = max\{0, -y\}$. To estimate the predicted reduction we use the estimated approximation of the objective function $q_\mu(\Delta x)$

$$q_\mu(\Delta x) = f_k + \nabla f_k^T \Delta x + \frac{1}{2}\Delta x^T \nabla^2_{xx}\mathcal{L}_k \Delta x + \mu \sum_{i\in\mathcal{I}} \left[c_i\left(x_k\right) + \nabla c_i\left(x_k\right)^T \Delta x\right]^- \tag{4.16}$$

We then calculate the ratio between the predicted and actual reduction.

$$\rho_k = \frac{\text{ared}_k}{\text{pred}_k} = \frac{\phi_1\left(x_k, \mu\right) - \phi_1\left(x_k + \Delta x_k, \mu\right)}{q_\mu(0) - q_\mu\left(\Delta x_k\right)}. \tag{4.17}$$

If $\rho > 0$ we accept the step and otherwise we reject it. After having accepted or rejected the trust region we must adjust it for the next iteration. To do this we calculate $\gamma(\rho)$ by following [BJ19].

$$\gamma(\rho) = \min\left(\max\left((2\rho - 1)^3 + 1, 0.25\right), 2\right) \tag{4.18}$$

We then update the trust region by

$$\Delta_{k+1} = \begin{cases} \gamma(\rho)\Delta_k & , \rho > 0 \\ \gamma(\rho)\|\Delta x_k\|_\infty & \text{otherwise} \end{cases}$$

Lastly we have to update the penalty parameter μ. We will do this following a l_1 powell update.

$$\mu_{k+1} = \max(\frac{1}{2}(\mu_k + \|z\|_\infty), \|z\|_\infty) \tag{4.19}$$

4.5.2 Problem specific optimizations

The specific problem structure in 4.2 includes both general and variable bounds but no equality constraints. This specific structure does not matter for most of the SQP theory developed in the previous section except for the solver for the sub problem. We can utilize the structure to develop a specialized solver similar to the solvers seen in exercise 2 and 3. Our sub-problem has the form

$$\begin{aligned} \min_{\Delta x} \quad & \tfrac{1}{2}\Delta x^T B_k \Delta x + \nabla_x f(x_k)^T \Delta x \\ \text{s.t.} \quad & \begin{bmatrix} I \\ -I \\ \nabla_x c(x_k) \\ -\nabla_x c(x_k) \end{bmatrix} \Delta x \geq \begin{bmatrix} -x_k + l \\ x_k - u \\ -c(x_k) + c_l \\ c(x_k) - c_u \end{bmatrix} \end{aligned} \tag{4.20}$$

We have developed an interior point algorithm for which the code can be seen in appendix A.3.4.1. All the problem specific optimizations are similar to the ones seen in exercise 2 and 3 and hence we will only touch upon the most significant optimization here. Because we do not have any equality constraints the augmented system becomes

$$\begin{bmatrix} \bar{H} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -\bar{r}_L \\ 0 \end{bmatrix} \Rightarrow$$
$$\bar{H}\Delta x = -\bar{r}_L$$

This means we have a much smaller system of equations to solve. Furthermore $\bar{H}$ is positive definite most of the time and we saw from the bechmark test in figure 1.4, that the LDL factorization was much faster for a positive definite matrix than an indefinite matrix. In exercise 3 our system was always positive definite due to convexity of the space. We cannot guarantee convexity here and hence the matrix can be indefinite. Nevertheless we still expect a speed up of the system. We therefore again test our interior point algorithm against quadprog's interior algorithm, on IQPs without equality constraints and with bounds. The code can be found in appendix A.4.5.

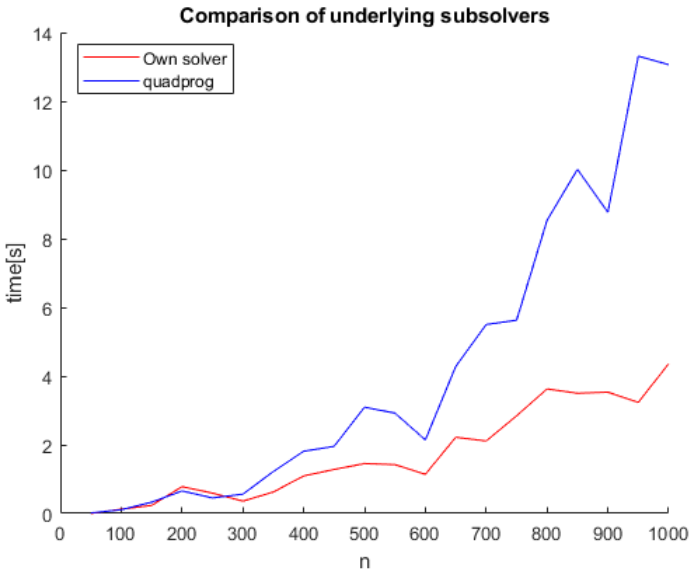


Figure 4.5: quadprog's interior point algorithm and our own specialized interior point algorithm tested on QP's for problems of the structure 4.2.

We see that the optimized solver is much faster than quadprog. We now just have to prove its correctness. To do this we implement a basic SQP algorithm with our solver and with quadprog. The SQP algorithm can be found in appendix A.3.4.2 and the code to test the correctness can be found in appendix A.4.5, under 'Test of interior point method'.

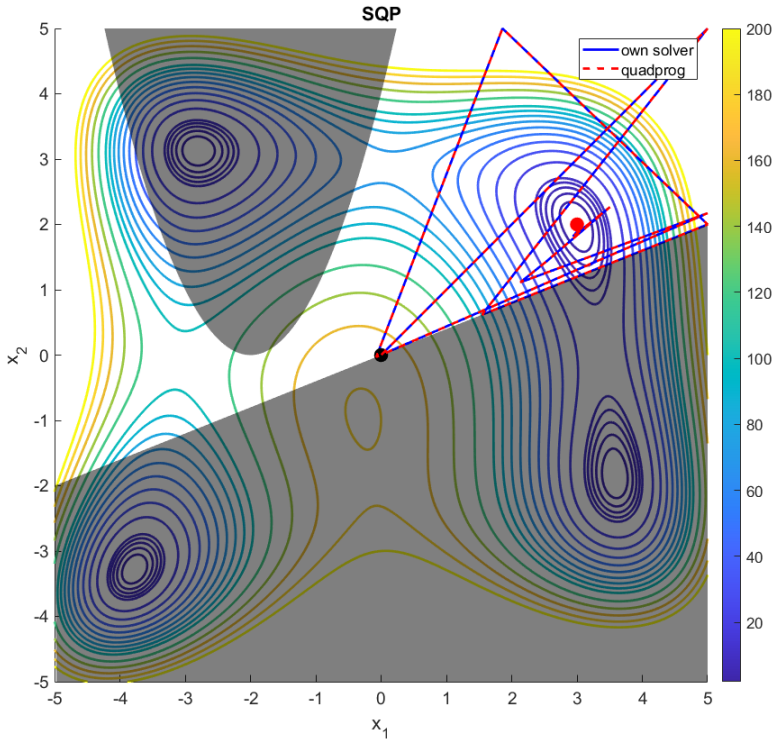


Figure 4.6: A plain vanilla SQP algorithm using quadprog and a specialized interior point algorithm.

We see in figure 4.6 and corresponding tables found in appendix A.2.1 that the solution paths are exactly the same using quadprog and our own solver. We hence conclude our sub solver is reliable.

4.5.3 Results

We have implemented a SQP solver-interface which includes 5 different algorithms. A basic SQP-BFGS method, a SQP-BFGS which can handle infeasible linearizations, a SQP-BFGS with line search, a SQP-BFGS with line search which also handles infeasible linearizations and a trust region based SQP. In the succeeding section we will go over results for the different algorithms. The solver interface and the associated algorithms can be found in following appendices.

Program	Appendix
The solver interface	A.4.6
The SQP-BFGS method	A.3.4.2
The SQP-BFGS method with infeasibility handling	A.3.4.3
The SQP-BFGS with line search	A.3.4.4
The SQP-BFGS with line search and infeasibility handling	A.3.4.5
The trust region SQP	A.3.4.6

Table 4.1: The different SQP files and their associated appendix

4.5.3.1 SQP-BFGS

In figure 4.2 we saw that 4.4 had 4 minima. We will therefore test the solvers from 4 initial points each close to one of the minima. SQP-BFGS is the first algorithm we test. This is a SQP algorithm where we update the hessian using a damped BFGS update but besides that it is as the frame work described in section 4.5.1.1. The code which is used to test the algorithm can be found in appendix A.4.5.

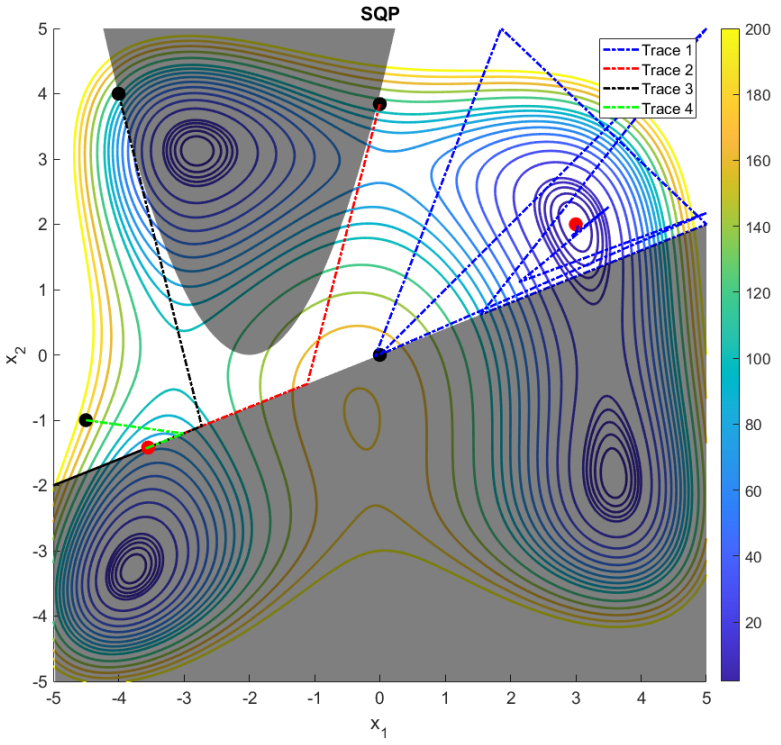


Figure 4.7: 4 paths of the SQP-BFGS algorithm found in A.3.4.2

We see that the two minima on the boundary of the non linear constraint is not possible to reach for the basic SQP-BFGS algorithm. Furthermore it is particular apparent for trace 1 that there is no line search. It is converging very slowly and from table 4.2 we can see it takes 18 iterations to converge. We will therefore extend the algorithm with a line search in the next section.

x_0	$(0,0)$	$(0,3.842)$	$(-4,4)$	$(-4.5,-1)$
Iterations	18	8	7	5
Function calls	38	18	16	12

Table 4.2: SQP-BFGS statistics

$x_0 = (0,0)'$	1	2	3	4	5	6	7	8
x_1	5	1.544	2.201	5	1.858	-0.075	1.591	5
x_2	5	0.618	0.880	2	5	-0.030	0.695	2.173
$x_0 = (0,0)'$	9	10	11	12	13	14	15	16
x_1	2.131	2.533	3.504	2.939	3.017	3.030	3.022	3.003
x_2	1.113	1.434	2.266	1.815	1.911	1.955	1.978	2.001
$x_0 = (0,0)'$	17	18						
x_1	3	3						
x_2	2.001	2						

Table 4.3: SQP trace 1

$x_0 = (0,3.8420)'$	1	2	3	4	5	6	7	8
x_1	-1.111	-2.007	-5	-3.281	-3.448	-3.563	-3.548	-3.549
x_2	-0.444	-0.803	-2	-1.312	-1.379	-1.425	-1.419	-1.419

Table 4.4: SQP trace 2

$x_0 = (-4,4)'$	1	2	3	4	5	6	7
x_1	2.727	-5	-3.281	-3.430	-3.566	-3.548	-3.549
x_2	-1.091	-2	-1.312	-1.372	-1.426	-1.419	-1.419

Table 4.5: SQP trace 3

$x_0 = (-4.5, -1)'$	1	2	3	4	5
x_1	-3.009	-3.584	-3.537	-3.548	-3.549
x_2	-1.204	-1.434	-1.415	-1.419	-1.419

Table 4.6: SQP trace 4

4.5.3.2 SQP with line search results

We now extend the SQP-BFGS algorithm with the line search algorithm described in section 4.5.1.3. This way we should be able to catch the 4 minimas of Himmelblau's test problem. To be able to compare the SQP-BFGS without a line search extension with the line search extension we will test on the same 4 initial points. The code which is used to test the algorithm can be found in appendix A.4.5.

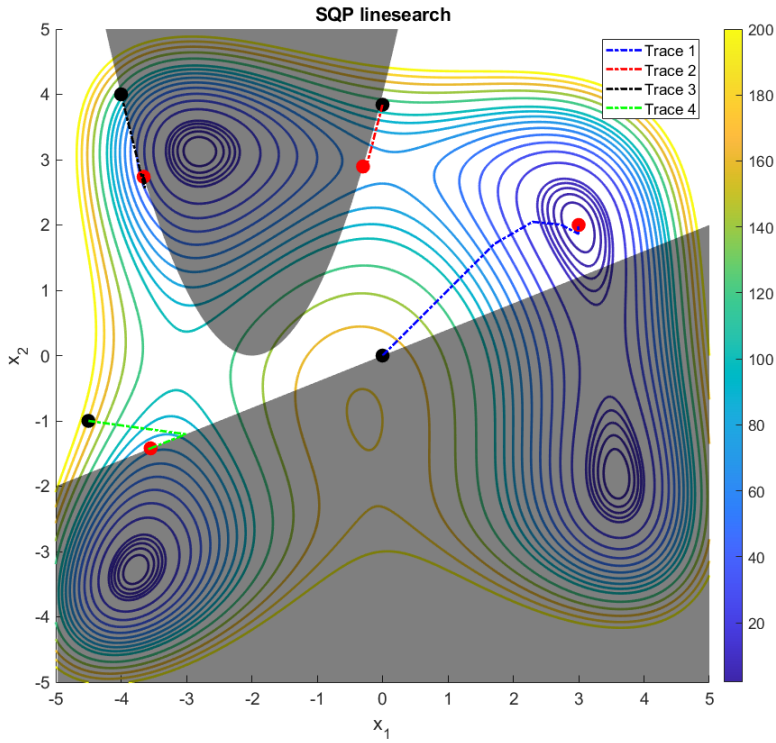


Figure 4.8: 4 paths of the SQP-BFGS algorithm with line search found in A.3.4.4

We see much more controlled traces compared to figure 4.7. They all converge to the closest minimum and in fewer iterations. We see from table 4.7 that even though we use fewer iterations we use a lot more function calls because every iteration of the

line search algorithm uses 2 function calls. We will now look into the effect of the non monotone strategy which is used by the line search algorithm.

x_0	$(0,0)$	$(0,3.842)$	$(-4,4)$	$(-4.5,-1)$
Iterations	9	5	6	5
Function calls	61	31	36	29

Table 4.7: SQP-BFGS with line search statistics

$x_0 = (0,0)'$	1	2	3	4	5	6	7	8
x_1	0.500	1.141	1.709	2.297	2.734	2.997	2.993	3.001
x_2	0.500	1.141	1.709	2.048	2.003	1.863	1.982	2.002
$x_0 = (0,0)'$	9							
x_1	3							
x_2	2							

Table 4.8: SQP with line search, trace 1

$x_0 = (0,3.8420)'$	1	2	3	4	5
x_1	-0.273	-0.308	-0.301	-0.298	-0.298
x_2	2.787	2.860	2.886	2.896	2.896

Table 4.9: SQP with line search, trace 2

$x_0 = (-4,4)'$	1	2	3	4	5	6
x_1	-3.641	-3.630	-3.667	-3.654	-3.655	-3.655
x_2	2.565	2.657	2.779	2.736	2.738	2.738

Table 4.10: SQP with line search, trace 3

$x_0 = (-4.5,-1)'$	1	2	3	4	5
x_1	-3.009	-3.584	-3.537	-3.548	-3.549
x_2	-1.204	-1.434	-1.415	-1.419	-1.419

Table 4.11: SQP with line search, trace 4

4.5.3.3 Non monotone strategy

We see from table 4.12 that by using the non monotone strategy described in section 4.5.1.3 we are able to reduce the number of function calls by over 150 calls in the

example of trace 4. From table 4.13 we see that if we do not apply the non monotone strategy the line search algorithm just keeps taking very small steps making no progress. The non monotone strategy then forces it to take a full step in the cases where the found step is below the given precision.

x_0	$(-4.5, -1)$	$(-4.5, -1)$
Non monotone strategy?	Yes	No
Iterations	5	14
Function calls	29	186

Table 4.12: Summary of the non monotone strategy's effect for trace 4

$x_0 = (-4.5, -1)'$	1	2	3	4	5	6	7	8
With NMS, step size	1.505	0.619	0.050	0.012	2e-04	NA	NA	NA
Without NMS, step size	1.505	0.619	0.050	0.012	7e-05	6e-05	5e-11	5e-12
$x_0 = (-4.5, -1)'$	9	10	11	12	13	14	15	16
With NMS, step size	NA	NA	NA	NA	NA	NA		
Without NMS, step size	5e-13	5e-09	5e-13	5e-16	6e-15	5e-05		

Table 4.13: Non monotone strategy(NMS), step length ($\|\alpha\Delta x\|_2$). NA means we have converged.

4.5.3.4 Handling infeasible linearization

If one choose the initial point to $x_0 = (-9, 6)$ the sub problem becomes infeasible due to the linearization. We have therefore implemented two algorithms with the infeasibility handling described in section 4.5.1.4. The algorithms are based on the SQP-BFGS with and without line search. The implementations can be found in appendix A.3.4.3 and A.3.4.5 and we test the SQP-BFGS algorithm with line search and infeasibility handling in appendix A.4.5. The result is plotted in figure 4.9.

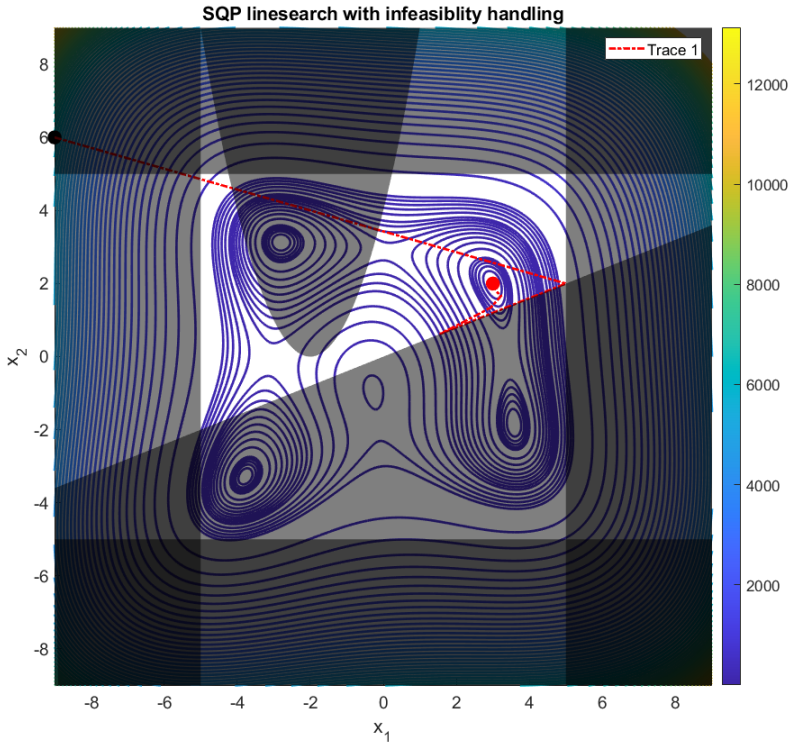


Figure 4.9: The SQP-BFGS algorithm with line search and infeasiblity handling test with initial point $(-9, 6)$. The algorithm can be found in A.3.4.5

4.5.3.5 SQP with trust region

Lastly we have the trust region SQP algorithm. We have implemented an algorithm based on section 4.5.1.5 where the code is shown in appendix A.3.4.6. For the tests we have used an initial trust region of 0.5 which leads to fast convergence to the closets minima, as it can be seen from figure 4.10. From table 4.16 we read that it converges in 6 iterations and with 32 function calls for all initial points except $x_0 = (0, 0)$. Here it uses 12 iterations with 58 function calls. We look a bit closer on trace 1 by assessing table 4.15. We see from this table that due to the small initial trust region the algorithm enlarges the trust region the first 2 iteration where in the third iteration the step is rejected and the trust region reduced to 0.5. The algorithm then again enlarges the trust region for two iteration, then reject the 6th iteration and for subsequent iterations all trust regions are accepted and the algorithm converges to the minimum.

If we are to compare the trust region based SQP algorithm to the other methods we converge must more controlled and faster than the SQP-BFGS without line search.

We though do a bit poorer job than the line search based SQP-BFGS algorithm. The trust region based SQP methods can though handle singular Jacobians and Hessians compared to the line search based methods.

The trust region based SQP can also converge from all start points as the previous algorithms with infeasibility handling.

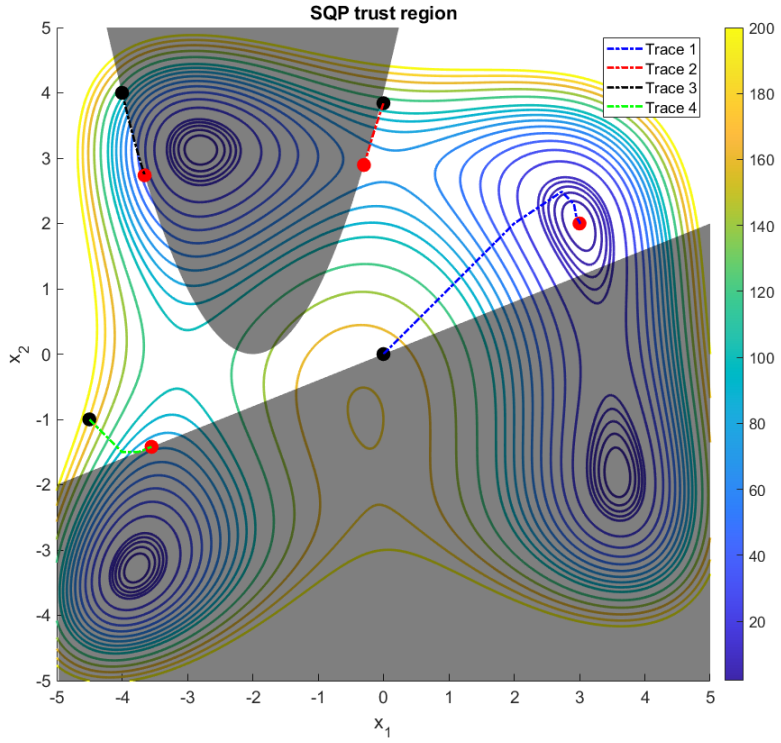


Figure 4.10: 4 paths of the trust region SQP which can be found in A.3.4.6

x_0	$(0,0)$	$(0,3.842)$	$(-4,4)$	$(-4.5,-1)$
Iterations	12	6	6	6
Function calls	58	32	32	32

Table 4.14: Trust region SQP, statistics

$x_0 = (0,0)'$	0	1	2	3	4	5	6	7
Δ_k	0.500	1	2	0.500	0.727	0.756	0.189	0.203
Accept/Reject	NA	1	1	0	1	1	0	1
$x_0 = (0,0)'$	8	9	10	11	12	13	14	15
Δ_k	0.406	0.813	1.173	2.347	4.685			
Accept/Reject	1	1	1	1	1			

Table 4.15: Trust regions for trace 1

$x_0 = (0,0)'$	1	2	3	4	5	6	7	8
x_1	0.5000	1.5000	1.5000	2	2.7270	2.7270	2.9160	2.9530
x_2	0.5000	1.5000	1.5000	2	2.4860	2.4860	2.2970	2.0940
$x_0 = (0,0)'$	9	10	11	12	13	14	15	16
x_1	3.0070	2.9980	3	3				
x_2	2.0150	1.9990	2	2				

Table 4.16: SQP with trust region, trace 1

$x_0 = (0, 3.8420)'$	1	2	3	4	5	6
x_1	-0.165	-0.257	-0.291	-0.298	-0.298	-0.298
x_2	3.342	3.030	2.919	2.897	2.896	2.896

Table 4.17: SQP with trust region, trace 2

$x_0 = (-4,4)'$	1	2	3	4	5	6
x_1	-3.875	-3.743	-3.680	-3.658	-3.655	-3.655
x_2	3.500	3.019	2.819	2.749	2.738	2.738

Table 4.18: SQP with trust region, trace 3

$x_0 = (-4.5,-11)'$	1	2	3	4	5	6
x_1	-4	-3.746	-3.587	-3.552	-3.549	-3.549
x_2	-1.500	-1.498	-1.435	-1.421	-1.419	-1.419

Table 4.19: SQP with trust region, trace 4

CHAPTER 5

Markowitz Portfolio Optimization

In this exercise we will use Markowitz Portfolio Optimization to optimize a portfolio by diversifying an investment into several securities. We consider a 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

Before we are able to optimize the portfolio we need to turn the given financial market into a solvable problem. From example 16.1 in [NW06] we know the financial market can be turned into a QP. We have that the returns and its related statistical first and second order moments, given as:

$$R = \sum_{i=1}^{n} x_i r_i, \quad \forall x_i \leq 0$$

$$\mathrm{E}[R] = \mathrm{E}[\sum_{i=1}^{n} x_i r_i] = \sum_{i=1}^{n} x_i \,\mathrm{E}[r_i] = x^T \mu$$

$$\mathrm{Var}[R] = \mathrm{E}[R - \mathrm{E}[R]^2] = x^T \Sigma x$$

where the covariance matrix, Σ is a $n \times n$ symmetric matrix defined by

$$\Sigma_{ij} = \rho_{ij} \sigma_i \sigma_j$$

Ideally we want to find a portfolio where $x^T \mu$ is large and $x^T \Sigma x$ is small, i.e. high expected return and low expected risk. We can combine these two into a maximization

problem which do not allow for shorting.

$$\max_x x^T\mu - \kappa x^T\Sigma x, \quad \text{subject to } \sum_{i=1}^{n} x_i = 1, x \geq 0 \tag{5.1}$$

where κ is a measure of risk tolerance. Or equivalently as a minimization problem.

$$\min_x \kappa x^T\Sigma x - x^T\mu, \quad \text{subject to } \sum_{i=1}^{n} x_i = 1, x \geq 0 \tag{5.2}$$

5.2 Exercise 5.2

For our given financial market we do not really have a maximum or minimum possible return due to its stochastic nature. Depending on ones assumptions, the maximum and minimum would differ. For example if one assumed an underlying gaussian distribution, a sensible set of extremes would be the minimum and maximum expected returns $\pm$ two times their standard deviation.

This example is probably on the simpler side of possible assumptions and to avoid this extra problem we will assume the given expected returns as deterministic. Hence the maximum possible return is 18.68 and the minimum possible return is 8.50.

5.3 Exercise 5.3

We now want to solve for the optimal portfolio giving 12 in return. To do this we reformulate 5.2 with the expected return as a constraint instead of an objective.

$$\begin{aligned} \min_x \quad & x^T\Sigma x \\ \text{s.t.} \quad & \sum_{i=1}^{5} x_i = 1, \\ & x \geq 0, \\ & \sum_{i=1}^{5} x_i\mu_i = 12 \end{aligned} \tag{5.3}$$

We can now solve the problem using quadprog and obtain the optimal portfolio and its associated return and risk. The Matlab code can be found in appendix A.4.7.

$$x = \begin{bmatrix} 0.0000 \\ 0.4765 \\ 0.2551 \\ 0.1234 \\ 0.1449 \end{bmatrix} \quad E[R] = 12 \quad Var[R] = 0.7654$$

5.4 Exercise 5.4

Peoples willingness to take risk differ and we therefore do not have one globally optimal answer when optimizing portfolios. However we can still talk about what is called pareto optimal. It defines a globally optimal set where every member of the set defines a solution which fulfills that we cannot make any variable of the solution better without at least making another worse. In our example it corresponds to not being able to increase return for a given portfolio without also increasing the risk.

This set of portfolios is called the efficient frontier. We can calculate it by solving 5.3, given in exercise 5.3, with R taking values in the interval $[min(\mu), max(\mu)]$. We have used equidistant steps of size 0.01 giving 1019 optimization problems. The Matlab code can be found in appendix A.4.7.

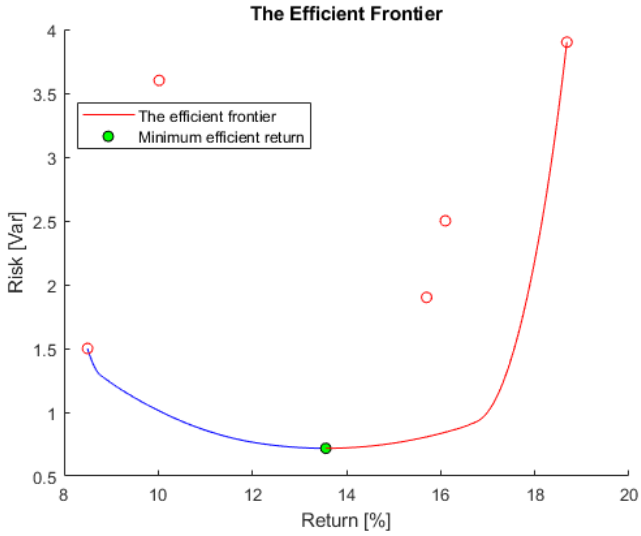


Figure 5.1: The efficient frontier

We see that the red line defines all portfolios for which we cannot increase return without also increasing risk and vice versa. We also see that we actually are not pareto optimal with returns under 13.56 indicated by the green dot. This effect is called hedging. When choosing assets such that they are negatively correlated gives us a more robust portfolio which is so called hedged. We find the portfolio with the minimum risk to be:

$$x = \begin{bmatrix} 0.0870 \\ 0.3076 \\ 0.3016 \\ 0.1000 \\ 0.2039 \end{bmatrix} \quad E[R] = 13.56 \quad Var[R] = 3.9$$

We see in figure 5.2 how the different portfolios are composed as a function of return.

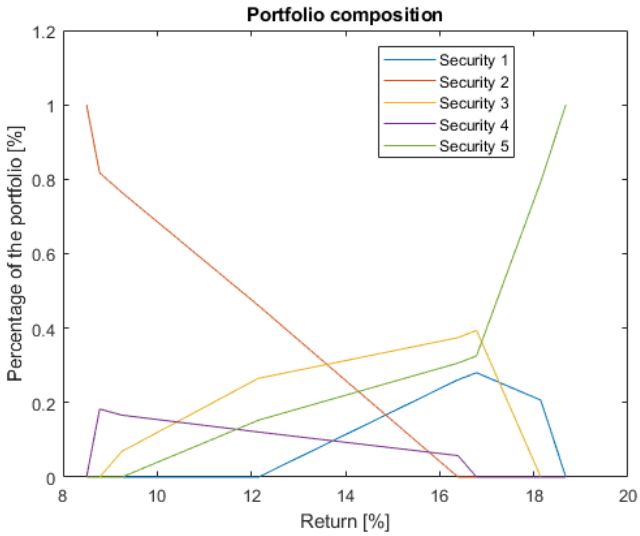


Figure 5.2: Portfolio compositions as a function of return

5.5 Exercise 5.5-5.7

It is a rarity that people can come up with a minimum return for which we can compose a pareto optimal portfolio for. It is much more likely they can define some degree of risk willingness. We can define this risk willingness to be a number $\alpha \in [0, 1]$ for which we can setup a bi-criterion optimization problem that attach risk α importance and return $1 - \alpha$ importance.

$$\min_x \alpha x^T \Sigma x - (1 - \alpha) x^T \mu, \quad \text{subject to } \sum_{i=1}^{n} x_i = 1, x \geq 0 \tag{5.4}$$

In the formulation 5.4 we do not allow for shorting because $x \geq 0$. If we dropped this inequality constrained we would obtain an EQP which allowed for shorting.

$$\min_x \alpha x^T \Sigma x - (1-\alpha) x^T \mu, \quad \text{subject to } \sum_{i=1}^{n} x_i = 1 \tag{5.5}$$

We will now calculate the risk-return curves for 5.4 and 5.5 using the algorithms we derived in exercise 1 and 2. The code is given in appendix A.4.7.

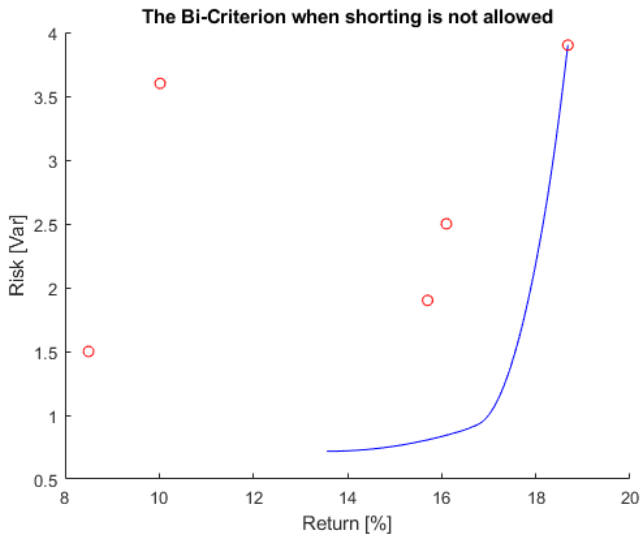


Figure 5.3: The risk-return curve for 5.4

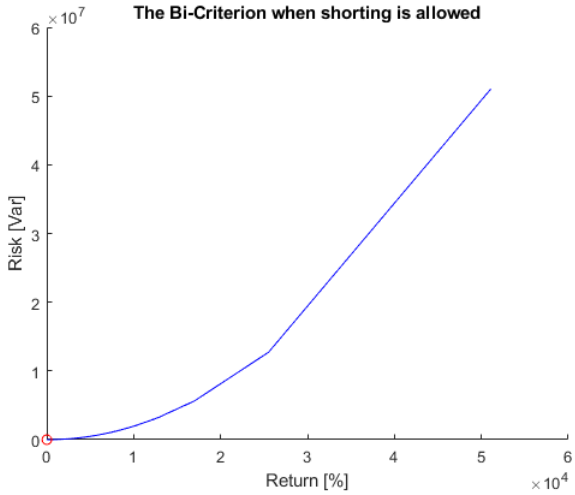


Figure 5.4: The risk-return curve for 5.5

We see in figure 5.4 that when we allow for shorting there is effectively no limit on potential returns. That said we also observe they come with an immense amount of risk attached. This level of risk is way beyond any sensible mindset so one should look into reformulating 5.5 if one is interested in shorting. On the other hand we see in figure 5.3 that when we disallow shorting the picture looks like figure 5.1.

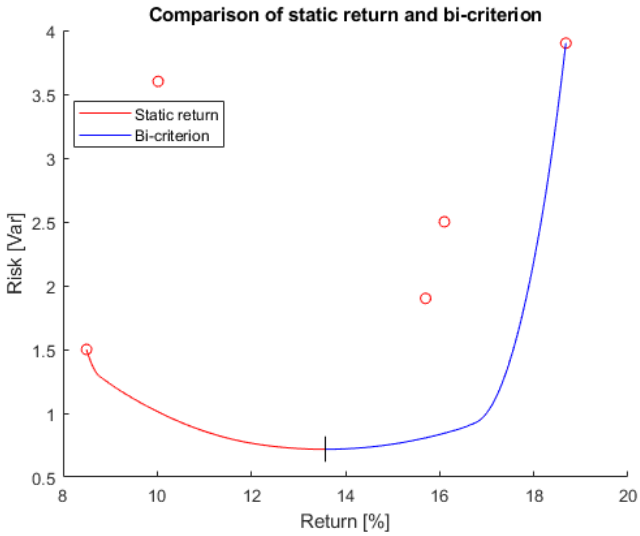


Figure 5.5: The risk-return curves for 5.5 and 5.3

In figure 5.5 we see the risk-return cruves for 5.5 and 5.3. We observe that when optimizing over both return and risk we obtain only pareto optimal solutions.

To check if our own solvers produces the correct answers we compare the found portfolios to quadprog and CVX. In figure 5.6 we see that log10 mean squared error when shorting is disallowed. We see that there is a very small difference between quadprog and our own solver but between our own solver and CVX the error is up to 10^{-5} which is still acceptable.

In figure 5.7 the log10 mean squared error, for when shorting is allowed, is plotted. We see that the error between quadprog and our own solver is very small. For CVX on the other hand to deviations are up to 10^{-2} but the same holds between quadprog and CVX. We therefore conclude that it is probably not our solver which is performing poorly but rather CVX even though it sounds weird.

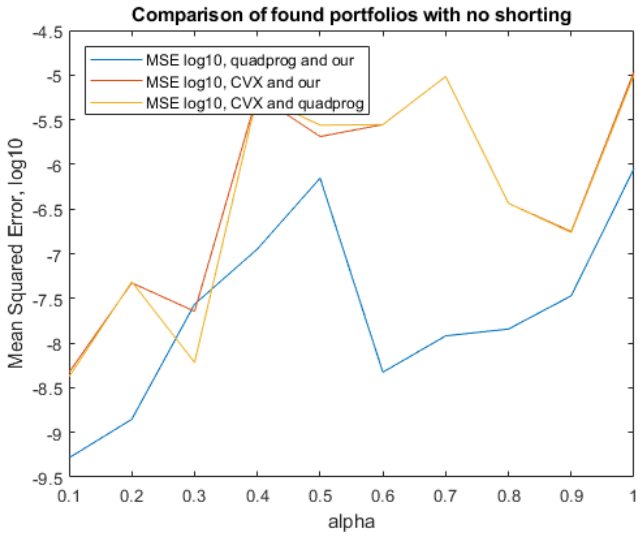


Figure 5.6: The log10 mean squared error for 5.4

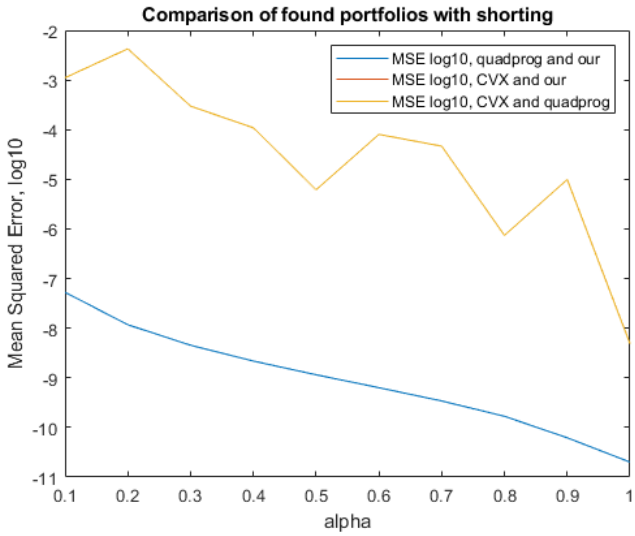


Figure 5.7: The log10 mean squared error for 5.5

5.6 Exercise 5.8-5.9

We now introduce a risk free asset with no return. This could for example correspond to having money in the bank with a deposit rate of zero. This gives us the new financial market:

Security	Covariance						Return
1	2.30	0.93	0.62	0.74	-0.23	0	15.10
2	0.93	1.40	0.22	0.56	0.26	0	12.50
3	0.62	0.22	1.80	0.78	-0.27	0	14.70
4	0.74	0.56	0.78	3.40	-0.56	0	9.02
5	-0.23	0.26	-0.27	-0.56	2.60	0	17.68
6	0	0	0	0	0	0	0.0

We can utilize this new financial asset to lower risk in our portfolios. To see this we recalculate the efficient frontier with the new risk free asset. The code can be found in appendix A.4.7.

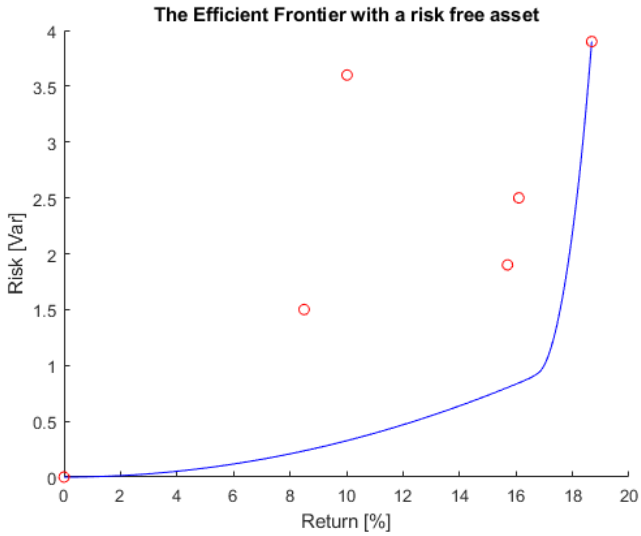


Figure 5.8: The efficient frontier with a risk free asset

We see that now the pareto optimal set extends all the way to zero risk and zero return, corresponding to putting all your money in the bank. We now try to plot the new efficient frontier with the efficient frontier without a risk free asset.

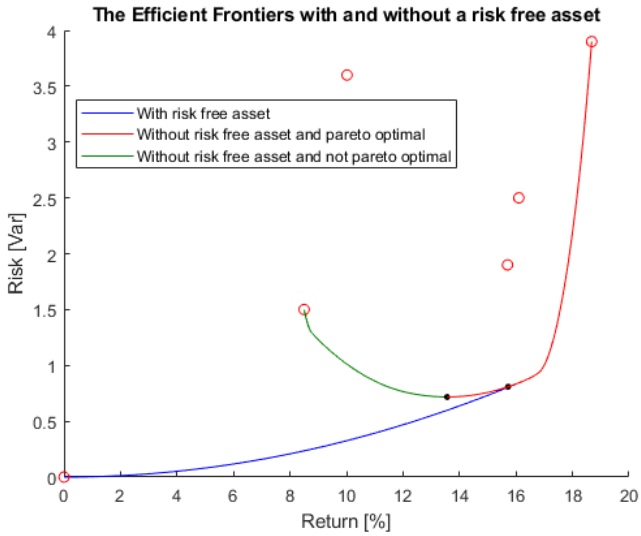


Figure 5.9: The efficient frontiers for a financial market with and without a risk free asset

We see from figure 5.9 that we can actually obtain a higher return with a lower risk meaning that, the portfolios corresponding to the line segment in between the two black dots, are no longer pareto optimal in the new financial market. The intuition behind this is that in the old market we had to put all our money into the given 5 assets even though they did not hedge each other perfectly. Now we can utilize the given assets correlations perfectly and the money we have left can be put into the bank. This can also be seen in 5.10 where the normal securities increases linearly in a static ratio with the rest taken up by the risk free asset. This can continue until the return demand gets so high that the hedged portfolio cannot satisfy the return demand. This correspond to the point around 17 where the risk free asset is 0 and the efficient frontier begins to increase in risk very rapidly.

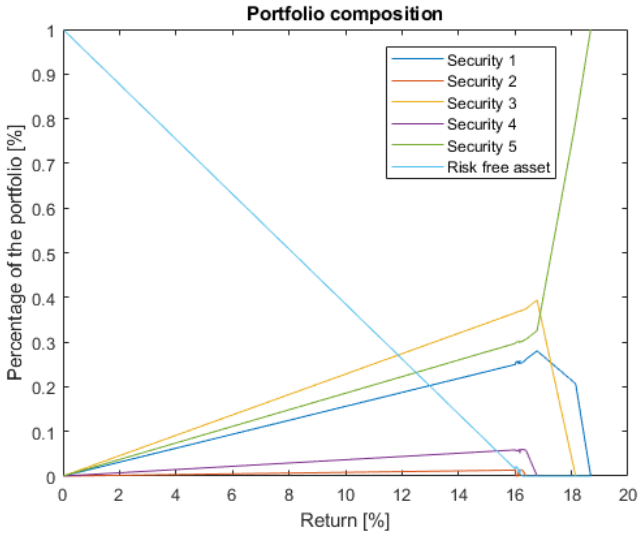


Figure 5.10: Portfolio compositions as a function of return with a risk free asset

5.7 Exercise 5.10-5.11

In the given assignment we are asked to calculate the optimal portfolio with an expected return of 14, in the new financial market.

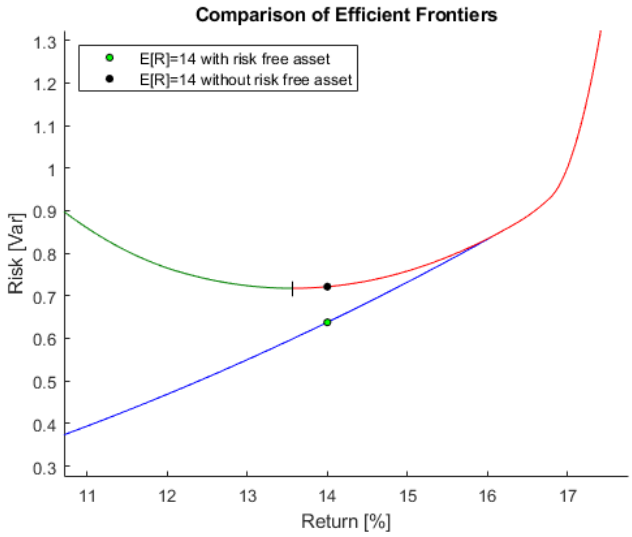


Figure 5.11: Two portfolios giving 14% in expected return. The black dot is in a financial market without a risk free asset and the green dot in a market with a risk free asset.

We see that as described in exercise 5.8-5.9 the risk free asset reduces the risk of the portfolio compared to the market without a risk free asset. In the new market the optimal portfolio with an expected return of 14 is given below.

$$x_{riskfree} = \begin{bmatrix} 0.2185 \\ 0.0116 \\ 0.3196 \\ 0.0512 \\ 0.2598 \\ 0.1393 \end{bmatrix} \quad E[R] = 14.0000 \quad Var[R] = 0.6377$$

In the old market without a risk free asset the optimal portfolio with an expected return of 14 had an associated risk of 0.7214. We hence see that the risk free asset in this case have brought the expected risk down with almost 0.1.

APPENDIX A

Appendix

A.1 Extra theory

A.1.1 Null Space derivation

From [Bro] we get that the null space method applies the constraint elimination idea the following way:

We QR-factorize A

$$A = QR = \begin{bmatrix} Q_{range} & Q_{null} \end{bmatrix} \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}$$

meaning

$$\begin{aligned} A^T x = b \iff \\ R^T Q^T x = \hat{R}^T Q_{range}^T x = b \end{aligned}$$

We can now write x as $x = Q_{range}u + Q_{null}v$ resulting in

$$\begin{aligned} \hat{R}^T Q_{range}^T x &= \hat{R}^T Q_{range}^T (Q_{range}u + Q_{null}v) \\ &= \hat{R}^T u = b \iff \\ u &= \hat{R}^{-T} b \end{aligned} \tag{A.1}$$

We see from A.1 that u is fixed but v can vary freely

$$\begin{aligned} x &= Q_{range}u + Q_{null}v \\ &= Q_{range}\hat{R}^{-T}b + Q_{null}v \end{aligned}$$

We define $\hat{x} = Q_{range}(\hat{R}^{-1})^T b$ and rewrite our objective function.

$$\begin{aligned} f(x) &= \frac{1}{2}x^T H x + g^T x \\ &= \frac{1}{2}(\hat{x} + Q_{null}v)^T H(\hat{x} + Q_{null}v) + g^T(\hat{x} + Q_{null}v) \\ &= \frac{1}{2}v^T(Q_{null}^T H Q_{null})v + (\hat{x}^T H Q_{null} + g^T Q_{null})v + \frac{1}{2}\hat{x}^T H \hat{x} + g^T \hat{x} \end{aligned}$$

This is an unconstrained problem in v so we can just solve for v.

$$v^* = -(Q_{null}^T H Q_{null})^{-1}(\hat{x} H Q_{null} + g^T Q_{null})$$

giving

$$x^* = \hat{x} + Q_{null} v^* = Q_{range} \hat{R}^{-T} b + Q_{null} v^*$$

To obtain λ^* we can use equation 16.20 on page 457 in [NW06].

$$\begin{aligned} \lambda^* &= (Q_{range}^T A)^{-1} Q_{range}^T (g + H x^*) \\ &= \hat{R}^{-1} Q_{range}^T (g + H x^*) \end{aligned}$$

A.1.2 Range Space Derivation

We write the KKT system as two equations

$$Hx - A\lambda = -g \tag{A.2}$$

$$-A^T x = -b \implies$$

$$A^T x = b \tag{A.3}$$

We then multiply A.2 with $A^T H^{-1}$

$$A^T x - A^T H^{-1} A\lambda = -A^T H^{-1} g \implies \tag{A.4}$$

$$A^T x = A^T H^{-1} A\lambda - A^T H^{-1} g \implies$$

$$x^* = H^{-1} A\lambda - H^{-1} g \tag{A.5}$$

We then subtract A.4 from A.3

$$A^T x - (A^T x - A^T H^{-1} A\lambda) = b + A^T H^{-1} g \implies$$

$$A^T H^{-1} A\lambda = b + A^T H^{-1} g \implies$$

$$\lambda^* = (A^T H^{-1} A)^{-1}(b + A^T H^{-1} g) \tag{A.6}$$

A.2 Tables

A.2.1 Comparison of quadprog and interior point method for 4.20

$x_0 = (0,0)'$	1	2	3	4	5	6	7	8
x_1	5	1.544	2.201	5	1.858	-0.075	1.591	5
x_2	5	0.618	0.880	2	5	-0.030	0.695	2.173
$x_0 = (0,0)'$	9	10	11	12	13	14	15	16
x_1	2.131	2.533	3.504	2.939	3.017	3.030	3.022	3.003
x_2	1.113	1.434	2.266	1.815	1.911	1.955	1.978	2.001
$x_0 = (0,0)'$	17	18	19	20	21	22	23	24
x_1	3	3						
x_2	2.001	2						

Table A.1: SQP own sub solver

$x_0 = (0,0)'$	1	2	3	4	5	6	7	8
x_1	5	1.544	2.201	5	1.858	-0.075	1.591	5
x_2	5	0.618	0.880	2	5	-0.030	0.695	2.173
$x_0 = (0,0)'$	9	10	11	12	13	14	15	16
x_1	2.131	2.533	3.504	2.939	3.017	3.030	3.022	3.003
x_2	1.1130	1.434	2.266	1.815	1.911	1.955	1.978	2.001
$x_0 = (0,0)'$	17	18	19	20	21	22	23	24
x_1	3	3						
x_2	2.001	2						

Table A.2: SQP quadprog

A.3 Algorithms

A.3.1 Algorithms for exercise 1

A.3.1.1 Dense LU solver

```
1 function [x, lambda] = EqualityQPSolverLUdense(H, g, A, b)
2 % EqualityQPSolverLUdense  dense LU solver
3 %
4 %          min  x'*H*x+g'x
5 %           x
6 %          s.t. A x  = b      (Lagrange multiplier: lambda)
```

```
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLUdense(H,g,A,b)
10 %
11 %          x                : Solution
12 %          lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19     % Create a KKT system
20     KKT = get_KKT(H,g,A,b);
21
22     % Fatorize the KKT matrix
23     [L,U,p] = lu(KKT,'vector');
24
25     % Solve for x and lambda
26     rhs = -[g;b];
27     solution(p) = U \ (L \ ( rhs(p)));
28     x = solution(1:size(H,1));
29     lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
30 end
```

Listing A.1: A dense LU solver

A.3.1.2 Sparse LU solver

```
1  function [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
2  % EqualityQPSolverLUsparse   Sparse LU solver
3  %
4  %          min  x'*H*x+g'x
5  %           x
6  %          s.t. A x  = b      (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
10 %
11 %          x                : Solution
12 %          lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19     % Create a sparse KKT system
20     KKT = get_KKT_sparse(H,g,A,b);
21
22     % Fatorize the KKT matrix
23     [L,U,p] = lu(KKT,'vector');
```

```
24
25      % Solve for x and lambda
26      rhs = [g;b];
27      solution(p) = U \ (L \ ( -rhs(p)));
28      x = solution(1:size(H,1));
29      lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
30  end
```

Listing A.2: A sparse LU solver

A.3.1.3 Dense LDL solver

```
1  function [x, lambda] = EqualityQPSolverLDLdense(H,g,A,b)
2  % EqualityQPSolverLDLdense    Dense LDL solver
3  %
4  %           min  x'*H*x+g'x
5  %            x
6  %           s.t. A x  = b       (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda] = EqualityQPSolverLDLdense(H,g,A,b)
10 %
11 %          x                : Solution
12 %          lambda           : Lagrange multipier
13
14 % Created: 06.06.2021
15 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
16 %           IMM, Technical University of Denmark
17
18 %%
19      % Create a KKT system
20      KKT = get_KKT(H,g,A,b);
21
22      % Fatorize the KKT matrix
23      [L,D,p] = ldl(KKT,'lower','vector');
24
25      % Solve for x and lambda
26      rhs =  -[g;b];
27      solution(p) = L' \ (D \ (L \rhs(p)));
28      x = solution(1:size(H,1));
29      lambda = solution(size(H,1)+1:size(H,1)+size(b,1));
```

Listing A.3: A dense LDL solver

A.3.1.4 Sparse LDL solver

```
1  function [x, lambda] = EqualityQPSolverLDLsparse(H, g, A, b)
2  % EqualityQPSolverLDLsparse    Sparse LDL solver
```

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

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

%%
    % Create a sparse KKT system
    KKT = get_KKT_sparse(H,g,A,b);

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

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

Listing A.4: A sparse LDL solver

A.3.1.5 Null Space solver

```
function [x,lambda,time_N] = EqualityQPSolverNullSpace(H,g,A,b)
% EqualityQPSolverNullSpace   Null Space solver
%
%           min  x'*H*x+g'x
%            x
%           s.t. A x  = b       (Lagrange multiplier: lambda)
%
%
% Syntax: [x,lambda,time_N] = EqualityQPSolverNullSpace(H,g,A,b)
%
%          x                  : Solution
%          lambda             : Lagrange multipier
%          time_N             : Time spend on qr factorization

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

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

```
21
22      % Factorize A
23      start = cputime;
24      [Q,R] = qr(A,'vector');
25      time_N = cputime-start;
26
27      % Solve for x and lambda
28      Qrange = Q(:,1:m);
29      Qnull = Q(:,m+1:n);
30      R = R(1:m,1:m);
31      Y = (R'\b);
32      Qnt = Qnull';
33      lpre = Qnt*H*Qnull;
34      L = chol(lpre);
35      mu=L'\(-Qnt*(H*Qrange*Y+g));
36      Z=L\mu;
37      x = Qrange*Y+Qnull*Z;
38      lambda = R\Qrange'*(g+H*x);
```

Listing A.5: A Null Space solver

A.3.1.6 Range Space solver

```
1  function [x, lambda, time_R] = EqualityQPSolverRangeSpace(H,g,A,b)
2  % EqualityQPSolverRangeSpace   Range Space solver
3  %
4  %           min  x'*H*x+g'x
5  %            x
6  %           s.t. A x  = b      (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda, time_R] = EqualityQPSolverRangeSpace(H,g,A,b)
10 %
11 %          x                : Solution
12 %          lambda           : Lagrange multipier
13 %          time_R           : Time spend on cholesky factorization
14
15 % Created: 06.06.2021
16 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
17 %           IMM, Technical University of Denmark
18
19 %%
20      % Factorize H
21      start = cputime;
22      R=chol(H);
23      time_R = cputime-start;
24
25      % Solve for x and lambda
26      mu=R'\g;
27      Hg=R\mu;
28      mu=R'\A;
29      HA=R\mu;
```

```
30      lambda = (A'*HA)\(b+A'*Hg);
31      x = HA*lambda-Hg;
```

Listing A.6: A Range Space solver

A.3.2 Algorithms for exercise 2

A.3.2.1 Interior point method a box constrained EQP

```
1  function [x,y,z,s, iter, ldltime] = ...
       primalDualInteriorMethod_box(H,g,A,b,l,u,x0,y0,z0,s0)
2  % primalDualInteriorMethod_box  An interior point solver based on ...
       Mehrota's predictor-corrector
3  %                                primal-dual interior point ...
       algorithm. It takes
4  %                                problems of the form
5  %
6  %           min    x'Hx+g'x
7  %            x
8  %           s.t     Ax  = b
9  %               u>=  x >= l
10 %
11 %
12 % Syntax: [x,y,z,s, iter, ldltime] = ...
       primalDualInteriorMethod_box(H,g,A,b,l,u,x0,y0,z0,s0)
13 %
14 %        x             : Solution
15 %        y             : Equality lagrange multipliers
16 %        z             : Inequality lagrange multipliers
17 %        s             : Slack variables
18 %        iter          : Iterations used
19 %        ldltime       : Time used on ldl factorization
20
21 % Created: 06.06.2021
22 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
23 %           IMM, Technical University of Denmark
24
25 %%
26     % Sets constants for the algorithm
27     mIn = length(x0);
28     m = length(y0);
29     epsilon = 0.000001;
30     ldltime = 0;
31     max_iter = 100;
32     eta = 0.995;
33     iter = 0;
34
35     % Initial values
36     x = x0;
37     y = y0;
38     z = z0;
```

```
39      s = s0;
40
41      % Makes sure non of the following matrix operations are singular
42      while(any(s==0))
43          x = x+0.000001;
44          s = [x-l;-x+u];
45      end
46
47      %Initilize constraint specific slacks and lagrange multipliers
48      e = ones(mIn*2,1);
49      sl = s(1:mIn);
50      su = s(mIn+1:mIn*2);
51      zl = z(1:mIn);
52      zu = z(mIn+1:mIn*2);
53
54      %initial residuals
55      rL = H*x+g-A*y-(z(1:mIn)-z(mIn+1:mIn*2));
56      rA = b-A'*x;
57      rC = s+[l; -u] - [x; -x];
58
59      % Start point heuristic
60      zsl = zl./sl;
61      zsu = zu./su;
62      Hbar = H + diag(zsl + zsu);
63      KKT = [Hbar -A; -A' zeros(m)];
64      KKT = sparse(KKT);
65      start = cputime;
66      [L,D,p] = ldl(KKT,'vector');
67      ldltime = ldltime + cputime-start;
68
69
70      % Affine step
71      rCs = (rC-s);
72      rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
73      rhs = -[rLbar ; rA];
74
75      solution(p) = L' \ (D \ (L \rhs(p)));
76
77      dxAff = solution(1:length(x))';
78
79      dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z./s).*rCs;
80      dsAff = -s-(s./z).*dzAff;
81
82      %Update of starting point
83      z = max(1,abs(z+dzAff));
84      s = max(1,abs(s+dsAff));
85
86      %Update of initial residuals
87      sl = s(1:mIn);
88      su = s(mIn+1:mIn*2);
89      zl = z(1:mIn);
90      zu = z(mIn+1:mIn*2);
91
92
93      rL = H*x+g-A*y-(zl-zu);
```

```
94      rA = b-A'*x;
95      rC = s+[l; -u] - [x; -x];
96
97      % Initial dual gap
98      dualGap = (z'*s)/(2*mIn);
99      dualGap0 = dualGap;
100
101
102     for i = 1:max_iter
103         iter = iter + 1;
104         zsl = zl./sl;
105         zsu = zu./su;
106         Hbar = H + diag(zsl + zsu);
107         KKT = [Hbar -A; -A' zeros(m)];
108         KKT = sparse(KKT);
109         start = cputime;
110         [L,D,p] = ldl(KKT,'vector');
111         ldltime = ldltime + cputime-start;
112
113         % Affine step
114         rCs = (rC-s);
115         rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
116
117         rhs = -[rLbar ; rA];
118         solution(p) = L' \ (D \ (L \rhs(p)));
119
120         dxAff = solution(1:length(x))';
121
122         dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z./s).*rCs;
123         dsAff = -s-(s./z).*dzAff;
124
125         %compute max alpha affine
126         dZS = [dzAff; dsAff];
127         alphas = (-[z;s]./dZS);
128         alphaAff = min([1;alphas(dZS<0)]);
129
130         dualGapAff = ((z+alphaAff*dzAff)'*(s+alphaAff*dsAff))/(2*mIn);
131         sigma = (dualGapAff/dualGap)^3;
132
133
134         % Affine-Centering-Correction Direction
135         rSZz =  s + dsAff.*dzAff./z-dualGap*sigma*e./z;
136         rCs = (rC-rSZz);
137         rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
138
139         rhs = -[rLbar ; rA];
140         solution(p) = L' \ (D \ (L \rhs(p)));
141
142         dx = solution(1:length(x))';
143         dy = solution(length(x)+1:length(x)+length(y))';
144
145         dz = - [zsl.*dx; -zsu.*dx] + (z./s).*rCs;
146         ds = -rSZz-(s./z).*dz;
147
148         %compute max alpha
```

```
149             dZS = [dz; ds];
150             alphas = (-[z;s]./dZS);
151             alpha = min([1;alphas(dZS<0)]);
152
153             alphaBar = eta*alpha;
154
155             % Update of position
156             x = x + alphaBar * dx;
157             y = y + alphaBar * dy;
158             z = z + alphaBar * dz;
159             s = s + alphaBar * ds;
160
161             % Update of residuals
162             sl = s(1:mIn);
163             su = s(mIn+1:mIn*2);
164             zl = z(1:mIn);
165             zu = z(mIn+1:mIn*2);
166
167
168             rL = H*x+g-A*y-(z(1:mIn)-z(mIn+1:mIn*2));
169             rA = b-A'*x;
170             rC = s+[l; -u] - [x; -x];
171
172             % Compute the dual gap
173             dualGap = (z'*s)/(2*mIn);
174
175             % Check for convergence
176             if(dualGap <= epsilon*0.01*dualGap0)
177                 return
178             end
179         end
180     end
```

Listing A.7: An interior point method for 2.2

A.3.3 Algorithms for exercise 3

A.3.3.1 Interior point method for a box constrained LP

```
1  function [x,y,z,s, iter] = LinearPDIM_box(g,A,b,l,u,x0,y0,z0,s0)
2  % LinearPDIM_box   An interior point solver based on Mehrota's ...
       predictor-corrector
3  %                  primal-dual interior point algorithm. It takes
4  %                  problems of the form
5  %
6  %            min     g'x
7  %             x
8  %            s.t      Ax  = b
9  %                u>=  x >= l
10 %
11 %
```

```
12 % Syntax: [x,y,z,s, iter] = LinearPDIM_box(g,A,b,l,u,x0,y0,z0,s0)
13 %
14 %         x              : Solution
15 %         y              : Equality lagrange multipliers
16 %         z              : Inequality lagrange multipliers
17 %         s              : Slack variables
18 %         iter           : Iterations used
19
20 % Created: 06.06.2021
21 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
22 %           IMM, Technical University of Denmark
23
24 %%
25     % Sets constants for the algorithm
26     mIn = length(u);
27     epsilon = 0.000000001;
28     max_iter = 100;
29     eta = 0.995;
30     iter = 0;
31
32     % Initial values
33     x = x0;
34     y = y0;
35     z = z0;
36     s = s0;
37
38     % Makes sure non of the following matrix operations are singular
39     while(any(s==0))
40         x = x+0.000001;
41         s = [x-l;-x+u];
42     end
43
44     %Initilize constraint specific slacks and lagrange multipliers
45     e = ones(mIn*2,1);
46     sl = s(1:mIn);
47     su = s(mIn+1:mIn*2);
48     zl = z(1:mIn);
49     zu = z(mIn+1:mIn*2);
50
51     %initial residuals
52     rL = g-A*y-(z(1:mIn)-z(mIn+1:mIn*2));
53     rA = b-A'*x;
54     rC = s+[l; -u] - [x; -x];
55
56     % Start point heuristic
57     zsl = zl./sl;
58     zsu = zu./su;
59
60     % Affine step for the start point heuristic
61     rCs = (rC-s);
62     rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
63     Hbar_diagonal_inverse = 1./(zsl+zsu);
64
65     % Calculate the factor in the normal equation
66     normalfactor = A' * (Hbar_diagonal_inverse .*  A);
```

```
67      R = chol(normalfactor);
68
69      mu_rhs = rA + A' * (Hbar_diagonal_inverse .* rLbar);
70      dyAff = R \ (R' \ mu_rhs);
71      dxAff = Hbar_diagonal_inverse .* (-rLbar + A*dyAff);
72
73      dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z./s).*rCs;
74      dsAff = -s-(s./z).*dzAff;
75
76      %Update of starting point
77      z = max(1,abs(z+dzAff));
78      s = max(1,abs(s+dsAff));
79
80      %Update of initial residuals
81      sl = s(1:mIn);
82      su = s(mIn+1:mIn*2);
83      zl = z(1:mIn);
84      zu = z(mIn+1:mIn*2);
85
86
87      rL = g-A*y-(z(1:mIn)-z(mIn+1:mIn*2));
88      rA = b-A'*x;
89      rC = s+[l; -u] - [x; -x];
90
91      % Initial dual gap
92      dualGap = (z'*s)/(2*mIn);
93      dualGap0 = dualGap;
94
95      for i = 1:max_iter
96          iter = iter + 1;
97          zsl = zl./sl;
98          zsu = zu./su;
99
100         % Affine step
101         rCs = (rC-s);
102         rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
103
104         % Calculate the factor in the normal equation
105         Hbar_diagonal_inverse = 1./(zsl+zsu);
106         normalfactor = A' * (Hbar_diagonal_inverse .*  A);
107         R = chol(normalfactor);
108
109         mu_rhs = rA + A' * (Hbar_diagonal_inverse .* rLbar);
110         dyAff = R \ (R' \ mu_rhs);
111         dxAff = Hbar_diagonal_inverse .* (-rLbar + A*dyAff);
112
113
114         dzAff = - [zsl.*dxAff; -zsu.*dxAff] + (z./s).*rCs;
115         dsAff = -s-(s./z).*dzAff;
116
117         %compute max alpha affine
118         dZS = [dzAff; dsAff];
119         alphas = (-[z;s]./dZS);
120         alphaAff = min([1;alphas(dZS<0)]);
121
```

```
122         dualGapAff = ((z+alphaAff*dzAff)'*(s+alphaAff*dsAff))/(2*mIn);
123         sigma = (dualGapAff/dualGap)^3;
124
125
126         % Affine-Centering-Correction Direction
127         rSZz = s + dsAff.*dzAff./z-dualGap*sigma*e./z;
128         rCs = (rC-rSZz);
129
130         rLbar = rL - zsl.*rCs(1:mIn) +zsu.*rCs(1+mIn:2*mIn);
131
132         % Calculate the factor in the normal equation
133         normalfactor = A' * (Hbar_diagonal_inverse .*  A);
134         R = chol(normalfactor);
135
136         mu_rhs = rA + A' * (Hbar_diagonal_inverse .* rLbar);
137
138         %This is normal equation stuff as well
139         dy = R \ (R' \ mu_rhs);
140         dx = Hbar_diagonal_inverse .* (-rLbar + A*dy);
141
142         dz = - [zsl.*dx; -zsu.*dx] + (z./s).*rCs;
143         ds = -rSZz-(s./z).*dz;
144
145         %compute max alpha
146         dZS = [dz; ds];
147         alphas = (-[z;s]./dZS);
148         alpha = min([1;alphas(dZS<0)]);
149
150         alphaBar = eta*alpha;
151
152         % Update of position
153         x = x + alphaBar * dx;
154         y = y + alphaBar * dy;
155         z = z + alphaBar * dz;
156         s = s + alphaBar * ds;
157
158         % Update of residuals
159         sl = s(1:mIn);
160         su = s(mIn+1:mIn*2);
161         zl = z(1:mIn);
162         zu = z(mIn+1:mIn*2);
163
164
165         rL = g-A*y-(z(1:mIn)-z(mIn+1:mIn*2));
166         rA = b-A'*x;
167         rC = s+[l; -u] - [x; -x];
168
169         % Compute the dual gap
170         dualGap = (z'*s)/(2*mIn);
171
172         % Check for convergence
173         if(dualGap ≤ epsilon*0.01*dualGap0)
174             return
175         end
176     end
```

```
177 end
```

Listing A.8: An interior point method for 3.2

A.3.4 Algorithms for exercise 4

A.3.4.1 Interior point method for the sub problem

```
1  function [x,z,feasible,i] = intSQP(B,df,dc,lk,uk,clk,cuk,x0)
2  % intSQP   An interior point solver based on Mehrota's ...
       predictor-corrector
3  %           primal-dual interior point algorithm. It takes
4  %            problems of the form
5  %
6  %            min    x'*H*x+g'x
7  %             x
8  %            s.t    gu≥ cx ≥ gl
9  %                    u≥  x ≥ l
10 %
11 %
12 % Syntax: [x,z,feasible,i] = intSQP(B,df,dc,lk,uk,clk,cuk,x0)
13 %
14 %         x             : Solution
15 %         z             : Lagrange multipliers
16 %         feasible      : Flag to indicate feasibility
17 %         i             : Iterations used
18
19 % Created: 06.06.2021
20 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
21 %           IMM, Technical University of Denmark
22
23 %%
24     % Sets constants for the algorithm
25     warning('off','all')
26     n = length(x0);
27     m = length(cuk);
28     mc = n*2+2*m;
29     epsilon = 0.0001;
30     max_iter = 100;
31     eta = 0.995;
32     feasible = 1;
33
34     % Initial values
35     x = x0;
36     z = ones(mc,1);
37     s = ones(mc,1);
38     d = [lk;-uk;clk;-cuk];
39     e = ones(2*n+2*m,1);
40
41     % Makes sure non of the following matrix operations are singular
42     while(any(s==0))
```

```
43          x = x+0.000001;
44          s = [x;-x; dc'*x; -dc'*x]-d;
45      end
46
47      %Initilize constraint specific slacks and lagrange multipliers
48      sl = s(1:n);
49      su = s(n+1:n*2);
50      scl = s(2*n+1:2*n+m);
51      scu = s(m+2*n+1:n*2+2*m);
52      zl = z(1:n);
53      zu = z(n+1:n*2);
54      zcl = z(2*n+1:2*n+m);
55      zcu = z(m+2*n+1:n*2+2*m);
56
57      %initial residuals
58      rL = B*x+df-(zl-zu+dc*zcl-dc*zcu);
59      rC = s+d - [x; -x; dc'*x; -dc'*x];
60
61      % Start point heuristic
62      zsl = diag(zl./sl);
63      zsu = diag(zu./su);
64      zslc = zcl./scl;
65      zsuc = zcu./scu;
66      zc = zslc + zsuc;
67      Hbar = B + zsl + zsu + bsxfun(@times,zc',dc)*dc';
68      [L,D,P] = ldl(Hbar,'lower');
69
70      % Affine step for the start point heuristic
71      rCs = (rC-s);
72      rLbar = rL - [ zsl -zsu bsxfun(@times,zslc',dc) ...
            bsxfun(@times,zsuc',-dc)]*rCs;
73
74      rhs = -rLbar;
75      dxAff = P*(L' \ (D \ (L \ (P'*rhs) )));
76
77      dzAff = - [ zsl; -zsu; bsxfun(@times,zslc,dc'); ...
            bsxfun(@times,zsuc,-dc')]*dxAff + (z./s).*rCs;
78      dsAff = -s-(s./z).*dzAff;
79
80      %Update of starting point
81      z = max(1,abs(z+dzAff));
82      s = max(1,abs(s+dsAff));
83
84      %Update of initial residuals
85      sl = s(1:n);
86      su = s(n+1:n*2);
87      scl = s(2*n+1:2*n+m);
88      scu = s(m+2*n+1:n*2+2*m);
89      zl = z(1:n);
90      zu = z(n+1:n*2);
91      zcl = z(2*n+1:2*n+m);
92      zcu = z(m+2*n+1:n*2+2*m);
93
94
95      rL = B*x+df-(zl-zu+dc*zcl-dc*zcu);
```

```
96      rC = s+d - [x; -x; dc'*x; -dc'*x];
97      rSZ = s.*z;
98
99      % Initial dual gap
100     dualGap = (z'*s)/(mc);
101     dualGap0 = dualGap;
102
103     for i = 1:max_iter
104         zsl = diag(zl./sl);
105         zsu = diag(zu./su);
106         zslc = zcl./scl;
107         zsuc = zcu./scu;
108         zc = zslc + zsuc;
109
110         % Factorization
111         Hbar = B + zsl + zsu + bsxfun(@times,zc',dc)*dc';
112         [L,D,P] = ldl(Hbar,'lower');
113
114         % Affine step
115         rCs = (rC-s);
116         rLbar = rL - [ zsl -zsu bsxfun(@times,zslc',dc) ...
                bsxfun(@times,zsuc',-dc)]*rCs;
117
118         rhs = -rLbar;
119         dxAff = P*(L' \ (D \ (L \ (P'*rhs) )));
120
121         dzAff = - [ zsl; -zsu; bsxfun(@times,zslc,dc'); ...
                bsxfun(@times,zsuc,-dc')]*dxAff + (z./s).*rCs;
122         dsAff = -s-(s./z).*dzAff;
123
124         % Compute max alpha affine
125         dZS = [dzAff; dsAff];
126         alphas = (-[z;s]./dZS);
127         alphaAff = min([1;alphas(dZS<0)]);
128
129         dualGapAff = ((z+alphaAff*dzAff)'*(s+alphaAff*dsAff))/(mc);
130         sigma = (dualGapAff/dualGap)^3;
131
132
133         % Affine-Centering-Correction Direction
134         rSZbar = rSZ + dsAff.*dzAff-sigma*dualGap*sigma*e;
135         rLbar = rL - [ zsl -zsu bsxfun(@times,zslc',dc) ...
                bsxfun(@times,zsuc',-dc)]*(rC-rSZbar./z);
136
137         rhs = -rLbar;
138         dx = P*(L' \ (D \ (L \ (P'*rhs) )));
139
140         dz = - [ zsl; -zsu; bsxfun(@times,zslc,dc'); ...
                bsxfun(@times,zsuc,-dc')]*dx + (z./s).*(rC-rSZbar./z);
141         ds = -rSZbar./z-(s./z).*dz;
142
143         %compute max alpha
144         dZS = [dz; ds];
145         alphas = (-[z;s]./dZS);
146         alpha = min([1;alphas(dZS<0)]);
```

```
147
148         alphaBar = eta*alpha;
149
150         % Update of position
151         x = x + alphaBar * dx;
152         z = z + alphaBar * dz;
153         s = s + alphaBar * ds;
154
155         % Update of residuals
156         sl = s(1:n);
157         su = s(n+1:n*2);
158         scl = s(2*n+1:2*n+m);
159         scu = s(m+2*n+1:n*2+2*m);
160         zl = z(1:n);
161         zu = z(n+1:n*2);
162         zcl = z(2*n+1:2*n+m);
163         zcu = z(m+2*n+1:n*2+2*m);
164
165
166         rL = B*x+df-(zl-zu+dc*zcl-dc*zcu);
167         rC = s+d - [x; -x; dc'*x; -dc'*x];
168         rSZ = s.*z;
169
170         % Compute the dual gap
171         dualGap = (z'*s)/(mc);
172
173         if any(isnan(z))
174             feasible = 0;
175             return
176         end
177
178         % Check for convergence
179         if(dualGap <= epsilon*0.01*dualGap0)
180            return
181         end
182     end
183 end
```

Listing A.9: An interior point method for 4.20

A.3.4.2 A SQP-BFGS algorithm

```
1 function [x,z,Hist] = SQP(x0,obj,con,l,u,cl,cu,log, subsolver,precision)
2 % SQP       A sequential quadratic programing algorithm with a damped ...
      BFGS
3 %           update of the hessian
4 %
5 %            min   f(x)
6 %             x
7 %            s.t   gu>= c(x) >= gl
8 %                   u>=  x >= l
9 %
```

```
10 %
11 % Syntax: [x,z,Hist] = SQP(x0,obj,con,l,u,cl,cu,log, subsolver,precision)
12 %
13 %         x             : Solution
14 %         z             : Lagrange multipliers
15 %         Hist          : Hist object with algorithm run-time information
16
17 % Created: 06.06.2021
18 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
19 %           IMM, Technical University of Denmark
20
21 %%
22     max_iter = 100;
23     n = length(x0);
24     epsilon = 10^(-precision);
25     functionCalls = 0;
26
27     % Define relevant functions call
28     x = x0;
29     [¬,df] = feval(obj,x);
30     [c,dc] = feval(con,x);
31     B = eye(n);
32     m = size(c,1);
33     z = ones(2*m+2*n,1);
34     lid = 1:m;
35     uid = (m+1):(2*m);
36     clid = (2*m+1):(2*m+n);
37     cuid = (2*m+n+1):(2*(n+m));
38
39     % Define options for quadprog
40     options = optimset('Display', 'off');
41
42     % Log objects
43     if log
44         xHist = zeros(n,max_iter+1);
45         pkHist = zeros(n,max_iter);
46         timePerformence = zeros(1,max_iter);
47         functionCalls = 2;
48
49         xHist(:,1) = x0;
50     end
51
52
53     % Start for loop
54     for i = 1:max_iter
55
56         % Update lower and upper bounds for the quadrastart = cputime; ...
               approximation
57         lk = -x+l;
58         uk = -x+u;
59         clk = -c+cl;
60         cuk = -c+cu;
61
62         % Solves local QP program
63         if subsolver
```

```
64              start = cputime;
65              [pk,zhat] = intSQP(B,df,dc,lk,uk,clk,cuk,x);
66              time = cputime-start;
67          else
68              start = cputime;
69              [pk,~,~,~,lambda] = quadprog(B,df,-[dc'; ...
                    -dc'],-[clk;-cuk],[],[],lk,uk,[], options);
70              time = cputime-start;
71
72              zhat = [lambda.lower; lambda.upper; lambda.ineqlin];
73          end
74
75          if any(isempty(pk) | isnan(pk) == true)
76              error('The program is infeasible. Try with infeasibility ...
                    handling')
77          end
78
79          pz = zhat-z;
80
81
82          % Update the current point
83          z = z + pz;
84          x = x + pk;
85
86
87          % For the quasi Newton update
88          dL = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
89
90          % Update values for next iteration
91          [~,df] = feval(obj,x);
92          [c,dc] = feval(con,x);
93          functionCalls = functionCalls +2;
94
95          % Quasi newton update of the hessian
96          dL2 = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
97
98          p = pk;
99          q = dL2-dL;
100         theta = 1;
101         Bp = (B*p);
102         pBp = p'*Bp;
103
104         if p'*q < 0.2*pBp
105             theta = (0.8*pBp)/(pBp-p'*q);
106         end
107         r = theta*q+(1-theta)*(Bp);
108         B = B + r*r'/(p'*r) - Bp*Bp'/pBp;
109
110         if log
111             pkHist(:,i) = pk;
112             xHist(:,i+1) = x0;
113         end
114
115         % log information
116         if log
```

```
117                 pkHist(:,i) = pk;
118                 xHist(:,i+1) = x;
119                 timePerformence(1,i) = time;
120             end
121
122             % Check for convergence
123             if norm(dL2, 'inf')<epsilon
124                 if log
125                     pkHist = pkHist(:,1:i);
126                     xHist = xHist(:,1:i+1);
127                     timePerformence = timePerformence(:,1:i);
128
129                     Hist = struct('xHist', xHist, 'pkHist', pkHist, ...
                            'timePerformence', timePerformence, 'Iterations' ...
                            ,i, 'functionCalls', functionCalls);
130                 else
131                     Hist = struct();
132                 end
133
134                 return
135             end
136         end
137
138     end
```

Listing A.10: A SQP-BFGS algorithm

A.3.4.3 A SQP-BFGS algorithm with infeasibility handling

```
1  function [x,z,Hist] = ...
       SQP_infes(x0,obj,con,l,u,cl,cu,log,precision,penalty)
2  % SQP_infes        A sequential quadratic programing algorithm with a ...
       damped BFGS
3  %                    update of the hessian and infeasibility handling
4  %
5  %             min    f(x)
6  %              x
7  %             s.t    gu≥ c(x) ≥ gl
8  %                     u≥  x ≥ l
9  %
10 %
11 % Syntax: [x,z,Hist] = ...
       SQP_infes(x0,obj,con,l,u,cl,cu,log,precision,penalty)
12 %
13 %          x              : Solution
14 %          z              : Lagrange multipliers
15 %          Hist           : Hist object with algorithm run-time information
16
17 % Created: 06.06.2021
18 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
19 %           IMM, Technical University of Denmark
20
```

```
21 %%
22     max_iter = 100;
23     n = length(x0);
24     epsilon = 10^(-precision);
25     functionCalls = 0;
26
27     % Define relevant functions call
28     x = x0;
29     [~,df] = feval(obj,x);
30     [c,dc] = feval(con,x);
31     B = eye(n);
32     m = size(c,1);
33     z = ones(2*m+2*n,1);
34     lid = 1:m;
35     uid = (m+1):(2*m);
36     clid = (2*m+1):(2*m+n);
37     cuid = (2*m+n+1):(2*(n+m));
38
39     % Log objects
40     if log
41         xHist = zeros(n,max_iter+1);
42         pkHist = zeros(n,max_iter);
43         timePerformence = zeros(1,max_iter);
44         functionCalls = 2;
45
46         xHist(:,1) = x0;
47     end
48
49     % Define options for quadprog
50     options = optimset('Display', 'off');
51
52
53
54
55     % Start for loop
56     for i = 1:max_iter
57
58         % Update lower and upper bounds for the quadrastart = cputime; ...
               approximation
59         lk = -x+l;
60         uk = -x+u;
61         clk = -c+cl;
62         cuk = -c+cu;
63
64         % Define infesibility program
65         Hinf = [B zeros(n,2*m); zeros(2*m,2*m+n)];
66         ginf = [df; penalty*ones(2*m,1)];
67         iden = eye(m);
68         Cinf = [dc -dc zeros(m,2*m);iden zeros(m,m) iden zeros(m,m); ...
               zeros(m,m) iden zeros(m,m) iden]';
69         dinf = [clk; -cuk; zeros(2*m,1)];
70         lkinf = [lk; zeros(2*m,1)];
71         ukinf = [uk; inf(2*m,1)];
72
73         % Solves local QP program
```

```
74          start = cputime;
75          [pk,~,~,~,zhat] = ...
                quadprog(Hinf,ginf,-Cinf,-dinf,[],[],lkinf,ukinf,[],options);
76          time = cputime-start;
77          zhat = [zhat.lower(1:n); zhat.upper(1:n); zhat.ineqlin(1:2*m)];
78          pk = pk(1:n);
79          pz = zhat-z;
80
81
82          % Update the current point
83          z = z + pz;
84          x = x + pk;
85
86
87          % For the quasi Newton update
88          dL = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
89
90          % Update values for next iteration
91          [~,df] = feval(obj,x);
92          [c,dc] = feval(con,x);
93
94          functionCalls = functionCalls +2;
95
96          % Quasi newton update of the hessian
97          dL2 = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
98
99          p = pk;
100         q = dL2-dL;
101         theta = 1;
102         Bp = (B*p);
103         pBp = p'*Bp;
104
105         if p'*q < 0.2*pBp
106             theta = (0.8*pBp)/(pBp-p'*q);
107         end
108         r = theta*q+(1-theta)*(Bp);
109         B = B + r*r'/(p'*r) - Bp*Bp'/pBp;
110
111         % log information
112         if log
113             pkHist(:,i) = pk;
114             xHist(:,i+1) = x;
115             timePerformence(1,i) = time;
116         end
117
118         % Check for convergence
119         if norm(dL2, 'inf')<epsilon
120             if log
121                 pkHist = pkHist(:,1:i);
122                 xHist = xHist(:,1:i+1);
123                 timePerformence = timePerformence(:,1:i);
124
125                 Hist = struct('xHist', xHist, 'pkHist', pkHist, ...
                        'timePerformence', timePerformence, 'Iterations' ...
                        ,i, 'functionCalls' ,functionCalls);
```

```
126                 else
127                     Hist = struct();
128                 end
129
130                 return
131             end
132         end
133
134     end
```

Listing A.11: A SQP-BFGS algorithm with infeasibility handling

A.3.4.4 A SQP-BFGS algorithm with line search

```
1  function [x,z,Hist] = SQP_ls(x0,obj,con,l,u,cl,cu,log, ...
       subsolver,precision, nonmonotone)
2  % SQP_ls        A sequential quadratic programing algorithm with a ...
       damped BFGS
3  %                update of the hessian and line search
4  %
5  %             min    f(x)
6  %              x
7  %             s.t    gu≥ c(x) ≥ gl
8  %                     u≥  x ≥ l
9  %
10 %
11 % Syntax: [x,z,Hist] = SQP_ls(x0,obj,con,l,u,cl,cu,log, ...
       subsolver,precision, nonmonotone)
12 %
13 %          x              : Solution
14 %          z              : Lagrange multipliers
15 %          Hist           : Hist object with algorithm run-time information
16
17 % Created: 06.06.2021
18 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
19 %           IMM, Technical University of Denmark
20
21 %%
22      max_iter = 50;
23      n = length(x0);
24      epsilon = 10^(-precision);
25      d = [l;-u;cl;-cu];
26      mu = 0;
27      functionCalls = 0;
28
29
30      % Define relevant functions call
31      x = x0;
32      [f,df] = feval(obj,x);
33      [c,dc] = feval(con,x);
34      B = eye(n);
35      m = size(c,1);
```

```
36      z = ones(2*m+2*n,1);
37      lid = 1:m;
38      uid = (m+1):(2*m);
39      clid = (2*m+1):(2*m+n);
40      cuid = (2*m+n+1):(2*(n+m));
41
42      % Define options for quadprog
43      options = optimset('Display', 'off');
44
45      % Log objects
46      if log
47          xHist = zeros(n,max_iter+1);
48          pkHist = zeros(n,max_iter);
49          timePerformence = zeros(1,max_iter);
50          stepLength = zeros(1,max_iter);
51          functionCalls = 2;
52
53          xHist(:,1) = x0;
54      end
55
56      %Start for loop
57      for i = 1:max_iter
58
59          % Update lower and upper bounds for the quadrastart = cputime; ...
                approximation
60          lk = -x+l;
61          uk = -x+u;
62          clk = -c+cl;
63          cuk = -c+cu;
64
65          % Solves local QP program
66          if subsolver
67              start = cputime;
68              [pk,zhat] = intSQP(B,df,dc,lk,uk,clk,cuk,x);
69              time = cputime-start;
70          else
71              start = cputime;
72              [pk,~,~,~,lambda] = quadprog(B,df,-[dc'; ...
                    -dc'],-[clk;-cuk],[],[],lk,uk,[], options);
73              time = cputime-start;
74
75              zhat = [lambda.lower; lambda.upper; lambda.ineqlin];
76          end
77
78
79          if any(isempty(pk) | isnan(pk) == true)
80              disp(i)
81              error('The program is infeasible. Try with infeasibility ...
                    handling')
82          end
83
84          % Line search
85          alpha = 1;
86          pz = zhat-z;
87          [c_l] = feval(con,x);
```

```
88          functionCalls = functionCalls +1;
89          c_ls = [x; -x; c_l; -c_l]-d;
90          mu = max(abs(z),1/2*(mu+abs(z)));
91          phi0 = phi(f,mu,c_ls);
92          Dphi0 = dphi(df,pk,mu,c_ls);
93
94          while true
95              x_ls = x + alpha*pk;
96              f_ls = obj(x_ls);
97              functionCalls = functionCalls +1;
98              [c_l] = feval(con,x_ls);
99              functionCalls = functionCalls +1;
100             c_ls = [x; -x; c_l; -c_l]-d;
101
102             phi1 = phi(f_ls,mu,c_ls);
103
104             if phi1 <= phi0 + 0.1*alpha*Dphi0
105                 break
106             else
107                 a = (phi1-(phi0+Dphi0*alpha))/alpha^2;
108                 alpha_min = -Dphi0/(2*a);
109
110                 alpha = min(0.9*alpha, max(alpha_min, 0.1*alpha));
111             end
112         end
113
114         % non monotone strategy
115         if (all(round(alpha*pk,precision)==zeros(n,1))) && nonmonotone
116             alpha = 1;
117         end
118
119
120         % Update the current point
121         z = z + alpha*pz;
122         x = x + alpha*pk;
123         mu = z;
124
125         % For the quasi Newton update
126         dL = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
127
128         % Update values for next iteration
129         [f,df] = feval(obj,x);
130         [c,dc] = feval(con,x);
131         functionCalls = functionCalls +2;
132
133         % Quasi newton update of the hessian
134         dL2 = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
135
136         p = alpha*pk;
137         q = dL2-dL;
138         theta = 1;
139         Bp = (B*p);
140         pBp = p'*Bp;
141
142         if p'*q < 0.2*pBp
```

```
143             theta = (0.8*pBp)/(pBp-p'*q);
144         end
145         r = theta*q+(1-theta)*(Bp);
146         B = B + r*r'/(p'*r) - Bp*Bp'/pBp;
147 
148         % log information
149         if log
150             pkHist(:,i) = pk;
151             xHist(:,i+1) = x;
152             timePerformence(1,i) = time;
153             stepLength(1,i) = sqrt(sum((alpha*pk).^2));
154         end
155 
156         % Check for convergence
157         if norm(dL2, 'inf')<epsilon
158             if log
159                 pkHist = pkHist(:,1:i);
160                 xHist = xHist(:,1:i+1);
161                 timePerformence = timePerformence(:,1:i);
162                 stepLength = stepLength(1,1:i);
163 
164                 Hist = struct('xHist', xHist, 'pkHist', pkHist, ...
                        'timePerformence', timePerformence, 'Iterations' ...
                        ,i, 'stepLength', stepLength, 'functionCalls', ...
                        functionCalls);
165             else
166                 Hist = struct();
167             end
168 
169             return
170         end
171     end
172 
173 end
174 % Functions for the line search algorithm
175 function [val] = phi(f,mu,c)
176 val = f+mu'*abs(min(0,c));
177 end
178 function [val] = dphi(df,pk,mu,c)
179 val = df'*pk-mu'*abs(min(0,c));
180 end
181 
182 function [val] = phialt(f,mu,c,z)
183 c = abs(min(0,c));
184 
185 val = f-z'*c+mu'*c.^2;
186 end
187 
188 function [val] = dphialt(df,pk,pz,mu,c,dc,z)
189 c = abs(min(0,c));
190 
191 valx = df-dc*z+dc*(mu.*c);
192 valz = -c;
193 
194 val = [valx; valz]'*[pk;pz];
```

```
195 end
```

Listing A.12: A SQP-BFGS algorithm with line search

A.3.4.5 A SQP-BFGS algorithm with line search and infeasibility handling

```
1  function [x,z,Hist] = ...
       SQP_ls_infes(x0,obj,con,l,u,cl,cu,log,precision,nonmonotone,penalty)
2  % SQP_ls_infes        A sequential quadratic programing algorithm with ...
       a damped BFGS
3  %                        update of the hessian, line search and ...
       infeasibility handling
4  %
5  %              min    f(x)
6  %               x
7  %              s.t    gu≥ c(x) ≥ gl
8  %                      u≥  x ≥ l
9  %
10 %
11 % Syntax: [x,z,Hist] = ...
       SQP_ls_infes(x0,obj,con,l,u,cl,cu,log,precision,nonmonotone,penalty)
12 %
13 %          x              : Solution
14 %          z              : Lagrange multipliers
15 %          Hist           : Hist object with algorithm run-time information
16
17 % Created: 06.06.2021
18 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
19 %           IMM, Technical University of Denmark
20
21 %%
22
23     max_iter = 100;
24     n = length(x0);
25     epsilon = 10^(-precision);
26     d = [l;-u;cl;-cu];
27     mu = penalty;
28     functionCalls = 0;
29
30
31     % Define relevant functions call
32     x = x0;
33     [f,df] = feval(obj,x);
34     [c,dc] = feval(con,x);
35     B = eye(n);
36     m = size(c,1);
37     z = ones(2*m+2*n,1);
38     lid = 1:m;
39     uid = (m+1):(2*m);
40     clid = (2*m+1):(2*m+n);
41     cuid = (2*m+n+1):(2*(n+m));
42
```

```
43      % Log objects
44      if log
45          xHist = zeros(n,max_iter+1);
46          pkHist = zeros(n,max_iter);
47          timePerformence = zeros(1,max_iter);
48          stepLength = zeros(1,max_iter);
49          functionCalls = 2;
50
51          xHist(:,1) = x0;
52      end
53
54      % Define options for quadprog
55      options = optimset('Display', 'off');
56
57
58      % Start for loop
59      for i = 1:max_iter
60
61          % Update lower and upper bounds for the quadrastart = cputime; ...
                approximation
62          lk = -x+l;
63          uk = -x+u;
64          clk = -c+cl;
65          cuk = -c+cu;
66
67          % Define infesibility program
68          Hinf = [B zeros(n,2*m); zeros(2*m,2*m+n)];
69          ginf = [df; penalty*ones(2*m,1)];
70          iden = eye(m);
71          Cinf = [dc -dc zeros(m,2*m);iden zeros(m,m) iden zeros(m,m); ...
                zeros(m,m) iden zeros(m,m) iden]';
72          dinf = [clk; -cuk; zeros(2*m,1)];
73          lkinf = [lk; zeros(2*m,1)];
74          ukinf = [uk; inf(2*m,1)];
75
76          % Solves local QP program
77          start = cputime;
78          [pk,¬,¬,¬,zhat] = ...
                quadprog(Hinf,ginf,-Cinf,-dinf,[],[],lkinf,ukinf,[],options);
79          time = cputime-start;
80
81          zhat = [zhat.lower(1:n); zhat.upper(1:n); zhat.ineqlin(1:2*m)];
82          pk = pk(1:n);
83
84          %line search
85          alpha = 1;
86          pz = zhat-z;
87          [c_l] = feval(con,x);
88          functionCalls = functionCalls +1;
89          c_ls = [x; -x; c_l; -c_l]-d;
90          mu = max(abs(z),1/2*(mu+abs(z)));
91          phi0 = phi(f,mu,c_ls);
92          Dphi0 = dphi(df,pk,mu,c_ls);
93
94          while true
```

```
95              x_ls = x + alpha*pk;
96              f_ls = obj(x_ls);
97              [c_l] = feval(con,x_ls);
98              functionCalls = functionCalls +2;
99              c_ls = [x; -x; c_l; -c_l]-d;
100
101
102             phi1 = phi(f_ls,mu,c_ls);
103
104
105
106             if phi1 <= phi0 + 0.1*alpha*Dphi0
107                 break
108             else
109                 a = (phi1-(phi0+Dphi0*alpha))/alpha^2;
110                 alpha_min = -Dphi0/(2*a);
111
112                 alpha = min(0.9*alpha, max(alpha_min, 0.1*alpha));
113             end
114         end
115
116         % non monotone strategy
117         if (all(round(alpha*pk,precision)==zeros(n,1))) && nonmonotone
118             alpha = 1;
119         end
120
121         % Update the current point
122         z = z + alpha*pz;
123         x = x + alpha*pk;
124         mu = z;
125
126         % For the quasi Newton update
127         dL = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
128
129         % Update values for next iteration
130         [f,df] = feval(obj,x);
131         [c,dc] = feval(con,x);
132
133         functionCalls = functionCalls +2;
134
135         % Quasi newton update of the hessian
136         dL2 = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
137
138         p = alpha*pk;
139         q = dL2-dL;
140         theta = 1;
141         Bp = (B*p);
142         pBp = p'*Bp;
143
144         if p'*q < 0.2*pBp
145             theta = (0.8*pBp)/(pBp-p'*q);
146         end
147         r = theta*q+(1-theta)*(Bp);
148         B = B + r*r'/(p'*r) - Bp*Bp'/pBp;
149
```

```
150         % Store logging information
151         if log
152             pkHist(:,i) = pk;
153             xHist(:,i+1) = x;
154             timePerformence(1,i) = time;
155             stepLength(1,i) = alpha;
156         end
157
158         % Check convergence
159         if norm(dL2, 'inf')<epsilon
160             if log
161                 pkHist = pkHist(:,1:i);
162                 xHist = xHist(:,1:i+1);
163                 timePerformence = timePerformence(:,1:i);
164                 stepLength = stepLength(1,1:i);
165
166                 Hist = struct('xHist', xHist, 'pkHist', pkHist, ...
                        'timePerformence', timePerformence, 'Iterations' ...
                        ,i, 'stepLength', stepLength, 'functionCalls' , ...
                        functionCalls);
167             else
168                 Hist = struct();
169             end
170
171             return
172         end
173     end
174
175 end
176 function [val] = phi(f,mu,c)
177 val = f+mu'*abs(min(0,c));
178 end
179 function [val] = dphi(df,pk,mu,c)
180 val = df'*pk-mu'*abs(min(0,c));
181 end
182
183 function [val] = phialt(f,mu,c,z)
184 c = abs(min(0,c));
185
186 val = f-z'*c+mu'*c.^2;
187 end
188
189 function [val] = dphialt(df,pk,pz,mu,c,dc,z)
190 c = abs(min(0,c));
191
192 valx = df-dc*z+dc*(mu.*c);
193 valz = -c;
194
195 val = [valx; valz]'*[pk;pz];
196 end
```

Listing A.13: A SQP-BFGS algorithm with line search and infeasibility handling

A.3.4.6 A trust region SQP algorithm

```
function [x,z,Hist] = ...
    SQP_trust(x0,obj,cons,l,u,cl,cu,log,precision,trust_region, penalty)
% SQP_trust       A sequential quadratic programing algorithm with a ...
    damped BFGS
%                   update of the hessian, line search and ...
    infeasibility handling
%
%            min   f(x)
%             x
%            s.t   gu≥ c(x) ≥ gl
%                   u≥  x ≥ l
%
%
% Syntax: [x,z,Hist] = ...
    SQP_trust(x0,obj,cons,l,u,cl,cu,log,precision,trust_region, penalty)
%
%         x             : Solution
%         z             : Lagrange multipliers
%         Hist          : Hist object with algorithm run-time information

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

%%
    max_iter = 100;
    n = length(x0);
    epsilon = 10^(-precision);
    mu = penalty;
    dk0 = trust_region;
    dk = dk0;
    time = 0;
    dL2 = Inf;

    % Define relevant functions call
    x = x0;
    [f,df] = obj(x);
    [c,dc] = cons(x);
    B = eye(n);
    m = size(c,1);
    mu_vec = mu*ones(n*2+m*2,1);
    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));

    % Log objects
    if log
        xHist = zeros(n,max_iter+1);
        pkHist = zeros(n,max_iter);
```

```
48          timePerformence = zeros(1,max_iter);
49          trustRegion = zeros(1,max_iter+1);
50          rhos = zeros(1,max_iter);
51          functionCalls = 2;
52
53          xHist(:,1) = x0;
54          trustRegion(:,1) = dk;
55      end
56
57      % Define options for quadprog
58      options = optimset('Display', 'off');
59
60
61      % Start for loop
62      for i = 1:max_iter
63
64          % Update lower and upper bounds for the quadrastart = cputime; ...
                approximation
65          lk = -x+l;
66          uk = -x+u;
67          clk = -c+cl;
68          cuk = -c+cu;
69
70          % Define infesibility program
71          Hinf = zeros(3*n+2*m);
72          Hinf(1:n,1:n) = B;
73          Cinf = [ eye(n) -eye(n) dc -dc zeros(n,2*m+2*n) eye(n)  ...
                -eye(n);eye(2*n+2*m) eye(2*n+2*m) zeros(2*m+2*n,n*2)]';
74          ginf = [df; mu_vec];
75          dinf = [lk; -uk;clk; -cuk; zeros(2*m+2*n,1); -dk*ones(2*m,1)];
76          start = cputime;
77          [pk,~,~,~,zhat] = ...
                quadprog(Hinf,ginf,-Cinf,-dinf,[],[],[],[],[],options);
78          time = time + cputime-start;
79
80          zhat = zhat.ineqlin(1:2*m+2*n);
81          pk = pk(1:n);
82
83          %Update penalty
84          zinf = vecnorm(z,'Inf');
85          mu = max(1/2*(mu+zinf),zinf);
86          mu_vec = mu*ones(n*2+m*2,1);
87
88          %Calculate relevant values for the trust region
89          [c_full, dc_full] = cons(x,true,dinf(1:2*n+2*m));
90          functionCalls = functionCalls +1;
91
92          [c_p_full, ~] = cons(x+pk,true,dinf(1:2*n+2*m));
93          functionCalls = functionCalls +1;
94
95          qp0 = f+df'*pk+1/2*pk'*B*pk+mu_vec'*max(0,-(c_full+dc_full'*pk));
96          q0 = f+mu_vec'*max(0,-(c_full));
97          phi1 = q0;
98
99          [f_p,~] = obj(x+pk);
```

```
100         functionCalls = functionCalls +1;
101         phi1p = f_p+mu_vec'*max(0,-(c_p_full));
102
103         rho = (phi1-phi1p)/(q0-qp0);
104
105         gamma = min(max((2*rho-1)^3+1,0.25),2);
106
107         % If trust region is accepted
108         if rho>0
109             % Update the current point
110             z = zhat;
111             x = x+pk;
112
113
114             % For the quasi Newton update
115             dL = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
116
117
118
119             % Update values for next iteration
120             [f,df] = feval(obj,x);
121             [c,dc] = feval(cons,x);
122
123             functionCalls = functionCalls +2;
124
125             % Quasi newton update of the hessian
126             dL2 = df - (z(lid)-z(uid)+dc*z(clid)-dc*z(cuid));
127
128             p = pk;
129             q = dL2-dL;
130             theta = 1;
131             Bp = (B*p);
132             pBp = p'*Bp;
133
134             if p'*q < 0.2*pBp
135                 theta = (0.8*pBp)/(pBp-p'*q);
136             end
137             r = theta*q+(1-theta)*(Bp);
138             B = B + r*r'/(p'*r) - Bp*Bp'/pBp;
139
140             % Update the trust region
141             dk = gamma*dk;
142
143         % If the trust region is not accepted
144         else
145             % Update the trust region
146             dk = gamma * vecnorm(pk,'Inf');
147         end
148
149         % Store relevant logging information
150         if log
151             pkHist(:,i) = pk;
152             xHist(:,i+1) = x;
153             timePerformence(1,i) = time;
154             trustRegion(:,i+1) = dk;
```

```
155             rhos(:,i) = rho;
156             time = 0;
157         end
158
159         % Check for convergence
160         if norm(dL2, 'inf')<epsilon
161             if log
162                 pkHist = pkHist(:,1:i);
163                 xHist = xHist(:,1:i+1);
164                 timePerformence = timePerformence(:,1:i);
165                 rhos = rhos(:,1:i);
166                 trustRegion = trustRegion(:,1:i+1);
167
168
169                 Hist = struct('xHist', xHist, 'pkHist', pkHist, ...
                        'timePerformence', timePerformence, 'Iterations' ...
                        ,i, 'functionCalls', functionCalls, 'rho', rhos, ...
                        'trustRegion',trustRegion);
170             else
171                 Hist = struct();
172             end
173
174             return
175         end
176     end
177
178 end
```

Listing A.14: A trust region SQP algorithm

A.4 Drivers and interfaces

A.4.1 Drivers for exercise 1

```
1  %{
2   This is the driver for exercise 1.
3   This file contains:
4       - A test for correctness using the problem given in exercise 1.4.
5       - A test for efficiency under growing problem using the recycling
6         problem given in week 5.
7       - A benchmark test for matrix factorizations.
8       - We vary the number of constraints instead of the number of ...
              variables
9   %}
10  %% Given problem
11  names = ["LDLdense", "LDLsparse", "LUdense", "LUsparse", "rangespace", ...
        "nullspace", "quadprog"];
12  tests = 20;
13  times = zeros(7,tests);
14  answers = zeros(5,tests,7);
```

```
bs = [];
i=1;
for b1 = linspace(8.5, 18.68, tests)
    bs = [bs;b1];
    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 = [b1;1];

    %We test all solvers one by one and save the answers
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "LDLdense");
    answers(:, i,1) = x;
    times(1, i) = cputime-start;
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "LDLsparse");
    answers(:, i,2) = x;
    times(2, i) = cputime-start;
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "LUdense");
    answers(:, i,3) = x;
    times(3, i) = cputime-start;
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "LUsparse");
    answers(:, i,4) = x;
    times(4, i) = cputime-start;
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "rangespace");
    answers(:, i,5) = x;
    times(5, i) = cputime-start;
    start = cputime;
    [x, lambda] = EqualityQPSolver(H,g,A,b, "nullspace");
    answers(:, i,6) = x;
    times(6, i) = cputime-start;
    options = optimset('Display', 'off');
    start = cputime;
    [x, lambda] = quadprog(H,g,[],[],A',b,[],[],[],options);
    answers(:, i,7) = x;
    times(7, i) = cputime-start;


    i = i+1;
end


%We compare answers to quadprog
answer_diff = zeros(6,20);
for i=1:6
    answer_diff(i,:) = mean(sqrt((answers(:,:,i)-answers(:,:,7)).^2));
end
```

```
70
71
72
73  hold off
74  for i=1:size(times,1)
75      plot(bs, times(i,:))
76      hold on
77  end
78  legend(names)
79  xlabel("b(1)")
80  ylabel("time [log s]")
81
82  %and plot the answers
83  figure
84  for i=1:6
85      plot(bs, (answer_diff(i,:)))
86      hold on
87  end
88  legend(names(1:6))
89  xlabel("b(1)")
90  ylabel("Error relative to quadprog")
91
92
93
94
95  %% Recycling problem
96  names = ["LDLdense", "LDLsparse", "LUdense", "LUsparse", "rangespace", ...
        "nullspace", "quadprog"];
97  tests = 20;
98  times = ones(7,tests)*tests;
99  ns = zeros(tests,1);
100 %We test every solver using the Recycling problem of different sizes n.
101 for i = 1:tests
102     disp(i)
103     n = i*(2000/tests);
104     ns(i) = n;
105
106     [H, g, A, b] = ProblemEQPRecycling(n, 0.2, 1);
107     for k =1:1
108         j=1;
109
110         start = cputime;
111         [x, lambda] = EqualityQPSolver(H,g,A,b, "LDLdense");
112         times(j, i) = cputime-start;
113         j = j+1;
114
115         start = cputime;
116         [x, lambda] = EqualityQPSolver(H,g,A,b, "LDLsparse");
117         times(j, i) = cputime-start;
118         j = j+1;
119         %
120         start = cputime;
121         [x, lambda] = EqualityQPSolver(H,g,A,b, "LUdense");
122         times(j, i) = cputime-start;
123         j = j+1;
```

```
124         %
125         start = cputime;
126         [x, lambda] = EqualityQPSolver(H,g,A,b, "LUsparse");
127         times(j, i) = min(cputime-start, times(j,i));
128         j = j+1;
129
130         %
131         start = cputime;
132         [x, lambda] = EqualityQPSolver(H,g,A,b, "rangespace");
133         times(j, i) = cputime-start;
134         j = j+1;
135
136         start = cputime;
137         [x, lambda] = EqualityQPSolver(H,g,A,b, "nullspace");
138         times(j, i) = cputime-start;
139         j = j+1;
140
141         options = optimset('Display', 'off');
142         start = cputime;
143         [x, lambda] = quadprog(H,g,[],[],A',b,[],[],[],options);
144         times(j, i) = cputime-start;
145         j = j+1;
146         %
147     end
148
149     disp("n="+n)
150 end
151
152 %We plot the time for each method.
153 figure
154 hold off
155 for i=1:size(times,1)
156     plot(ns, times(i,:))
157     hold on
158 end
159 legend(names, 'Location','northwest')
160 xlabel("n")
161 ylabel("time [s]")
162 title("Growing size problem")
163
164
165 %% Factorization benchmark
166 tests = 10;
167 times = ones(5,tests)*100;
168 ns = zeros(tests,1);
169 %We bench LU vs LDL versus Cholesky of varying sizes
170 for i = 1:tests
171     n = i*(5000/tests);
172     ns(i) = n;
173
174     [H,g,A,b,x,lambda] = generateRandomEQP(n,n);
175     KKT = [H -A;-A', zeros(size(A,2), size(A,2))];
176     for k =1:1
177         j=1;
178
```

```
179             start = cputime;
180             x = lu(KKT,'vector');
181             times(j, i) = min(cputime-start, times(j,i));
182             j = j+1;
183
184             start = cputime;
185             x = lu(H,'vector');
186             times(j, i) = min(cputime-start, times(j,i));
187             j = j+1;
188
189             start = cputime;
190             x = ldl(KKT,'vector');
191             times(j, i) = min(cputime-start, times(j,i));
192             j = j+1;
193
194             start = cputime;
195             x = ldl(H,'vector');
196             times(j, i) = min(cputime-start, times(j,i));
197             j = j+1;
198
199             start = cputime;
200             x = chol(H);
201             times(j, i) = min(cputime-start, times(j,i));
202             j = j+1;
203
204
205         end
206
207         disp("n="+n)
208     end
209
210
211     %We then plot the time for each method and target.
212     hold off
213     for i=1:5%size(times,1)
214         semilogy(ns, times(i,:))
215         hold on
216     end
217     legend(["LU, KKT", "LU, H", "LDL, KKT", "LDL, H", "Cholesky, H"])
218     xlabel("n")
219     ylabel("time [s]")
220     title("Benchmark of factorizations")
221
222     %% m dependent
223     names = ["rangespace", "nullspace"];
224     tests = 10;
225     times = ones(2,tests)*100;
226     ms = zeros(tests,1);
227     top = 3000;
228     %To compare range space and null space, we solve random EQPs with varying
229     %number of constraints.
230     for i = 1:tests
231         m = i*(top/tests);
232         ms(i) = m;
233
```

```
234         [H, g, A, b] = generateRandomEQP(top, m);
235         for k =1:1
236             j=1;
237
238
239             start = cputime;
240             [x, lambda, facTime_R] = EqualityQPSolver(H,g,A,b, "rangespace");
241             times(j, i) = min(cputime-start, times(j,i));
242             j = j+1;
243
244             start = cputime;
245             [x, lambda, facTime_N] = EqualityQPSolver(H,g,A,b, "nullspace");
246             times(j, i) = min(cputime-start, times(j,i));
247             j = j+1;
248
249         end
250
251         disp("m="+m)
252     end
253
254     figure
255     hold off
256     for i=1:2
257         plot(ms, times(i,:))
258         hold on
259     end
260     plot(ms, times(1,:)-facTime_R)
261     plot(ms, times(2,:)-facTime_N)
262     xline(1950)
263     legend([names,'range space -Cholesky','null space -QR','Theoretical ...
            tipping point'], 'Location','northwest')
264     xlabel("m")
265     ylabel("time [s]")
266     title("Growing constraints problem, n=3000")
```

Listing A.15: Driver for exercise 1

A.4.2 Solver interface for exercise 1

```
1  function [x, lambda,time] = EqualityQPSolver(H, g, A, b, solver)
2  % EqualityQPSolver      The solver interface for the equality EQP solvers
3  %
4  %          min  x'*H*x+g'x
5  %           x
6  %          s.t. A x  = b      (Lagrange multiplier: lambda)
7  %
8  %
9  % Syntax: [x, lambda,time] = EqualityQPSolver(H, g, A, b, solver)
10 %
11 %         x             : Solution
12 %         z             : Lagrange multipliers
```

```
%          time          : Time used on factorization in some of the
%                            algorithms
%
% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, 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,time] = EqualityQPSolverRangeSpace(H,g,A,b);
elseif solver == "nullspace"
    [x, lambda,time] = EqualityQPSolverNullSpace(H,g,A,b);
else
    error("solver " + solver + "does not exist; possible values are ...
        LDLdense, LDLsparse, LUdense, LUsparse, rangespace, and ...
        nullspace")
end
```

Listing A.16: Solver interface for exercise 1

A.4.3 Driver for exercise 2

```
%{
This is the driver for exercise 2.
This file contains:
    - A test for correctness using the problem given in exercise 1.4.
      To test for this problem set the variable largeProb to false. The
      variable is found in the top of the script.
    - A test for efficiency under growing problem using a random EQP
      generator with bounds. To test for this problem set the
      variable largeProb to true. The size of every increment is
      controlled by n_large. The script will run 20 iterations
%}
%% Testing correctness
largeProb = false;
n_large = 50;


rng(12)
times = zeros(20, 3);
bs = [];
iterations = zeros(20, 3);
```

```
21  i = 1;
22  solution = zeros(20,1);
23  ldltimes = zeros(20,1);
24
25 %Plot settings
26  plotIterations = true;
27  plotTimes = true;
28  plotSolution = true;
29  plotldlvstime = false;
30  plotcvx = false;
31
32 %We test the QP solver for various values of b for the given problem.
33  for b1 = linspace(8.5, 18.68, 20)
34
35      %Create a quadratic program
36      if ¬largeProb
37      n = 5;
38      H = [5.0 1.86 1.24 1.48 -0.46; ...
39           1.86 3.0 0.44 1.12 0.52; ...
40           1.24 0.44 3.8 1.56 -0.54; ...
41           1.48 1.12 1.56 7.2 -1.12; ...
42           -0.46 0.52 -0.54 -1.12 7.8];
43      g = [-16.1; -8.5; -15.7; -10.02; -18.68];
44      A = [16.1 8.5 15.7 10.02 18.68; 1.0 1.0 1.0 1.0 1.0]';
45      b = [b1;1];
46
47
48      l = zeros(5,1);
49      u = ones(5,1);
50      end
51
52      if largeProb
53          n = n_large*i;
54          [H, g, A, b] = generateRandomEQP(n,n/2);
55          l = zeros(n,1);
56          u = ones(n,1);
57      end
58
59     % Solve by CVX
60     if plotcvx
61     start = cputime;
62
63      cvx_begin quiet
64          %cvx_precision low
65          variable x(n)
66          minimize( 1/2 * x' * H *x + g'*x)
67          subject to
68              A' * x == b
69              l ≤ x
70              x ≤ u
71      cvx_end
72
73      times(i,1) = cputime-start;
74      iterations(i,1) = cvx_slvitr;
75     end
```

```
76
77      %solve by quadprog
78
79      options = optimset('Display', 'off');
80      options = optimset(options, 'Algorithm', 'interior-point-convex');
81      start = cputime;
82      [x2, optval, exitflag,output] = quadprog(H, g, [],[], A', b,l,u, ...
            0, options);
83
84      times(i,2) = cputime-start;
85      iterations(i,2) = output.iterations;
86
87
88
89      %solve by own solver
90      x0 = zeros(n,1);
91      s0 = ones(2*n,1);
92      y0 = ones(length(b),1);
93      z0 = ones(2*n,1);
94
95      start = cputime;
96      [x,y,z,s, iter, ldltime] = ...
            primalDualInteriorMethod_box(H,g,A,b,l,u,x0,y0,z0,s0);
97      times(i,3) = cputime-start;
98      iterations(i, 3) = iter;
99      ldltimes(i,1) = ldltime;
100
101     solution(i,1) = mean(sqrt((x-x2).^2));
102
103
104     bs = [bs;b1];
105     disp("Iteration " +i +"/" + 20);
106     disp("Mean error: " + mean(sqrt((x-x2).^2)))
107     disp("quadprog time: " + times(i,2))
108     disp("our time: " + times(i,3))
109     i = i+1;
110
111     if exitflag ~= 1
112         disp("Iteration with b(1)="+b1+" is infeasible!!")
113     end
114 end
115
116
117 % PLOT ITERATIONS
118 if largeProb
119     bs = n_large:n_large:(20*n_large);
120 end
121 if plotIterations
122     figure;
123     disp("Plotting # of iterations!")
124     if plotcvx
125         plot(bs, iterations(:,1))
126     end
127     hold on
128     plot(bs, iterations(:,2))
```

```
129     plot(bs, iterations(:,3))
130     hold off
131     if plotcvx
132         legend(["cvx iterations", "quadprog interior-point ...
                iterations", "Our solver"])
133     else
134         legend(["quadprog interior-point iterations", "Our solver"])
135     end
136
137     ylabel("iterations")
138     ylim([0 max(max(iterations))+2])
139     if largeProb
140          xlabel("n")
141          title("iterations vs n")
142     else
143         xlabel("b(1)")
144         title("iterations vs b(1)")
145     end
146
147 end
148
149 if plotTimes
150     disp("Plotting times!")
151     figure;
152     %Note: Time plotted is total wall time between starting and ending
153     %solver
154     % Not CPU time (since CPU time counts the number of cores used)
155     if largeProb
156         if plotcvx
157             plot(bs, (times(:,1)))
158         end
159         hold on
160         plot(bs, (times(:,2)))
161         plot(bs, (times(:,3)))
162         hold off
163         if plotcvx
164             legend(["cvx time", "quadprog interior-point time", "Our ...
                    solver"])
165         else
166             legend(["quadprog interior-point time", "Our solver"])
167         end
168
169         ylabel("t [s]")
170          xlabel("n")
171          title("time vs n")
172     else
173         if plotcvx
174             semilogy(bs, times(:,1))
175         end
176         hold on
177         semilogy(bs, times(:,2))
178         semilogy(bs, times(:,3))
179         hold off
180         if plotcvx
```

```
181             legend(["cvx time", "quadprog interior-point time", "Our ...
                    solver"])
182         else
183             legend(["quadprog interior-point time", "Our solver"])
184         end
185
186         ylabel("t [s]")
187         xlabel("b(1)")
188         title("time vs b(1)")
189     end
190 end
191
192 if plotSolution
193     figure;
194     disp("Plotting solution!")
195     plot(bs, solution)
196     title("Correctness")
197     ylabel("Mean Squared Error")
198     legend("Error between quadprog and our solver")
199     if largeProb
200         xlabel("n")
201     else
202         xlabel("b(1)")
203     end
204 end
205
206 if plotldlvstime
207     disp('plotting ldl vs time')
208     figure;
209     hold off
210     plot(bs, (times(:,2)))
211     hold on
212     plot(bs, (times(:,3)))
213     plot(bs, (times(:,3))-(ldltimes(:,1)))
214     hold off
215     legend(["quadprog interior-point time", "Our solver", "Our solver ...
            without factorization"])
216     ylabel("t [s]")
217      xlabel("n")
218      title("time vs n")
219 end
220
221 %% Large problem
222 %We look at the runtime of different algorithhms for a varying size ...
      problem
223 figure
224 largeProb = true;
225 n_large = 50;
226
227
228 rng(12)
229 times = zeros(20, 3);
230 bs = [];
231 iterations = zeros(20, 3);
232 i = 1;
```

```
233 solution = zeros(20,1);
234 ldltimes = zeros(20,1);
235
236 %Plot settings
237 plotIterations = true;
238 plotTimes = true;
239 plotSolution = true;
240 plotldlvstime = false;
241 plotcvx = false;
242
243 for b1 = linspace(8.5, 18.68, 20)
244
245     %Create a quadratic program
246     if ¬largeProb
247     n = 5;
248     H = [5.0 1.86 1.24 1.48 -0.46; ...
249          1.86 3.0 0.44 1.12 0.52; ...
250          1.24 0.44 3.8 1.56 -0.54; ...
251          1.48 1.12 1.56 7.2 -1.12; ...
252          -0.46 0.52 -0.54 -1.12 7.8];
253     g = [-16.1; -8.5; -15.7; -10.02; -18.68];
254     A = [16.1 8.5 15.7 10.02 18.68; 1.0 1.0 1.0 1.0 1.0]';
255     b = [b1;1];
256
257
258     l = zeros(5,1);
259     u = ones(5,1);
260     end
261
262     if largeProb
263         n = n_large*i;
264         [H, g, A, b] = generateRandomEQP(n,n/2);
265         l = zeros(n,1);
266         u = ones(n,1);
267     end
268
269    % Solve by CVX
270    if plotcvx
271    start = cputime;
272
273     cvx_begin quiet
274         %cvx_precision low
275         variable x(n)
276         minimize( 1/2 * x' * H *x + g'*x)
277         subject to
278             A' * x == b
279             l ≤ x
280             x ≤ u
281     cvx_end
282
283     times(i,1) = cputime-start;
284     iterations(i,1) = cvx_slvitr;
285    end
286
287     %solve by quadprog
```

```
288
289       options = optimset('Display', 'off');
290       options = optimset(options, 'Algorithm', 'interior-point-convex');
291       start = cputime;
292       [x2, optval, exitflag,output] = quadprog(H, g, [],[], A', b,l,u, ...
              0, options);
293
294       times(i,2) = cputime-start;
295       iterations(i,2) = output.iterations;
296
297
298
299       %solve by own solver
300       x0 = zeros(n,1);
301       s0 = ones(2*n,1);
302       y0 = ones(length(b),1);
303       z0 = ones(2*n,1);
304
305       start = cputime;
306       [x,y,z,s, iter, ldltime] = ...
              primalDualInteriorMethod_box(H,g,A,b,l,u,x0,y0,z0,s0);
307       times(i,3) = cputime-start;
308       iterations(i, 3) = iter;
309       ldltimes(i,1) = ldltime;
310
311       solution(i,1) = mean(sqrt((x-x2).^2));
312
313
314       bs = [bs;b1];
315       disp("Iteration " +i +"/" + 20);
316       disp("Mean error: " + mean(sqrt((x-x2).^2)))
317       disp("quadprog time: " + times(i,2))
318       disp("our time: " + times(i,3))
319       i = i+1;
320
321       if exitflag ~= 1
322           disp("Iteration with b(1)="+b1+" is infeasible!!")
323       end
324   end
325
326
327   % PLOT ITERATIONS
328   if largeProb
329       bs = n_large:n_large:(20*n_large);
330   end
331   if plotIterations
332       figure;
333       disp("Plotting # of iterations!")
334       if plotcvx
335           plot(bs, iterations(:,1))
336       end
337       hold on
338       plot(bs, iterations(:,2))
339       plot(bs, iterations(:,3))
340       hold off
```

```
341     if plotcvx
342         legend(["cvx iterations", "quadprog interior-point ...
                iterations", "Our solver"])
343     else
344         legend(["quadprog interior-point iterations", "Our solver"])
345     end
346
347     ylabel("iterations")
348     ylim([0 max(max(iterations))+2])
349     if largeProb
350          xlabel("n")
351          title("iterations vs n")
352     else
353         xlabel("b(1)")
354         title("iterations vs b(1)")
355     end
356
357 end
358
359 if plotTimes
360     disp("Plotting times!")
361     figure;
362     %Note: Time plotted is total wall time between starting and ending
363     %solver
364     % Not CPU time (since CPU time counts the number of cores used)
365     if largeProb
366         if plotcvx
367             plot(bs, (times(:,1)))
368         end
369         hold on
370         plot(bs, (times(:,2)))
371         plot(bs, (times(:,3)))
372         hold off
373         if plotcvx
374             legend(["cvx time", "quadprog interior-point time", "Our ...
                    solver"])
375         else
376             legend(["quadprog interior-point time", "Our solver"])
377         end
378
379         ylabel("t [s]")
380          xlabel("n")
381          title("time vs n")
382     else
383         if plotcvx
384             semilogy(bs, times(:,1))
385         end
386         hold on
387         semilogy(bs, times(:,2))
388         semilogy(bs, times(:,3))
389         hold off
390         if plotcvx
391             legend(["cvx time", "quadprog interior-point time", "Our ...
                    solver"])
392         else
```

```
393             legend(["quadprog interior-point time", "Our solver"])
394         end
395
396         ylabel("t [s]")
397         xlabel("b(1)")
398         title("time vs b(1)")
399     end
400 end
401
402 if plotSolution
403     figure;
404     disp("Plotting solution!")
405     plot(bs, solution)
406     title("Correctness")
407     ylabel("Mean Squared Error")
408     legend("Error between quadprog and our solver")
409     if largeProb
410         xlabel("n")
411     else
412         xlabel("b(1)")
413     end
414 end
415
416 if plotldlvstime
417     disp('plotting ldl vs time')
418     figure;
419     hold off
420     plot(bs, (times(:,2)))
421     hold on
422     plot(bs, (times(:,3)))
423     plot(bs, (times(:,3))-(ldltimes(:,1)))
424     hold off
425     legend(["quadprog interior-point time", "Our solver", "Our solver ...
            without factorization"])
426     ylabel("t [s]")
427     xlabel("n")
428     title("time vs n")
429 end
```

Listing A.17: Driver for exercise 2

A.4.4 Driver for exercise 3

```
1 %{
2 This is the driver for exercise 3.
3 This file contains:
4     - A test for efficiency and correctness under a growing problem using
5       a random LP generator with bounds. The size of every increment is
6       controlled by n_large and the number of iterations is controlled
7       by iter_test.
8 %}
```

```
%% Test efficiency and correctness
%Configure how many tests and how large an LP to test on
iter_test = 20;
n_large = 10;
sizes = n_large:n_large:(iter_test*n_large);
times = zeros(iter_test, 4);
iterations = zeros(iter_test, 4);
solution = zeros(iter_test,2);

%Plot settings
plotIterations = true;
plotTimes = true;
plotLogTimes = false;
plotSolution = true;
plotcvx = true;
plotsimplex = true;

if plotcvx
    plotsimplex = true;
end

%We test on a number of random LP problems of different sizes with our
%solver, cvx, and linprog both using simplex and interior-point.
for i = 1:iter_test

    %Create a quadrastart = cputime; program and solve it using cvx.
    n = n_large*i;
    [H, g, A, b] = generateRandomEQP(n,n/2);
    l = zeros(n,1);
    u = ones(n,1);



    if plotcvx
        start = cputime;
        cvx_begin quiet
            %cvx_precision low
            variable x(n)
            minimize(g'*x)
            subject to
                A' * x == b
                l <= x
                x <= u
        cvx_end

        times(i,1) = cputime-start;
        iterations(i,1) = cvx_slvitr;
    end

    start = cputime;
    options = optimset('Display', 'off');
    options = optimset(options, 'Algorithm', 'dual-simplex');
    [x2, optval, exitflag,output] = linprog(g, [],[], A', b,l,u, ...
        options);
    times(i,2) = cputime-start;
```

```
63      iterations(i,2) = output.iterations;
64
65      start = cputime;
66      options = optimset('Display', 'off');
67      options = optimset(options, 'Algorithm', 'interior-point');
68      [x3, optval, exitflag,output] = linprog(g, [],[], A', b,l,u, ...
            options);
69      times(i,3) = cputime-start;
70      iterations(i,3) = output.iterations;
71
72
73      x0 = zeros(n,1);
74      s0 = ones(2*n,1);
75      y0 = ones(length(b),1);
76      z0 = ones(2*n,1);
77
78      start = cputime;
79      [x,y,z,s, iter] = LinearPDIM_box(g,A,b,l,u,x0,y0,z0,s0);
80      times(i, 4) = cputime-start;
81      iterations(i, 4) = iter;
82
83      disp("Iteration " +i+"/"+iter_test);
84      disp("Mean error wrt simplex: " + mean(sqrt((x-x2).^2)))
85      disp("Mean error wrt interior: " + mean(sqrt((x-x3).^2)))
86
87      solution(i,1) = mean(sqrt((x-x2).^2));
88      solution(i,2) = mean(sqrt((x-x3).^2));
89
90      if exitflag ~= 1
91          disp("Iteration "+i+" is infeasible!!")
92      end
93  end
94
95
96  % PLOT ITERATIONS
97  if plotIterations
98      disp("Plotting # of iterations!")
99      figure;
100     if(plotcvx)
101         plot(sizes, iterations(:,1))
102         hold on
103     end
104     if(plotsimplex)
105         plot(sizes, iterations(:,2))
106         hold on
107     end
108     plot(sizes, iterations(:,3))
109     hold on
110     plot(sizes, iterations(:,4))
111     hold off
112     if(plotcvx)
113         legend(["cvx iterations", "linprog dual-simplex iterations", ...
                "linprog interior-point iterations", "our solver"])
114     elseif(plotsimplex)
```

```
115         legend(["linprog dual-simplex iterations", "linprog ...
                interior-point iterations", "our solver"])
116     else
117         legend([ "linprog interior-point iterations", "our solver"])
118     end
119     ylabel("iterations")
120     xlabel("n")
121     title("iterations vs n")
122
123 end
124
125 %Plot time spent for the different solvers
126 if plotTimes
127     disp("Plotting times!")
128     figure;
129     if(plotcvx)
130         plot(sizes, times(:,1))
131         hold on
132     end
133     plot(sizes, times(:,2))
134     hold on
135     plot(sizes, times(:,3))
136     plot(sizes, times(:,4))
137     hold off
138     if(plotcvx)
139         legend(["cvx time", "linprog dual-simplex time", "linprog ...
                interior-point time", "our solver"])
140     else
141         legend(["linprog dual-simplex time", "linprog interior-point ...
                time", "our solver"])
142     end
143     ylabel("t [s]")
144     xlabel("n")
145     title("time vs n")
146 end
147
148 if plotLogTimes
149     disp("Plotting times!")
150     clear log
151     figure;
152     if(plotcvx)
153         loglog(sizes, times(:,1))
154         hold on
155     end
156     loglog(sizes, times(:,2))
157     hold on
158     loglog(sizes, times(:,3))
159     loglog(sizes, times(:,4))
160     hold off
161     if(plotcvx)
162         legend(["cvx time", "linprog dual-simplex time", "linprog ...
                interior-point time", "our solver"])
163     else
164         legend(["linprog dual-simplex time", "linprog interior-point ...
                time", "our solver"])
```

```
165     end
166     ylabel("t [s]")
167     xlabel("n")
168     title("time vs n")
169 end
170
171 if plotSolution
172     disp("Plotting solution!")
173     figure;
174     plot(sizes, solution)
175     title("Correctness")
176     ylabel("Mean Squared Error")
177     xlabel("n")
178     legend(["Relative to Simplex","Relative to interior point"])
179 end
```

Listing A.18: Driver for exercise 3

A.4.5 Driver for exercise 4

```
1  close all
2  %%
3  %{
4  This is the driver for exercise 4.
5  This file contains:
6      - A contour plot of the Himmelblau problem (Exercise 4.4)
7      - A comparison of fmincon, CasADi and our own SQP-BFGS (Exercise 4.5)
8      - A test of the efficiency of subsolvers
9      - A test of the correctness of subsolvers
10     - A test of SQP BFGS
11     - A test of SQP BFGS Line Search
12     - A test of non-monotone strategy
13     - A test of SQP BFGS Line Search with infeasiblity handling
14     - A test of SQP BFGS Trust Region
15     - A test of SQP BFGS Trust Region from infeasible start
16
17 %}
18
19 %% Exercise 4.4, Himmelblau's test problem plotted
20
21 fig = figure('Position', [100, 0, 1000,1000]);
22 hold on
23 edges = 7;
24 %Plot the contours themselves
25 himmelblauContours(edges)
26
27 % Plot all the points of interest (max, min, saddle)
28 h1 = plot(-3.073025751,-0.08135304429,'d','MarkerFaceColor','r', ...
29       'MarkerEdgeColor','r','markersize',10);
30 h2 = plot(0.08667750456,2.884254701,'d','MarkerFaceColor','r', ...
31       'MarkerEdgeColor','r','markersize',10);
```

```
32 h3 = plot(3,2,'s','MarkerFaceColor','g', ...
33       'MarkerEdgeColor','r','markersize',15);
34 h4 = plot(-0.2983476136,2.895620844,'s','MarkerFaceColor','g', ...
35       'MarkerEdgeColor','r','markersize',15);
36 h5 = plot(-1.424243078,0.3314960331,'h','MarkerFaceColor','y', ...
37       'MarkerEdgeColor','r','markersize',15);
38 h6 = plot(-3.654605171,2.737718273,'s','MarkerFaceColor','g', ...
39       'MarkerEdgeColor','r','markersize',15);
40 h7 = plot(-3.548537388,-1.419414955,'s','MarkerFaceColor','g', ...
41       'MarkerEdgeColor','r','markersize',15);
42 h8 = plot(-0.4869360343,-0.1947744137,'h','MarkerFaceColor','y', ...
43       'MarkerEdgeColor','r','markersize',15);
44 h9 = plot(3.216440661,1.286576264,'h','MarkerFaceColor','y', ...
45       'MarkerEdgeColor','r','markersize',15);
46  legend([h8,h7,h2],{'Maximum', 'Minimum', 'Saddle Point'})
47
48 % show constraints as dark areas over contour
49 x1lim = edges; x1 = -x1lim:0.01:x1lim;
50 x2lim = edges; x2 = -x1lim:0.01:x2lim;
51 x1f = -edges:0.01:edges;
52 h1 = fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
53 h2 = patch('Faces',[1 2 3], 'Vertices', [5.348 x1lim; 4.324 -7; 100 ...
       -100], 'FaceColor', 'black', 'FaceAlpha', 0.5, 'EdgeColor', 'none');
54 h3 = patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim ...
       0.4*x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
55 h4 = patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim+7; x1lim ...
       0.4*x1lim + 7; x1lim +999], 'FaceColor', 'black', 'FaceAlpha', ...
       0.5, 'EdgeColor', 'none');
56 set( get( get( h4, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
57 set( get( get( h1, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
58 set( get( get( h2, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
59 set( get( get( h3, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
60
61 h5 = patch('Faces',[1 2 3 4], 'Vertices', [-edges -5; edges -5; edges ...
      -edges; -edges -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
62 h6 = patch('Faces',[1 2 3 4], 'Vertices', [-edges edges; -5 edges; -5 ...
      -edges; -edges -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
63 h7 = patch('Faces',[1 2 3 4], 'Vertices', [-edges edges; edges edges; ...
       edges 5; -edges 5], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
64 h8 = patch('Faces',[1 2 3 4], 'Vertices', [5 edges; edges edges; edges ...
      -edges; 5 -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
       'EdgeColor', 'none');
65 set( get( get( h5, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
66 set( get( get( h6, 'Annotation'), 'LegendInformation' ), ...
       'IconDisplayStyle', 'off' );
```

```
67  set( get( get( h7, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
68  set( get( get( h8, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
69  % set axis as needed
70  axis([-edges edges -edges edges])
71  title('Himmelblaus Test Problem')
72
73  %% Exercise 4.5, Comparison of fmincon, CasADi and our own SQP-BFGS
74  %Solve Himmelblau using fmincon, CasADi, and or own solver.
75  sympref('FloatingPointOutput',1);
76  l = [-5;-5];
77  u = [5;5];
78  cl = [0;0];
79  cu = [47;70];
80  addpath('C:\Users\anton\\OneDrive - Københavns Universitet\Uni\Uni\8. ...
        semester\Constrained optimization\Casadi')
81  import casadi.*
82
83  %Solve with fmincon
84  options = optimset('Display', 'off');
85  xfmin = fmincon(@objfminconHimmel, [0 0], [], [], [], [], l, u, ...
        @consfminconHimmel, options);
86
87  %Solve with our solver
88  options = struct('log',false, 'infesibility_handling', false, ...
        'method', 'SQP', 'subsolver', 'own solver');
89  x0 = [0;0];
90  [xown,~] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
91
92
93  x1 = SX.sym('x1');
94  x2 = SX.sym('x2');
95  nlp = struct ('x',[x1;x2], ...
        'f',(x1^2+x2-11)^2+(x1+x2^2-7)^2,'g',[(x1+2)^2-x2; -4*x1+10*x2]);
96
97  %Solve with CasADi
98  options = struct;
99  options.ipopt.print_level = 0;
100 options.print_time = 0;
101 S = nlpsol('S', 'ipopt', nlp,options);
102 r = S('x0', [0,0], 'lbg',0,'ubg',inf);
103 x_Cas =full(r.x);
104
105 %Print the solutions
106 fprintf('CasADi, solution: [%f,%f] \n', x_Cas(1), x_Cas(2));
107 fprintf('fmincon, solution: [%f,%f] \n', xfmin(1), xfmin(2));
108 fprintf('Own solver, solution: [%f,%f] \n', xown(1), xown(2));
109
110
111
112 %% Efficiency of subsolvers
113 %Compare quadprog and our internal solver stepwith growing size random ...
        EQP
114 % problems.
```

```
115 options = optimset('Display', 'off');
116 times = zeros(20, 2);
117 n_large = 50;
118
119 %Solve with both our and quadprog for several sizes
120 for i=1:20
121         n = n_large*i;
122         [H, g, A, b] = generateRandomEQP(n,n/2);
123         C = A;
124         dl = b-3;
125         du = b+3;
126         l = zeros(n,1);
127         u = ones(n,1);
128         x0 = zeros(n,1);
129
130         start = cputime;
131         [x,z,~,~] = intSQP(H,g,C,l,u,dl,du,x0);
132         times(i,1) = cputime-start;
133
134         start = cputime;
135         [x,~,~,~,z] = quadprog(H,g,-[C'; -C'],-[dl;-du],[],[],l,u,x0, ...
                options);
136         times(i,2) = cputime-start;
137
138         disp("Iteration " +i +"/" + 20);
139 end
140
141
142 %Plot subsolver time spent on subproblem
143 figure
144 hold on
145 h1 = plot(n_large:n_large:20*n_large,times(:,1),'r');
146 h2 = plot(n_large:n_large:20*n_large,times(:,2),'b');
147 legend([h1,h2],{'Own solver', 'quadprog'})
148 title('Comparison of underlying subsolvers')
149 ylabel('time[s]')
150 xlabel('n')
151
152 l = [-5;-5];
153 u = [5;5];
154 cl = [0;0];
155 cu = [47;70];
156
157 %% Test of the correctness of subsolvers
158 %Compare our subsolver to quadprog subsolver, compare - they take the ...
      same
159 %path
160 close all
161 sympref('FloatingPointOutput',1);
162 l = [-5;-5];
163 u = [5;5];
164 cl = [0;0];
165 cu = [47;70];
166 % Comparison of own solver and the quadprog
167 x0 = [0.0;0.0];
```

```
168 options = struct('log',true, 'infesibility_handling', false, 'method', ...
        'SQP', 'subsolver', 'own solver');
169 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
170 xHist = Hist.xHist;
171 xHistLatex_1 = latex(sym(round(xHist,3)));
172 time_own = Hist.timePerformence;
173 %Trace using our solver
174 himmelPlot(x0,x,xHist, 'SQP', 'own solver','b',5,'')
175
176
177 options = struct('log',true, 'infesibility_handling', false, 'method', ...
        'SQP', 'subsolver', 'quadprog');
178 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
179 xHist = Hist.xHist;
180 xHistLatex_2 = latex(sym(round(xHist,3)));
181 time_quad = Hist.timePerformence;
182 %Trace using quadprog
183 himmelTrace(x0,x,xHist, 'r', 'quadprog','—')
184
185 legend show
186
187
188 %% Test of SQP BFGS
189 %Test and plot four traces of SQP BFGS all with starting point near each
190 % their own minimum of the Himmelblau problem
191 close all
192 sympref('FloatingPointOutput',1);
193 l = [-5;-5];
194 u = [5;5];
195 cl = [0;0];
196 cu = [47;70];
197
198 % Several traces
199 options = struct('log',true, 'infesibility_handling', true, 'method', ...
        'SQP', 'subsolver', 'own solver');
200 x0 = [0;0];
201 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
202 Hist_1 = Hist;
203 xHist = Hist.xHist;
204 xHistLatex_1 = latex(sym(round(xHist,3)));
205 himmelPlot(x0,x,xHist, 'SQP', 'Trace 1')
206
207
208 x0 = [0;3.8416];
209 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
210 Hist_2 = Hist;
211 xHist = Hist.xHist;
212 xHistLatex_2 = latex(sym(round(xHist,3)));
213 himmelTrace(x0,x,xHist, 'r', 'Trace 2')
214
215
216 x0 = [-4;4];
217 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
218 Hist_3 = Hist;
219 xHist = Hist.xHist;
```

```
xHistLatex_3 = latex(sym(round(xHist,3)));
himmelTrace(x0,x,xHist, 'k', 'Trace 3')

x0 = [ -4.5;-1];
start = cputime;
[x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
endtime_4 = cputime-start;
functionCalls_4 = Hist.functionCalls;
Hist_4 = Hist;
xHist = Hist.xHist;
xHistLatex_4 = latex(sym(round(xHist,3)));
himmelTrace(x0,x,xHist, 'g', 'Trace 4')

legend show

%% Test of SQP BFGS Line Search
%Test and plot four traces of line search converging to each their own
%minimum.
close all
sympref('FloatingPointOutput',1);
l = [-5;-5];
u = [5;5];
cl = [0;0];
cu = [47;70];
% Several traces
options = struct('log',true, 'infesibility_handling', false, 'method', ...
    'SQP_ls', 'subsolver', 'own solver');
x0 = [0;0];
[x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
Hist_1 = Hist;
xHist = Hist.xHist;
xHistLatex_1 = latex(sym(round(xHist,3)));
stepHist_1 = latex(sym(round(Hist.stepLength,3)));
himmelPlot(x0,x,xHist, 'SQP linesearch', 'Trace 1')


x0 = [0;3.8416];
[x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
Hist_2 = Hist;
xHist = Hist.xHist;
xHistLatex_2 = latex(sym(round(xHist,3)));
stepHist_2 = latex(sym(round(Hist.stepLength,3)));
himmelTrace(x0,x,xHist, 'r', 'Trace 2')


x0 = [-4;4];
[x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
Hist_3 = Hist;
xHist = Hist.xHist;
xHistLatex_3 = latex(sym(round(xHist,3)));
stepHist_3 = latex(sym(round(Hist.stepLength,3)));
himmelTrace(x0,x,xHist, 'k', 'Trace 3')

x0 = [ -4.5;-1];
[x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
```

```
274 Hist_4 = Hist;
275 xHist = Hist.xHist;
276 xHistLatex_4 = latex(sym(round(xHist,3)));
277 stepHist_4 = latex(sym(round(Hist.stepLength,3)));
278 himmelTrace(x0,x,xHist, 'g', 'Trace 4')
279
280 legend show
281
282 %% Test of non-monotone strategy
283 %Compare non-monotone and monotone strategy for line search
284 close all
285 sympref('FloatingPointOutput',1);
286 l = [-5;-5];
287 u = [5;5];
288 cl = [0;0];
289 cu = [47;70];
290
291 %Non monotone line search
292 x0 = [ -4.5;-1];
293 options = struct('log',true, 'infesibility_handling', false, 'method', ...
        'SQP_ls', 'subsolver', 'own solver', 'non_monotone', true);
294 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
295 xHist = Hist.xHist;
296 Hist_1 = Hist;
297 steps1 = Hist.stepLength;
298 xHistLatex_1 = latex(sym(round(xHist,3)));
299 stepHist_1 = latex(sym(round(Hist.stepLength,3)));
300 himmelPlot(x0,x,xHist, 'SQP linesearch', 'With non monotone strategy')
301
302 %Monotone line search
303 options = struct('log',true, 'infesibility_handling', false, 'method', ...
        'SQP_ls', 'subsolver', 'own solver', 'non_monotone', false);
304 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
305 xHist = Hist.xHist;
306 Hist_2 = Hist;
307 steps2 = Hist.stepLength;
308 xHistLatex_2 = latex(sym(round(xHist,3)));
309 stepHist_2 = latex(sym(round(Hist.stepLength,3)));
310 himmelTrace(x0,x,xHist, 'r', 'Without non monotone strategy')
311
312
313 legend show
314
315
316 %% Test of SQP BFGS Line Search with infeasiblity handling
317 %Testing line search with infeasibility handling with just one trace.
318 close all
319 sympref('FloatingPointOutput',1);
320 l = [-5;-5];
321 u = [5;5];
322 cl = [0;0];
323 cu = [47;70];
324 % When infeasiblity handling is turned on, quadprog is the only solver ...
        for
325 %  the sub problem which is avalible.
```

```
326
327 x0 = [-9;6];
328 try
329     options = struct('log',true, 'infesibility_handling', false, ...
            'method', 'SQP_ls');
330     [x,z, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
331 catch
332     disp('We see that the subproblem is infeasible due to an ...
            infeasible linearization');
333 end
334 options = struct('log',true, 'infesibility_handling', true, 'method', ...
        'SQP_ls');
335 [x,z, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
336 xHist = Hist.xHist;
337 xHistLatex_1 = latex(sym(round(xHist,3)));
338 stepHist_1 = latex(sym(round(Hist.stepLength,3)));
339 himmelPlot(x0,x,xHist, 'SQP linesearch with infeasiblity handling', ...
        'Trace 1','r',9)
340 legend show
341
342 %% SQP BFGS Trust Region
343 %Four traces using Trust Region converging to each their own minimum
344 close all
345 sympref('FloatingPointOutput',1);
346 l = [-5;-5];
347 u = [5;5];
348 cl = [0;0];
349 cu = [47;70];
350 % When one uses the SQP based on a trust region, quadprog is the only ...
        solver for
351 %  the sub problem which is avalible.
352 options = struct('log',true, 'method', 'SQP_trust', 'trust_region',0.5);
353 x0 = [0;0];
354 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
355 xHist = Hist.xHist;
356 Hist_1 = Hist;
357 xHistLatex_1 = latex(sym(round(xHist,3)));
358 himmelPlot(x0,x,xHist, 'SQP trust region', 'Trace 1')
359
360
361 x0 = [0;3.8416];
362 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
363 xHist = Hist.xHist;
364 Hist_2 = Hist;
365 xHistLatex_2 = latex(sym(round(xHist,3)));
366 himmelTrace(x0,x,xHist, 'r', 'Trace 2')
367
368
369 x0 = [-4;4];
370 [x,z, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
371 xHist = Hist.xHist;
372 Hist_3 = Hist;
373 xHistLatex_3 = latex(sym(round(xHist,3)));
374 himmelTrace(x0,x,xHist, 'k', 'Trace 3')
375
```

```
376 x0 = [ -4.5;-1];
377 [x,¬, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
378 xHist = Hist.xHist;
379 Hist_4 = Hist;
380 xHistLatex_4 = latex(sym(round(xHist,3)));
381 himmelTrace(x0,x,xHist, 'g', 'Trace 4')
382
383 legend show
384
385 %% Test of SQP BFGS Trust Region from infeasible start
386 %Testing just one trace from infeasible start using trust region and
387 %infeasibility handling
388 close all
389 sympref('FloatingPointOutput',1);
390 l = [-5;-5];
391 u = [5;5];
392 cl = [0;0];
393 cu = [47;70];
394 % When infeasiblity handling is turned on, quadprog is the only solver ...
      for
395 %  the sub problem which is avalible.
396
397 x0 = [-2.805118; 3.131312];
398 options = struct('log',true, 'method', 'SQP_trust', 'trust_region',5, ...
      'penalty',1000);
399 [x,z, Hist] = SQPSolver(x0,@objHimmel,@consHimmel,l,u,cl,cu,options);
400 xHist = Hist.xHist;
401 xHistLatex_1 = latex(sym(round(xHist,3)));
402 himmelPlot(x0,x,xHist, 'SQP trust region with infeasible start', ...
      'Trace 1','r',9)
403 legend show
404
405
406 %% objective, constraints, derivitives and plotting functions
407 function [f,df] = objHimmel(x)
408
409 x1 = x(1);
410 x2 = x(2);
411
412 temp1 = x1^2+x2-11;
413 temp2 = x1+x2^2-7;
414
415 f = temp1^2+temp2^2;
416
417 df = zeros(2,1);
418 df(1) = 4*x1*temp1+2*temp2;
419 df(2) = 2*temp1+4*x2*temp2;
420
421 end
422
423 function [c,dc] = consHimmel(x,full,d)
424
425 if nargin<2
426     full = false;
427     d = 0;
```

```
428 end
429
430 if nargin<3 && full
431     error('One also needs to input d to det the full dc')
432 end
433
434
435 x1 = x(1);
436 x2 = x(2);
437
438 c = zeros(2,1);
439 c(1) = (x1+2)^2-x2;
440 c(2) = -4*x1+10*x2;
441
442
443
444
445 dc = zeros(2,2);
446 dc(1,1) = 2*x1+4;
447 dc(1,2) = -1;
448 dc(2,1) = -4;
449 dc(2,2) = 10;
450
451 dc = dc';
452
453 if full
454     c = [x; -x; c; -c];
455     c = c-d;
456
457     dc = [eye(2) -eye(2) dc -dc];
458 end
459
460 end
461
462 function [f] = objfminconHimmel(x)
463
464 x1 = x(1);
465 x2 = x(2);
466
467 temp1 = x1^2+x2-11;
468 temp2 = x1+x2^2-7;
469
470 f = temp1^2+temp2^2;
471
472 df = zeros(2,1);
473 df(1) = 4*x1*temp1+2*temp2;
474 df(2) = 2*temp1+4*x2*temp2;
475
476 end
477
478 function [c, ceq] = consfminconHimmel(x)
479
480 x1 = x(1);
481 x2 = x(2);
482
```

```
483 c = zeros(2,1);
484 c(1) = (x1+2)^2-x2;
485 c(2) = -4*x1+10*x2;
486 c(1) = -c(1);
487 c(2) = - c(2);
488 ceq = [0];
489
490 %dc = zeros(2,2);
491 %dc(1,1) = 2*x1+4;
492 %dc(1,2) = -1;
493 %dc(2,1) = -4;
494 %dc(2,2) = 10;
495
496 %dc = dc';
497
498 end
499
500
501 function [] = himmelTrace(x0,x,xHist, tracecolor, legendname,linepattern)
502 if nargin<6
503     linepattern = '-.';
504 end
505 hold on
506 h0 = plot(x0(1,:),x0(2,:),'k.','markersize',40);
507 h1 = plot(x(1,:),x(2,:),'r.','markersize',40);
508 plot(xHist(1,:),xHist(2,:),[tracecolor,linepattern],'linewidth',2,'DisplayName',legend
509 set( get( get( h0, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
510 set( get( get( h1, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
511 %legend(h2, legendname)
512 hold off
513 end
514
515
516 function [] = himmelPlot(x0,x,xHist, titlename,legendname, color, ...
        edges ,linepattern)
517 if nargin <6
518     color = 'b';
519 end
520 if nargin <7
521     edges = 5;
522 end
523 if nargin<8
524     linepattern = '-.';
525 end
526 fig = figure('Position', [100, 0, 1000,1000]);
527 hold on
528 himmelblauContours(edges)
529 h1 = plot(x0(1,:),x0(2,:),'k.','markersize',40);
530 h2 = plot(x(1,:),x(2,:),'r.','markersize',40);
531 plot(xHist(1,:),xHist(2,:),strcat(color,linepattern),'linewidth',2,'DisplayName',legen
532 %legend(h1, legendname)
533 set( get( get( h1, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
```

```
534 set( get( get( h2, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
535 % show constraints
536 x1lim = edges; x1 = -x1lim:0.01:x1lim;
537 x2lim = edges; x2 = -x1lim:0.01:x2lim;
538 x1f = -edges:0.01:edges;
539 h3 = fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
540 set( get( get( h3, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
541 h4 = patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim ...
        0.4*x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
542 set( get( get( h4, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
543
544 h5 = patch('Faces',[1 2 3 4], 'Vertices', [-edges -5; edges -5; edges ...
        -edges; -edges -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
545 h6 = patch('Faces',[1 2 3 4], 'Vertices', [-edges edges; -5 edges; -5 ...
        -edges; -edges -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
546 h7 = patch('Faces',[1 2 3 4], 'Vertices', [-edges edges; edges edges; ...
        edges 5; -edges 5], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
547 h8 = patch('Faces',[1 2 3 4], 'Vertices', [5 edges; edges edges; edges ...
        -edges; 5 -edges], 'FaceColor', 'black', 'FaceAlpha', 0.5, ...
        'EdgeColor', 'none');
548 set( get( get( h5, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
549 set( get( get( h6, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
550 set( get( get( h7, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
551 set( get( get( h8, 'Annotation'), 'LegendInformation' ), ...
        'IconDisplayStyle', 'off' );
552 % set axis as needed
553 axis([-edges edges -edges edges])
554 title(titlename)
555 hold off
556 end
```

Listing A.19: Driver for exercise 4

A.4.6 Solver interface for exercise 4

```
1 function [x,z,Hist] = SQPSolver(x0,obj,con,l,u,cl,cu, options)
2 %{
3 This is the solver inteface for all the SQP algorithms. The interface has
4  the input 'option' argument which functions as follows:
5     Options:
```

```
6       log - Do you want to log the process
7           values: true/false
8
9       method - Which SQP method
10          values: {'SQP'          (Plain vanilla SQP)
11                  ,'SQP_ls'       (Line search SQP)
12                  ,'SQP_trust'    (Trust region SQP)}
13
14      infesibility_handling - For infesibility handling SQP and SQP_ls
15          values: true/false
16
17      precision - spans from 10^-1 to 10^-9
18          values: Integer between 1 and 9
19
20      subsolver - For SQP and SQP_ls without infesibility handling there is
21                  a self made interior point algorithm.
22          values: 'own solver' or 'quadprog'
23
24      trust_region - For SQP trust region one can set the inital trust ...
            region
25          valuse: positive reals
26
27      non_monotone - Activate or deactivatethe non monotone strategy for
28                     the line search SQP
29          values: true/false
30
31      penalty - Set the inital penalty
32          values: reals above 100.
33
34
35  Created: 06.06.2021
36  Authors : Anton Ruby Larsen and Carl Frederik Grønvald
37            IMM, Technical University of Denmark
38  %}
39      if nargin <8
40          options = struct();
41      end
42      if sum(strcmp(fieldnames(options), 'log')) == 1
43          if islogical(options.log)
44              log = options.log;
45          else
46              error('The option log should be a boolean')
47          end
48      else
49          log = 0;
50      end
51      if sum(strcmp(fieldnames(options), 'method')) == 1
52          if ismember(options.method, {'SQP', 'SQP_ls', 'SQP_trust'})
53              method = options.method;
54          else
55              error('The option method should be one of {SQP, SQP_ls, ...
                    SQP_trust}')
56          end
57      else
58          method = 'SQP_ls';
```

```
59      end
60      if sum(strcmp(fieldnames(options), 'infesibility_handling')) == 1
61         if islogical(options.infesibility_handling)
62             infesibility_handling = options.infesibility_handling;
63         else
64             error('The options infesibility_handling should be a boolean')
65         end
66      else
67          infesibility_handling = true;
68      end
69      if sum(strcmp(fieldnames(options), 'precision')) == 1
70          if isnumeric(options.precision) && ...
                floor(options.precision)==options.precision && ...
                options.precision>0 && options.precision<6
71              precision = options.precision;
72          else
73              error('The option precision should be a positive integer ...
                    below 6.')
74          end
75      else
76          precision = 3;
77      end
78      if sum(strcmp(fieldnames(options), 'subsolver')) == 1
79          if ismember(options.subsolver , {'own solver', 'quadprog'})
80              if strcmp(options.subsolver,'own solver')
81                  %true is own solver
82                  subsolver = true;
83              else
84                  subsolver = false;
85              end
86          else
87              msg = ['The option subsolver should be either be ', ...
                    char(39), 'own solver', char(39) ,' or ', char(39), ...
                    'quadprog' ,char(39)];
88              error(msg)
89          end
90      else
91          subsolver = true;
92      end
93      if sum(strcmp(fieldnames(options), 'trust_region')) == 1
94          if isnumeric(options.trust_region) && options.trust_region>0
95              trust_region = options.trust_region;
96          else
97              error('The option trust_region should be a positive real.')
98          end
99      else
100         trust_region = 0.5;
101     end
102     if sum(strcmp(fieldnames(options), 'non_monotone')) == 1
103         if islogical(options.non_monotone)
104             non_monotone = options.non_monotone;
105         else
106             error('The option non_monotone should be a boolean.')
107         end
108     else
```

```
109             non_monotone = true;
110         end
111             if sum(strcmp(fieldnames(options), 'penalty')) == 1
112             if isnumeric(options.penalty) && options.penalty>100
113                 penalty = options.penalty;
114             else
115                 error('The option penalty should be a real over 100.')
116             end
117         else
118             penalty = 100;
119         end
120
121
122
123         if method == "SQP" && infesibility_handling==0
124             try
125             [x,z,Hist] = SQP(x0,obj,con,l,u,cl,cu,log,subsolver,precision);
126             catch
127                 close all
128                 error('The program is infeasible. Try with infeasibility ...
                        handling')
129             end
130
131         elseif method == "SQP_ls" && infesibility_handling==0
132             try
133             [x,z,Hist] = ...
                    SQP_ls(x0,obj,con,l,u,cl,cu,log,subsolver,precision, ...
                    non_monotone);
134             catch
135                 close all
136                 error('The program is infeasible. Try with infeasibility ...
                        handling')
137             end
138
139         elseif method == "SQP" && infesibility_handling==1
140             [x,z,Hist] = ...
                    SQP_infes(x0,obj,con,l,u,cl,cu,log,precision,penalty);
141
142         elseif method == "SQP_ls" && infesibility_handling==1
143             [x,z,Hist] = SQP_ls_infes(x0,obj,con,l,u,cl,cu,log,precision, ...
                    non_monotone,penalty);
144
145         elseif method == "SQP_trust"
146             [x,z,Hist] = SQP_trust(x0,obj,con,l,u,cl,cu,log,precision, ...
                    trust_region,penalty);
147
148         end
149
150 end
```

Listing A.20: Solver interface for exercise 4

A.4.7 Driver for exercise 5

```
close all

%%
%{
This is the driver for exercise 5.
This file contains:
    - Optimal portfolio and its risk for exercise 5.3
    - Computation of the efficient frontier and optimal portfolio as a
    function of the return
    - Solution of efficient frontier of the bi-criterion problem using ...
        both
    EQP and QP solvers
    - Computation of the efficient frontier and optimal portfolio as a
    function of return when we add a risk-free asset
    - Finding and plotting optimal point for R=14 with a risk-free asset
%}

%% Exercise 5.1-5.3
%Find the solution when going for R=12, and the accompanying risk
returns = [16.1, 8.5, 15.7, 10.02, 18.68];
R = 12;
covariance = [2.5 .93 .62 .74 -.23;
              0.93 1.5 0.22 0.56 0.26;
              .62 .22 1.9 .78 -0.27;
              .74 .56 .78 3.6 -0.56;
              -0.23 0.26 -0.27 -0.56 3.9];

H = covariance;
f = [];

A1 = returns;
b1 = R;

A2 = [1,1,1,1,1];
b2 = 1;

Aeq = [A1; A2];
beq = [b1; b2];

Aineq = -eye(5);
bineq = zeros(5,1);

x = quadprog( H, f, Aineq, bineq, Aeq, beq)

port_risk = x'*covariance*x

%% Exercise 5.4, Computing the efficient frontier

Rs = min(returns):0.01:max(returns);
port_risks = zeros(length(Rs),1);
options = optimset('Display', 'off');
```

```
51  optimalPorts = zeros(length(Rs),5);
52
53  %Compute the efficient frontier for each possible return
54  for i = 1:length(Rs)
55  beq = [Rs(i); b2];
56  x = quadprog( H, f, Aineq, bineq, Aeq, beq,[],[],[],options);
57  port_risks(i) = x'*covariance*x;
58  optimalPorts(i,:) = x;
59
60  end
61
62  %% Exercise 5.4, Plotting the efficient frontier
63  figure;
64  hold on
65
66  opt_return1 = Rs(port_risks==min(port_risks));
67  opt_risk1 = min(port_risks);
68
69  effRs = Rs(find(Rs == opt_return1):end);
70  effRisk = port_risks(find(Rs == opt_return1):end);
71
72  %Plot efficient frontier for each possible return
73  plot(Rs,port_risks,'b', returns, diag(covariance),'ro')
74  h2 = plot(opt_return1,opt_risk1 ,'ko','MarkerFaceColor', 'g');
75  h1 = plot(effRs,effRisk , 'r');
76  title('The Efficient Frontier')
77  xlabel('Return [%]')
78  ylabel('Risk [Var]')
79  legend([h1,h2],{'The efficient frontier','Minimum efficient return'}, ...
        'Location', 'northwest')
80  hold off
81
82  %% Exercise 5.4, Plotting the optimal port folio as a function of return
83  figure
84  h1 = plot(Rs,optimalPorts(:,1));
85  hold on
86  h2 = plot(Rs,optimalPorts(:,2));
87  h3 = plot(Rs,optimalPorts(:,3));
88  h4 = plot(Rs,optimalPorts(:,4));
89  h5 = plot(Rs,optimalPorts(:,5));
90  legend([h1,h2,h3,h4,h5], {'Security 1','Security 2','Security ...
        3','Security 4','Security 5'})
91  xlabel('Return [%]')
92  ylabel('Percentage of the portfolio [%]')
93  title('Portfolio composition')
94
95  %% Exercise 5.4, Minimum risk
96  % Find the portfolio with the smallest possible risk.
97  beq = [opt_return1; b2];
98  x_min = quadprog( H, f, Aineq, bineq, Aeq, beq,[],[],[],options)
99  port_risks_min = x'*covariance*x
100
101 %% Exercise 5.5-5.7, Bi-criterion Optimization
102 %Setup for bi-criterion
103 H = covariance;
```

```
f = -returns;

Aeq = [1,1,1,1,1];
beq = 1;

Aineq = -eye(5);
bineq = zeros(5,1);


%% Exercise 5.5-5.7, Solving for different alpha with and without ...
    short-selling
trials = 10;
x_short = zeros(trials,5,3);
x_nonshort = zeros(trials,5,3);

alphas = 1:1:trials;
alphas = alphas./(trials);
port_risk_short = zeros(trials,1,3);
port_risk_nonshort = zeros(trials,1,3);
port_return_short = zeros(trials,1,3);
port_return_nonshort = zeros(trials,1,3);

x0 = zeros(5,1);
s0 = ones(2*5,1);
y0 = ones(length(beq),1);
z0 = ones(2*5,1);
n = 5;
if trials<11
    cvx_solve = true;
else
    cvx_solve = false;
end

%optimal port. weights found by quadprog
for i = 1:trials
    if mod(i,trials/10)==0
        disp(i)
    end
    % Without shorting, i.e. an IQP problem
    x_nonshort(i,:,1) = quadprog( alphas(i).*H, (1-alphas(i)).*f', ...
        Aineq, bineq, Aeq, beq,[],[],[],options);
    x_nonshort(i,:,2) = primalDualInteriorMethod_box(alphas(i).*H, ...
        (1-alphas(i)).*f',Aeq',beq,zeros(5,1),ones(5,1),x0,y0,z0,s0);
    if cvx_solve
        cvx_begin quiet
            %cvx_precision low
            variable x(n)
            minimize( 1/2 * x' *  (alphas(i).*H) *x + ...
                ((1-alphas(i)).*f)*x)
            subject to
                Aeq * x == beq
                zeros(5,1) <= x
                x <= ones(5,1)
        cvx_end
        x_nonshort(i,:,3) = x;
```

```
155     end
156
157     % With shorting, i.e. an EQP problem
158     x_short(i,:,1) = quadprog( alphas(i).*H, (1-alphas(i)).*f', [], ...
            [], Aeq, beq,[],[],[],options);
159     x_short(i,:,2) = EqualityQPSolver(alphas(i).*H, ...
            (1-alphas(i)).*f',Aeq', beq, 'rangespace');
160     if cvx_solve
161         cvx_begin quiet
162             %cvx_precision low
163             variable x(n)
164             minimize( 1/2 * x' *  (alphas(i).*H) *x + ...
                    ((1-alphas(i)).*f)*x)
165             subject to
166                 Aeq * x == beq
167         cvx_end
168         x_short(i,:,3) = x;
169     end
170
171     port_risk_short(i,2) = x_short(i,:,2)*covariance*x_short(i,:,2)';
172     port_risk_nonshort(i,2) = ...
            x_nonshort(i,:,2)*covariance*x_nonshort(i,:,2)';
173
174     port_return_short(i,2) = -f*x_short(i,:,2)';
175     port_return_nonshort(i,2) = -f*x_nonshort(i,:,2)';
176 end
177
178
179 %Plot the MSE btwn our method, quadprog, and cvx, solving the EQP problem
180 figure
181 h1 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_short(:,:,1)-x_short(:,:,2)).^2),2)));
182 hold on
183 h2 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_short(:,:,3)-x_short(:,:,2)).^2),2)));
184 h3 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_short(:,:,3)-x_short(:,:,1)).^2),2)));
185 title('Comparison of found portfolios with shorting')
186 xlabel('alpha')
187 ylabel('Mean Squared Error, log10')
188 legend([h1,h2,h3],{'MSE log10, quadprog and our','MSE log10, CVX and ...
        our','MSE log10, CVX and quadprog'}, 'Location', 'northeast')
189
190 %And when solving the full QP problem
191 figure
192 h1 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_nonshort(:,:,1)-x_nonshort(:,:,2)).^2),2
193 hold on
194 h2 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_nonshort(:,:,3)-x_nonshort(:,:,2)).^2),2
195 h3 = ...
        plot((1:trials)./trials,log10(sum(sqrt((x_nonshort(:,:,3)-x_nonshort(:,:,1)).^2),2
196 title('Comparison of found portfolios with no shorting')
197 xlabel('alpha')
198 ylabel('Mean Squared Error, log10')
```

```
199 legend([h1,h2,h3],{'MSE log10, quadprog and our','MSE log10, CVX and ...
        our','MSE log10, CVX and quadprog'}, 'Location', 'northeast')
200
201
202 %% Exercise 5.5-5.7, Plot the efficient frontiers
203 %Plot efficient frontiers and portfolios when allowing or disallowing
204 %shorting, solving the Bicriterion
205 clear log
206
207 trials = 1000;
208 x_short = zeros(trials,5,3);
209 x_nonshort = zeros(trials,5,3);
210
211 alphas = 1:1:trials;
212 alphas = alphas./(trials);
213 port_risk_short = zeros(trials,1,3);
214 port_risk_nonshort = zeros(trials,1,3);
215 port_return_short = zeros(trials,1,3);
216 port_return_nonshort = zeros(trials,1,3);
217
218 x0 = zeros(5,1);
219 s0 = ones(2*5,1);
220 y0 = ones(length(beq),1);
221 z0 = ones(2*5,1);
222 n = 5;
223 if trials<11
224     cvx_solve = true;
225 else
226     cvx_solve = false;
227 end
228
229 %optimal port. weights found by quadprog
230 for i = 1:trials
231     if mod(i,trials/10)==0
232         disp(i)
233     end
234     % Without shorting, i.e. an IQP problem
235     x_nonshort(i,:,1) = quadprog( alphas(i).*H, (1-alphas(i)).*f', ...
            Aineq, bineq, Aeq, beq,[],[],[],options);
236     x_nonshort(i,:,2) = primalDualInteriorMethod_box(alphas(i).*H, ...
            (1-alphas(i)).*f',Aeq',beq,zeros(5,1),ones(5,1),x0,y0,z0,s0);
237     if cvx_solve
238         cvx_begin quiet
239             %cvx_precision low
240             variable x(n)
241             minimize( 1/2 * x' *  (alphas(i).*H) *x + ...
                    ((1-alphas(i)).*f)*x)
242             subject to
243                 Aeq * x == beq
244                 zeros(5,1) <= x
245                 x <= ones(5,1)
246         cvx_end
247         x_nonshort(i,:,3) = x;
248     end
249
```

```
250     % With shorting, i.e. an EQP problem
251     x_short(i,:,1) = quadprog( alphas(i).*H, (1-alphas(i)).*f', [], ...
            [], Aeq, beq,[],[],[],options);
252     x_short(i,:,2) = EqualityQPSolver(alphas(i).*H, ...
            (1-alphas(i)).*f',Aeq', beq, "rangespace");
253     if cvx_solve
254         cvx_begin quiet
255             %cvx_precision low
256             variable x(n)
257             minimize( 1/2 * x' *  (alphas(i).*H) *x + ...
                    ((1-alphas(i)).*f)*x)
258             subject to
259                 Aeq * x == beq
260         cvx_end
261         x_short(i,:,3) = x;
262     end
263
264     port_risk_short(i,2) = x_short(i,:,2)*covariance*x_short(i,:,2)';
265     port_risk_nonshort(i,2) = ...
            x_nonshort(i,:,2)*covariance*x_nonshort(i,:,2)';
266
267     port_return_short(i,2) = -f*x_short(i,:,2)';
268     port_return_nonshort(i,2) = -f*x_nonshort(i,:,2)';
269 end
270
271 %Plot the efficient frontier of the bi-criterion problem when shorting is
272 %allowed
273 figure
274 hold on
275 plot(port_return_short(:,2),port_risk_short(:,2),'b', returns, ...
        diag(covariance),'ro')
276 title('The Bi-Criterion when shorting is allowed')
277 xlabel('Return [%]')
278 ylabel('Risk [Var]')
279 hold off
280
281 %And the efficient frontier of the Bi-Criterion problem when shorting is
282 %not allowed
283 figure
284 hold on
285 plot(port_return_nonshort(:,2),port_risk_nonshort(:,2),'b', returns, ...
        diag(covariance),'ro')
286 title('The Bi-Criterion when shorting is not allowed')
287 xlabel('Return [%]')
288 ylabel('Risk [Var]')
289 hold off
290
291 % Plot comparing the solutions to the Bi-Criterion with no shorting and
292 % optimizing for risk given a return.
293 figure
294 hold on
295 h1 = plot(Rs,port_risks,'r');
296 plot(returns, diag(covariance),'ro')
297 h2 = plot(port_return_nonshort(:,2),port_risk_nonshort(:,2),'b');
298 plot(opt_return1,opt_risk1 ,'k|','MarkerSize', 14)
```

```
299 title('Comparison of static return and bi-criterion')
300 xlabel('Return [%]')
301 ylabel('Risk [Var]')
302 legend([h1,h2],{'Static return','Bi-criterion'}, 'Location', 'northwest')
303 hold off
304
305
306 %% Exercise 5.8-5.11, Introducing a risk free security with return 0
307 %Setup for the problem
308 returns = [16.1, 8.5, 15.7, 10.02, 18.68];
309 covariance = [2.5 .93 .62 .74 -.23;
310               0.93 1.5 0.22 0.56 0.26;
311               .62 .22 1.9 .78 -0.27;
312               .74 .56 .78 3.6 -0.56;
313               -0.23 0.26 -0.27 -0.56 3.9];
314
315 returns = [returns, 0];
316
317 covariance = [covariance, zeros(5,1)];
318 covariance = [covariance; zeros(1,6)];
319
320 H = covariance;
321 f = [];
322
323 A1 = returns;
324 b1 = min(A1);
325
326 A2 = [1,1,1,1,1,1];
327 b2 = 1;
328
329 Aeq = [A1; A2];
330 beq = [b1; b2];
331
332 Aineq = -eye(6);
333 bineq = zeros(6,1);
334 %% Exercise 5.8-5.11, Computing the efficient frontier with a ...
        risk-free asset
335
336 Rs_free = min(returns):0.01:max(returns);
337 port_risks_free = zeros(length(Rs_free),1);
338 optimalPorts = zeros(length(Rs),6);
339 %Just compute it with quadprog
340 for i = 1:length(Rs_free)
341 beq = [Rs_free(i); b2];
342 x = quadprog( H, f, Aineq, bineq, Aeq, beq,[],[],[],options);
343 port_risks_free(i) = x'*covariance*x;
344 optimalPorts(i,:) = x;
345 end
346
347 %% Exercise 5.8-5.11, Plotting the efficient frontier with a risk-free ...
        asset
348 %Then plot the efficient frontier with a risk-free asset
349 figure
350 hold on
351 plot(Rs_free,port_risks_free,'b', returns, diag(covariance),'ro')
```

```
352 title('The Efficient Frontier with a risk free asset')
353 xlabel('Return [%]')
354 ylabel('Risk [Var]')
355 hold off
356
357 lower_risk_pareto = ...
        port_risks(min(find(abs(round(port_risks_free(851:end)-port_risks,2))==0)));
358 lower_return_pareto = ...
        Rs(min(find(abs(round(port_risks_free(851:end)-port_risks,2))==0)));
359
360 figure
361 hold on
362 h1 = plot(Rs_free,port_risks_free,'b');
363 scatter( returns, diag(covariance),'ro')
364 h3 = plot(Rs,port_risks,'color',[0 0.5 0]);
365 plot(opt_return1, opt_risk1,'ko','MarkerFaceColor', 'k', 'MarkerSize',3);
366 plot(lower_return_pareto,lower_risk_pareto,'ko','MarkerFaceColor', ...
        'k', 'MarkerSize',3);
367 h2 = plot(effRs,effRisk , 'r');
368 hold off
369 title('The Efficient Frontiers with and without a risk free asset')
370 xlabel('Return [%]')
371 ylabel('Risk [Var]')
372 legend([h1,h2, h3],{'With risk free asset','Without risk free asset ...
        and pareto optimal','Without risk free asset and not pareto ...
        optimal'}, 'Location', 'northwest')
373
374 %% Exercise 5.8-5.11, Portfolio compositions
375 %Plot the optimal portfolio when we have a risk-free security
376 figure
377 h1 = plot(Rs_free,optimalPorts(:,1));
378 hold on
379 h2 = plot(Rs_free,optimalPorts(:,2));
380 h3 = plot(Rs_free,optimalPorts(:,3));
381 h4 = plot(Rs_free,optimalPorts(:,4));
382 h5 = plot(Rs_free,optimalPorts(:,5));
383 h6 = plot(Rs_free,optimalPorts(:,6));
384 legend([h1,h2,h3,h4,h5, h6], {'Security 1','Security 2','Security ...
        3','Security 4','Security 5', 'Risk free asset'})
385 xlabel('Return [%]')
386 ylabel('Percentage of the portfolio [%]')
387 title('Portfolio composition')
388
389
390 %% Exercise 5.8-5.11, Finding the optimum for R=14
391
392 beq = [14; b2];
393 x = quadprog( H, f, Aineq, bineq, Aeq, beq);
394 opt_risk2 = x'*covariance*x;
395
396
397 %% Exercise 5.8-5.11, Ploting
398 %Plot the location R=14 with and without a risk-free asset, so we can
399 %compare
400 figure
```

```
hold on
opt_return2 = 14;
plot(Rs_free,port_risks_free,'b', returns, diag(covariance),'mo')
plot(Rs,port_risks,'color',[0 0.5 0])
h2 = plot(effRs,effRisk , 'r');
p1 = plot(opt_return2, opt_risk2,'ko','MarkerFaceColor', 'g', ...
    'MarkerSize',4);
plot(opt_return1, opt_risk1,'k|','MarkerFaceColor', 'k', 'MarkerSize',8);
p2 = plot(Rs(find(Rs == 14)), port_risks(find(Rs == ...
    14)),'ko','MarkerFaceColor', 'k', 'MarkerSize',4);
title('Comparison of Efficient Frontiers')
xlabel('Return [%]')
ylabel('Risk [Var]')
legend([p1,p2],{'E[R]=14 with risk free asset','E[R]=14 without risk ...
    free asset'},'Location','northwest')
hold off
```

Listing A.21: Driver for exercise 5

A.5 Extra function

A.5.1 A random EQP generator

```
function [H,g,A,b,x,lambda] = generateRandomEQP(n,m)
% generateRandomEQP    Generate a random EQP with n variables and m
%                      constraints
%
%
% Syntax: [H,g,A,b,x,lambda] = generateRandomEQP(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: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

%%
    % Create a symmetric and pos def Hessian
    H = rand(n);
    H = 0.5*(H+H') + n*eye(n);

    % Create a full rank A matrix
    A = 10*rand(n);
    A = 0.5*(A+A')+n*eye(n);
    A = A(:,1:m);
```

```
28
29      % Create a random solution to the system
30      x = rand(n,1);
31      lambda = rand(m,1);
32
33      % Create the matching g and b
34      KKT = [H -A;-A' zeros(m)];
35      sol = [x; lambda];
36      rhs = KKT*sol;
37
38      g = -rhs(1:n);
39      b = -rhs(n+1:n+m);
40  end
```

Listing A.22: A progam which generates random EQPs

A.5.2 Generate an EQP Recycling problem

```
1  function [H,g,A,b] = ProblemEQPRecycling(n, uhat, d0)
2  % ProblemEQPRecycling   Generate an instance of the recycling problem
3  %                       described in exercise 1.5
4  %
5  %
6  % Syntax: [H,g,A,b] = ProblemEQPRecycling(n, uhat, d0)
7  %
8  %         H               : Hessian
9  %         g               : Linear term of the objective
10 %         A               : Matrix of the constraints
11 %         b               : lhs of the constraints
12
13 % Created: 06.06.2021
14 % Authors : Anton Ruby Larsen and Carl Frederik Grønvald
15 %           IMM, Technical University of Denmark
16
17 %%
18      H = eye(n+1);
19      g = uhat*ones(n+1,1);
20      A = eye(n);
21      sub = eye(n);
22      sub = circshift(sub,1,2);
23      A = A - sub;
24      A = padarray(A,[0 1],0,'post');
25      A(end-1,end) = -1;
26      A(end-1,end-1) = -1;
27      A(end-1,end-2) = 1;
28      A = A';
29      b = zeros(n,1);
30      b(n) = -d0;
31  end
```

Listing A.23: A progam which generates EQP Recycling problems

A.5.3 Generate KKT matrix

```
function [KKT] = get_KKT(H,g,A,b)
% get_KKT    Generate a KKT matrix
%
%
% Syntax: [KKT] = get_KKT(H,g,A,b)
%
%          KKT              : The KKT matrix

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

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

Listing A.24: Generate KKT matrix

A.5.4 Generate sparse KKT matrix

```
function [KKT] = get_KKT_sparse(H,g,A,b)
% get_KKT_sparse    Generate a sparse KKT matrix
%
%
% Syntax: [KKT] = get_KKT_sparse(H,g,A,b)
%
%          KKT              : The sparse KKT matrix

% Created: 06.06.2021
% Authors : Anton Ruby Larsen and Carl Frederik Grønvald
%           IMM, Technical University of Denmark

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

Listing A.25: Generate sparse KKT matrix

A.6 The exam assignment

02612 Constrained Optimization 2021
Exam Assignment
Hand-in deadline: June 7, 2021, 12:30

John Bagterp Jørgensen

April 6, 2021

1 Equality Constrained Convex QP

In this problem, we consider the equality constrained convex QP

$$\min_x \quad \phi = \frac{1}{2}x'Hx + g'x \tag{1.1a}$$
$$s.t. \quad A'x = b \tag{1.1b}$$

with $H \succ 0$.

1. What is the Lagrangian function for this problem?
2. What is the first order necessary optimality conditions for this problem? Are they also sufficient and why?
3. Implement solvers for solution of the problem (1.1) that are based on an LU-factorization (dense), LU-factorization (sparse), LDL-factorization (dense), LDL-factorization (sparse), a range-space factorization, and a null-space factorization. You must provide pseudo-code and source code for your implementation. The solvers for the individual factorizatons must have the interface `[x,lambda]=EqualityQPSolverXX(H,g,A,b)` where `XX` can be e.q. `LUdense`, `LUsparse`, etc. You must make a system that can switch between the different solvers as well. It should have an interface like `[x,lambda]=EqualityQPSolver(H,g,A,b,solver)`, where `solver` is a flag used to switch between the different factorizations.

4. Test your algorithms on the test problem with the data

```
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 =
15
1
```

   Compute the solution for different values of `b(1)` in the range `[8.5 18.68]`.

5. Test your implementation on a size dependent problem structure and report the results. You are free to chose the problems that you want to use for testing your algorithm.

2 Quadratic Program (QP)

We consider the quadratic program (QP) in the form (assume that A has full column rank)

$$\min_x \quad \phi = \frac{1}{2}x'Hx + g'x \tag{2.1a}$$

$$s.t. \quad A'x = b \tag{2.1b}$$

$$l \leq x \leq u \tag{2.1c}$$

1. What is the Lagrangian function for this problem (2.1)?
2. Write the nesessary and sufficient optimality conditions for this problem (2.1).
3. Write pseudo-code for a primal-dual interior-point algorithm for solution of this problem (2.1). Explain each major step in your algorithm.
4. Implement the primal-dual interior-point algorithm for (2.1) and test it. You must provide commented code as well as driver files to test your code, documentation that it works, and performance statistics.
5. Compare the performance of your primal-dual interior-point algorithm and `quadprog` from Matlab (or equivalent QP library functions). Provide scripts that demonstrate how you compare the software and comment on the tests and the results.
6. Consider a QP in the form (2.1). Use H, g, A and b from problem 1. Let `l=zeros(5,1)` and `u = ones(5,1)`. Compute the solution for different values of `b(1)` in the range `[8.5 18.68]`. Test the primal-dual interior-point QP algorithm and the library QP algorithm e.g. `quadprog` and `cvx` for this problem. Plot the solution as well as solution statistics (number of iterations, cpu time, etc).

3 Linear Program (LP)

In this problem we consider a linear program in the form (assume that A has full column rank)

$$\min_x \quad \phi = g'x \tag{3.1a}$$

$$s.t. \quad A'x = b \tag{3.1b}$$

$$l \leq x \leq u \tag{3.1c}$$

1. What is the Lagrangian function for this problem (3.1)?
2. Write the nesessary and sufficient optimality conditions for this problem (3.1).
3. Write pseudo-code for a primal-dual interior-point algorithm for solution of this problem (3.1). Explain each major step in your algorithm.
4. Implement the primal-dual interior-point algorithm and test it. You must provide commented code as well as driver files to test your code, documentation that it works, and performance statistics.
5. Compare the performance of your primal-dual interior-point algorithm and `linprog` from Matlab (or equivalent LP library functions). Provide scripts that demonstrate how you compare the software and comment on the tests and the results. You should solve the problem using your primal-dual interior-point algorithm and a library algorithm e.g. `linprog` and `cvx`.

4 Nonlinear Program (NLP)

We consider a nonlinear program in the form

$$\min_x \quad f(x) \tag{4.1a}$$

$$s.t. \quad g_l \leq g(x) \leq g_u \tag{4.1b}$$

$$x_l \leq x \leq x_u \tag{4.1c}$$

We assume that the involved functions are sufficiently smooth for the algorithms discussed in this course to work. Assume that $\nabla g(x)$ has full column rank.

1. What is the Lagrangian function for the nonlinear program (4.1)?
2. What is the necessary first order optimality conditions for the nonlinear program (4.1)?
3. What are the sufficient second order optimality conditions for the nonlinear program (4.1)?
4. Consider Himmelblau's test problem. Convert this problem into the form (4.1). Provide the contour plot of the problem and locate all stationary points.
5. Solve the test problem using a library function for nonlinear programs, e.g. `fmincon` in Matlab and `CasADi`.
6. Explain, discuss and implement an SQP procedure with a damped BFGS approximation to the Hessian matrix for the problem (4.1). Make a table with the iteration sequence for different starting points. Plot the iteration sequence in a contour plot. Discuss the results.
7. Explain, discuss and implement the SQP procedure with a damped BFGS approximation to the Hessian matrix and line search for the problem (4.1). Make a table with the iteration sequence. Make a table with relevant statistics (function calls etc). Plot the iteration sequence in a contour plot. Discuss the results.
8. Explain, discuss, and implement a Trust Region based SQP algorithm for this problem (4.1). Make a table with the iteration sequence. Make a table with relevant statistics (function calls etc). Plot the iteration sequence in a contour plot. Discuss the results

5 Markowitz Portfolio Optimization

This exercise illustrates use of quadratic programming in a financial application. By diversifying an investment into several securities it may be possible to reduce risk without reducing return. Identification and construction of such portfolios is called hedging. The Markowitz Portofolio Optimization problem is very simple hedging problem for which Markowitz was awarded the Nobel Price in 1990.

Consider a 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

Optimal solution as function of return

1. For a given return, R, formulate Markowitz' Portfolio optimization problem as a quadratic program.
2. What is the minimal and maximal possible return in this financial market?
3. Compute a portfolio with return, $R = 12.0$, and minimal risk. What is the optimal portfolio and what is the risk (variance)?
4. Compute the efficient frontier, i.e. the risk as function of the return. Plot the efficient frontier as well as the optimal portfolio as function of return.

Bi-criterion optimization

1. Formulate the optimization problem as a bi-criterion of the variance and the return, i.e. an objective function in the form $\phi = \alpha V\{\boldsymbol{R}\} - (1-\alpha)E\{\boldsymbol{R}\}$, where $\boldsymbol{R}$ is the stochastic portfolio return.
2. Compute the solution using your algorithms from Problem 1 and 2 for different values of α in the interval $[0, 1]$. Plot the solutions and also report the solver statistics.
3. Compare the solution to the solution obtained using `quadprog` and `cvx`. Also compare the solver statistics to the solver statistics for your own algorithm (Problem 2).

Risk-free asset
In the following we add a risk free security to the financial market. It has return $r_f = 0.0$.

1. What is the new covariance matrix and return vector.

2. Compute the efficient frontier, plot it as well as the (return,risk) coordinates of all the securities. Comment on the effect of a risk free security. Plot the optimal portfolio as function of return.

3. What is the minimal risk and optimal portfolio giving a return of $R = 14.00$. Plot this point in your optimal portfolio as function of return as well as on the efficient frontier diagram.

4. Discuss the solution and an appropriate solver for the problem.

Report

You are allowed to work on the assignment in groups. You must hand in an individual report that you write yourself for the assignment. The following must be uploaded to CampusNet: 1) one pdf file of the report, 2) one zip-file containing all Matlab and Latex code etc used to prepare the report. In addition you must print the pdf file of your report and hand it in to may mail box in Building 303B Room 112 (in case DTU is open by June 2021).

Labels, fontsize, and visibility of all figures must be made in a professional manner. Include key matlab code in the report (use syntax high lighting - and print the report in color), and provide all matlab code in the appendix that you can refer to. The report should include a desciption and discussion of the mathematical methods and algorithms that you use, as well as a discussion of the results that you obtain. We want you to demonstrate that you can critically reflect on the methods used, their properties, and the results that you obtain.

The deadline for handing in the report is Monday June 7, 2021 at 12:30.

Bibliography

[BJ19] Dimitri Boiroux and John Bagterp Jørgensen. "Sequential l_1 Quadratic Programming for Nonlinear Model Predictive Control." In: (2019).

[Bro] Dr Richard Brown. *UC math 352, University of Canterbury*. https://www.youtube.com/playlist?list=PLh464gFUoJWOmBYla3zbZbc4nv2AXez6X.

[HV07] Esben Lundsager Hansen and Carsten Völcker. "Numerical Algorithms for Sequential Quadratic Optimization." Technical University of Denmark, 2007.

[Jør21] John Bagterp Jørgensen. *Numerical Methods for Constrained Optimization*. first. Springer, 2021.

[MN21] Joaquim R. R. A. Martins and Andrew Ning. *Engineering Design Optimization*. first. 2021.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.

[SY10] Klaus Schittkowski and Ya-xiang Yuan. "Sequential Quadratic Programming Methods." In: (June 2010).

[Wri97] Stephen J. Wright. *Primal-Dual Interior-Points Methods*. Philadelphia, Pa, USA: SIAM, 1997.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ