DTU Compute
Department of Applied Mathematics and Computer Science

Exam Assignment

02612 Constrained Optimization 2022

Sara Húnfjörð Jósepsdóttir (s212952)

Kongens Lyngby 2022

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

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

Preface

The code in this exam assignment was done in collaboration with Helga Þórey Björnsdóttir (s212952), Helena Bugge Nicolaisen (s173715), Joachim Pors Andreasen (s184289) and Olgeir Ingi Árnason (s212564).

Kongens Lyngby, May 30, 2022

Sara Húnfjörð Jósepsdóttir (s212952)

Contents

CHAPTER 1

Equality Constrained Convex Quadratic Program

In this chapter, the equality constrained convex quadratic program (EQP) is considered:

$$\min_x \ \phi = \frac{1}{2}x^T Hx + g^T x \tag{1.1a}$$

$$s.t. \ \ A^T x = b \tag{1.1b}$$

with $H \succ 0$.

1.1 The Lagrange Function

A very useful method for solving minimization and maximization problems in constrained optimization is the Lagrange function. It is defined as:

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

where $f(x)$ is the objective function, and each constraint $c_i(x)$, has an associated Lagrange multiplier, λ_i, with $i \in \mathcal{E} \cup \mathcal{I}$. The $\mathcal{E}$ and $\mathcal{I}$ are finite sets of indices representing the inequality and equality constraints [1]. The Lagrange function for problem 1.1 can thus be formulated as:

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

1.2 First Order Optimality Conditions

Necessary conditions for a constrained optimal solution are the Karush-Kuhn-Tucker (KKT) conditions. They are equivalent to the first order conditions for unconstrained optimization problems according to Proposition 2.10 in lecture notes [1] and are stated as:

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

$$c_i(x) = 0 \qquad i \in \mathcal{E} \tag{1.4b}$$

$$c_i(x) \geq 0 \qquad i \in \mathcal{I} \tag{1.4c}$$

$$\lambda_i \geq 0 \qquad i \in \mathcal{I} \tag{1.4d}$$

$$c_i(x) = 0 \ \vee \ \lambda_i = 0 \qquad i \in \mathcal{I} \tag{1.4e}$$

where (1.4e) is called the complementary condition. Now, as (1.3) is an equality constraint problem, only equality constraints are considered and thus (1.4a) and (1.4b) are only relevant. From formulation (1.3), the KKT conditions for the problem can then be stated as:

$$\nabla_x \mathcal{L}(x,\lambda) = Hx + g^T - \lambda^T A = 0, \tag{1.5a}$$

$$\begin{aligned} c(x) &= \nabla_\lambda \mathcal{L}(x,\lambda) \\ &= -A^T x + b = 0. \end{aligned} \tag{1.5b}$$

Seeing as (1.5b) is the partial derivative of (1.3) with respect to λ it can be concluded that the first order conditions of an EQP are satisfied at all stationary points of the Lagrangian. This is due to the fact that the Lagrangian is continuously differentiable in an open neighbourhood of x, see Lecture 2A [2].

A convex function is essentially a function that given a line segment between any two points on the graph, it does not lie below the graph between the two points [3]. The function therefore contains only one minimum and no saddle points. This minimum is not only local but global as is stated in Section 2.5 in the lecture notes [1]. It can therefore be concluded that the first order conditions are both necessary and sufficient for a convex EQP.

1.3 Solvers for an Equality Constrained Quadratic Program

The KKT conditions for the solution $x \in \mathbb{R}^n$ of the EQP in (1.1) give rise to the following linear system of equations:

$$\underbrace{\begin{bmatrix} H & -A^T \\ -A & 0 \end{bmatrix}}_{=:\ \mathrm{K}} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix}. \tag{1.6}$$

where $\lambda \in \mathbb{R}^m$ is the associated Lagrange multiplier and K is called the KKT matrix. For a convex EQP $H \in \mathbb{R}^{n \times n}$ is symmetric and positive definite and thus K is also symmetric. The KKT matrix is therefore symmetric indefinite as it has both positive and negative eigenvalues for $H \succ 0$. Additionally, if the problem has more than 1 constraint ($m \geq 1$), the KKT matrix is always indefinite [4]. In the perceding sections, four methods are implemented to solve the KKT system in (1.6) and their performances compared.

1.3.1 LU Factorization

LU factorization uses Gauss elimination with partial pivoting to obtain the L and U factors where L is the lower triangular and U is the upper triangular of the matrix. This makes computation faster and easier. Although, the disadvantage with this approach is that it ignores the symmetry of the KKT matrix and has numeric instability. The algorithm to solve for x and λ in (1.6) is listed in Algorithm 1. The matlab implementation of the EQP solvers using dense and sparse LU factorization are listed in Listing 1.2 and Listing 1.1.

Algorithm 1: An algorithm for solving an EQP using LU factorization.

Data: H, g, A, b
Result: x, λ

1 Set $KKT = \begin{bmatrix} H & -A^T \\ -A & 0 \end{bmatrix}$
2 Compute $[L, U] = lu(KKT)$ `// LU factorize the KKT matrix`
3 Compute $\begin{bmatrix} x \\ \lambda \end{bmatrix} = -(\begin{bmatrix} g \\ b \end{bmatrix} /L)/U$

```
function [x, lambda] = EqualityQPSolverLUdense(H,g,A,b)
% EqualityQPSolverLUdense Solver for a convex equality constrained QP
% using LU factorization
%
%            min 0.5*x'H*x + g'x
```

```
%             x
%           s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverLUdense(H,g,A,b)

%  Solves equality constrained QP using LU factorization
% and treating the system as a dense matrix.

[n,m] = size(A);

KKTmat = [H, -A; -A', zeros(m)];

rhs = [-g; -b];

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

% Back substitute
XandLambda = U\(L\rhs(p));

% Extract x and lambda
x = XandLambda(1:n,1);
lambda = XandLambda(n+1:n+m,1);

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

% Solve f o r x and lambda
rhs = -[g ; b] ;
solution = U \ (L \ ( rhs (p) ) ) ;
x = solution(1:size(H,1)) ;
lambda = solution(size(H,1)+1: size(H,1)+ size(b,1)) ;

end
```

Listing 1.1: A sparse EQP solver using LU factorization.

```
function [x, lambda] = EqualityQPSolverLUsparse(H,g,A,b)
% EqualityQPSolverLUsparse Sparse solver for a convex equality constrained
% QP using LU factorization
%
%           min 0.5*x'H*x + g'x
%            x
%           s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverLUsparse(H,g,A,b)

%  Solves equality constrained QP using LU factorization
% and treating the system as a sparse matrix.

[n,m] = size(A);


KKTmat = [H, -A; -A', zeros(m)];
```

```
KKTmat = sparse(KKTmat);

rhs = [-g; -b];

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

% Back substitute
XandLambda = U\(L\rhs(p,1));

% Extract x and lambda
x = XandLambda(1:n,1);
lambda = XandLambda(n+1:n+m,1);

end
```

Listing 1.2: A dense EQP solver using LU factorization.

1.3.2 LDL factorization

LDL factorization is a better approach than the LU factorization as it only requires the matrix to be symmetric. The factorization decomposes the matrix into a unit lower triangular L, a block-diagonal D and an upper triangular component L^T:

$$P^T K P = L D L^T,$$

the symmetric permutations represented in P makes the computation numerically stable and maintains sparsity in the case of a large sparse KKT matrix. Additionally, the computational cost of the symmetric indefinite factorization is about half the cost of sparse LU decomposition [4]. The algorithm to solve for x and λ in (1.6) is listed in Algorithm 2. The matlab implementation of the EQP solvers using dense and sparse LU factorization are listed in Listing 1.3 and Listing 1.4.

Algorithm 2: An algorithm for solving an EQP using LDL factorization.

Data: H, g, A, b
Result: x, λ

1 Set $KKT = \begin{bmatrix} H & -A^T \\ -A & 0 \end{bmatrix}$
2 Compute $[L, D] = ldl(KKT)$ `// LDL factorize the KKT matrix`
3 Compute $\begin{bmatrix} x \\ \lambda \end{bmatrix} = -\big(\big(\begin{bmatrix} g \\ b \end{bmatrix} / L\big)/D\big)/L^T$

```
function [x,lambda] = EqualityQPSolverLDLdense(H,g,A,b)
% EqualityQPSolverLDLdense Solver for convex equality constrained QP
% using LDL factorization
%
```

```
%             min 0.5*x'H*x + g'x
%              x
%             s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverLDLdense(H,g,A,b)

[n, m] = size(A);

% Define KKT system
KKTmat = [H, -A; -A' zeros(m)];
rhs = [-g; -b];

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

% Back substitute
XandLambda(p,1) = L'\(D\(L\rhs(p,1)));

% Extract x and lambda
x = XandLambda(1:n,1);
lambda = XandLambda(n+1:n+m,1);

end
```

Listing 1.3: A dense EQP solver using LDL factorization.

```
function [x,lambda] = EqualityQPSolverLDLsparse(H,g,A,b)
% EqualityQPSolverLDLsparse Solver for a convex equality constrained QP
% using LDL factorization
%
%             min 0.5*x'H*x + g'x
%              x
%             s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverLDLsparse(H,g,A,b)

[n, m] = size(A);
KKTmat = sparse([H, -A; -A' zeros(m)]);

rhs = [-g; -b];

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

% Back substitute
XandLambda(p,1) = L'\(D\(L\rhs(p,1)));

% Extract x and lambda
x = XandLambda(1:n,1);
lambda = XandLambda(n+1:n+m,1);

end
```

Listing 1.4: A sparse EQP solver using LDL factorization.

1.3.3 The Range-Space Method

The range-space method uses Cholesky factorization to factorize the H matrix and therefore requires the Hessian matrix to be positive definite. It is useful when the Hessian is easily invertible and well-conditioned. The procedure is most efficient when the number of equality constraints, m, is small compared to the number of variables, n [4]. Algorithm 3 summarizes the range-space method for solving an EQP [5]. The matlab code is listed in Listing 1.5.

Algorithm 3: An algorithm for solving an EQP using the range-space method.

Data: H, g, A, b
Result: x, λ

1 Compute $L = chol(H)$ `// Cholesky factorize H`
2 Compute K by solving $LK = A$
3 Compute w by solving $Lw = g$
4 Compute $H = K^T K$
5 Compute $z = K^T w + b$
6 Compute $M = chol(H)$ `// Cholesky factorize H`
7 Compute q by solving $Mq = z$
8 Compute λ by solving $M^T \lambda = q$
9 Compute $r = K\lambda - w$
10 Compute x by solving $L^T x = r$

```
function [x,lambda] = EqualityQPSolverLDLsparse(H,g,A,b)
% EqualityQPSolverLDLsparse Solver for a convex equality constrained QP
% using LDL factorization
%
%            min 0.5*x'H*x + g'x
%             x
%            s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverLDLsparse(H,g,A,b)

[n, m] = size(A);
KKTmat = sparse([H, -A; -A' zeros(m)]);

rhs = [-g; -b];

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

% Back substitute
XandLambda(p,1) = L'\(D\(L\rhs(p,1)));

% Extract x and lambda
x = XandLambda(1:n,1);
```

```
lambda = XandLambda(n+1:n+m,1);

end
```

Listing 1.5: A sparse EQP solver using the range-space method

1.3.4 The Null-Space Method

The null-space method solves the KKT system (1.6) using the null space of $A \in \mathbb{R}^{n\times m}$. The method does not require the Hessian, H, to be positive definite as in the range-space method but only positive semi-definite. It is therefore not restricted to strictly convex quadratic programs. Algorithm 4 summarizes the null-space method for solving an EQP [5]. The matlab code is listed in Listing 1.6.

Algorithm 4: An algorithm for solving an EQP using the null-space method.

Data: H, g, A, b
Result: x, λ

1 QR factorize $A = \begin{bmatrix} Y & Z \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix}$
2 Cholesky factorize $Z^T H Z = LL^T$
3 Compute p_y by solving $R^T p_y = b$
4 Compute $g_z = -Z^T(HYP_y + g)$
5 Compute r by solving $Lr = gz$
6 Compute p_z by solving $L^T p_z = r$
7 Compute $x = Yp_y + Zp_z$
8 Compute λ by solving $R\lambda = Y^T(Gx + g)$

```
function [x, lambda] = EqualityQPSolverNullSpace(H,g,A,b)
% EqualityQPSolverNullSpace Solver for a convex equality constrained QP
% using the Null Space procedure
%
%          min 0.5*x'H*x + g'x
%            x
%           s.t. A'x = b
%
% Syntax: [x,lambda] = EqualityQPSolverNullSpace(H,g,A,b)

[n,m] = size(A);

% Factorize A
[Q,R] = qr (A, 'vector') ;

Q1 = Q(:,1:m);
Q2 = Q(:,m+1:n); % null space
```

```
R = R(1:m,1:m);

L = chol(Q2'*H*Q2);
Py = (R'\b);
gz= -Q2'*(H*Q1*Py+g);
r = L'\gz;
Pz = L\r;

% Solve for x and lambda
x = Q1*Py+Q2*Pz ;
lambda = R\Q1'*(g+H*x) ;

end
```

Listing 1.6: An EQP solver using the null-space method.

1.3.5 Testing the EQP Solvers on a Test Problem

The EQP solvers were tested on a test problem with the data:

$$H = \begin{bmatrix} 5.0000 & 1.8600 & 1.2400 & 1.4800 & -0.4600 \\ 1.8600 & 3.000 & 0.4400 & 1.1200 & 0.5200 \\ 1.2400 & 0.4400 & 3.8000 & 1.5600 & -0.5400 \\ 1.4800 & 1.1200 & 1.5600 & 7.2000 & -1.1200 \\ -0.4600 & 0.5200 & -0.5400 & -1.1200 & 7.8000 \end{bmatrix}$$

$$g = \begin{bmatrix} -16.1000 \\ -8.5000 \\ -15.7000 \\ -10.0200 \\ -18.6800 \end{bmatrix}$$

$$A = \begin{bmatrix} 16.1000 & 1.0000 \\ 8.5000 & 1.0000 \\ 15.7000 & 1.0000 \\ 10.0200 & 1.0000 \\ 18.6800 & 1.0000 \end{bmatrix}$$

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

The test problem was solved for 20 equally spaced values of $b(1)$ in the range $[8.5, 18.68]$ and the error of the approximate solution estimated by comparing it with *quadprog*. The driver for the test can be found in Appendix A.2.1. The comparison can be seen in Figure 1.1. As can be seen the solvers all have the same performance with an error of 10^{-12} compared to *quadprog* which confirms that they are working correctly.

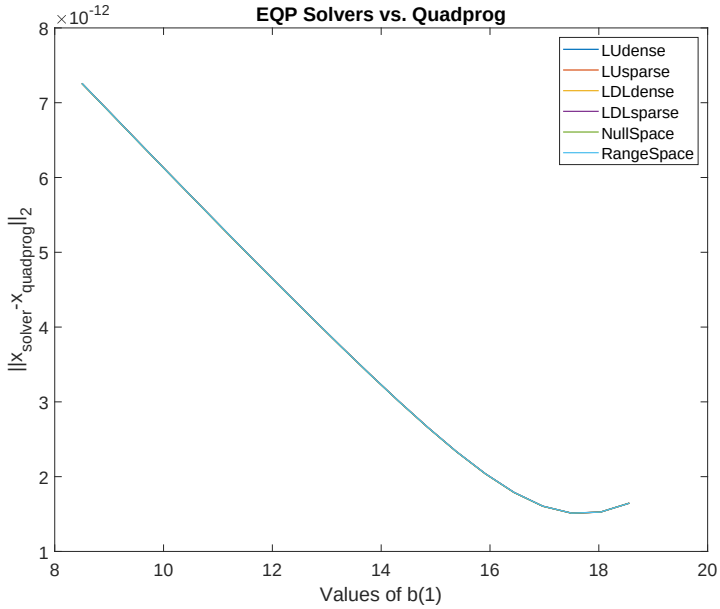

Figure 1.1: Comparing solutions generated by the implemented EQP solvers with Matlab's *quadprog* for the test problem defined in Section 1.3.5.

Finally, an interface was created to easily switch between the solvers and is listed in Appendix A.2.2.

1.3.6 Testing the EQP Solvers on a Size Dependent Problem

The driver for the tests implemented in this section can be found in Appendix A.2.1. To assess their performance, the implemented EQP solvers were tested on a size dependent problem. The problem used was the "Recycling System" from week 5 in the course, defined as:

$$\min_{u} \ \frac{1}{2}\sum_{i=1}^{n+1}(u_i - \bar{u})^2, \tag{1.7a}$$

$$s.t. \ \ -u_1 + u_n = -d_0, \tag{1.7b}$$

$$u_i - u_{i+1} = 0, \quad i = 1, 2, ..., n-2, \tag{1.7c}$$

$$u_{n-1} - u_n - u_{n+1} = 0, \tag{1.7d}$$

where d_0 and $\bar{u}$ are parameters of the problem. The size of the problem can then be adjusted by selecting $n \geq 3$. To solve the problem it needs to be converted to matrix form as in (1.1). This can be done by looking at the objective function, $f(x)$:

$$\begin{aligned} f(x) &= \frac{1}{2}\sum_{i=1}^{n+1}(u_i - \bar{u})^2 \\ &= \frac{1}{2}(u_1^2 + \bar{u}^2 - 2u_1\bar{u} + u_2^2 - 2u_2\bar{u} + \ldots + u_{n+1}^2 + \bar{u}^2 - 2u_{n+1}\bar{u}) \\ &= \frac{1}{2}\left(\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n+1} \end{bmatrix}^T I^{(n+1)\times(n+1)} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n+1} \end{bmatrix} - 2\begin{bmatrix} \bar{u} \\ \bar{u} \\ \vdots \\ \bar{u} \end{bmatrix}^T \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n+1} \end{bmatrix} + (n+1)\bar{u}^2\right) \\ &= \frac{1}{2}u^T I^{(n+1)\times(n+1)}u - \bar{u}^{(n+1)\times 1}u + \frac{n+1}{2}\bar{u}^2. \end{aligned} \tag{1.8}$$

Now, from the constraints in (1.7) the A matrix and b vector can be extracted. The parameters for the quadratic problem are thus:

$$H = I^{(n+1)\times(n+1)} \tag{1.9}$$

$$x = \begin{bmatrix} u_1 & u_2 & \dots & u_{n+1} \end{bmatrix}^T \tag{1.10}$$

$$g = -\begin{bmatrix} \bar{u}_1 & \bar{u}_2 & \dots & \bar{u}_{n+1} \end{bmatrix}^T \tag{1.11}$$

$$A = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 1 & \dots & 0 & 0 \\ & & & \ddots & & \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & -1 & 1 \\ 1 & 0 & 0 & \dots & 0 & -1 \\ 0 & 0 & 0 & \dots & 0 & -1 \end{bmatrix}^{(n+1)\times n} \tag{1.12}$$

$$b = \begin{bmatrix} -d_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^{(n+1)\times 1} \tag{1.13}$$

$$\tag{1.14}$$

The matlab function to generate the parameters for the recycling system can be found in Appendix A.3.1. The test was done for 20 equidistant values of n in the range $[10, 2000]$ with $\bar{u} = 0.2$ and $d_0 = 1$. Figure 1.2 shows the comparison for the recycling system as an increasing problem size. As was expected, the computation time increases as the problem size increases. This is because the matrices grow larger which requires more computation time. The fastest solvers are the sparse LU and LDL solvers and the dense solvers along with the null space solver are the slowest. This is surprising as the LDL solver is supposed to take up half of the LU solver's computation time as mentioned above. The range space method could be performing slower because as mentioned before it works best for when the number of variables is much larger than number of constraints. In this problem the number of constraints is only 1 dimension lower than for the variables which explains the behaviour.

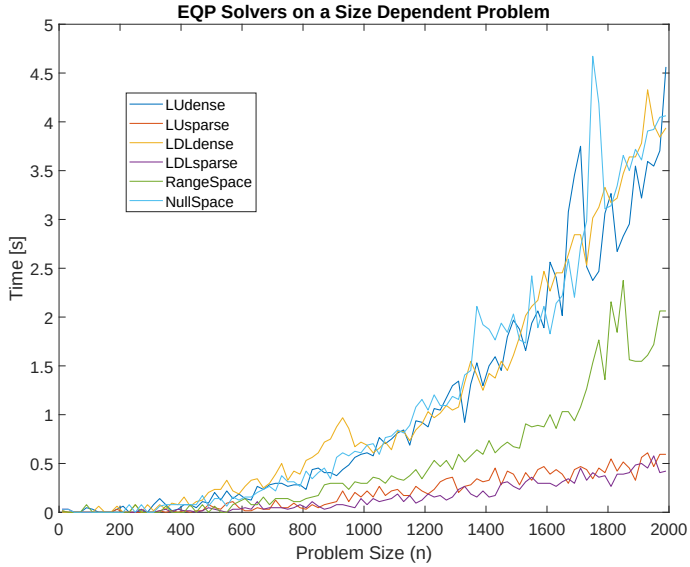

Figure 1.2: Comparing the performance (CPU time) of the EQP solvers on a size dependent problem.

To investigate further, the effect of difference in dimensionality between the variables and constraints were tested using a randomly generated convex QP. To generate a random convex EQP, a function was created that outputs randomly generated parameters H, g, A, b, x and λ, see Appendix A.3.2. The function takes the dimensions as input but ensures $n \leq m$. For the Hessian matrix to be symmetric and positive definite the function first generates a $n \times n$ matrix X with random integers and multiplies it with its transpose. Since all integers are less than 1 by construction and a symmetric diagonally dominant matrix is also symmetric positive definite then:

$$H = \frac{1}{2} XX' + nI$$

Now, the solvers were tested on a random convex EQP with fewer constraints than variables, that is $m \approx n/4$ and for the same range of dimension values as in the previous test. The results can be seen in Figure 1.3. Surprisingly, the LU sparse solver is now extremely slow and it is apparent that this solver is not fit for this type of problem in high dimension. The range-space method is now the fastest out of all the solvers which confirms that it works best for fewer constraints than variables. Additionally, the difference in computation time for the dense LDL and LU solvers is more apparent now, with LDL being faster. To conclude, the number of variables and constraints seem to have a great impact on the computational performance of the EQP solvers.

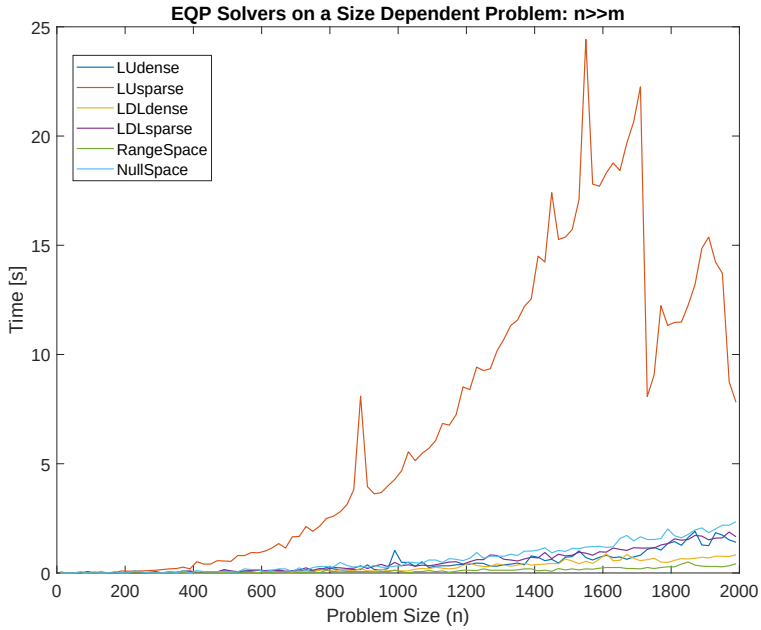

Figure 1.3: Comparing the performance (CPU time) of the EQP solvers on a size dependent problem with fewer constraints than variables.

CHAPTER 2

Quadratic Program

In this chapter the following quadratic program (QP) is considered:

$$\min_x \quad \phi = \frac{1}{2}x^T Hx + g^T x \tag{2.1a}$$
$$s.t. \quad A^T x = b \tag{2.1b}$$
$$l \leq x \leq u \tag{2.1c}$$

The feasible region defined by (2.1c) has a rectangular shape and is therefore often referred to as a "box". For ease of calculations in the perceding sections the lower and upper bounds presented in (2.1c) can be reformulated as inequality constraints of the form $C^T x \geq d$ [6]. The problem then becomes:

$$\min_x \quad \phi = \frac{1}{2}x^T Hx + g^T x \tag{2.2a}$$
$$s.t. \quad A^T x = b \tag{2.2b}$$
$$\begin{bmatrix} I \\ -I \end{bmatrix} x \geq \begin{bmatrix} l \\ -u \end{bmatrix} \tag{2.2c}$$

2.1 The Lagrange Function

The problem in this chapter differs from the one presented in Chapter 1 as now there are both equality and inequality constraints. The Lagrange function for an optimization program with inequality and equality constraint is:

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

Where μ represents the Lagrange multiplier for the inequality constraints and λ for the equality constraints. For the QP presented in this chapter the Lagrange function in quadratic form is [4]:

$$\mathcal{L}(x, \lambda, \mu) = \frac{1}{2} x^T H x + g^T x - \lambda^T (A^T x - b) - \mu^T (C^T x - d) \tag{2.4}$$

$$= \frac{1}{2} x^T H x + g^T x - \lambda^T (A^T x - b) - \mu^T \begin{bmatrix} x - l \\ u - x \end{bmatrix}. \tag{2.5}$$

2.2 Optimality Conditions

The necessary first order conditions for the problem in (2.2c) are the same as for the EQP in Chapter 1 but now with the addition of the inequality conditions provided in (1.4). They are:

$$\nabla_x \mathcal{L} = \nabla f(x) - \sum_{i \in \mathcal{E}} \lambda_i \nabla c_i(x) - \sum_{i \in \mathcal{I}} \mu_i \nabla c_i(x) = 0 \tag{2.6a}$$

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

$$c_i(x) \geq 0 \qquad i \in \mathcal{I} \tag{2.6c}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I} \tag{2.6d}$$

$$c_i(x) = 0 \ \vee \ \mu_i = 0 \qquad i \in \mathcal{I}. \tag{2.6e}$$

where (2.6d) is called the dual feasibility condition. According to lecture 2B [2] there are three possible constraint qualifications for (2.6e). First the inactive inequality constraint:

$$c_i(x) > 0, \quad \mu_i = 0 \qquad i \in \mathcal{I}. \tag{2.7}$$

The weakly active inequality constraint:

$$c_i(x) = 0, \quad \mu_i = 0 \qquad i \in \mathcal{I}, \tag{2.8}$$

which indicates that removing the inequality constraint does not affect the solution. Finally, the strongly active inequality constraint:

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

which on the other hand means that if this holds then removing the inequality constraint affects the solution. Now, if the program is convex then the first order conditions are guaranteed to be sufficient. However, if the program is not convex then the second order conditions are required to determine if the optimum obtained is a minimum. See Section 4.3 for more on the second order conditions.

2.3 Primal-Dual Interior-Point Algorithm for a Quadratic Program

When solving a QP there are two methods that are widely used. These are the active set and interior-point methods. A specific type of an interior-point method, the primal-dual interior-point method will be the focus of this chapter. This method is known to be efficient and quite useful for many types of problems. The section starts with a general introduction to these methods before diving into the algorithm. The following derivations are mostly based on Lecture 6 on "Convex Quadratic Programming" [2] and the book by Nocedal & Wright [4].

2.3.1 The Primal-Dual Framework

Consider a QP problem of the form:

$$\min_x \ \frac{1}{2}x^T Hx + g^T x \tag{2.10a}$$
$$s.t. \ A^T x = b \tag{2.10b}$$
$$C^T x \geq d \tag{2.10c}$$

with the Lagrange function given in (2.4) and optimality conditions as in (2.6). Now, the inequality constraint can be converted to an equality constraint by introducing slack variables:

$$s := C^T x - d \geq 0, \tag{2.11}$$

which implies:

$$-C^T x + s + d = 0, \tag{2.12}$$
$$s \geq 0. \tag{2.13}$$

The optimality conditions can then be expressed as:

$$r_L = Hx + g - Ay - Cz = 0 \tag{2.14}$$
$$r_A = -A^T x + b = 0 \tag{2.15}$$
$$r_C = -C^T x + s + d = 0 \tag{2.16}$$
$$r_{SZ} = SZe = 0 \tag{2.17}$$
$$s \geq 0, z \geq 0 \tag{2.18}$$

where $SZe = 0$ is an expression of the complementary conditions $s_i z_i = 0$, $i = 1, 2, ..., m_c$, and:

$$S = \begin{bmatrix} s_1 & & \\ & \ddots & \\ & & s_{m_c} \end{bmatrix}, \quad Z = \begin{bmatrix} z_1 & & \\ & \ddots & \\ & & s_{m_c} \end{bmatrix}, \quad e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{2.19}$$

Now, to derive the primal-dual interior-point methods the optimality conditions can be restated as:

$$F(x, y, z, s) = \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix} = \begin{bmatrix} Hx + g - Ay - Cz \\ -A^T x + b \\ -C^T x + s + d \\ SZe \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.20a}$$

$$(z, s) \geq 0. \tag{2.20b}$$

General primal-dual methods generate iterates that strictly satisfy the bounds $(z, s) > 0$ to avoid spurious solutions. The term primal-dual comes from this property. The reason for this is because they do not provide useful information regarding the solution of (2.10), and are thus excluded from the search region [4]. An important component in the primal-dual interior-point method is the duality measure, which measures the desirability of each point in the search region. It is defined as:

$$\mu = \frac{z^T s}{m_c}. \tag{2.21}$$

The search direction is determined using Newton's method by the solution of the following system of equations:

$$J(x, y, z, s) = \begin{bmatrix} \nabla x \\ \nabla y \\ \nabla z \\ \nabla s \end{bmatrix} = -F(x, y, z, s), \tag{2.22}$$

which can be further expressed as:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \nabla x \\ \nabla y \\ \nabla z \\ \nabla s \end{bmatrix} = - \begin{bmatrix} r_L \\ r_A \\ r_C \\ r_{SZ} \end{bmatrix}. \tag{2.23}$$

The Newton step is then as follows:

$$\begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix} := \begin{bmatrix} x \\ y \\ z \\ s \end{bmatrix} + \alpha \begin{bmatrix} \nabla x \\ \nabla y \\ \nabla z \\ \nabla s \end{bmatrix}, \quad (s, z) \geq 0 \tag{2.24}$$

The goal is to select α to be s.t. $(z, s) \geq 0$ holds.

2.3.2 The Central Path

The central path plays a vital role in primal-dual algorithms as only using the above framework often leads to poor convergence. It allows for larger steps as it biases the steps toward the interior of the nonnegative orthant defined by the constraint $(z, s) \geq 0$. This sets the scene for a large reduction in μ in the next iteration. The central path is obtained by adding a scalar $\tau > 0$ to the KKT conditions:

$$r_L = Hx + g - Ay - Cz = 0 \tag{2.25a}$$
$$r_A = -A^T x + b = 0 \tag{2.25b}$$
$$r_C = -C^T x + s + d = 0 \tag{2.25c}$$
$$SZe = \tau \tag{2.25d}$$
$$s \geq 0, z \geq 0 \tag{2.25e}$$

Notice the only change is the τ term on the right-hand side. From (2.25) the central path is then defined as:

$$\mathcal{C} = \{(x_\tau, y_\tau, s_\tau, z_\tau) | \tau > 0\} \tag{2.26}$$

Now, incorporating this into Newton's search direction:

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

In order to calculate the centering scalar τ a centering parameter $\sigma \in [0, 1]$ is used along with the duality measure μ yielding:

$$\tau = \sigma\mu. \tag{2.28}$$

There are many strategies available for choosing the centering parameter σ. One such strategy is Mehrotra's predictor-corrector which is explained in the next section.

2.3.3 Mehrotra's Modification

Merothra's predictor-corrector algorithm was originally developed for linear programs. It is however the most popular interior-point method for convex QP [4]. The method adds two key features to the primal-dual framework:

1. The addition of a corrector step to the primal-dual search direction.
2. An adaptive choice of the centering parameter σ.

As is stated above, the centering parameter, σ, is chosen adaptively and before calculating the search direction itself. The iteration starts by computing what is called an affine-scaling direction by setting $\sigma = 0$:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \nabla x^{aff} \\ \nabla y^{aff} \\ \nabla z^{aff} \\ \nabla s^{aff} \end{bmatrix} = - \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} + \tau e \end{bmatrix}. \tag{2.29}$$

Taking a full step in this direction yields:

$$\begin{aligned} &(z_i + \nabla z_i^{aff}(s_i + \nabla s_i^{aff})) \\ &= z_i s_i + z_i \nabla s_i^{aff} + s_i \nabla_i^{aff} + \nabla z_i^{aff} \nabla s_i^{aff} = \nabla z_i^{aff} \nabla s_i^{aff}. \end{aligned} \tag{2.30}$$

Notice that the updated value of $z_i s_i$ is not the ideal value of 0. To correct for this deviation the system can be solved to obtain a corrector-step defined as:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \nabla x^{cor} \\ \nabla y^{cor} \\ \nabla z^{cor} \\ \nabla s^{cor} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\nabla S^{aff} \nabla Z^{aff} e \end{bmatrix}. \tag{2.31}$$

The corrector step is then added to the affine-scaling step. The combined step has the potential to reduce the duality measure even more. Now, if the affine step does not violate the nonnegativity condition $(x, z) > 0$ and yields a large reduction in μ there is not much need for centering and a small value of σ is chosen. If the step does not yield much progress then more centering is enforced by choosing σ closer to 1. The scheme for doing this is to first choose α^{aff} as the largest value satisfying:

$$z + \alpha^{aff} \nabla z^{aff} \geq 0, \tag{2.32}$$

$$s + \alpha^{aff} \nabla s^{aff} \geq 0. \tag{2.33}$$

The value of the duality gap obtained using these step lengths is then:

$$\mu^{aff} = \frac{(z + \alpha^{aff}\nabla z^{aff})^T(s + \alpha^{aff}\nabla s^{aff})}{m_c}. \tag{2.34}$$

Finally, the centering parameter is chosen according to the heuristic:

$$\sigma = \left(\frac{\mu^{aff}}{\mu}\right)^3, \tag{2.35}$$

and the centered search direction is calculated. The computation of the centering and corrector step are often combined such as not to increase computation cost. In summary, the total step in Mehrotra's modification yields the following system:

$$\begin{bmatrix} H & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \nabla x \\ \nabla y \\ \nabla z \\ \nabla s \end{bmatrix} = \begin{bmatrix} -r_L \\ -r_A \\ -r_C \\ -r_{SZ} - \nabla Z^{aff}\nabla S^{aff}e + \sigma\mu e \end{bmatrix}. \tag{2.36}$$

Solving the system in the interior-point method can be expensive. To increase efficiency of the algorithm the system can be augmented into the following form:

$$\begin{bmatrix} H + C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \nabla x \\ \nabla y \end{bmatrix} = \begin{bmatrix} -r_L + C(S^{-1}Z)(r_C - Z^{-1}\bar{z}_{SZ}) \\ -r_A \end{bmatrix}, \tag{2.37}$$

where $\bar{r}_{SZ} = r_{SZ} - \nabla Z^{aff}\nabla S^{aff}e + \sigma\mu e$. This corresponds to an EQP which is easier to solve. Algorithm 5 summarizes the above theory. The pseudo-code solves QPs of the form provided in (2.10).

Algorithm 5: Primal-dual predictor-corrector interior-point algorithm for quadratic programs.

Data: H, g, A, b, C, d, x, y, z, s
Ensure $(z, s) > 0$
Set $\eta = 0.995$
Set m_c as the number of inequality constraints
Compute residuals
$r_L = Hx + g - Ay - C^T z$, $r_A = b - A^T x$, $r_C = s + d - Cx$, $r_{SZ} = SZ\mathbf{e}$
Set $\mu = \frac{z^T s}{m_c}$ `// Duality gap measure`
Check for convergence
while *not STOP* **do**
  Compute $\bar{H} = H + C(S^{-1}Z)C^T$
  Set $KKT = \begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix}$
  Compute $[L, D] = ldl(KKT)$ `// LDL factorize the KKT matrix`
  Affine direction
  Compute $\bar{r}_L = r_L - C(S^{-1}Z)C^T$
  Compute $\bar{b} = -\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
  Compute $\begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \end{bmatrix} = L^{-T}(D^{-1}(L^{-1}\bar{b}))$
  Compute $\Delta z^{aff} = -(S^{-1}Z)C^T\Delta x^{aff} + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
  Compute $\Delta s^{aff} = -Z^{-1}r_{SZ} - Z^{-1}S\Delta z^{aff}$
  Find largest α^{aff} *s.t.* $z + \alpha^{aff} \geq 0$ *and* $s + \alpha^{aff}\Delta s^{aff} \geq 0$
  Affine duality gap and centering parameter
  Compute $\mu^{aff} = (z + \alpha^{aff}\Delta z^{aff})^T(s + \alpha^{aff}\Delta s^{aff})/m_c$
  Compute $\sigma = (\mu^{aff}/\mu)^3$
  Affine-centering-correction direction
  Compute $\bar{r}_{SZ} = r_{SZ} + \Delta S^{aff}\Delta Z^{aff}\mathbf{e} - \sigma\mu\mathbf{e}$
  Compute $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$
  Compute $\bar{b} = -\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
  Compute $\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = L^{-T}(D^{-1}(L^{-1}\bar{b}))$
  Compute $\Delta z = -(S^{-1}Z)C^T\Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
  Compute $\Delta s = -Z^{-1}\bar{r}_{SZ} - Z^{-1}S\Delta z$
  Find largest α *s.t.* $z + \alpha \geq 0$ *and* $s + \alpha\Delta s \geq 0$
  Update parameters
  $x := x + \eta\alpha\Delta x, \quad y := y + \eta\alpha\Delta y$
  $z := z + \eta\alpha\Delta z, \quad s := s + \eta\alpha\Delta s$
  Update residuals
  $r_L := Hx + g - Ay - C^T z$, $r_A := b - A^T x$, $r_C := s + d - Cx$, $r_{SZ} := SZ\mathbf{e}$
  Compute $\mu = \frac{z^T s}{m_c}$
  Check for convergence
end

2.3.4 Initial Point Heuristic

The choice of a starting point is an important issue which greatly effects the robustness and efficiency of the algorithm. The selection can be done in several ways. One such way is presented in Lecture 6 [2] and is listed in Algorithm 6.

Algorithm 6: Initial point heuristics for the primal-dual interior-point algorithm in Algorithm 5.

Data: x_0, y_0, z_0, s_0
Result: x, y, z, s
Require $(z_0, s_0) > 0$
Compute residuals
$r_L = Hx_0 + g - Ay_0 - C^T z_0$, $r_A = b - A^T x_0$, $r_C = s + d - Cx_0$, $r_{SZ} = SZ\mathbf{e}$
Compute $\bar{H} = H + C(S^{-1}Z)C^T$
Set $KKT = \begin{bmatrix} \bar{H} & -A \\ -A^T & 0 \end{bmatrix}$
Compute $[L, D] = ldl(KKT)$ `// LDL factorize the KKT matrix`
Affine direction
Compute $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{sz})$
Compute $\bar{b} = -\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
Compute $\begin{bmatrix} \Delta x^{aff} \\ \Delta y^{aff} \end{bmatrix} = L^{-T}(D^{-1}(L^{-1}\bar{b}))$
Compute $\Delta z^{aff} = -(S^{-1}Z)C^T\Delta x^{aff} + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
Compute $\Delta s^{aff} = -Z^{-1}\bar{r}_{SZ} - Z^{-1}S\Delta z^{aff}$
Update point
$x := x_0, \quad y := y_0$
$z := max\{1, |z_0 + \nabla z^{aff}|\}, \quad s := max\{1, |s_0 + \nabla s^{aff}|\}$

2.3.5 Testing the Primal-Dual Interior-Point QP Algorithm

The primal-dual interior-point algorithm presented in Algorithm 5 was implemented in matlab, see function code in Appendix A.1.1. To test if it was providing correct solutions the test problem in Section 1.3.5 was used with the same values of $b(1)$. The solution was compared to the solution from *quadprog* using the interior-point-convex method and the active-set method. The results are presented in Figure 2.1 and it can be seen that the errors behave the same for both methods. The error is very low, on the scale of 10^{-10}, which confirms that the solver is working properly. The driver for generating the tests in this section and the following subsections is listed in Appendix A.2.3.

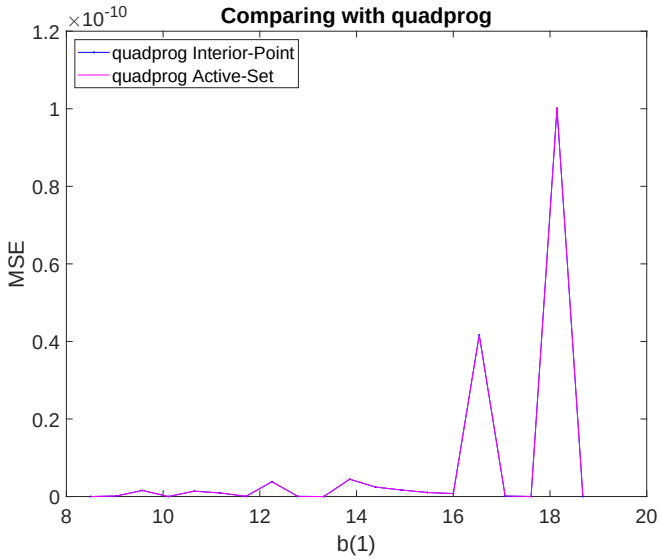

Figure 2.1: Testing the performance of the implemented primal-dual interior-point QP solver by comparing it with *quadprog* using the interior-point-convex and active-set methods.

2.3.5.1 Comparing with Quadprog for a Size Depenent Problem

To further test the performance of the algorithm, it was compared with *quadprog* using the interior-point and active-set method for an increasing problem size. The test was done on the "Recycling system" described in Section 1.3.6 with the addition of lower and upper bounds on x: $0 \leq x \leq 1$. The same parameters were used with 50 equally spaced values of n in the interval $[10, 500]$.

According to the results in Figure 2.3 the iterations are few and constant for both solvers. The reason for *quadprog* having 0 iterations was due to the solution being found during presolve, that is some combination of the bounds or constraints in the objective function immediately lead to the solution. As was expected *quadprog* is not influenced by the increasing problem size while the solver slows down as the problem size increases, see Figure 2.2. One reason is that *quadprog* is programmed in C which is a much faster language than Matlab. From Figure 2.4 it can be seen that the error between the solutions is extremely low. From this it can be concluded that the solver works faster for lower dimensional problems but the overall performance is good.

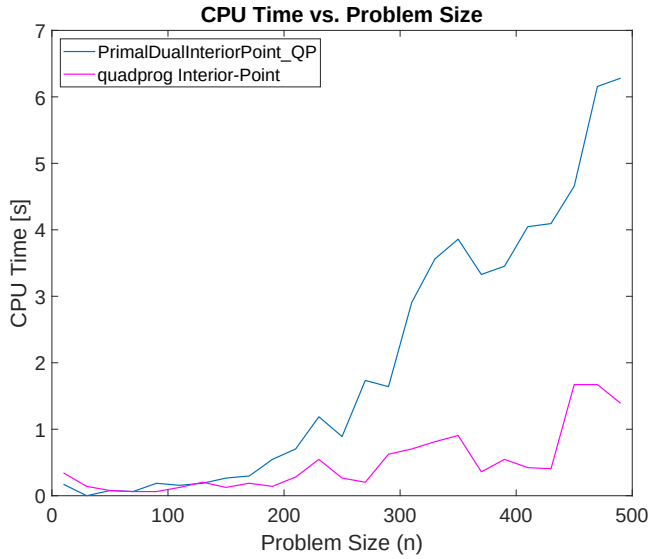

Figure 2.2: Comparing CPU time between the implemented primal-dual interior-point QP solver and *quadprog* on a size dependent problem.

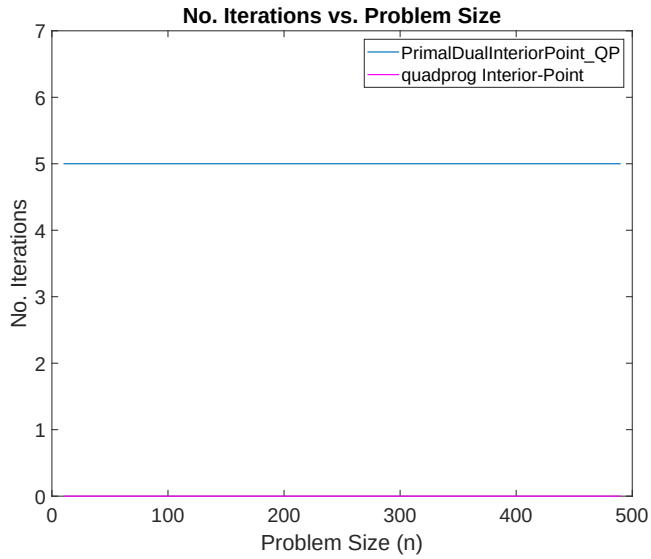

Figure 2.3: Comparing iterations used by the implemented primal-dual interior-point QP solver and *quadprog* on a size dependent problem.

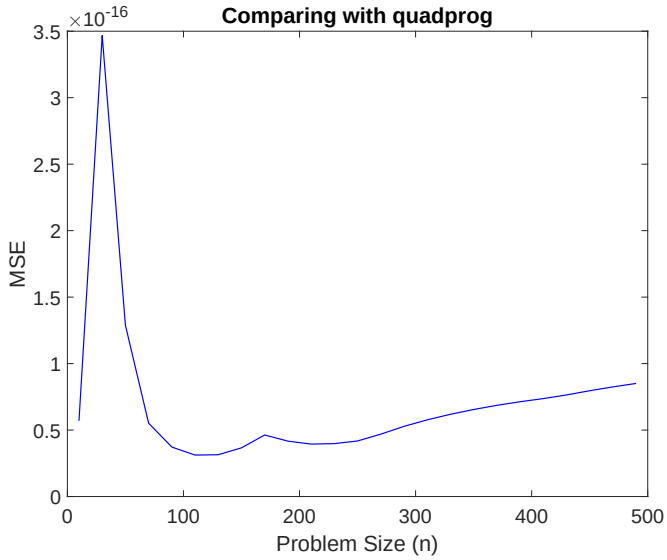

Figure 2.4: Mean squared error between the implemented primal-dual interior-point QP solver and *quadprog* on a size dependent problem.

2.3.5.2 Comparing with Other QP Libraries

The primal-dual interior-point QP solver was tested on the test problem described in Section 1.3.5 with the addition of lower and upper bounds: $0 \leq x \leq 1$. The solution was then compared to the QP libraries CVX, Gurobi and quadprog. For CVX the default *SDPT3* solver was used. As in Section 1.3.5 this was done for 20 equally spaced values of $b(1)$ in the range $[8.5, 18.68]$. According to Figure 2.5, CVX is by far the slowest. This could be due to CVX being a more general purpose solver rather than a specialized one, as *SDPT3* can be used for both LP and QP problems. Regarding the number of iterations, from Figure 2.6 it can be seen that they all fluctuate in a similar range, with Gurobi obtaining the solution in 0 iterations which seems weird at first. However, Figure 2.7 confirms that Gurobi was working as it should as the error compared to the implemented solver is very low and follows the same path as *quadprog*. In summary, the implemented solver is working quite efficiently even outperforming CVX in some cases.

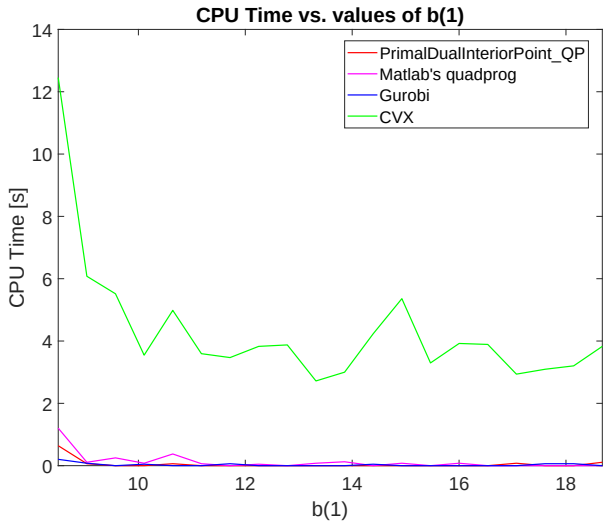

Figure 2.5: Comparing CPU time of the implemented primal-dual interior-point algorithm and *quadprog* on the test problem in Section 1.3.5.

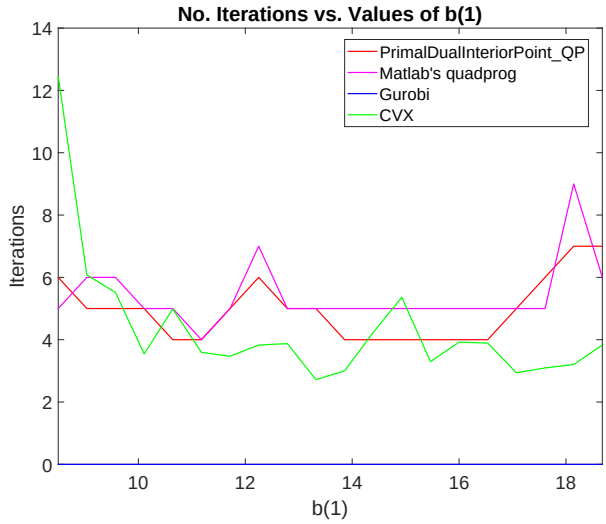

Figure 2.6: Comparing the number of iterations used by the implemented primal-dual interior-point QP algorithm and *quadprog* on the test problem in Section 1.3.5.

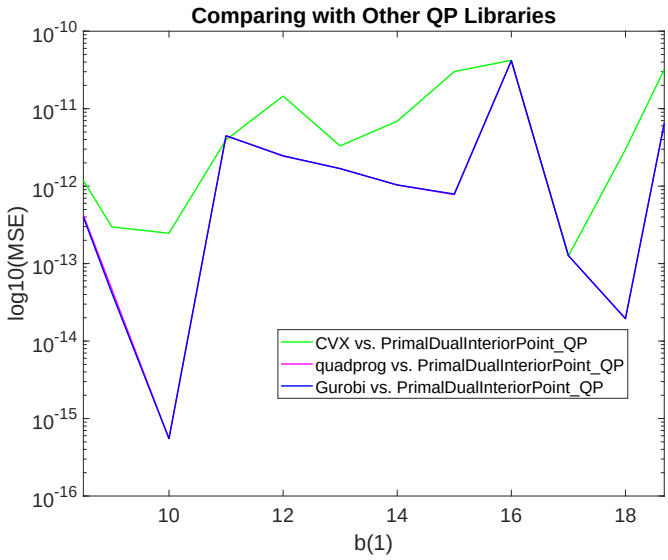

Figure 2.7: Comparing solutions from the implemented primal-dual interior-point algorithm with *quadprog*, CVX and Gurobi, using the test problem in Section 1.3.5.

CHAPTER 3

Linear Program

In this chapter the following linear program (LP) is considered:

$$\min_x \quad \phi = g^T x \tag{3.1a}$$
$$s.t. \quad A^T x = b \tag{3.1b}$$
$$l \leq x \leq u \tag{3.1c}$$

As in Chapter 2, for ease of calculations the lower and upper bounds can be reformulated as inequality constraints of the form $C^T x \geq d$ [6]. The problem then becomes:

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

3.1 The Lagrange Function

The Lagrange function for this problem is the same as for the one presented in Chapter 2 with the only difference being the objective function which is now linear instead of quadratic. The Lagrange function is thus:

$$\mathcal{L}(x, \lambda, \mu) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{I}} \mu_i c_i(x) \tag{3.3a}$$
$$= g^T x - \lambda^T (A^T x - b) - \mu^T (C^T x - d) \tag{3.3b}$$
$$= g^T x - \lambda^T (A^T x - b) - \mu^T \left(\begin{bmatrix} x - l \\ u - x \end{bmatrix} \right) \tag{3.3c}$$

3.2 Optimality Conditions

The linear program has inequality and equality constraints as in Chapter 2 and therefore the necessary first order optimality conditions are the same, that is:

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

$$c_i(x) = 0 \qquad i \in \mathcal{E} \tag{3.4b}$$

$$c_i(x) \geq 0 \qquad i \in \mathcal{I} \tag{3.4c}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I} \tag{3.4d}$$

$$c_i(x) = 0 \ \vee \ \mu_i = 0 \qquad i \in \mathcal{I} \tag{3.4e}$$

According to the lecture notes a linear program of the form presented in (3.1) is a convex program [1]. Given the same arguments as in Section 1.2, the first order conditions are both necessary and sufficient.

3.3 Primal-Dual Interior-Point Algorithm for a Linear Program

Dealing with a linear program is much easier than with a quadratic program as both the objective function and constraints are linear. This saves computation time and makes implemeting the program easier. In this chapter, the primal-dual interior-point algorithm is implemented with Mehrotra's modification. The derivations are very similar to the ones in the previous chapter, see Section 2.3 for the full derivation. As the quadratic term, H, is no longer present the KKT conditions become:

$$r_L = g - Ay - Cz = 0 \tag{3.5a}$$

$$r_A = -A^T x + b = 0 \tag{3.5b}$$

$$r_C = -C^T x + s + d = 0 \tag{3.5c}$$

$$SZe = \tau \tag{3.5d}$$

$$s \geq 0, z \geq 0 \tag{3.5e}$$

The system to be solved in (2.36) then takes a new form:

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

The augmented system is now:

$$\begin{bmatrix} C(S^{-1}Z)C^T & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} \nabla x \\ \nabla y \end{bmatrix} = \begin{bmatrix} -\bar{r}_L \\ -r_A \end{bmatrix}. \tag{3.7}$$

where $-\bar{r}_L = -r_L + C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{SZ})$. Solving for ∇x yields two equations:

$$\nabla x = (-\bar{r}_L + A\nabla y)(C(S^{-1}Z)C^T)^{-1} \tag{3.8}$$

$$-A^T\nabla x = -r_A \tag{3.9}$$

Now, substituting (3.8) into (3.9) yields the normal equation:

$$A(C(S^{-1}Z)C^T)^{-1}A^T\nabla y = r_A + A^T(C(S^{-1}Z)C^T)^{-1}\bar{r}_L \tag{3.10}$$

Due to the lower and upper bounds in problem (3.1), the inequality matrix is defined as $C = [I - I]^T$ and therefore $C(S^{-1}Z)C^T$ becomes a diagonal matrix. This means that a Cholesky factorization can be used to solve for ∇y which is more computationally efficient than the previous LDL factorization. The pseudo-code for implementing the primal-dual interior-point method for linear programs can be seen in Algorithm 7.

Algorithm 7: Primal-dual predictor-corrector interior-point algorithm for linear programs.

Data: g, A, b, C, d, x, y, z, s
Require $(z, s) > 0$
Set $\eta = 0.995$
Set m_c as the number of inequality constraints
Compute residuals
$r_L = g - Ay - C^T z$, $r_A = b - A^T x$, $r_C = s + d - Cx$, $r_{sz} = SZ\mathbf{e}$
Compute $\mu = \frac{z^T s}{m_c}$
while *not STOP do* **do**
  Compute $R = AC(S^{-1}Z)C^T A^T$
  Compute $\bar{r}_L = r_L - C(S^{-1}Z)C^T$
  Compute $\bar{b} = -\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
  Compute $L = chol(R)$; `// Cholesky factorize`
  Compute $\Delta y^{aff} = L^{-T}(L^{-1}\bar{b}))$
  Compute $\Delta x^{aff} = (-\bar{r}_L + A\Delta y^{aff})(C(S^{-1}Z)C^T)^{-1}$
  Compute $\Delta z^{aff} = -(S^{-1}Z)C^T\Delta x^{aff} + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
  Compute $\Delta s^{aff} = -Z^{-1}r_{SZ} - Z^{-1}S\Delta z^{aff}$
  Find largest α^{aff} *s.t.* $z + \alpha^{aff} \geq$ *and* $s + \alpha^{aff}\Delta s^{aff} \geq 0$
  Affine duality gap and centering parameter
  Compute $\mu^{aff} = (z + \alpha^{aff}\Delta z^{aff})^T(s + \alpha^{aff}\Delta s^{aff})/m_c$
  Compute $\sigma = (\mu^{aff}/\mu)^3$
  Affine-centering-correction direction
  Compute $\bar{r}_{SZ} = r_{SZ} + \Delta S^{aff}\Delta Z^{aff}\mathbf{e} - \sigma\mu\mathbf{e}$
  Compute $\bar{r}_L = r_L - C(S^{-1}Z)(r_C - Z^{-1}\bar{r}_{SZ})$
  Compute $\bar{b} = -\begin{bmatrix} \bar{r}_L \\ r_A \end{bmatrix}$
  Compute $\Delta y = L^{-T}(L^{-1}\bar{b})$
  Compute $\Delta x^{=}(-\bar{r}_L + A\Delta y)(C(S^{-1}Z)C^T)^{-1}$
  Compute $\Delta z = -(S^{-1}Z)C^T\Delta x + (S^{-1}Z)(r_C - Z^{-1}r_{SZ})$
  Compute $\Delta s = -Z^{-1}\bar{r}_{SZ} - Z^{-1}S\Delta z$
  Find largest α *s.t.* $z + \alpha \geq$ *and* $s + \alpha\Delta s \geq 0$
  Update parameters
  $x := x + \eta\alpha\Delta x, \quad y := y + \eta\alpha\Delta y$
  $z := z + \eta\alpha\Delta z, \quad s := s + \eta\alpha\Delta s$
  Update residuals
  $r_L := g - Ay - C^T z$, $r_A := b - A^T x$, $r_C := s + d - Cx$, $r_{sx} := SZ\mathbf{e}$
  Compute $\mu = \frac{z^T s}{m_c}$
end
Check for convergence

3.3.1 Testing the Primal-Dual Interior-Point LP Algorithm

The primal-dual interior-point LP algorithm was implemented in Matlab and the code can be found in Appendix A.1.2. It was then tested on a given test problem with the following data:

$$
\begin{aligned}
g &= \\
&-16.1000 \\
&-8.5000 \\
&-15.7000 \\
&-10.0200 \\
&-18.6800 \\
A &= \\
&1.0000 \\
&1.0000 \\
&1.0000 \\
&1.0000 \\
&1.0000 \\
b &= \\
&1
\end{aligned}
$$

$l = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$, $u = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T$.

The driver for this test can be seen in Appendix A.2.4. The solver was compared with Matlab's linear program solver, *linprog*, to confirm that it was working as it should. The solution was:

$$
x = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{bmatrix}
$$

and the mean squared error between the solutions $MSE = 4 * 10^{-15}$. Additionally, it took the implemented solver 6 iterations and *linprog* 1 iteration. This confirms that the solver is working correctly. The performance is further investigated in the next section.

3.3.1.1 Comparing with Linprog on a Size Dependent Problem

The implemented solver was compared with Matlab's *linprog* using two different methods, the dual-simplex method and the interior-point method. See Appendix A.2.4 for the driver to perform this test. This was done for an increasing problem size using the aforementioned "Recycling System" as a test problem, see Appendix A.3.1. The same setup was used as in Section 2.3.5.1. The results are visualized in Figure 3.1, Figure 3.2, and Figure 3.3. As was expected *linprog* is not influenced by the increasing problem size. The implemented solver starts to slow down after about $n = 120$, this could be due to the Cholesky factorization as factorizing large matrices takes up much computation. The error is on the scale of 10^{-15} as was computed in the previous section and seems to be the lowest for very small problems. The two methods from *linprog* have similar behaviour. From this it can be concluded that the solver works efficiently but could be improved to work better for large problems.

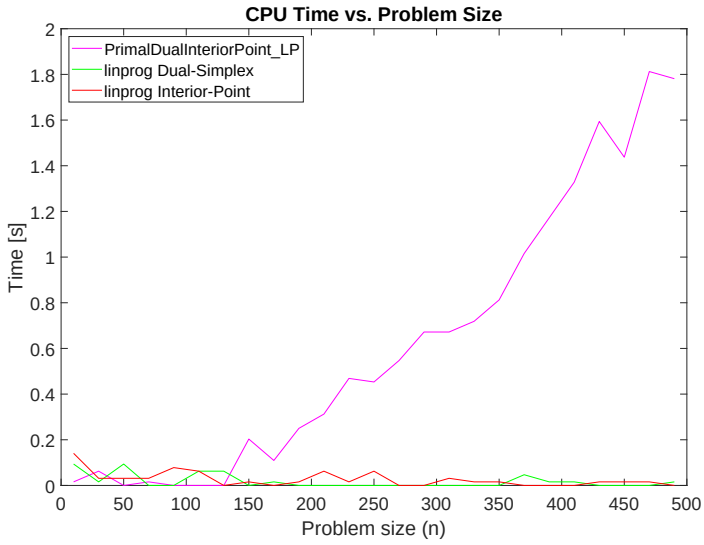

Figure 3.1: Comparing the CPU time of the implemented primal-dual interior-point LP algorithm and *linprog* on a size dependent problem.

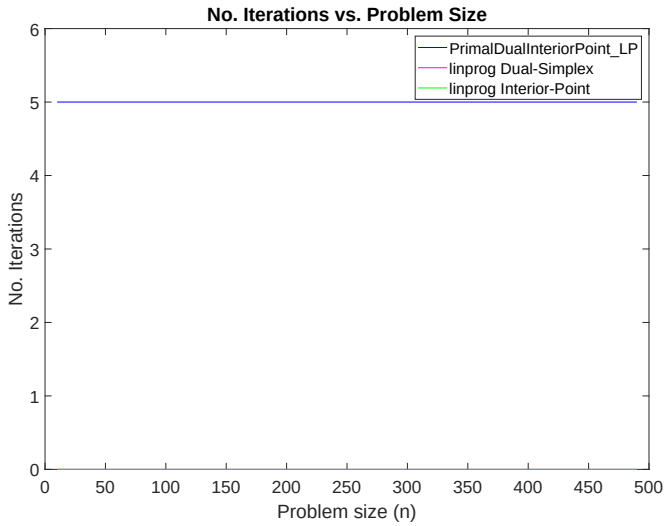

Figure 3.2: Comparing the number of iterations used by the implemented primal-dual interior-point LP algorithm and *linprog* on a size dependent problem.

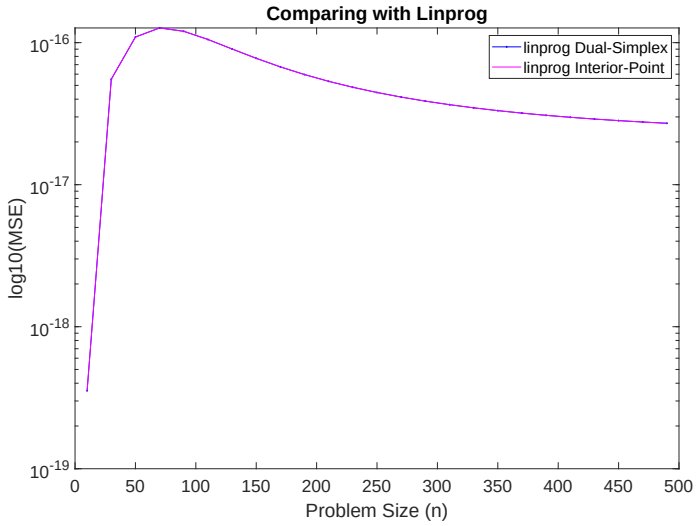

Figure 3.3: Comparing solutions of the implemented primal-dual interior-point LP algorithm verus *linprog* on a size dependent problem.

CHAPTER 4

Nonlinear Program

In this chapter the following nonlinear program (NLP) is considered:

$$\min_x \quad f(x) \tag{4.1a}$$
$$s.t. \quad g_l \leq g(x) \leq g_u \tag{4.1b}$$
$$x_l \leq x \leq x_u \tag{4.1c}$$

In the preceding sections, the involved functions are assumed to be sufficiently smooth in addition to $\nabla g(x)$ having full column rank. For ease of calculations the program can be reformulated as [6]:

$$\min_x \quad f(x) \tag{4.2a}$$
$$s.t. \quad \begin{bmatrix} g(x) \\ -g(x) \end{bmatrix} \geq \begin{bmatrix} g_l \\ -g_u \end{bmatrix} \tag{4.2b}$$
$$\begin{bmatrix} x \\ -x \end{bmatrix} \geq \begin{bmatrix} x_l \\ -x_u \end{bmatrix}. \tag{4.2c}$$

4.1 The Lagrange Function

Now there are only inequality constraints and therefore the Lagrange function for the nonlinear program given in (4.1) is defined as:

$$\mathcal{L}(x, \lambda, \mu) = f(x) - \mu^T \begin{bmatrix} g(x) - g_l \\ g_u - g(x) \\ x - x_l \\ x_u - x \end{bmatrix}. \tag{4.3}$$

4.2 First Order Optimality Conditions

The objective function being twice differentiable means that the first order conditions hold [4]. Although, the conditions are necessary but not necessarily sufficient as nonlinear programs are not guaranteed convex programs. From Lecture 9A on "SQP" [2] then for (4.1) the first order conditions can stated as:

$$\nabla_x \mathcal{L}(x,\mu) = \nabla f(x) - \mu^T \begin{bmatrix} I & -I & \nabla g(x) & -\nabla g(x) \end{bmatrix} \tag{4.4a}$$

$$\nabla_\mu \mathcal{L}(x,\mu) = \begin{bmatrix} g(x) - g_l \\ g_u - g(x) \\ x - x_l \\ x_u - x \end{bmatrix} \geq 0 \tag{4.4b}$$

$$\mu_i \geq 0 \qquad i \in \mathcal{I} \tag{4.4c}$$

$$g_i(x) = 0 \ \vee \ \mu_i = 0 \qquad i \in \mathcal{I} \tag{4.4d}$$

4.3 Second Order Optimality Conditions

According to Proposition 2.13 in the lecture notes [1], the necessary second order optimality conditions for the problem are:

$$h^T \nabla^2_{xx} \mathcal{L}(x,\mu) h \geq 0 \qquad \forall h \in \mathcal{F}(x). \tag{4.5}$$

where $\mathcal{F}(x)$ denotes the set of feasible directions, h, in a given feasible point, $x \in \Omega$, Ω being the feasible region. One of the major limitations of solving non-convex constrained optimization problems is that there is no guaranteed global minimum, only local minimum. To ensure local optimality for a non-convex program, the sufficient second order conditions are required. According to Proposition 2.14 in the lecture notes [1] they are:

$$h^T \nabla^2_{xx} \mathcal{L}(x,\mu) h > 0 \qquad \forall h \in \mathcal{F}(x). \tag{4.6}$$

4.4 Himmelblau's Test Problem

Himmelblau's test problem is a two-dimensional problem and is therefore good for visualization. It will be used for testing the SQP methods implemented in this chapter. The problem is defined as:

$$\min_{(x_1,x_2)} \ f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \tag{4.7a}$$

$$s.t. \ \ g_1(x_1,x_2) = (x_1+2)^2 - x_2 \geq 0 \tag{4.7b}$$

$$g_2(x_1,x_2) = -4x_1 + 10x_2 \geq 0 \tag{4.7c}$$

The problem can be converted into a nonlinear program of the same form as in (4.1) by adding lower and upper bounds on the constraints. The problem then becomes:

$$\min_{(x_1,x_2)} \quad f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \tag{4.8a}$$

$$s.t. \quad 0 \leq g_1(x_1, x_2) \leq 10^8 \tag{4.8b}$$

$$0 \leq g_2(x_1, x_2) \leq 10^8 \tag{4.8c}$$

$$-5 \leq x_1 \leq 5 \tag{4.8d}$$

$$-5 \leq x_2 \leq 5. \tag{4.8e}$$

The contour plot of the objective function, $f(x)$ is shown in Figure 4.1 along with the constraints and stationary points. The feasible region is within the non-shaded region on the contour plot.

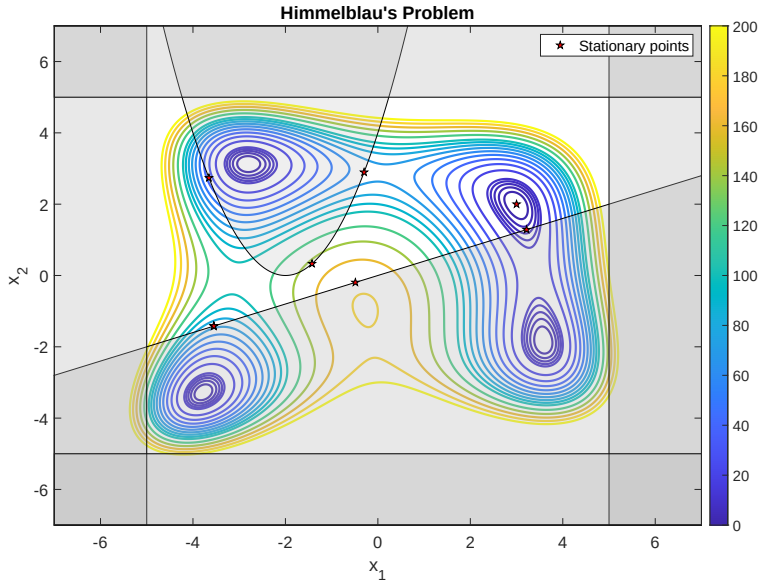

Figure 4.1: The Himmelblau problem in (4.8). The non-shaded area contains the feasible region. The stars represent the stationary points.

4.5 Testing Library Functions on Himmelblau's Problem

In this section, the nonlinear programming functions *fmincon* and CasADi were tested on Himmelblau's problem in (4.8). The driver for performing this test can be found in Appendix A.2.5. CasAdi is an efficient open-source tool that can be used for nonlinear optimization and *fmincon* is a nonlinear programming solver from Matlab. According to Figure 4.1 the problem has 4 minima and therefore the algorithms were tested for different starting points to try to locate all the minima. The results are listed in Table 4.1 below and as can be seen both solvers manage to locate all four minima.

Table 4.1: Testing CasADi and *fmincon* on Himmelblau's problem.

x_0	$[0,0]^T$	$[-4,2]^T$	$[0,3.5]^T$	$[-4,-0.5]^T$
CasADi x^*	$[3.000, 2.000]^T$	$[-3.6546, 2.7377]^T$	$[-0.2983, 2.896]^T$	$[-3.549, -1.420]^T$
fmincon x^*	$[3.000, 2.000]^T$	$[-3.6546, 2.7377]^T$	$[-0.2983, 2.896]^T$	$[-3.549, -1.420]^T$

4.6 Sequential Quadratic Programming

The sequential quadratic programming (SQP) approach is one of the most effective methods to solve nonlinearly constrained optimization problems. It does this by solving a sequence of quadratic subproblems on the current iterate, x_k, to generate a new iterate x_{k+1}. The design of the subproblem is such that it yields good steps. The simplest version, often called the *local SQP* method, is to apply Newton's method on the KKT optimality conditions [4]. However, computing the Hessian of the Lagrangian, $\nabla^2_{xx}\mathcal{L}_k$ and inverting the KKT matrix requires much computation. This also poses a problem if the problem is large or the KKT matrix is singular. There exist a variety of modifications to tackle those problems. In the preceding sections, the SQP and three modifications using BFGS, line search and trust region are discussed. The derivations are heavily based on Lecture 9 on "Sequential Quadratic Programming" [2] as well as the book by Nocedal & Wright [4].

4.6.1 The Local SQP

Before diving into the modifications of the SQP method, the general framework of the SQP needs to be explained. First, consider the equality-constrained problem:

$$\min_x \ f(x) \tag{4.9}$$

$$s.t. \ \ h(x) = 0 \tag{4.10}$$

where f and h are smooth functions. The Lagrangian of the problem is:

$$\mathcal{L}(x, \lambda) = f(x) - \lambda^T h(x), \tag{4.11}$$

and the corresponding KKT conditions can be written as a system of nonlinear equations such that $F(x, \lambda) = 0$:

$$F(x,\lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x,\lambda) \\ \nabla_\lambda \mathcal{L}(x,\lambda) \end{bmatrix} = \begin{bmatrix} \nabla f(x) - \nabla h(x)\lambda \\ -h(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{4.12}$$

Now it is possible to use Newton's method to solve the nonlinear equations in (4.12) by solving the system:

$$\nabla F(x_k, \lambda_k) \begin{bmatrix} \nabla x \\ \nabla \lambda \end{bmatrix} = -F(x_k, \lambda_k) \tag{4.13a}$$

$$\Rightarrow \begin{bmatrix} \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k) & -\nabla h(x_k) \\ -\nabla h(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} \nabla x \\ \nabla \lambda \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x_k, \lambda_k) \\ -h(x_k) \end{bmatrix}. \tag{4.13b}$$

To simplify $\nabla y = y - y_k$ can be inserted into (4.13) and then after some derivation the system becomes:

$$\begin{bmatrix} \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k) & -\nabla h(x_k) \\ -\nabla h(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} \nabla x \\ \lambda \end{bmatrix} = - \begin{bmatrix} \nabla f(x_k) \\ -h(x_k) \end{bmatrix}. \tag{4.14a}$$

This is equivalent to the QP:

$$\min_{\nabla x} \ \frac{1}{2} \nabla x \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k) \nabla x + \nabla f(x_k)^T \nabla x \tag{4.15a}$$

$$s.t. \ [\nabla h(x_k)]^T \nabla x = -h(x_k) \tag{4.15b}$$

Now it is possible to find a search direction by solving (4.15) for ∇x and $\nabla \lambda$. The above framework can then be easily extended to include inequality constraints:

$$\min_{\nabla x} \ \frac{1}{2} \nabla x \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k) \nabla x + \nabla f(x_k)^T \nabla x \tag{4.16a}$$

$$s.t. \ \nabla h(x_k)^T \nabla x = -h(x_k) \tag{4.16b}$$

$$\nabla g(x_k)^T \nabla x \geq -g(x_k) \tag{4.16c}$$

4.6.2 SQP with a Damped BFGS Approximation

In order to increase the efficiency of the SQP algorithm it can be modified by using a Broyden–Fletcher–Goldfarb–Shanno approximation, or BFGS. The BFGS method is a quasi-Newton method that approximates the inverse of the Hessian, denoted B_k for the kth step. It does this by obtaining information from the gradient of the Lagrangian, $\nabla\mathcal{L}$. The update in BFGS makes use of the vectors p_k and q_k:

$$p_k = x_{k+1} - x_k, \qquad q_k = \nabla_x\mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x\mathcal{L}(x_k, \lambda_{k+1}), \tag{4.17}$$

and the update for the next iterate is found by solving:

$$B_{k+1} = B_k \frac{B_k p_k (B_k p_k)^T}{p_k^T B_k p_k} + \frac{q_k q_k^T}{p_k^T q_k}. \tag{4.18}$$

In order for a QP to have a unique solution the problem must be strictly convex. The damped version of the BFGS ensures this by first introducing a variable r such that:

$$r_k = \theta_k y_k + (1 - \theta_k) B_k s_k, \tag{4.19}$$

where θ is a postive parameter, defined as:

$$\theta_k = \begin{cases} 1, & \text{if } p_k^T q_k \geq 0.2 p_k^T B_k p_k, \\ \frac{0.8 p_k^T B_k p_k}{p_k^T B_k p_k - p_k^T q_k}, & \text{if } p_k^T q_k < 0.2 p_k^T B_k p_k. \end{cases} \tag{4.20}$$

Now r_k replaces q_k in the above equation (4.18) to yield a new update of B_k:

$$B_{k+1} = B_k - \frac{B_k p_k p_k^T B_k}{p_k^T B_k p_k} + \frac{r_k r_k^T}{p_k^T r_k}. \tag{4.21}$$

This formulation guarantees positive definiteness of B_{k+1} by the choice of θ_k which ensures closeness between the two approximations, B_k and B_{k+1} [4].

4.6.3 SQP with BFGS and Line Search

The SQP algorithm can be used with line search in order to control the step size and improve the convergence properties. However, the choice of step length is not as clear as in the unconstrained case. The goal is for the next iterate to decrease the objective function as well as satisfying the constraints. These aims often conflict, which is where the merit function comes in. Used with line search, the merit function controls the size of the step. The one used in this chapter is Powell's exact l_1-merit function, which for the problem given in (4.1) is defined as:

$$P(x, \mu) = f(x) + \mu^T |min\{0, g(x)\}|, \tag{4.22}$$

where $f(x)$ is the objective function and $\mu \geq |z|$ is the penalty parameter, updated by:

$$\mu = max\{|z|, \tfrac{1}{2}(\mu + |z|)\}.$$

For accepting the step αx_k, the Armijo condition must be satisfied:

$$P(x_k + \alpha, \mu) \leq P(x_k, \mu) + c_1 \alpha \frac{dP}{d\alpha}(x_k, \mu), \tag{4.23}$$

where $c_1 \in [0, 1]$ and:

$$\frac{dP}{d\alpha}(x_k, \mu) = \nabla f(x_k)^T \nabla x_k - \mu^T |min\{0, g(x_k)\}|. \tag{4.24}$$

This is done to ensure sufficient decrease of the objective function. The full line search procedure with the merit function as provided in Lecture 9A [2] can be seen in Algorithm 8, where a quadratic approximation is implemented to calculate the step length α.

Algorithm 8: A line search algorithm with Powell's exact l_1 merit function.

Data: $f(x_k), \nabla f(x_k), g(x_k), \mu, \nabla x_k$
Result: x
Set $\alpha = 1$
Set $c = \phi(0) = f(x_k) + \mu |min\{0, g(x_k)\}|$
Set $b = \phi'(0) = \nabla f(x_k)^T \nabla x_k - \mu^T |min\{0, g(x_k)\}|$
while *not STOP* **do**
  Update $x = x_k + \alpha \nabla x_k$
  Compute $f_k = f(x)$
  Compute $g_k = g(x)$
  Compute $\phi(\alpha) = f_k + \mu^T |min\{0, g_k\}|$
  if $\phi(\alpha) \leq \phi(0) + 0.1\phi'(0)\alpha$ **then**
    STOP
  else
    Compute $\alpha = \frac{\phi(\alpha) - (c + b\alpha)}{\alpha^2}$
    Compute $\alpha_{min} = \frac{-b}{2a}$
    Set $\alpha = min\{0.9\alpha, max\{\alpha_{min}, 0.1\alpha\}\}$
  end
  Update parameters
  Set $c = \phi(0) = f(x_k) + \mu |min\{0, g(x_k)\}|$
  Set $b = \phi'(0) = \nabla f(x_k)^T \nabla x_k - \mu^T |min\{0, g(x_k)\}|$
  Set $\mu = max\{|z|, \frac{1}{2}(\mu + |z|)\}$
end

4.6.4 SQP with Trust Region

Trust region SQP methods do not require the Hessian matrix of the Lagrangian to be positive definite which makes them an attractive approach. They can also control the quality of the steps even when the Jacobian and Hessian are singular [4]. There are many variations of trust region SQP methods, one such is the sequential l_1 quadratic programming (Sl_1QP) approach which maintains consistency of the constraints. The QP subproblem is defined as:

$$\min_{p} \quad q_\mu(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T \nabla^2_{xx} \mathcal{L}_k p + \mu \sum_{i \in \mathcal{I}} [c_i(x_k) + \nabla c_i(x_k)^T p]^- \tag{4.25a}$$

$$s.t. \quad ||p||_\infty \leq \nabla_k \tag{4.25b}$$

for some penalty parameter μ and the notation $[y]^- = max\{0, -y\}$. This problem can then be reformulated by introducing the slack variable t:

$$\min_{p_k, t} \quad \frac{1}{2} p_k^T \nabla^2_{xx} \mathcal{L}_k p_k + \nabla f^T p_k + \mu \tag{4.26a}$$

$$s.t. \quad c_i(x_k) + \nabla c_i(x_k)^T p_k \geq -t_i \qquad i \in \mathcal{I} \tag{4.26b}$$

$$t \geq 0 \tag{4.26c}$$

$$||p||_\infty \leq \nabla_k. \tag{4.26d}$$

The l_1 merit function can then defined as:

$$\phi_1(x; \mu) = f(x) + \mu \sum_{i \in \mathcal{I}} [c_i(x)]^{-1}. \tag{4.27}$$

This merit function is then used to determine the ratio ρ_k that judges the acceptability of a step, defined as the ratio between the actual versus the predicted reduction:

$$\rho_k = \frac{ared_k}{pred_k} = \frac{\phi_1(p_k, \mu) - \phi_1(p_k + \nabla p_k)}{q_\mu(0) - q_\mu(\nabla p_k)}. \tag{4.28}$$

The step is then accepted or rejected according to standard trust-region rule: if $\rho > 0$ it's accepted, otherwise rejected. The trust region must then be adjusted for the next iteration. First, $\gamma(\rho)$ is calculated:

$$\gamma(\rho) = min\{max\{(2\rho - 1)^3 + 1, 0.25\}, 2\}. \tag{4.29}$$

and the trust region then updated by the aforementioned rule:

$$\nabla_{k+1} = \begin{cases} \gamma(\rho)\nabla_k, & \text{if } \rho < 0. \\ \gamma(\rho)||p||_\infty, & \text{otherwise.} \end{cases} \tag{4.30}$$

Finally, the penalty parameter is updated in the same way as described above for Powell's l_1 merit function:

$$\mu = max\{||z||_\infty, \frac{1}{2}(\mu + ||z||_\infty)\}. \tag{4.31}$$

The step, p_k , in the Sl_1QP algorithm depends on μ and the penalty parameter therefore plays an important role in the efficiency of the method. This is different from the other methods which only use the penalty function to determine the acceptance of a trial point. The value must be chosen carefully as too small values can lead the algorithm away from the solution, and too large values slow down the progress [4].

4.7 Testing the SQP Algorithms on Himmelblau's Problem

In the preceding sections, the SQP algorithms derived above were tested on Himmelblau's problem in (4.8). As stated above, the problem has 4 minima and therefore the algorithms were tested for the four different starting points seen in Table 4.1 to try to locate all the minima. Their performances were then evaluated. The driver to generate the following tests can be seen in Appendix A.2.5 and the algorithms for each implementation of the SQP method in Appendix A.1.3.

4.7.1 SQP with Damped BFGS Approximation

Figure 4.2 shows the results using SQP with damped BFGS approximation on Himmelblau's test problem for the four different starting points. The full iteration sequences can be seen in Table 4.3. When the solver starts at $x_0 = [0, 0]^T$ it starts by taking very large steps, hitting the boundaries of the constraints for the first few iterations and then converges slowly to the minimum. The solver did not manage to locate all the minima, only three minimum points were reached. The solver statistics can be seen in Table 4.2. From this it is apparent this solver has quite slow convergence.

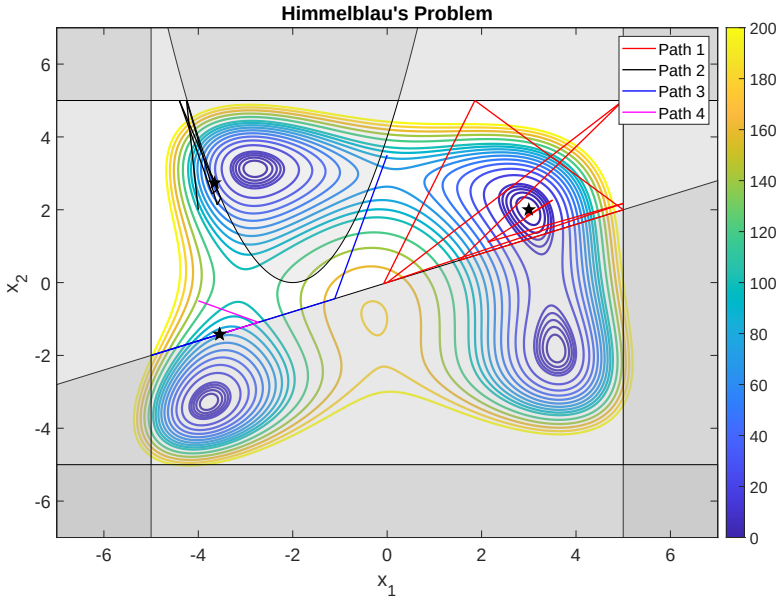

Figure 4.2: SQP with damped BFGS approximation on Himmelblau's test problem. The colored lines show the path for each starting point and the stars represent the minimizers.

Table 4.2: Solver statistics for the SQP method with BFGS on Himmelblau's test problem in (4.8).

$\mathbf{x_0}$	$[0,0]^T$	$[-4,2]^T$	$[0,3.5]^T$	$[-4,-0.5]^T$
No. Iterations	21	13	11	8
Function evaluations	42	26	22	16

Table 4.3: Iteration sequence for different starting points using the SQP method with BFGS on Himmelblau's test problem in (4.8).

	$x_0 = [0, 0]^T$		$x_0 = [-4, 2]^T$		$x_0 = [0, 3.5]^T$		$x_0 = [-4, -0.5]^T$	
#Iter	x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2
1	0	0	-4	2	0	3.5	-4	-0.5
2	5	5	-4.25	5	-1.1111	-0.4444	-2.7273	-1.0909
3	1.5441	0.61765	-3.5991	2.1335	-2.0072	-0.8029	-3.4559	-1.3824
4	2.2007	0.88029	-3.5258	2.3227	-5	-2	-3.6028	-1.4411
5	5	2	-4.4014	5	-3.2812	-1.3125	-3.5461	-1.4184
6	1.8584	5	-3.7085	2.4387	-3.4476	-1.379	-3.5485	-1.4194
7	-0.07469	-0.0299	-3.6008	2.5508	-3.5634	-1.4253	-3.5485	-1.4194
8	1.5909	0.69532	-3.7292	2.9737	-3.5478	-1.4191	-3.5485	-1.4194
9	5	2.1725	-3.649	2.7128	-3.5485	-1.4194	-	-
10	2.1313	1.1133	-3.6537	2.7347	-3.5485	-1.4194	-	-
11	2.5331	1.4344	-3.6546	2.7378	-3.5485	-1.4194	-	-
12	3.5038	2.2659	-3.6546	2.7377	-	-	-	-
13	2.9387	1.8153	-3.6546	2.7377	-	-	-	-
14	3.0171	1.911	-	-	-	-	-	-
15	3.0297	1.9551	-	-	-	-	-	-
16	3.0219	1.9778	-	-	-	-	-	-
17	3.0035	2.0014	-	-	-	-	-	-
18	3.0003	2.0006	-	-	-	-	-	-
19	3	2	-	-	-	-	-	-
20	3	2	-	-	-	-	-	-
21	3	2	-	-	-	-	-	-

4.7.2 SQP with BFGS and Line Search

Figure 4.3 shows the results using SQP with BFGS and line search on Himmelblau's test problem. Compared to the results above this is already an improvement. The solver manages to locate all minima in quite few iterations, see Table 4.4. Table 4.5 also shows that it only takes a few iterations for the solver to get very close to the minimum, with good precision. The only downside is the increased number of function evaluations compared to using only the BFGS, which for large problems could cause an increase in computation time. The line search has indeed done a good job in improving convergence by controlling the step size.

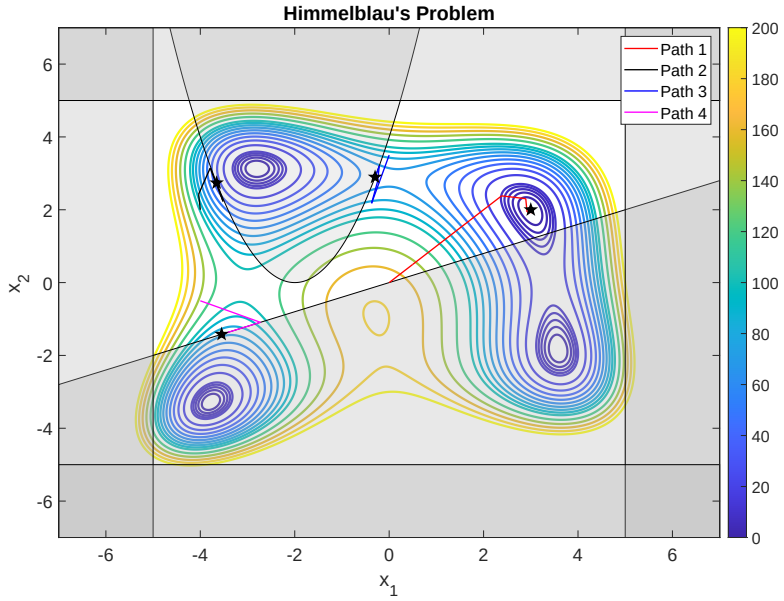

Figure 4.3: SQP with damped BFGS approximation and line search on Himmelblau's test problem. The colored lines show the iteration sequence and the star represents the minimizer.

Table 4.4: Solver statistics for the SQP method with BFGS and line search on Himmelblau's test problem in (4.8).

$\mathbf{x_0}$	$[0,0]^T$	$[-4,2]^T$	$[0,3.5]^T$	$[-4,-0.5]^T$
No. Iterations	15	10	9	8
Function evaluations	86	40	40	30

Table 4.5: Iteration sequence for different starting points using the SQP method with BFGS and line search on Himmelblau's test problem in (4.8).

	$x_0 = [0, 0]^T$		$x_0 = [-4, 2]^T$		$x_0 = [0, 3.5]^T$		$x_0 = [-4, -0.5]^T$	
#Iter	x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2
1	0	0	-4	2	0	3.5	-4	-0.5
2	0.5	0.5	-4.0357	2.4282	-0.37241	2.1779	-2.7273	-1.0909
3	1.1408	1.1408	-3.7905	3.1456	-0.24393	2.7827	-3.4559	-1.3824
4	1.709	1.709	-3.5177	2.229	-0.20917	3.2059	-3.6028	-1.4411
5	2.0899	2.0899	-3.6246	2.628	-0.29861	2.8867	-3.5461	-1.4184
6	2.3809	2.3809	-3.6675	2.7787	-0.29869	2.8945	-3.5485	-1.4194
7	2.6661	2.3394	-3.6539	2.735	-0.29836	2.8956	-3.5485	-1.4194
8	2.8932	2.303	-3.6546	2.7377	-0.29835	2.8956	-3.5485	-1.4194
9	2.9206	1.9389	-3.6546	2.7377	-0.29835	2.8956	-	-
10	3.0264	1.9216	-3.6546	2.7377	-	-	-	-
11	2.9875	1.977	-	-	-	-	-	-
12	2.9991	1.9997	-	-	-	-	-	-
13	2.9998	2.0003	-	-	-	-	-	-
14	3	2	-	-	-	-	-	-
15	3	2	-	-	-	-		-

4.7.3 SQP with Trust Region

Figure 4.4 shows the results using SQP with trust region on Himmeblau's problem. For the trust region strategy the damped BFGS approximation was used to approximate the Hessian. As can be seen the path is much cleaner now than what was seen for the other methods and Table 4.7 shows that the solver requires very few steps to get good precision. Notice again though in Table 4.6 the high number of function evaluations compared to the SQP with only BFGS. They are however not as extreme as for the SQP solver with line search. Based on these findings, it can be concluded that the SQP solver with trust region has the best performance out of all three implemented solvers for this problem.

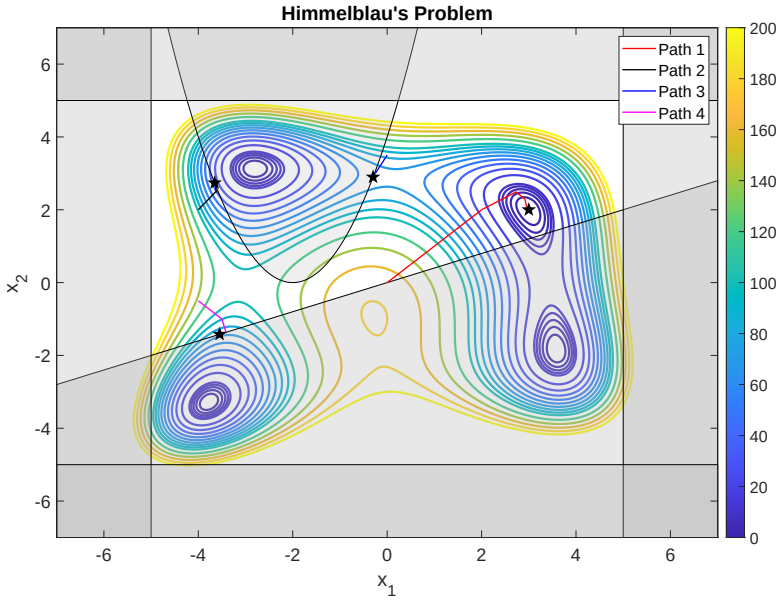

Figure 4.4: SQP with damped BFGS approximation and trust region strategy on Himmelblau's test problem. The colored lines show the iteration sequence and the star represents the minimizer.

Table 4.6: Solver statistics for the SQP method with trust region on Himmelblau's test problem in (4.8).

$\mathbf{x_0}$	$[0,0]^T$	$[-4,2]^T$	$[0,3.5]^T$	$[-4,-0.5]^T$
No. Iterations	14	9	7	9
Function evaluations	37	24	20	26

Table 4.7: Iteration sequence for different starting points using the SQP method with trust region on Himmelblau's test problem in (4.8).

	$x_0 = [0,0]^T$		$x_0 = [-4,2]^T$		$x_0 = [0,3.5]^T$		$x_0 = [-4,-0.5]^T$	
#Iter	x_1	x_2	x_1	x_2	x_1	x_2	x_1	x_2
1	0	0	-4	2	0	3.5	-4	-0.5
2	0.5	0.5	-3.625	2.5	-0.25	3	-3.5	-1
3	1.5	1.5	-3.625	2.5	-0.2906	2.9204	-3.4151	-1.3661
4	1.5	1.5	-3.6325	2.6651	-0.29784	2.8973	-3.424	-1.3696
5	2	2	-3.6605	2.7565	-0.29834	2.8956	-3.4655	-1.3862
6	2.7267	2.486	-3.6544	2.737	-0.29835	2.8956	-3.5539	-1.4215
7	2.7267	2.486	-3.6546	2.7377	-0.29835	2.8956	-3.5483	-1.4193
8	2.9158	2.297	-3.6546	2.7377	-	-	-3.5485	-1.4194
9	2.953	2.0919	-3.6546	2.7377	-	-	-3.5485	-1.4194
10	3.0076	2.0144	-	-	-	-	-	-
11	2.9983	1.9993	-	-	-	-	-	-
12	3	2	-	-	-	-	-	-
13	3	2	-	-	-	-	-	-
14	3	2	-	-	-	-	-	-

4.8 Testing the SQP Algorithms on Nonlinear Test Problems

To further test the efficiency of the SQP solvers implemented in this chapter, they were tested on two nonlinear test problems. For comparison the solution obtained was compared to the solution from *fmincon*. The driver and corresponding functions for performing the tests are listed in Appendix A.2.5. The first nonlinear test problem considered is defined as:

$$\min_x \quad 3sin(x_1) + x_1x_2 + x_2^2 \tag{4.32a}$$

$$s.t. \quad 0 \leq x_1 + x_2 - 1 \leq 3 \tag{4.32b}$$

$$0 \leq x_1 + 4x_2 - 2 \leq 3 \tag{4.32c}$$

$$-1 \leq x_1 \leq 2 \tag{4.32d}$$

$$-1 \leq x_2 \leq 2 \tag{4.32e}$$

A contour plot of the objective function can be seen in Figure 4.5. A starting point of $x_0 = [0,0]^T$ was used. The results can be seen in Table 4.8. All solvers managed to obtain a minimum given the constraints. The solvers even required fewer function evaluations and iterations than *fmincon*. Contrary to the conclusion for

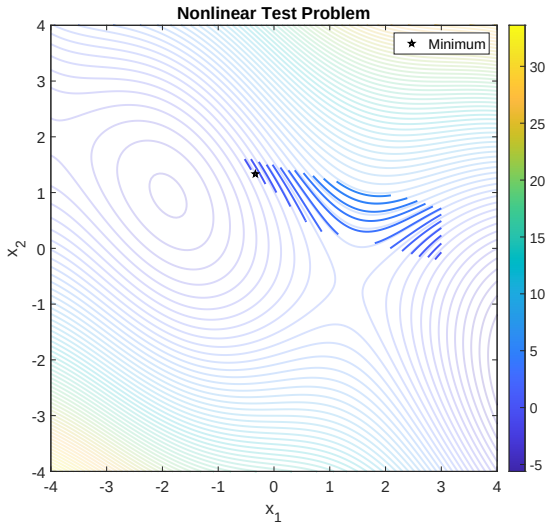

Figure 4.5: A contour plot of the nonlinear test function in (4.32). The feasible region is the non-shaded area.

the Himmelblau's problem, the SQP with BFGS had the best performance for this problem.

Table 4.8: Testing the SQP solvers and *fmincon* on the nonlinear test function in (4.32).

	x^*	$f(x^*)$	#Iters	Function evals
fmincon	$[-0.3333 1.3333]^T$	0.3518	8	28
SQP BFGS	$[-0.3333 1.3333]^T$	0.3517	2	6
SQP Line Search	$[-0.3333 1.3333]^T$	0.3517	3	16
SQP Trust Region	$[-0.3333 1.3333]^T$	0.3517	4	14

Next, the solvers were tested on Rosenbrock's function with the addition of two nonlinear constraints and lower and upper bounds. They were then compared to *fmincon*. The problem is defined as:

$$\min_{x} \quad 100(x_2 - x_1^2) + (1 - x_1)^2 \tag{4.33a}$$
$$s.t. \quad -1 \leq (x_1 + 1)^2 - x_2 \leq 3 \tag{4.33b}$$
$$-1 \leq (x_1 - 1) - x_2 \leq 3 \tag{4.33c}$$
$$-1 \leq x_1 \leq 0.8 \tag{4.33d}$$
$$-1 \leq x_2 \leq 0.8 \tag{4.33e}$$

The contour plot of the function can be seen in Figure 4.6. The test was performed for a starting point $x_0 = [0, 0]^T$. The results can be seen in Table 4.9. It seems the SQP with only the BFGS approximation has not managed to find a minimum. Additionally, the trust region strategy still manages to outperform the line search with regards to the function evaluations and even performs better than *fmincon*. However, for this problem the number of iterations were higher for the trust region than the line search but only slightly. To look better into the performance difference the iteration sequences can be seen in Table 4.10. Apparently, the reason for the higher number of iterations with the trust region strategy is due to the solver not finding an acceptable step for the first four iterations. This illustrates the importance of choosing a good starting point. To conclude, the SQP with BFGS does not work well for very strict constraints such as in this problem, while the SQP with line search and trust region

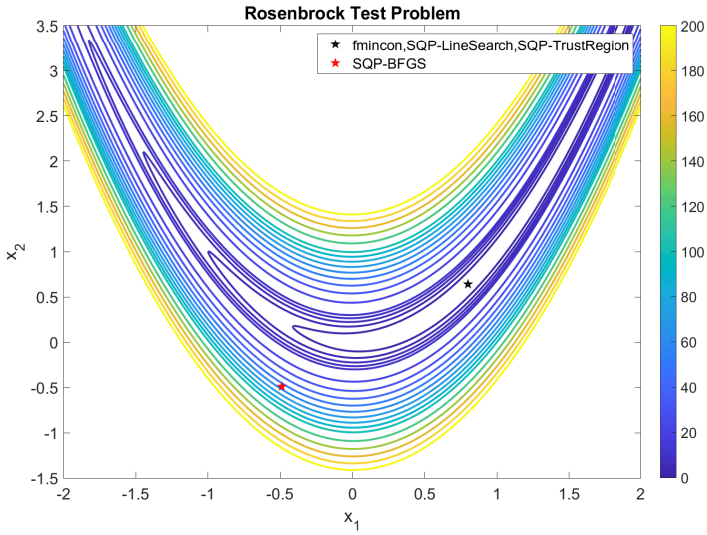

Figure 4.6: A contour plot of the Rosenbrock function in (4.33).

manage to perform quite efficiently.

Table 4.9: Testing the SQP solvers and *fmincon* on Rosenbrock's function in (4.33).

	x^*	$f(x^*)$	#Iters	Function evals
fmincon	$[0.800, 0.6400]^T$	0.0400	24	77
SQP BFGS	$[-0.4989, -0.4989]^T$	55.4639	17	36
SQP Line Search	$[0.800, 0.6400]^T$	0.0400	18	82
SQP Trust Region	$[0.800, 0.6400]^T$	0.0400	21	59

Table 4.10: Comparing iteration sequences between the SQP solver with line search and with trust region on Rosenbrock's problem.

	SQP Line Search		**SQP Trust Region**	
Iteration No.	x_1	x_2	x_1	x_2
1	0	0	0	0
2	0.0080	-0.0080	0	0
3	0.0098	-0.0097	0	0
4	0.0239	-0.0043	0	0
5	0.0591	-0.0113	0.0078	-0.0078
6	0.1121	0.0062	0.0157	-0.0001
7	0.2119	0.0122	0.0310	0.0004
8	0.2694	0.0567	0.0610	0.0010
9	0.4065	0.1323	0.1197	0.0023
10	0.4572	0.2120	0.2142	0.0245
11	0.5909	0.3234	0.3385	0.0829
12	0.5639	0.3137	0.3905	0.1506
13	0.6195	0.3793	0.5291	0.2518
14	0.7277	0.5112	0.5126	0.2570
15	0.7340	0.5340	0.5850	0.3369
16	0.8000	0.6352	0.7009	0.4691
17	0.8000	0.6408	0.6976	0.4832
18	0.8000	0.6400	0.7677	0.5846
19	0.8000	0.6400	0.8000	0.6369
20	-	-	0.8000	0.6425
21	-	-	0.8000	0.6400

CHAPTER 5

Markovitz Portfolio Optimization

In this chapter the use of quadratic programming in financial application is illustrated where Markovitz' Portfolio optimization problem is formulated as a quadratic program. The financial market with five securities considered in this chapter can be seen in Table 5.1. The driver for implementing the tests in this chapter can be found in Appendix A.2.6.

Table 5.1: 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 Optimal Solution as Function of Return

Example 16.1 in Nocedal & Wright [4] shows how to formulate a portfolio problem into a quadratic progam. The return on the portfolio is given by:

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

and assuming investment of all available funds and not allowing short-selling, the constraints can be defined as:

$$\sum_{i=1}^{n} x_i = 1,$$
$$x \geq 0.$$

The expected return and variance of the portfolio are given by:

$$E[R] = E\big[\sum_{i=1}^{n} x_i r_i\big] = \sum_{i=1}^{n} x_i E[r_i] = x^T \mu,$$
$$Var[R] = E[(R - E(R))^2] = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j \sigma_i \sigma_j \rho_{ij} = x^T H x,$$

where H is an $n \times n$ symmetric positive definite matrix and is defined as the covariance between assets i and j, $H_{ij} = \rho_{ij}\sigma_i\sigma_j$. The expected return and variance are important to be able to measure the desirability of the portfolio.

Now, ideally one would like to find a portfolio for which the variance or risk, $x^T H x$, is small while the expected return, $x^T \mu$, is large. By combining the two aforementioned aims a single objective function can be made and to obtain the optimal portfolio the following minimization problem is solved:

$$\min_x \quad -x^T \mu - \kappa x^T H x \tag{5.1a}$$
$$s.t. \quad \sum_{i=1}^{n} x_i = 1 \tag{5.1b}$$
$$x \geq 0 \tag{5.1c}$$

where x_i is the fraction (weight) of the portfolio invested in asset i and κ is a nonnegative risk tolerance parameter whose value depends on the preferences of the individual investor. Smaller value of κ means more risk but there is a possibility of higher expected return, while a large value minimizes risk by increasing the weight on the variance. According to Table 5.1 the minimum possible return is expected to be $R_{min} = 8.50$ and the maximal possible return $R_{max} = 18.68$.

5.1.1 Solving the Problem for a given Return

The problem given in (5.1c) was solved for a given return $R = 12.00$ to obtain the optimal portfolio with minimal risk. To do this the problem needed to be reformulated to include the return as a constraint:

$$\min_x \quad -x^T \mu - \kappa x^T H x \tag{5.2a}$$
$$s.t. \quad \sum_{i=1}^{n} x_i \mu = R \tag{5.2b}$$
$$\sum_{i=1}^{n} x_i = 1 \tag{5.2c}$$
$$x \geq 0 \tag{5.2d}$$

with $n = 5$ assets/securities. Using *quadprog* the optimal portfolio was obtained along with the minimal risk, $Var[R]$, and expected return:

$$x = \begin{bmatrix} 0.000 \\ 0.4765 \\ 0.2551 \\ 0.1234 \\ 0.1449 \end{bmatrix}, \quad Var[R] = 0.7654\ , \quad E[R] = 12.00.$$

Investors must determine the level of diversification that suits them best and this can be determined through what is called the efficient frontier. The efficient frontier is a graphical representation of a set of optimal portfolios. An optimal portfolio, according to Markovitz's theory, is designed such that there is a perfect balance of risk and return. That is, given a certain level of expected return offer the lowest risk and vice versa. A portfolio that falls outside of the efficient frontier is considered sub-optimal as it either carries too much risk compared to its return, or too little return compared to its risk [7].

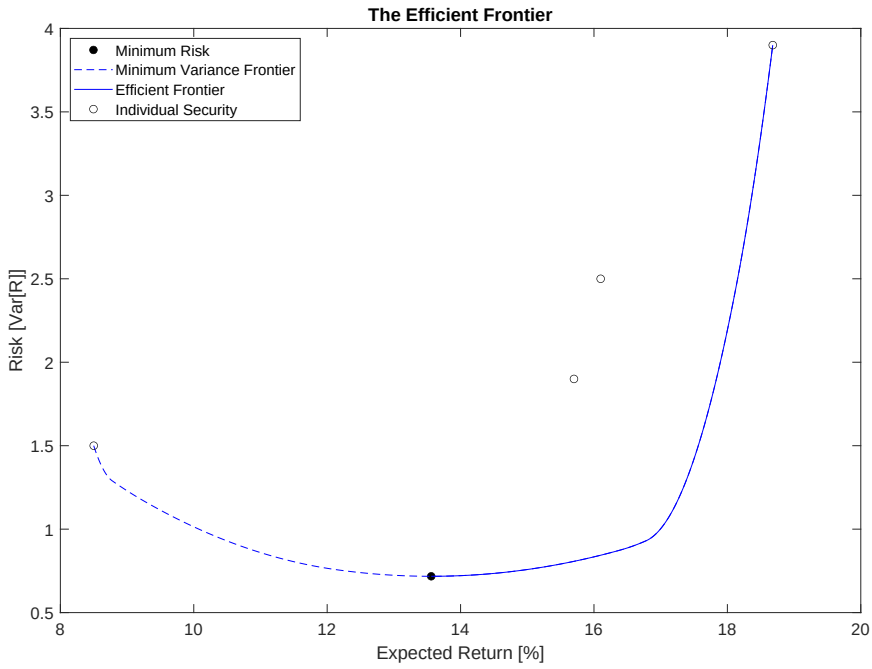

Figure 5.1: The efficient frontier for the financial market given in Table 5.1.

5.1.2 The Efficient Frontier

The efficient frontier can be found by iterating over an interval between the minimum and maximum expected returns for 10,000 equidistant points and computing the associated risks. The efficient frontier, i.e., the risk as function of the return, for the five securities can be seen in Figure 5.1. Each point on the plot represents a portfolio, and the ones closest to the efficient frontier have the potential to produce the greatest return with the lowest degree of risk. As can be seen the efficient frontier is the upper portion of the minimum variance frontier starting from the minimum risk, also called the global minimum variance (filled black point). Using *quadprog* the optimal portfolio with minimum risk was obtained:

$$x_{opt} = \begin{bmatrix} 0.0870 \\ 0.3076 \\ 0.3016 \\ 0.1000 \\ 0.2039 \end{bmatrix}, \quad Var[R] = 0.7176 \ , \quad E[R] = 13.56.$$

Figure 5.2 shows the optimal portfolios for the five securities as function of return for the financial market in Table 5.1.

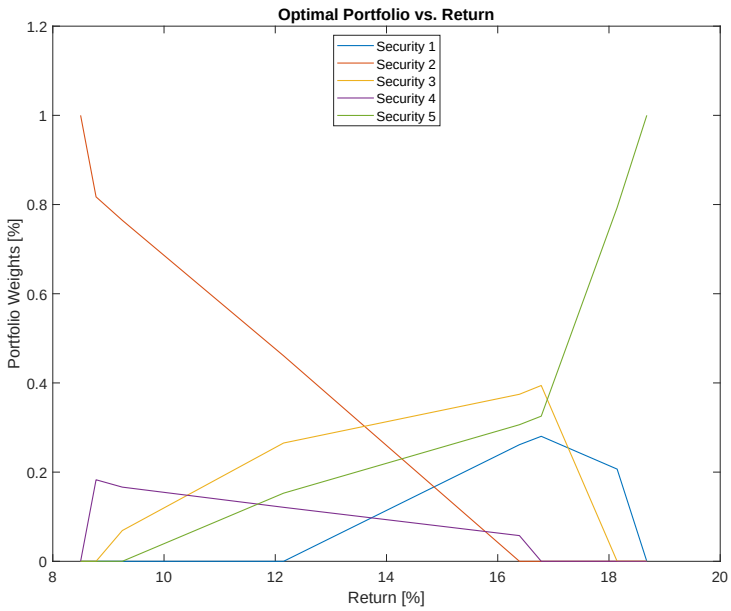

Figure 5.2: The optimal portfolios as a function of return for the five securities of the financial market given in Table 5.1.

5.2 Bi-Criterion Optimization

The optimization problem given in (5.1c) can be reformulated as a bi-criterion of the variance and the return. It then takes the form:

$$\min_x \ \alpha x^T H x - (1-\alpha) x^T \mu \tag{5.3a}$$

$$s.t. \ \sum_{i=1}^{n} x_i = 1 \tag{5.3b}$$

$$x \geq 0, \tag{5.3c}$$

where R is now the stochastic portfolio return and $\alpha \in [0, 1]$ is the risk proxy that controls the weight on the risk and return.

5.2.1 Comparing Algorithms on the Bi-Criterion Problem

A comparison was made for the algorithms implemented in Chapter 1 and Chapter 2 on the bi-criterion optimization problem, that is by allowing short-selling and not allowing short-selling. This is because solving the problem as an equality constrained quadratic program involves dropping the inequality constraints in (5.2d) and thus allowing short-selling. This problem is defined as:

$$\min_x \ \alpha x^T H x - (1-\alpha) x^T \mu \tag{5.4a}$$

$$s.t. \ \sum_{i=1}^{n} x_i = 1 \tag{5.4b}$$

For the EQP solver the Range Space method was used as H is a positive definite matrix, see Listing 1.5. The optimal portfolios were found using 1000 equally spaced values of $\alpha \in [0, 1]$ and the corresponding efficient frontiers computed using the two different algorithms can be visualized in Figure 5.3 and Figure 5.4. According to Figure 5.3, the short-selling makes the expected returns increase almost infinitely with extreme risks. This makes sense as short-selling can cause a trader to lose more than 100% of his original investment. However, there is possibility of high profits which explains why this is a common approach [8].

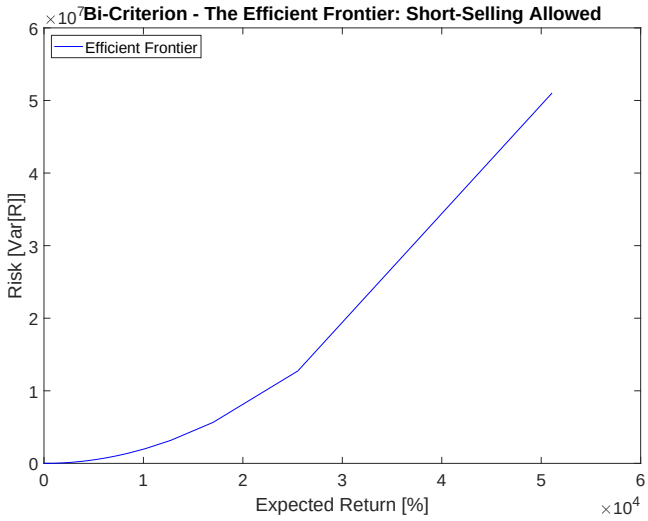

Figure 5.3: The efficient frontier for the bi-criterion problem in 5.4 when short-selling is allowed.

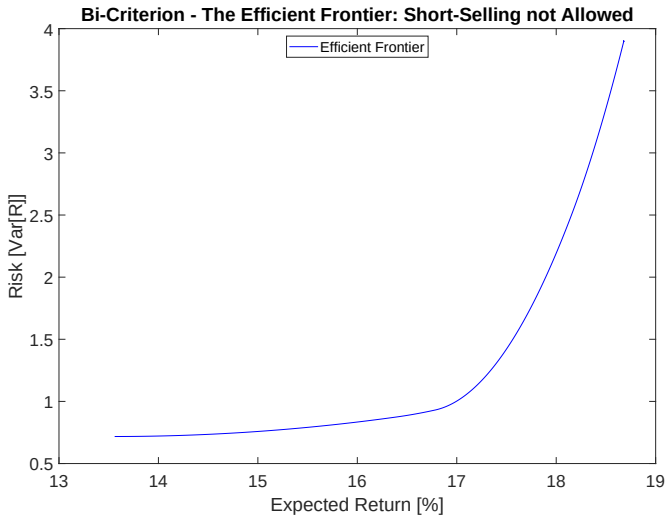

Figure 5.4: The efficient frontier for the bi-criterion problem in 5.3 when short-selling is not allowed.

Figure 5.4 shows the efficient frontier for the bi-criterion problem when short-selling is not allowed. As can be seen the bi-criterion causes the optimal portfolios to all fall in the efficient frontier, this is what is often called ”pareto efficiency”.

5.2.2 Comparing with Other Libraries

To test the performance of the EQP solver for this problem it was compared with *quadprog*. This was done for 20 equally spaced values of $\alpha \in [0, 1]$. Figure 5.5 shows the CPU time for both methods. When the time hits 0 it is due to the solver being too fast for Matlab's *cputime* to give any information. Figure 5.6 shows the error between the methods, which apparently does not deviate from more than 10^{-14}. Notice though that the error is highest when α is close to 0. This is due to the covariance matrix becoming close to singular and the Cholesky factorization in the range-space method requires the matrix to be positive definite in order to work efficiently. In this case the LDL factorization would be more useful as it only requires symmetry of the matrix. Surprisingly, this was also then *quadprog* was slowest. Still, it can be concluded that the EQP range-space solver is most efficient for larger values of α.

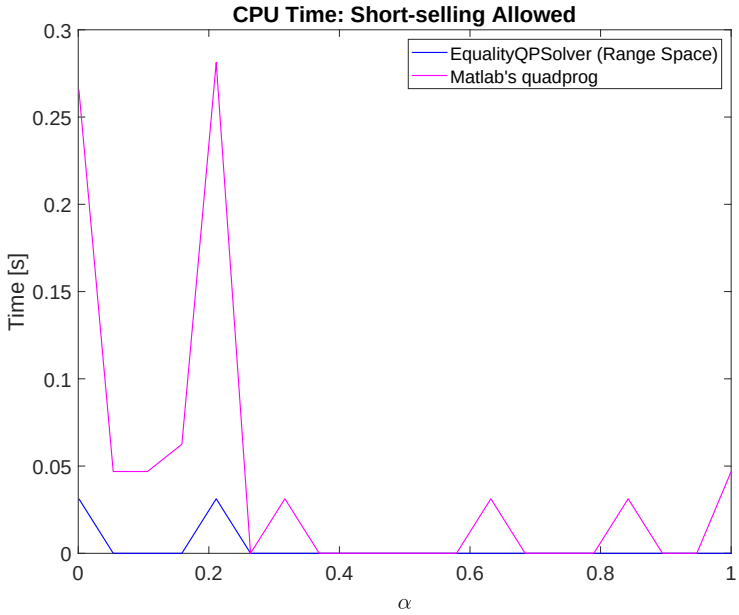

Figure 5.5: Comparison of the optimal portfolios computed using the EQP algorithm implemented in Chapter 1 with Matlab's *quadprog*. Short-selling allowed.

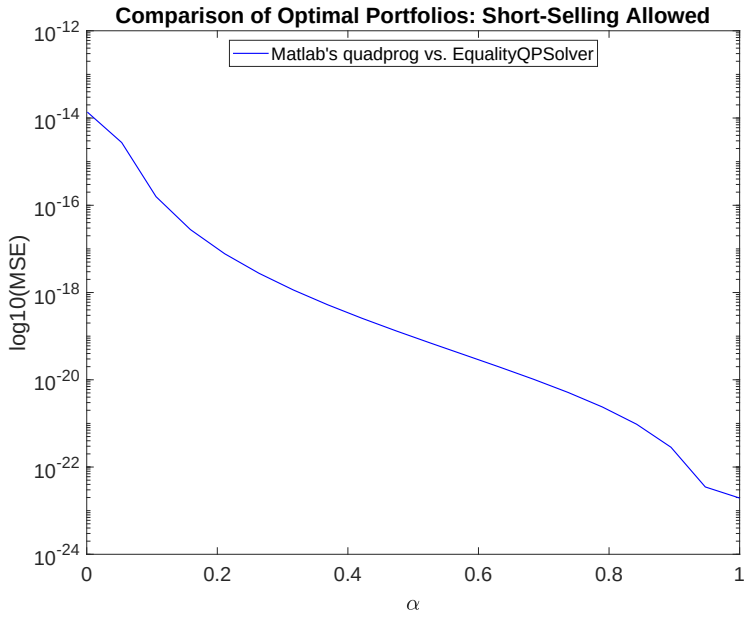

Figure 5.6: Comparison of the optimal portfolios computed using the EQP algorithm implemented in Chapter 1 with Matlab's *quadprog*, MOSEK, Gurobi and CVX, along with CPU time for each solver.

Now, to test the performance of the primal-dual interior-point QP solver for this problem it was compared with the QP libraries *quadprog*, MOSEK, CVX and Gurobi. This was done for the same values of α. For CVX the default *SDPT3* solver was used and for ease of implementation MOSEK was used within CVX. Figure 5.7 shows the CPU time for each method, with CVX and MOSEK being by far the slowest. The reason for MOSEK being slow could be due to it being used within CVX, although it still performs faster overall. As for the error between the implemented solver and the QP libraries, which can be seen in Figure 5.8, it is lowest for lower values of α. It never deviates from more than about 10^{-7} and the highest error is mostly between the solver and MOSEK. Overall, the solver is performing well for this type of problem and is not as influenced by α as the EQP solver.

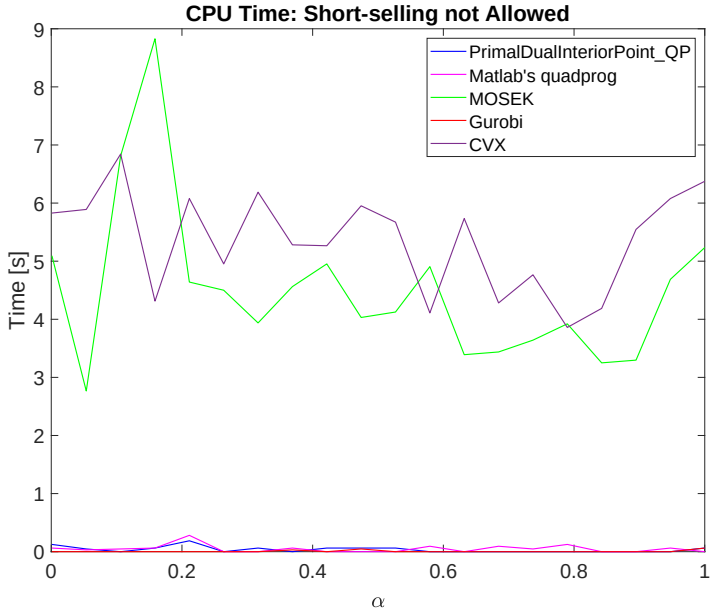

Figure 5.7: Comparison of CPU time for the primal-dual interior-point QP algorithm implemented in Chapter 2 with Matlab's *quadprog*, MOSEK, Gurobi and CVX. Short selling not allowed.

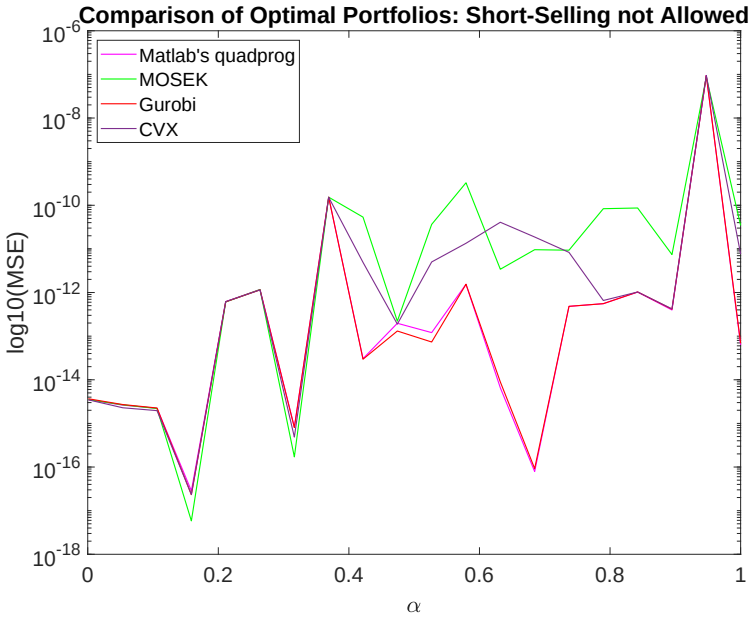

Figure 5.8: Mean squared error between the primal-dual interior-point QP algorithm implemented in Chapter 2 and Matlab's *quadprog*, MOSEK, Gurobi and CVX. Short selling not allowed.

5.3 Risk-Free Asset

A risk-free security is added to the financial markter given in Table 5.1 with return $r_f = 0.0$. This yields a new financial market with six securities that can be seen in Table 5.2.

Table 5.2: A financial market with 6 securities (added risk-free security).

Security	Covariance						Return
1	2.50	0.93	0.62	0.74	-0.23	0.00	16.10
2	0.93	1.50	0.22	0.56	0.26	0.00	8.50
3	0.62	0.22	1.90	0.78	-0.27	0.00	15.70
4	0.74	0.56	0.78	3.60	-0.56	0.00	10.02
5	-0.23	0.26	-0.27	-0.56	3.90	0.00	18.68
6	0.00	0.00	0.00	0.00	0.00	0.00	0.00

5.3.1 The Efficient Frontier

Figure 5.9 shows the efficient frontier for the updated financial market with risk-free security, along with the individual securities. This was computed with 10,000 equidistant points as in Section 5.1. The optimal portfolio for each security as function of return can be seen in Figure 5.10. As can be seen the first five securities all increase in a linear fashion while the risk-free security has an erratic behaviour, going against the motion of the other securities. The efficient frontier is visualized in Figure 5.9 where the frontier now goes all the way down to zero risk and return compared to before without the risk-free asset.

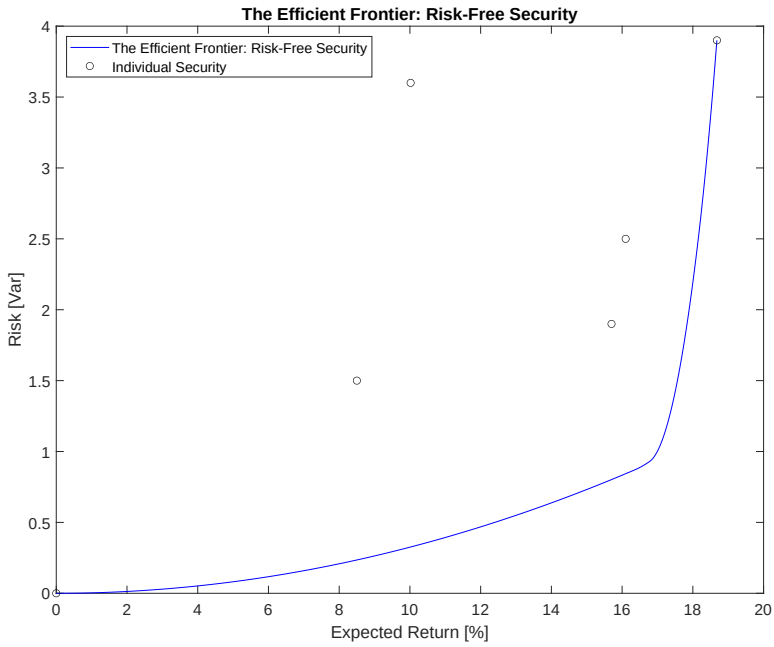

Figure 5.9: The efficient frontier for the financial market with risk-free security given in Table 5.2.

In general, at the predicted level, risk free investments are considered fairly certain to gain. This gain is essentially known and thus the rate of return is often considerably lower to reflect the lower amount of risk. A risk-free security is one that is free of various possible sources of risk and for which the expected return versus the actual return are expected to be roughly the same. Additionally, “they tend to have lower rates of return, since their safety means investors don’t need to be compensated for taking a chance” [9].

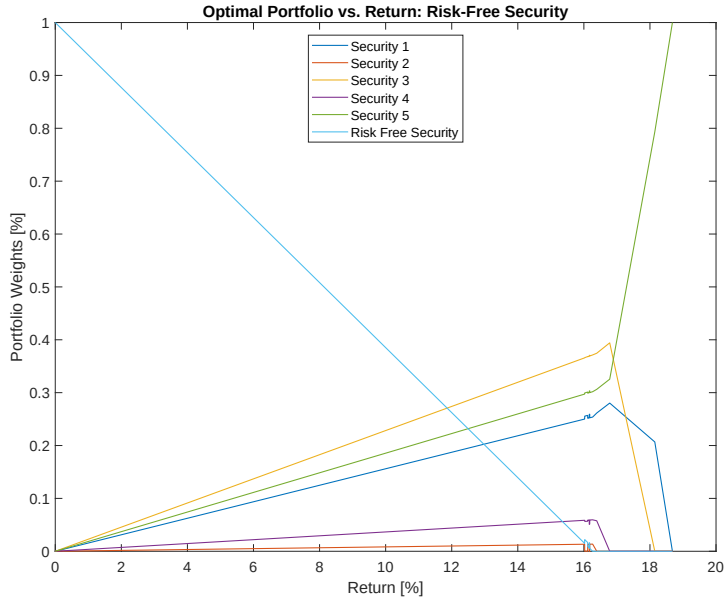

Figure 5.10: The optimal portfolio as a function of return for the securities in the financial market with risk-free security given in Table 5.2.

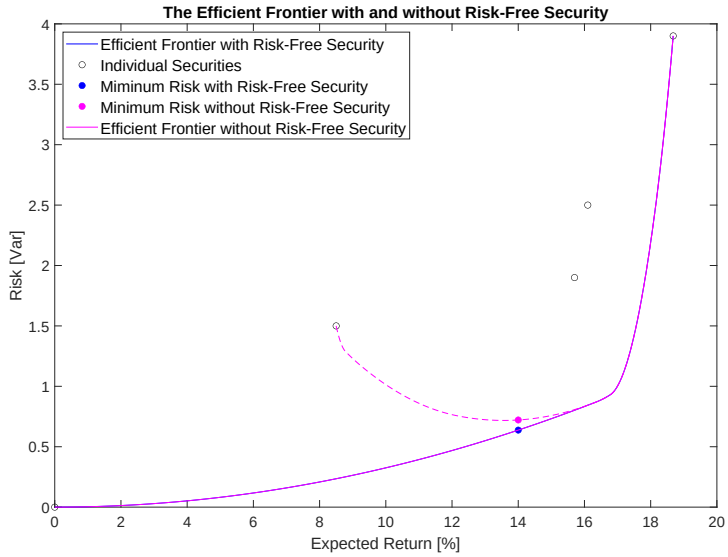

Figure 5.11: The efficient frontier with and without risk-free security along with the minimum risk given a return $R = 14.00$.

5.3.2 Solving the Problem for a given Return

The problem given in (5.2d) is now solved for a given return $R = 14.00$. The results are visualized in Figure 5.11 where the efficient frontier is compared to the efficient frontier in the previous financial market. The corresponding minimal risks are represented as a magenta filled point and blue filled point and as can be seen the risk is lower in the new market. Using *quadprog* the optimal portfolio along with the minimal risk and expected return are:

$$x_{rf} = \begin{bmatrix} 0.2185 \\ 0.0116 \\ 0.3196 \\ 0.0512 \\ 0.2598 \\ 0.1393 \end{bmatrix}, \quad Var[R] = 0.6377\ , \quad E[R] = 14.00.$$

Evidently, adding the risk-free security has decreased the risk down approximately 0.10 compared to the financial market without a risk-free security which for $R = 14.00$ had a minimal risk of $Var[R] = 0.7214$. These results reflect what was stated in the above-mentioned paragraph on risk-free security.

In the new financial market the covariance matrix is no longer positive definite and thus if short-selling is allowed, then an EQP solver using LDL factorization or the null-space method would be most applicable. The corresponding KKT matrix will still be symmetric indefinite and thus *quadprog* and the above-mentioned primal-dual interior-point QP solver are both appropriate methods in this case with short selling not being allowed.

APPENDIX A

Appendix

A.1 Algorithms

A.1.1 Problem 2 Algorithms

```
function [x,stat] = PrimalDualInteriorPoint_QP(H,g,A,b,C,d,x0,y0,z0,s0)
% PrimalDualInteriorPoint_QP A primal-dual interior point solver based
% on Mehrotra's predictor-corrector modification.
%
%                  min 1/2*x'Hx + g'x
%                    s.t.  A'x = b
%                         l <= x <= u
%
% x : Solution
% y : Lagrange multipliers for equality constraints
% z : Lagrange multipliers for inequality constraints
% s : Slack variables
% stat: Collects variable information
%
% Syntax: [x,stat] = PrimalDualInteriorPoint_QP(H,g,A,b,C,d,x0,y0,z0,s0)

% Constants
max_iters = 1e3;
epsilon = 1e-6; % Tolerance
eta = 0.995; % Dampening factor

% Get dimensions of x, equality and inequality constraints
mX = length(x0);
mEQ = size(A,2);
mINEQ = size(C,2);


%% Initial point heuristic

% Initial values
x = x0;
y = y0;  % Lagrange multipliers for equality constraints
z = z0;  % Lagrange multipliers for inequality constraints
s = s0;  % Slack variables

% Compute residuals
```

```
rL = H*x + g - A*y - C'*z; % Lagrangian gradient
rA = b - A'*x;
rC = s + d - C*x;
rsz = s.*z;                % Complementry inequality slack

% Compute LDL factorization of modified KKT system
Hbar = H+C'*diag(z./s)*C;
KKT = [Hbar, -A;-A', zeros(mEQ)];
[L, D, p] = ldl(KKT, 'lower', 'vector');

% Compute affine search direction based on initial values
rLbar = rL - C'*diag(z./s)*(rC - rsz./z);

% Solve system and get delta directions
rhs = -[rLbar; rA];
sol(p,1) = L'\(D\(L\rhs(p,1)));
Dx = sol(1:mX);
Dz = -diag(z./s)*C*Dx + diag(z./s)*(rC - rsz./z);
Ds = -rsz./z-diag(s./z)*Dz;

% Update starting point
z = max(1, abs(z+Dz));
s = max(1, abs(s+Ds));


% Storage variables
stat.converged = 0;
stat.iter = 0;
stat.x = x; stat.y = y;
stat.z = z; stat.s = s;


%% Main program

% Duality gap measure
mu = (z'*s)/mINEQ;

while( stat.converged && stat.iter <= max_iters)

    % Compute LDL factorization of modified KKT system
    Hbar = H+C'*diag(z./s)*C; %Hbar = H+C*diag(z./s)*C;
    KKT = [Hbar, -A; -A', zeros(mEQ)];
    [L, D, p] = ldl(KKT, 'lower', 'vector');

    % Solve system and get delta directions
    rLbar = rL - C'*diag(z./s) * (rC - rsz./z);
    rhs = -[rLbar; rA];

    solAff(p,1) = L'\(D\(L\rhs(p,1)));
    DxAff = solAff(1:mX);
    DzAff = -diag(z./s)*C*DxAff + diag(z./s)*(rC - rsz./z);
    DsAff = -rsz./z - diag(s./z) * DzAff;

    % Compute largest valid alpha affine
    DzsAff = [DzAff;DsAff]; %DeltaUnion = [DZ; DS];
```

```
92      alphas = -[z;s]./DzsAff; %(-[z;s]./DeltaUnion);
93      alpha_aff = min([1;alphas(DzsAff<0)]);
94
95      % Compute the affine duality gap
96      mu_aff = (z + alpha_aff*DzAff)' * (s+alpha_aff*DsAff)/mINEQ;
97
98      % Compute the centering parameter
99      sigma = (mu_aff/mu)^3;
100
101     % Affine centering-correction direction
102     rszbar = rsz + DsAff.*DzAff - sigma*mu;
103     rLbar = rL - C'*diag(z./s)*(rC-rszbar./z);
104     rhs = -[rLbar; rA];
105
106     sol(p,1) = L'\(D\(L\rhs(p,1)));
107     Dx = sol(1:mX);
108     Dy = sol(mX+(1:mEQ));
109     Dz = -diag(z./s)*C*Dx+diag(z./s)*(rC - rszbar./z);
110     Ds = -rszbar./z - diag(s./z) * Dz;
111
112     % Finding the largest valid alpha and apply eta-dampening
113     Dzs = [Dz;Ds];
114     alphas = (-[z;s]./Dzs);
115     alpha = min([1;alphas(Dzs<0)])*eta;
116
117     % Take step
118     x = x + alpha*Dx;
119     y = y + alpha*Dy;
120     z = z + alpha*Dz;
121     s = s + alpha*Ds;
122
123     % Update residuals and duality gap measure
124     rL = H*x + g - A*y - C'*z;
125     rA = b - A'*x;
126     rC = s + d - C*x;
127     rsz = s.*z;
128     mu = (z'*s)/mINEQ;
129
130     % Check for convergence
131     stat.converged = all([norm(rL,inf), norm(rA,inf), ...
132                           norm(rC,inf), abs(mu)] <= epsilon);
133
134     % Store variables
135     stat.x = [stat.x, x];
136     stat.y = [stat.y y];
137     stat.z = [stat.z z];
138     stat.s = [stat.s s];
139     stat.iter = stat.iter + 1;
140
141 end
142 end
```

Listing A.1: A primal-dual interior-point QP solver based on Mehrotra's modification for problem 2.

A.1.2 Problem 3 Algorithms

```
function [x,stat] = PrimalDualInteriorPoint_LP(g,A,b,C,d,x0,y0,z0,s0)
% PrimalDualInteriorPoint_LP A primal-dual interior point solver based
% on Mehrotra's perdictor-corrector modification.
% It solves linear problems of the form:
%
%                    min  g'x
%                 s.t.  A'x = b
%                       l <= x <= u
%
% x : Solution
% y : Lagrange multipliers for equality constraints
% z : Lagrange multipliers for inequality constraints
% s : Slack variables
% stat: Collects variable information
%
% Syntax: [x,stat] = PrimalDualInteriorPoint_LP(g,A,b,C,d,x0,y0,z0,s0)

% Constants
max_iters =1e3;
epsilon = 1e-6; % Tolerance
eta = 0.995; % Dampening factor

% Get dimensions of x, equality and inequality constraints
mX = length(x0);
mEQ = size(A,2);
mINEQ = size(C,2);


%% Initial point heuristic

% Initial values
x = x0;
y = y0;
z = z0;
s = s0;

% Compute residuals
rL = g - A*y - C'*z; % Lagrangian gradient
rA = b-A'*x;         % Equality constraint
rC = s + d - C*x;    % Inequality constraint
rsz = s.*z;          % Complementry inequality slack

% Affine step for start point

% Affine step for start point
Cdiag_inv = inv(C'*diag(z./s)*C);
rLbar = rL - C'*diag(z./s)*(rC - rsz./z);
norm_eq_LHS = A'*Cdiag_inv*A;
norm_eq_RHS = rA+A'*Cdiag_inv*rLbar;

L = chol(norm_eq_LHS);
```

```
52
53 % Compute affine search direction based on initial values
54 Dy = L\(L'\norm_eq_RHS);
55 Dx = Cdiag_inv*(-rLbar+A*Dy);
56 Dz = -diag(z./s)*C*Dx + diag(z./s)*(rC - rsz./z);
57 Ds = -rsz./z-diag(s./z)*Dz;
58
59
60 % Update starting point
61 z = max(1, abs(z+Dz));
62 s = max(1, abs(s+Ds));
63
64 % Storage variables
65 stat.converged = 0;
66 stat.iter = 0;
67 stat.x = x;
68 stat.y = y;
69 stat.z = z;
70 stat.s = s;
71
72
73 %% Main program
74 % Duality gap measure
75 mu = (z'*s)/mINEQ;
76
77
78 while( stat.converged && stat.iter <= max_iters)
79
80     %--- Affine direction--
81
82     % Compute Choleski factorization
83     Cdiag_inv = inv(C'*diag(z./s)*C);
84
85     rLbar = rL - C'*diag(z./s)*(rC - rsz./z);
86     norm_eq_LHS = A'*Cdiag_inv*A;
87     norm_eq_RHS = rA+A'*Cdiag_inv*rLbar;
88
89     L = chol(norm_eq_LHS);
90
91     % Compute affine search direction based on initial values
92     DyAff = L\(L'\norm_eq_RHS);
93     DxAff = Cdiag_inv*(-rLbar+A*DyAff);
94     DzAff = -diag(z./s)*C*DxAff + diag(z./s)*(rC - rsz./z);
95     DsAff = -rsz./z - diag(s./z) * DzAff;
96
97     % Compute largest valid alpha affine
98     DzsAff = [DzAff;DsAff];
99     alphas = -[z;s]./DzsAff;
100    alpha_aff = min([1;alphas(DzsAff<0)]);
101
102    % Computing the affine duality gap
103    mu_aff = (z + alpha_aff*DzAff)' * (s+alpha_aff*DsAff)/mINEQ;
104
105    % Compute the centering parameter
106    sigma = (mu_aff/mu)^3;
```

```
107
108     % Affine Centering-Correction Direction
109     rszbar = rsz + DsAff.*DzAff - sigma*mu;
110
111     Cdiag_inv = inv(C'*diag(z./s)*C);
112
113     rLbar = rL - C'*diag(z./s)*(rC - rszbar./z);
114     norm_eq_LHS = A'*Cdiag_inv*A;
115     norm_eq_RHS = rA+A'*Cdiag_inv*rLbar;
116
117     L = chol(norm_eq_LHS);
118
119     % Compute affine search direction based on initial values
120     Dy = L\(L'\norm_eq_RHS);
121     Dx = Cdiag_inv*(-rLbar+A*Dy);
122     Dz = -diag(z./s)*C*Dx+diag(z./s)*(rC-rszbar./z);
123     Ds = -rszbar./z - diag(s./z) * Dz;
124
125     % Finding the largest valid alpha and apply eta-dampening
126     Dzs = [Dz;Ds];
127     alphas = (-[z;s]./Dzs);
128     alpha = min([1;alphas(Dzs<0)])*eta;
129
130     % Take step
131     x = x + alpha*Dx;
132     y = y + alpha*Dy;
133     z = z + alpha*Dz;
134     s = s + alpha*Ds;
135
136     % Update residuals and duality gap measure
137     rL = g - A*y - C'*z;
138     rA = b - A'*x;
139     rC = s + d - C*x;
140     rsz = s.*z;
141     mu = (z'*s)/mINEQ;
142
143     % Check for convergence
144     stat.converged = (norm(rL,inf) <= epsilon) && ...
145             (norm(rA,inf) <= epsilon) && ...
146             (abs(mu) <= epsilon);
147
148     % Store variables
149     stat.x = [stat.x, x];
150     stat.y = [stat.y y];
151     stat.z = [stat.z z];
152     stat.s = [stat.s s];
153     stat.iter = stat.iter + 1;
154
155 end
156 end
```

Listing A.2: A primal-dual interior-point LP solver based on Mehrotra's modification for problem 3.

A.1.3 Problem 4 Algorithms

```
function [x,stat] = SQP_BFGS(x0,lb,ub,gl,gu,obj,con)
% SQP_BFGS Solves a sequential quadratic programming problem
% with a damped BFGS approximation to the Hessian matrix.
% Solves for nonlinear programs of the form:
%
%               min f(x)
%        s.t. gl <= g(x) <= gu
%              l <= x <= u
%
% Syntax: [x,y,info,k]=SQP_BFGS(x,y,obj,con)


x = x0;

% Evaluate func and constraints
[f,df] = feval(obj,x);
[g,dg] = feval(con,x);

n = length(x);
m = length(g);

%m = size(c,1) ;
z = ones(2*m+2*n,1);
lid = 1:m;
uid = (m+1):(2*m) ;
glid = (2*m+1):(2*m+n) ;
guid = (2*m+n+1):(2*(n+m)) ;

% For quadprog
options = optimset('display','off');

eta = 1e-06;
maxiter = 100;
k = 0;
B = eye(n); % Hessian approximation initial value


% Lagrangian gradient
%dL = df - dg*y;

% Store data
stat.X = x;
stat.F = f;
stat.dF = df;
stat.dL = [];
stat.iter = 0;
stat.function_calls = 2;
%stat.Y = y;

dL = 10000;
penalty = 1000;
```

```
xprev =100;
stat.converged=0;

%% Main program
while ( stat.converged &&  norm(x-xprev,'inf') > eta)

    xprev=x;
    % Update lower and upper bounds
        % Bounds for x
    lbk = -x+lb;
    ubk = -x+ub;
    % Bounds for g(x)
    glk = -g+gl;
    guk = -g+gu;

    %Compute pk with quadprog
    C=-[dg';-dg'];
    d=-[glk;-guk];


    [dx, , , ,lambda] = quadprog(B,df,C,d,[],[],lbk,ubk,[],options);

    if any(isempty(dx)) % | isnan(dx) == true)
        error('Problem is infeasible.')

    end


    zhat = [lambda.lower; lambda.upper; lambda.ineqlin];

    pz = zhat -z;

    % Take step
    zold = z;
    z = z + pz;
    xold = x;
    x = x + dx;

    % Lagrangian gradient with y_k+1 and old x
    %dL = df - dc*y;
    %dL = df - (z(lid)-z(uid)+dg*z(glid)-dg*z(guid));
    dL = (df' - zhat'*[eye(n); -eye(n); dg ; -dg])';

    % Evaluate functions
    [f,df] = feval(obj,x);
    [g,dg] = feval(con,x);
    stat.function_calls = stat.function_calls+ 2;

    dLnew =  (df' - zhat'*[eye(n); -eye(n); dg ; -dg])';

    % --- Damped BFGS updating ----
    % Compute p and q
    p = dx; %x - xold;
    q = dLnew - dL;

```

```
107     % Update Hessian matrix by the modified BFGS procedure
108     Bp = B*p;
109     criteria = (p'*q) >= (0.2*(p'*Bp)); % Here we are ensuring positive
            definiteness
110     if criteria
111         theta = 1;
112     else
113         theta = (0.8*p'*Bp) / (p'*Bp - p'*q);
114     end
115     r = theta*q + (1-theta)*Bp;
116
117     % Update B
118     B = B + (r*r')/(p'*r) - (Bp*Bp')/(p'*Bp);
119
120     % Store data
121     stat.X  = [stat.X  x];
122     %stat.Y  = [stat.Y y];
123     stat.F  = [stat.F f];
124     stat.dF = [stat.dF df];
125     stat.dL = [stat.dL dL];
126     stat.iter = stat.iter+1;
127
128
129     stat.converged = (norm(df,'inf') <= eta);
130     k=k+1;
131
132 end
133 end
```

Listing A.3: An SQP solver using damped BFGS approximation for problem 4.

```
1  function [x,lambda,stat] = SQP_LineSearch(fun,x,lb,ub,gl,gu,nonlcon)
2  % SQP_LineSearch Solves a sequential quadratic programming problem
3  % with a damped BFGS approximation to the Hessian matrix and
4  % using backtracking line search.
5  % Solves for nonlinear programs of the form:
6  %
7  %               min f(x)
8  %        s.t. gl <= g(x) <= gu
9  %              l <= x <= u
10 %
11 % Syntax: [x,lambda,stat] = SQP_LineSearch(fun,x,lb,ub,gl,gu,nonlcon)
12
13 % Function variables
14 tol = 1e-6;
15 rho=0.5;
16 nu = 0.1;
17 ta = 0.5;
18 max_iters=100;
19
20
21
22 % Silence output from quadprog
```

```
options = optimset('Display','off');


[f,df] = feval(fun,x);
[g,dg] = feval(nonlcon,x);

% Obtain dimensions
n=length(x);
m = length(g);

% Initialize variables
Bk = eye(n);
lambda = zeros(n*2+2*m,1);
bounds = [lb;-ub;gl;-gu];

% Storage variables
stat.converged=0;
stat.function_calls = 2;
stat.iter = 0;
stat.X = x;
stat.alpha = 1;
stat.lambda = lambda;


%% Main program
xprev = 100;

while( stat.converged &&  norm(x-xprev,'inf') > tol)%stat.iter <= max_iters)

    xprev = x;
    % Bounds for x
    lbk = lb-x;
    ubk = ub-x;
    % Bounds for g(x)
    glk = gl-g;
    guk = gu-g;

    %Compute pk with quadprog
    A=-[dg';-dg'];
    b=-[glk;-guk];

    [pk, , , ,dlambda] = quadprog(Bk,df,A,b,[],[],lbk,ubk,[],options);


    % Specific for the lb and ub and inequality constraints
    lambda_est = [dlambda.lower; dlambda.upper; dlambda.ineqlin];
    plambda = lambda_est-lambda;

   % Back tracking line search:
    stop = false;
    g_LS = [x;-x;g;-g]-bounds;


    mu=lambda;

```

```
78      c = f + mu'*abs(min(0,g_LS));
79      b = df'*pk-mu'*abs(min(0,g_LS));
80
81      alpha = 1;
82      while  stop
83
84          x_LS=x+alpha*pk;
85          f_LS = feval(fun,x_LS);
86          [g_LS dg_LS] =  feval(nonlcon,x_LS);
87          g_LS = [x_LS;-x_LS;g_LS;-g_LS]-bounds;
88           stat.function_calls = stat.function_calls+ 2;
89
90
91          phi = f_LS+lambda'*abs(min(0,g_LS));
92
93          if phi <= (c + 0.1*c'*alpha)
94              stop = true;
95          else
96              a = phi-(c+b*alpha)/(alpha^2);
97              alpha_min = -b/(2*a);
98              alpha = min(0.9*alpha, max(alpha_min,0.1*alpha));
99          end
100
101          c = f + mu'*abs(min(0,g_LS));
102          b = df'*pk-mu'*abs(min(0,g_LS));
103
104          mu = max(abs(lambda),0.5*(mu+abs(lambda)));
105
106     end
107
108
109     %Update the current points
110     x = x+alpha*pk;
111     lambda = lambda + alpha*plambda;
112
113
114     % Implementing damped approximation of the Hessian matrix
115     dL = (df' - lambda_est'*[eye(n); -eye(n); dg ; -dg])';
116
117     %Calculate new function values
118     [f,df] = feval(fun,x);
119     [g,dg] = feval(nonlcon,x);
120     stat.function_calls = stat.function_calls + 2;
121
122     dL_next = (df' - lambda_est'*[eye(n); -eye(n); dg ; -dg])';
123
124     q = dL_next-dL;
125     p = pk;
126
127     a = (p'*q);
128     b = p'*(Bk*p);
129    % Update Hessian matrix by the modified BFGS procedure
130     Bp = Bk*p;
131     criteria = (p'*q) >= (0.2*(p'*Bp)); % Here we are ensuring positive
            definiteness
```

```
    if criteria
        theta = 1;
    else
        theta = (0.8*p'*Bp) / (p'*Bp - p'*q);
    end
    r = theta*q + (1-theta)*Bp;

    % Update B
    Bk = Bk + (r*r')/(p'*r) - (Bp*Bp')/(p'*Bp);

    stat.converged = (norm(df,'inf') <= tol);
    stat.X = [stat.X, x];
    stat.lambda = [stat.lambda, lambda];
    stat.alpha = [stat.alpha, alpha];
    stat.iter = stat.iter+1;

end
end
```

Listing A.4: An SQP solver using damped BFGS approximation and line search for problem 4.

```
function [x,lambda,stat] = SQP_TrustRegion(fun,x,lb,ub,gl,gu,nonlcon)
% SQP_TrustRegion Solves a sequential quadratic programming problem
% with a damped BFGS approximation to the Hessian matrix and
% using trust region strategy.
% Solves for nonlinear programs of the form:
%
%               min f(x)
%        s.t. gl <= g(x) <= gu
%              l <= x <= u
%
% [x,lambda,stat] = SQP_TrustRegion(fun,x,lb,ub,gl,gu,nonlcon)

% Function parameters
tol = 1e-6;
rho=0.5;
nu = 0.1;
ta = 0.5;
max_iters=100;


% Silence quadprog output
options = optimset('Display','off');

[f,df] = feval(fun,x);
[g,dg] = feval(nonlcon,x);


% Obtain dimensions
n=length(x);
m = length(g);

```

```
Bk = eye(n);
bounds = [lb;-ub;gl;-gu];
lambda = zeros(n*2+2*m,1);
tr = 0.5; %initial trust region -> (0,1)


% Storage variables
stat.TR = tr;
stat.lambda = lambda;
stat.converged = 0;
stat.iter = 0;
stat.X = x;
stat.function_calls = 2;



%% Main program
i=0;
xprev = 100;
while( stat.converged &&  norm(x-xprev,'inf') > tol)%stat.iter <= max_iters)


    % Bounds for x
    lbk = lb-x;
    ubk = ub-x;
    % Bounds for g(x)
    glk = gl-g;
    guk = gu-g;


    % Implement trust region contstraint: ||p||_inf < Delta_k
    ubk([ubk>tr])=tr;
    lbk([lbk<(-tr)])=-tr;

    %Compute pk with quadprog
    A=-[dg';-dg'];
    b=-[glk;-guk];

    [pk, , , ,dlambda] = quadprog(Bk,df,A,b,[],[],lbk,ubk,[],options);


    %Update trust region

    f_next= feval(fun,x+pk);
    stat.function_calls = stat.function_calls + 1;
    if pk==0
        rho=1;
    else
        rho = (f_next-f)/((0.5*pk'*Bk*pk)+df'*pk);
    end

    gamma = min(max((2*rho-1)^3+1,0.25),2);

    if rho>0
        %accept step
```

```
87          xprev=x;
88          % Specific for the lb and ub and inequality constraints
89          lambda_est = [dlambda.lower; dlambda.upper; dlambda.ineqlin];
90          plambda = lambda_est-lambda;
91
92          %Update the current points
93          x = x+pk;
94          lambda = lambda + plambda;
95
96            % Implementing damped approximation of the Hessian matrix
97           dL = (df' - lambda_est'*[eye(n); -eye(n); dg ; -dg])';
98
99           %Calculate new function values
100         [f,df] = feval(fun,x);
101         [g,dg] = feval(nonlcon,x);
102         stat.function_calls = stat.function_calls + 2;
103
104
105         dL_next = (df' - lambda_est'*[eye(n); -eye(n); dg ; -dg])';
106
107         q = dL_next-dL;
108         p = pk;
109
110         % Update Hessian matrix by the modified BFGS procedure
111         Bp = Bk*p;
112         criteria = (p'*q) >= (0.2*(p'*Bp)); % Here we are ensuring positive
                definiteness
113         if criteria
114             theta = 1;
115         else
116             theta = (0.8*p'*Bp) / (p'*Bp - p'*q);
117         end
118         r = theta*q + (1-theta)*Bp;
119
120         %Update B
121         Bk = Bk + (r*r')/(p'*r) - (Bp*Bp')/(p'*Bp);
122         tr = gamma*tr;
123
124     else
125         %reject step
126         tr  = gamma*norm(pk,'inf');
127     end
128
129
130       stat.converged = (norm(df,'inf') <= tol);
131       stat.X = [stat.X, x];
132       stat.lambda = [stat.lambda, lambda];
133       stat.iter = stat.iter+1;
134       stat.TR = [stat.TR,tr];
135
136 end
137 end
```

Listing A.5: An SQP solver using damped BFGS approximation and trust region strategy for problem 4.

A.2 Drivers and Solver Interfaces

A.2.1 Problem 1 Driver

```
%% Problem 1: Equality Constrained Convex QP
close all; clear all; clc;

% -- In this problem the following methods are implemented: --
% LU factorization (dense and sparse version)
% LDL factorization (dense and sparse version)
% Range-space method
% Null-space method

%% Test algorithms on test problem

H = [5.00, 1.86, 1.24, 1.48, -0.46;...
     1.86, 3.00, 0.44, 1.12, 0.52;...
     1.24, 0.44, 3.80, 1.56, -0.54;...
     1.48, 1.12, 1.56, 7.20, -1.12;...
     -0.46, 0.52, -0.54, -1.12, 7.80];
g = [-16.1; -8.5; -15.7; -10.02; -18.68];
A = [16.1, 1.0; 8.5, 1.0; 15.7, 1.0; 10.02, 1.0; 18.68, 1.0];
b = [15;1];


% Compare algorithms for different values of b(1) for problem 1.1

n=20;
b1vals = linspace(8.5,18.56,n);

% Initialize error variables
ERRx = zeros(n,6);
OPTIONS = optimset('Display','off'); % for quadprog
solvers = {'LUdense','LUsparse','LDLdense','LDLsparse','NullSpace','
    RangeSpace'};
for i = 1:n

    b = [b1vals(i), 1]';

    % Compare using quadprog
    [xqp, lambdaqp] = quadprog(H,g,[],[],A',b,[],[],[],OPTIONS);

    for j = 1:6
        [x, lambda] = EqualityQPSolver(H,g,A,b,solvers{j});

        % Compute error between solver and quadprog
        ERRx(i, j) = norm(x-xqp);
    end
end


% Plot comparison to quadprog
```

```
48
49 figure; h=gcf;
50 for k = 1:6
51     plot(b1vals, ERRx(:,k),'linewidth',1.1)
52     hold on
53 end
54 legend(solvers)
55 title('EQP Solvers vs. Quadprog','FontSize',25)
56 ylabel('||x_{solver}-x_{quadprog}||_2','FontSize',20)
57 xlabel('Values of b(1)','FontSize',20)
58 set(gca, 'FontSize',16)
59
60 % Save figure
61 %set(h,'PaperOrientation','landscape');
62 %print(h,'EQPSolverVSQuadprog', '-dpdf','-fillpage')
63
64
65 %% Test your implementation on a size dependent structure and report the
       results
66 close all; clear all; clc
67
68 % Testing on recycling problem from week 5
69
70 n=10:20:2000;
71 u_bar=0.2;
72 d0=1;
73 solvers = {'LUdense','LUsparse','LDLdense','LDLsparse','RangeSpace',...
74     'NullSpace'};
75
76 figure;h=gcf;
77 for s=1:6
78     solver = solvers{s};
79     time = [];
80
81     for i=1:length(n)
82         [H,g,A,b] = objfun_RecycleSystem(n(i),u_bar,d0);
83         tStart = cputime;
84         [x,lambda]=EqualityQPSolver(H,g,A,b,solver);
85         time(i) = cputime - tStart;
86     end
87
88     plot(n,time,'LineWidth',1)
89     hold on
90 end
91
92 legend(solvers, 'location','best')
93 %title('Computational Time on a Size Dependent Problem')
94 title('EQP Solvers on a Size Dependent Problem', 'FontSize',25)
95 ylabel('Time [s]','FontSize',20), xlabel('Problem Size (n)','FontSize',20)
96 %xlim([0 3800])
97 set(gca, 'FontSize',16)
98
99 % Save figure
100 set(h,'PaperOrientation','landscape');
101 print(h,'CPU_ProblemSize_EQP', '-dpdf','-fillpage')
```

```
102
103
104 %% Test on a random convex EQP with fewer constraints than variables.
105 clear all; clc;
106
107 solvers = {'LUdense','LUsparse','LDLdense','LDLsparse','RangeSpace',...
108     'NullSpace'};
109 range=10:20:2000;
110 Ntests = length(range);
111
112 figure;h=gcf;
113 for s=1:6
114
115     solver = solvers{s};
116
117     i = 1;
118     for n=range
119         [H,g,A,b,x,lambda] = GenerateRandomConvexQP(n,floor(n/5));
120         tStart = cputime;
121         [x,lambda]=EqualityQPSolver(H,g,A,b,solver);
122         time(i) = cputime - tStart;
123         i = i+1;
124     end
125
126     plot(range,time,'LineWidth',1)
127     hold on
128
129 end
130
131 legend(solvers, 'location','best')
132 title('EQP Solvers on a Size Dependent Problem: n>>m','FontSize',25)
133 ylabel('Time [s]','FontSize',20), xlabel('Problem Size (n)','FontSize',20)
134 set(gca, 'FontSize',15)
```

Listing A.6: Driver for problem 1, Chapter 1.

A.2.2 Problem 1 Solver Interface

```
1  function [x,lambda] = EqualityQPSolver(H,g,A,b,solver)
2  % EqualityQPSolver Solver for an equality constrained quadratic program
3  %
4  %            min 0.5*x'H*x + g'x
5  %             x
6  %            s.t. A'x = b
7  %
8  %     Solver can be one of the following:
9  %     LDLdense, LUdense, LDLsparse, LUsparse, NullSpace, RangeSpace
10 %
11 % Syntax: [x,lambda] = EqualityQPSolver(H,g,A,b,solver)
12
13 switch upper(solver)
```

```
    case 'LDLDENSE'
        [x,lambda] = EqualityQPSolverLDLdense(H,g,A,b);
    case 'LUDENSE'
        [x,lambda] = EqualityQPSolverLUdense(H,g,A,b);
    case 'LDLSPARSE'
        [x,lambda] = EqualityQPSolverLDLsparse(H,g,A,b);
    case 'LUSPARSE'
        [x,lambda] = EqualityQPSolverLUsparse(H,g,A,b);
    case 'NULLSPACE'
        [x,lambda] = EqualityQPSolverNullSpace(H,g,A,b);
    case 'RANGESPACE'
        [x,lambda] = EqualityQPSolverRangeSpace(H,g,A,b);
    otherwise
        error_message = "Solver " + solver + ...
            " does not exist. Possible solvers are: "+...
            "LDLdense, LUdense, LDLsparse, LUsparse, NullSpace, RangeSpace";
        error(error_message)

end
```

Listing A.7: A solver interface for implementing different solvers for the solution of an equality constrained quadratic program.

A.2.3 Problem 2 Driver

```
%% Problem 2: Quadratic Program (QP)

%% Exercise 2.4: Compare primal-dual interior-point QP solver with quadprog
close all; clear all; clc;

options = optimset('Display','off','Algorithm','interior-point-convex');
ASoptions = optimset('Display','off','Algorithm','active-set');

% Using parameters from problem 1 to test the algorithm
H = [5.00, 1.86, 1.24, 1.48, -0.46;...
     1.86, 3.00, 0.44, 1.12, 0.52;...
     1.24, 0.44, 3.80, 1.56, -0.54;...
     1.48, 1.12, 1.56, 7.20, -1.12;...
    -0.46, 0.52, -0.54, -1.12, 7.80];
g = [-16.1; -8.5; -15.7; -10.02; -18.68];
A = [16.1, 1.0; 8.5, 1.0; 15.7, 1.0; 10.02, 1.0; 18.68, 1.0];
b = [15;1];

N = 20;
b1vals = linspace(8.5, 18.68, N); % 20 diff values of b(1)

n = 5;

% Inequality constraints
l = zeros(n,1);
u = ones(n,1);
```

```
C = [eye(n);-eye(n)];
d = [l;-u];

% Set initial parameters
x0 = zeros(n,1);
s0 = ones(2*n,1) ;
y0 = ones(length(b),1);
z0 = ones(2*n,1);

i = 1;
for b1 = b1vals

    b(1) = b1;

    [x_own,stat] = PrimalDualInteriorPoint_QP(H,g,A,b,C,d,x0,y0,z0,s0);
    [xqp, , ,output] = quadprog(H,g,[],[],A',b,l,u,x0,options);
    [x_AS,fval,exitflag,output] = quadprog(H,g,[],[],A',b,...
        l,u,x0,ASoptions);


    Errqp(i) = immse(x_own,xqp);
    ErrAS(i) = immse(x_own,x_AS);

    i=i+1;
end


figure,
h=gcf;
plot(b1vals, Errqp,'b.-','linewidth',1),hold on,
plot(b1vals, Errqp,'m','linewidth',1),
ylabel('MSE','FontSize',24),
xlabel('b(1)','FontSize',24),
title('Comparing with quadprog','FontSize', 32),
legend('quadprog Interior-Point','quadprog Active-Set','location','northwest
    ')
set(gca, 'FontSize',20) % Increase font size for axis ticks


%% Exercise 2.5: Comparing with quadprog on a size dependent problem

% Using the recycling system from week 5 in course

u_bar=0.2;
d0=1;
range=10:20:500;

Ntests = length(range);
options = optimset('algorithm','interior-point-convex','display','off');

% Initialize storage variables
ErrorIPvsQP = zeros(Ntests,1);
ItersIPvsQP =  zeros(Ntests,2);
TimeIPvsQP = zeros(Ntests,2);

```

```
i = 1;

% Compare for size dependent problem
for n = range

    % Generate QP program
    [H,g,A,b] = objfun_RecycleSystem(n,u_bar,d0);

    % Set lower and upper bounds
    l  = zeros(n,1);
    u = ones(n,1);

    % Inequality for primal dual algorithm
    C = [eye(n);-eye(n)];
    d = [l;-u];

    % Set initial parameters
    x0 = zeros(n,1);
    s0 = ones(2*n,1) ;
    y0 = ones(length(b),1);
    z0 = ones(2*n,1);

    tStart = cputime;
    [x,stat] = PrimalDualInteriorPoint_QP(H,g,A,b,C,d,x0,y0,z0,s0);
    TimeIPvsQP(i,1) = cputime - tStart;

    tStart = cputime;
    [xqp,fval,exitflag,output] = quadprog(H,g,[],[],A',b,l,u,x0,options);
    TimeIPvsQP(i,2) = cputime - tStart;


    ErrorIPvsQP(i,1) = immse(x,xqp);%norm(x-xqp);

    ItersIPvsQP(i,1) = stat.iter;
    ItersIPvsQP(i,2) = output.iterations;

    i = i+1;
end

% Plot the performance statistics

figure,
h=gcf;
plot(range, ErrorIPvsQP(:,1),'b','Linewidth',1.2),
ylabel('MSE','FontSize',24),
xlabel('Problem Size (n)','FontSize',24),
title('Comparing with quadprog','FontSize', 32)
xlim([0 500])
set(gca, 'FontSize',20) % Increase font size for axis ticks


figure,
h2 = gcf;
plot(range, TimeIPvsQP(:,1),'Linewidth',1.2),
hold on
```

```
plot(range, TimeIPvsQP(:,2),'m','Linewidth',1.2),
ylabel('CPU Time [s]','FontSize',24),
xlabel('Problem Size (n)','FontSize',24),
legend('PrimalDualInteriorPoint\_QP', 'quadprog Interior-Point',...
    'location','northwest')
title('CPU Time vs. Problem Size','FontSize', 32)
xlim([0 500])
set(gca, 'FontSize',20)

figure,
h3 = gcf;
plot(range, ItersIPvsQP(:,1),'Linewidth',1.2),
hold on
plot(range, ItersIPvsQP(:,2),'m','Linewidth',1.2),
ylim([0 7])
ylabel('No. Iterations','FontSize',24),xlabel('Problem Size (n)','FontSize'
    ,24),
legend('PrimalDualInteriorPoint\_QP', 'quadprog Interior-Point')
title('No. Iterations vs. Problem Size','FontSize', 26)
xlim([0 500])
set(gca, 'FontSize',20)



%% Exercise 2.6: Using test problem from problem 1 to compare with
% QP libraries

% Test problem
H = [5.00, 1.86, 1.24, 1.48, -0.46;...
     1.86, 3.00, 0.44, 1.12, 0.52;...
     1.24, 0.44, 3.80, 1.56, -0.54;...
     1.48, 1.12, 1.56, 7.20, -1.12;...
     -0.46, 0.52, -0.54, -1.12, 7.80];
g = [-16.1; -8.5; -15.7; -10.02; -18.68];
A = [16.1, 1.0; 8.5, 1.0; 15.7, 1.0; 10.02, 1.0; 18.68, 1.0];
b = [15;1];


n=5;

% Inequality constraints
l = zeros(n,1);
u = ones(n,1);
C = [eye(n);-eye(n)];
d = [l;-u];

% Set initial parameters
x0 = zeros(n,1);
s0 = ones(2*n,1) ;
y0 = ones(length(b),1);
z0 = ones(2*n,1);

options = optimset('display','off');

% Compute solution for different vlues of b(1)
```

```
N = 20;
Time = zeros(N,4);
Iters = zeros(N,4);
b1vals = linspace(8.5, 18.68, N); % 20 diff values of b(1)

i = 1;
for b1 = b1vals

    b(1) = b1;

    tStart = cputime;
    [x_own,stat] = PrimalDualInteriorPoint_QP(H,g,A,b,C,d,x0,y0,z0,s0);
    Time(i,1) = cputime - tStart;

    tStart = cputime;
    [xqp,fval,exitflag,output] = quadprog(H,g,[],[],A',b,l,u,x0,options);
    Time(i,2) = cputime - tStart;

    % Gurobi
    bmodel.modelsense = 'min';
    bmodel.sense = '=';          % Only equality constraints
    bmodel.Q = 0.5*sparse(H);
    bmodel.obj = g;
    bmodel.A = sparse(A');
    bmodel.rhs = b;             % vector b or d in the linear constraints
    bmodel.lb = l;
    bmodel.ub = u;
    bparams.OutputFlag = 0;
    tStart = cputime;
    bresult = gurobi(bmodel, bparams);
    Time(i,3) = cputime - tStart;
    xGur = bresult.x;

    % Test cvx's default solver SDPT3
    tStart = cputime;
    cvx_begin quiet % Surpress output
        variable x(n)
        minimize(1/2*x'*H*x + g'*x)
        subject to
            A'*x == b;
            l <= x;
            x <= u;
    cvx_end
    Time(i,4) = cputime - tStart;

    Iters(i,1) = stat.iter;
    Iters(i,2) = output.iterations;
    Iters(i,3) = bresult.itercount;
    Iters(i,4) = cvx_slvitr;

    ErrCvx(i) = immse(x_own,x); %norm(x_own-x);
    Errqp(i) =  immse(x_own,xqp);
    ErrGurobi(i) = immse(x_own,xGur);

    i=i+1;
```

```
end

% Plot output
figure,
h = gcf;
plot(b1vals, Time(:,1),'r','LineWidth',1.3), hold on
plot(b1vals, Time(:,2),'m','LineWidth',1.3),
plot(b1vals, Time(:,3),'b','LineWidth',1.3),
plot(b1vals, Time(:,4),'g','LineWidth',1.3),
ylabel('CPU Time [s]','FontSize',24),xlabel('b(1)','FontSize',24),
xlim([b1vals(1) b1vals(end)])
legend('PrimalDualInteriorPoint\_QP', 'Matlab''s quadprog','Gurobi','CVX'
    ,...
    'location','best')
title('CPU Time vs. values of b(1)','FontSize', 32)
set(gca, 'FontSize',20)

figure,
h2 = gcf;
plot(b1vals, Iters(:,1),'r','LineWidth',1.3),hold on
plot(b1vals, Iters(:,2),'m','LineWidth',1.3),
plot(b1vals, Iters(:,3),'b','LineWidth',1.3),
plot(b1vals, Time(:,4),'g','LineWidth',1.3),
ylabel('No. Iterations','FontSize',24),xlabel('b(1)','FontSize',24),
xlim([b1vals(1) b1vals(end)])
legend('PrimalDualInteriorPoint\_QP', 'Matlab''s quadprog','Gurobi','CVX'
    ,...
    'location','best')
title('No. Iterations vs. Values of b(1)','FontSize', 32)
set(gca, 'FontSize',20)


% Plot error between own solver and other solvers
figure; h3=gcf;
semilogy(1:N, ErrCvx, 'g','LineWidth',1.4), hold on
semilogy(1:N, Errqp,'m','LineWidth',1.4),
semilogy(1:N, ErrGurobi,'b','LineWidth',1.4),
title(...
    'Comparing with Other QP Libraries',...
    'FontSize', 32),
ylabel('log10(MSE)','FontSize',24), xlabel('b(1)','FontSize',24)
xlim([b1vals(1) b1vals(end)])
legend('CVX vs. PrimalDualInteriorPoint\_QP',...
    'quadprog vs. PrimalDualInteriorPoint\_QP', ...
    'Gurobi vs. PrimalDualInteriorPoint\_QP',...
    'Location','best','fontsize',17)
set(gca, 'FontSize',20)
```

Listing A.8: Driver for problem 2, Chapter 2.

A.2.4 Problem 3 Driver

```
%% Problem 3: Linear Program (LP)
close all; clear all;clc

%% Exercise 3.4-3.5: Implement the primal-dual interior point algorithm
% and test it

% Test LP algorthm on test data

g=[-16.1;
    -8.5;
    -15.7;
    -10.02;
    -18.68];

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

b=1;
l=zeros(5,1);
u=ones(5,1);

C = [eye(n);-eye(n)];
d = [l;-u];

% Set initial parameters
x0 = zeros(n,1) ;
s0 = ones(2*n,1);
y0 = ones(length(b),1);
z0 = ones(2*n,1);


N=10;
b = 1;

options = optimset('display','off');

% Compare with matlab's linprog

[x,stat] = PrimalDualInteriorPoint_LP(g,A,b,C,d,x0,y0,z0,s0);

[xlin1,fval,exitflag,output] = linprog(g,[],[],A',b,l,u,x0,options);

fprintf('Implemented Solver vs. Linprog:\n')
disp([x, xlin1])

fprintf('MSE error: %.e\n', immse(x,xlin1))

%% Exercise 3.6: Comparing method with linprog on a size dependent problem
close all; clear all; clc


IPoptions = optimoptions('linprog','Algorithm','interior-point',...
    'display','off');
DSoptions = optimoptions('linprog','Algorithm','dual-simplex',...
```

```
    'display','off');

u_bar=0.2;
d0=1;
n=10:20:500;

% Initialize storage variable
time=zeros(length(n),3);
error=zeros(length(n),2);
iters = zeros(length(n),3);

for i=1:length(n)

    % Generate QP problem
    [H,g,A,b] = objfun_RecycleSystem(n(i),u_bar,d0);

    % Set lower and uppper bounds
    l=zeros(n(i),1);
    u=ones(n(i),1);

    % Initial parameters
    x0 = zeros(n(i),1) ;
    s0 = ones(2*n(i),1);
    y0 = ones(length(b),1);
    z0 = ones(2*n(i),1);


    C = [eye(n(i));-eye(n(i))];
    d = [l;-u];

    tStart = cputime;
    [x,stat] = PrimalDualInteriorPoint_LP(g,A,b,C,d,x0,y0,z0,s0);
    time(i,1)=cputime - tStart;
    iters(i,1) = stat.iter;

    tStart = cputime;
    [xlin1, , ,output]= linprog(g,[],[],A',b,l,u,[],IPoptions);
    time(i,2)=cputime - tStart;
    iters(i,2) = output.iterations;

    tStart = cputime;
    [xlin2,fval,exitflag,output]= linprog(g,[],[],A',b,l,u,[],DSoptions);
    time(i,3)=cputime - tStart;
    iters(i,3) = output.iterations;

    error(i,1) = immse(x,xlin1);
    error(i,2) = immse(x,xlin2);

end

% Plot results
figure; h=gcf;
plot(n,time(:,1),'m','LineWidth',1), hold on,
plot(n,time(:,2),'g','LineWidth',1)
plot(n,time(:,3),'r','LineWidth',1)
```

```
title('CPU Time vs. Problem Size','FontSize',30)
legend('PrimalDualInteriorPoint\_LP','linprog Dual-Simplex',...
    'linprog Interior-Point','location','northwest')
xlabel('Problem size (n)','FontSize',24)
ylabel('Time [s]','FontSize',24)
set(gca,'fontsize',18)


figure; h2=gcf;
plot(n,iters(:,1),'b','LineWidth',1),hold on
plot(n,iters(:,2),'m','LineWidth',1),
plot(n,iters(:,3),'g','LineWidth',1),
ylim([0 6])
title('No. Iterations vs. Problem Size','FontSize',32)
legend('PrimalDualInteriorPoint\_LP','linprog Dual-Simplex',...
    'linprog Interior-Point')
xlabel('Problem size (n)','FontSize',24),
ylabel('No. Iterations','fontsize',24)
set(gca,'fontsize',18)

figure; h3=gcf;
semilogy(n,error(:,1),'b.-','LineWidth',1), hold on,
semilogy(n,error(:,2),'m','LineWidth',1)
title('Comparing with Linprog','FontSize',32)
legend('linprog Dual-Simplex','linprog Interior-Point')
xlabel('Problem Size (n)','FontSize',24),
ylabel('log10(MSE)','FontSize',24)
set(gca,'fontsize',18)
```

Listing A.9: Driver for problem 3, Chapter 3.

A.2.5 Problem 4 Driver

```
%% Problem 4
close all; clear all; clc

% Consider Himmelblau's function.

x = -7:0.005:7;
y = -7:0.005:7;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];

fig = figure;
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar

hold on

yc1 = (x+2).^2;
```

```
yc2 = (4.*x)/10;

%With gl and gu:
yc1_b = yc1-10^8;
yc2_b = yc2+10^7;


lb =-5;
ub = 5;

hold on
    fill(x,yc1,[0.7 0.7 0.7],'facealpha',0.2)
    fill([x x(end) x(1)],[yc2 y(1) y(1)],[0.7 0.7 0.7],'facealpha',0.3)
    fill([x x(end) x(1)],[yc1_b y(1) y(1)],[0.7 0.7 0.7],'facealpha',0.3)
    fill([x x(end) x(1)],[yc2_b y(end) y(end)],[0.7 0.7 0.7],'facealpha'
        ,0.3)
    % plot the x1 and x2 bounds
    fill([x(1) lb lb x(1)],[y(end) y(end) y(1) y(1)],[0.7 0.7 0.7],'
        facealpha',0.3)
    fill([ub x(end) x(end) ub],[y(end) y(end) y(1) y(1)],[0.7 0.7 0.7],'
        facealpha',0.3)
    fill([x(end) x(end) x(1) x(1)],[y(1) lb lb y(1)],[0.7 0.7 0.7],'
        facealpha',0.3)
    fill([x(end) x(end) x(1) x(1)],[ub y(end) y(end) ub],[0.7 0.7 0.7],'
        facealpha',0.3)

hold off
ylim([-7, 7])
ylabel('x_2')
xlabel('x_1')

hold on


% Locate stationary points - From lecture 2C

st_pts = zeros(7,2);

st_pts(1,:) = [3,2];
st_pts(2,:) = [-3.5485, -1.4191];
st_pts(3,:) = [-3.6546, 2.7377];
st_pts(4,:) = [-0.2983, 2.8956];
st_pts(5,:) = [-1.4242, 0.3315];
st_pts(6,:) = [-0.4869,  -0.1948];
st_pts(7,:) = [3.2164, 1.2866];
st_pts(8,:) =[-3.5485, -1.4191];
st_pts(9,:) =[-3.5485, -1.4191];

% Plot stationary points
for i=1:9
    h = plot(st_pts(i,1),st_pts(i,2),'kp','MarkerSize',7,'MarkerFaceColor','
        r');
    hold on
end

```

```
68 legend ([h],{'Stationary points'}),
69 title('Himmelblau''s Problem','fontsize',28)
70 xlabel('x_1','fontsize',18)
71 ylabel('x_2','fontsize',18),
72 set(gca, 'fontsize',15)
73
74 set(fig,'PaperOrientation','landscape');
75 print(fig,'HimmelblauContourPlot', '-dpdf','-fillpage')
76
77 %% Solve Himmelblaus test problem on library functions
78
79
80 % Add Casadi to path
81 %addpath("C:\Users\pengu\Documents\DTU\2_ONN\ScientificComputingDE\casadi-
       matlabR2016a")
82 addpath("G:\Other computers\Hp_pavilion\DTU\2_ONN\ScientificComputingDE\
       casadi-matlabR2016a")
83
84 import casadi.*
85
86 % Symbols/expressions for CasADi
87 x1 = MX.sym('x1');
88 x2 = MX.sym('x2');
89
90 t1 = x1*x1 + x2 - 11;
91 t2 = x1 + x2*x2 - 7;
92 f = t1*t1 + t2*t2;
93
94 c1 = (x1+2)^2 - x2;
95 c2 = -4*x1 + 10*x2;
96 g = [c1; c2];
97
98 nlp = struct; % NLP declaration
99 nlp.x = [x1;x2]; % decision vars
100 nlp.f = f; % objective
101 nlp.g = g; % constraints
102
103 % To silence output
104 options = struct;
105 options.ipopt.print_level = 0;
106 options.print_time = 0;
107
108 % Create solver instance
109 F = nlpsol('F','ipopt',nlp,options);
110
111 x0 = {[0;0],[-4;2],[0;3.5],[-4;-0.5]}; % initial point
112
113 lb=[-5;-5];
114 ub=[5;5];
115
116 options = optimoptions( 'fmincon',...
117     'Display','none',...
118     'Algorithm','interior-point');
119
120
```

```
% Solve using fmincon and casadi
for i=1:4
    % Solve the problem using a guess
    res = F('x0',x0{i},'ubg',1e8,'lbg',0,'lbx',[-5;-5],'ubx',[5;5]);
    x_casadi = res.x;
    [x_fmincon] = fmincon(@objfungradHimmelblau,x0{i},[],[],[],[],lb,ub,...
        @confunHimmelblau_fmincon,options);
    fprintf('Starting point: [%.4f,%.4f]\n',x0{i})
    fprintf('CasADi: x* = ')
    disp(x_casadi)
    fprintf('fmincon: x* = [%.4f %.4f] \n\n',x_fmincon)
end



%% Implementing the SQP procedure for himmelblau's test problem
close all; clear all;clc;


n=2;
lambda0 = zeros(8,1);
x0 = {[0;0],[-4;2],[0;3.5],[-4;-0.5]}; % initial point

lb = [-5;-5];
ub = [5;5];
gl = [0,0]';
gu = [1e8,1e8]';

for i = 1:4
    [xbfgs(:,i),stat{i,1}] = SQP_BFGS(x0{i},lb,ub,gl,gu,
        @objfungradHimmelblau,@confunHimmelblau);

    [xline(:,i),lambda,stat{i,2}] = SQP_LineSearch(@objfungradHimmelblau,x0{
        i},lb,ub,gl,gu,@confunHimmelblau);

    [xtr(:,i),lambda2,stat{i,3}] = SQP_TrustRegion(@objfungradHimmelblau,x0{
        i},lb,ub,gl,gu,@confunHimmelblau);

end


titles = [{'Himmelblau_BFGS'},{'Himmelblau_LineSearch'},{'
    Himmelblau_TrustRegion'}];

texts = [{'SQP_BFGS with x0 = [%.2f,%.2f]^T\n'},{'SQP_LineSearch with x0 =
    [%.2f,%.2f]^T\n'},...
    {'SQP_TrustRegion with x0 = [%.2f,%.2f]^T\n'}];


outputSol = true; % Set to true to output solution for each algorithm

% Generate contour plot of himmelblau's problem with each path
for j=1:3
    [seq(1),fig] = plot_iteration_sequence(stat{1,j}.X,[{'r-'},{'Path 1'}],
        false);
```

```
    seq(2) = plot_iteration_sequence(stat{2,j}.X,[{'k-'},{'Path 2'}],true);
    seq(3) = plot_iteration_sequence(stat{3,j}.X,[{'b-'},{'Path 2'}],true);
    seq(4) = plot_iteration_sequence(stat{4,j}.X,[{'m-'},{'Path 2'}],true);
    legend([seq(1),seq(2),seq(3),seq(4)],'Path 1','Path 2','Path 3','Path 4'
        )

    if outputSol
        for k = 1:4
            fprintf(texts{j},x0{k})
            T=table([1:stat{k,j}.iter+1]', stat{k,j}.X(1,:)',stat{k,j}.X
                (2,:)',...
                'VariableNames', {'Iteration','x_1','x_2'});
            disp(T)
            fprintf('Function calls: %.f\n',stat{k,j}.function_calls)
        end
    end
end

%% Implementing the SQP procedure on a nonlinear test problem
close all; clear all; clc;

% Plot contour plot of function
x = -4:0.05:4;
y = -4:0.05:4;
[X,Y] = meshgrid(x,y);
F = 3.*sin(X) + X.*Y + Y.^2;
v = [0:2:100 100:10:400 400:20:600];
fig = figure;
[c,h]=contour(X,Y,F,60,'linewidth',2); hold on; axis equal
colorbar

% Put constraints
% subdue with an overlay
hf=fill([1 1 -1 -1]*4,[-1 1 1 -1]*4,'w','facealpha',0.8);
% plot just the ROI
feasible = (X+Y-1>=0) & (X+Y-1<=4) & (X+4*Y-2>=0) & (X+4*Y-2<=4) & (X>=-2) &
    (X<=3) &(Y>=-2) & (Y<=3); %(Y>=0) & (X+2*Y<=2);
F( feasible) = NaN;
contour(X,Y,F,12,'linewidth',2);


x0 = [0, 0]';
lb = [-1;-1];
ub = [2;2];
gl = [0,0]';
gu = [3,3]';


[x,statbfgs] = SQP_BFGS(x0,lb,ub,gl,gu,@objfunRandomNonlinear,
    @confunRandomNonlinear);

[xfmincon,fval,exitflag,output] = fmincon(@objfunRandomNonlinear,x0
    ,[],[],[],[],lb,ub,@confunRandomNonlinear_fmincon);

```

```
[xline,lambda,statline] = SQP_LineSearch(@objfunRandomNonlinear,x0,lb,ub,gl,
    gu,@confunRandomNonlinear);

[xtr,lambda,stattr] = SQP_TrustRegion(@objfunRandomNonlinear,x0,lb,ub,gl,gu,
    @confunRandomNonlinear);

fprintf('SQP_BFGS x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f\n',x,
    objfunRandomNonlinear(x),...
    statbfgs.iter,statbfgs.function_calls)
fprintf('SQP_LineSearch x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f \
    n',xline,...
    objfunRandomNonlinear(xline),statline.iter,statline.function_calls)
fprintf('SQP_TrustRegion x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f
    \n',xtr,...
    objfunRandomNonlinear(xtr),stattr.iter,stattr.function_calls)
fprintf('SQP_fmincon x*: %.5f %.5f  f(x*): %.4f, iters: %.f, fevals: %.f \n'
    ,xfmincon,...
    objfunRandomNonlinear(xfmincon),output.iterations,output.funcCount)

hold on,
ax=plot(xfmincon(1), xfmincon(2),'kp','LineWidth',1.5);
title('Nonlinear Test Problem','fontsize',28)
xlabel('x_1','fontsize',20)
ylabel('x_2','fontsize',20),
set(gca, 'fontsize',16)
aa.MarkerEdgeColor = 'k';
aa.MarkerFaceColor = 'k';
legend(ax,'Minimum')

%% Implementing the SQP procedure for Rosenbrock's problem
close all; clear all;clc;

% Plot contour plot
x = -5:0.005:5;
y = -5:0.005:5;

[X,Y]=meshgrid(x,y);
F = (100.*(Y-X.^2).^2 + (1-X).^2);
v = [0:2:10 10:10:100 100:20:200];
fig = figure;
[c,h]=contour(X,Y,F,v,'linewidth',2);
colorbar
axis([-2 2 -1.5 3.5])


% Testing Rosenbrock function and comparing with fmincon

x0 = [0, 0]';
lb = [-1;-1];
ub = [0.8;0.8];
gl = [-1,-1]';
gu = [3,3]';


[x,statbfgs] = SQP_BFGS(x0,lb,ub,gl,gu,@rosenbrock,@confunRosenbrock);
```

```
268
269 [xfmincon,fval,exitflag,output] = fmincon(@rosenbrock,x0,[],[],[],[],lb,ub,
        @confunRosenbrock_fmincon);
270
271 [xline,lambda,statline] = SQP_LineSearch(@rosenbrock,x0,lb,ub,gl,gu,
        @confunRosenbrock);
272
273 [xtr,lambda,stattr] = SQP_TrustRegion(@rosenbrock,x0,lb,ub,gl,gu,
        @confunRosenbrock);
274
275
276 fprintf('SQP_BFGS x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f\n',x,
        rosenbrock(x),...
277     statbfgs.iter,statbfgs.function_calls)
278 fprintf('SQP_LineSearch x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f \
        n',xline,...
279     rosenbrock(xline),statline.iter,statline.function_calls)
280 fprintf('SQP_TrustRegion x*: %.5f %.5f f(x*): %.4f, iters: %.f, fevals: %.f
        \n',xtr,...
281     rosenbrock(xtr),stattr.iter,stattr.function_calls)
282 fprintf('SQP_fmincon x*: %.5f %.5f  f(x*): %.4f, iters: %.f, fevals: %.f \n'
        ,xfmincon,...
283     rosenbrock(xfmincon),output.iterations,output.funcCount)
284
285 % Plot minima
286 hold on,
287 ax=plot(xfmincon(1), xfmincon(2),'kp','linewidth',1.5);
288 ax2=plot(x(1),x(2),'rp','linewidth',1.5);
289 legend([ax,ax2],'fmincon,SQP-LineSearch,SQP-TrustRegion','SQP-BFGS')
290 title('Rosenbrock Test Problem','fontsize',28)
291 xlabel('x_1','fontsize',20)
292 ylabel('x_2','fontsize',20),
293 aa.MarkerEdgeColor = 'k';
294 aa.MarkerFaceColor = 'k';
295 set(gca, 'fontsize',16)
```

Listing A.10: Driver for problem 4, Chapter 4.

```
1  %% Functions for constraints in Himmelblau's problem
2  function [c,ceq] = confunHimmelblau_fmincon(x)
3
4  lb = 0;
5  ub = 1e8;
6
7  c = zeros(4,1);
8  c(1)=-(x(1)+2)^2-x(2)-ub;
9  c(2)=-(x(1)+2)^2+x(2)+lb;
10 c(3)=-4*x(1)+10*x(2)-ub;
11 c(4)=4*x(1)-10*x(2)+lb;
12 ceq = 0;
13
14 end
15
```

```
function [c,dc] = confunHimmelblau(x)

c = zeros(2,1);
dc = zeros(2,2);

% Inequality constraints c(x) >= 0
tmp = x(1)+2;
c(1,1) = (tmp^2 - x(2));
c(2,1) = (-4*x(1) + 10*x(2));

if nargout >1
    dc(1,1)=2*(x(1)+2);
    dc(1,2)=-1;
    dc(2,1)=-4;
    dc(2,2)=10;
    dc = dc';
end

end

%% Objective function Himmelblau's problem
function [f,dfdx,H]= objfungradHimmelblau(x)

tmp1 = x(1)*x(1)+x(2)-11;
tmp2 = x(1)+x(2)*x(2)-7;
f = tmp1^2 + tmp2^2;

% compute the gradient of f
if nargout > 1
    dfdx = zeros(2,1);
    dfdx(1,1) = 4*tmp1*x(1) + 2*tmp2;
    dfdx(2,1) = 2*tmp1 + 4*tmp2*x(2);
end

H= zeros(2);

if nargout > 2
    H(1,1)=4*(x(1)^2+x(2)-11)+8*x(1)^2;
    H(1,2)=4*x(1)+4*x(2);
    H(2,1)=4*x(1)+4*x(2);
    H(2,2)=4*(x(1)+x(2)^2-7)+8*x(2)^2+2;
end
end

%% Objective function for Rosenbrock's problem
function [f,df,d2f] = rosenbrock(x)

    f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;

    if nargout>1
        df = zeros(2,1);
        df(1,1) = 2*(200*x(1)^3-200*x(1)*x(2)+x(1)-1);
        df(2,1) = 200*(x(2)-x(1)^2);

    end
```

```
71      if nargout>2
72          d2f = [-400*(x(2)-x(1)^2)+800*x(1)^2 + 2, -400*x(1) ; -400*x(1),
                200];
73      end
74  end
75
76
77  %% Constraint and objective functions for random nonlinear test problem
78
79  function [f,df] = objfunRandomNonlinear(x,p)
80      %% Function as is specified in week 1.
81      f = 3*sin(x(1)) + x(1)*x(2) + x(2)^2;
82      % compute the gradient of f
83      if nargout > 1
84          df = zeros(2,1);
85          df(1,1) = 3*cos(x(1)) + x(2);
86          df(2,1) = x(1) + 2*x(2);
87      end
88  end
89
90  function [f,df] = confunRandomNonlinear(x,p)
91      %% Constraints as specified in week 1 (with tweaks).
92      f = zeros(2,1);
93      df = zeros(2,2);
94      % Inequality constraints c(x) <= 0
95      f(1,1) = x(1)+x(2)-1;
96      f(2,1) = x(1)+4*x(2)-2;
97      % Compute constraint gradients
98      if nargout > 1
99          df(1,1) = 1;
100         df(2,1) = 1;
101         df(1,2) = 1;
102         df(2,2) = 4;
103     end
104 end
105
106 function [c,ceq] = confunRandomNonlinear_fmincon(x)
107 lb=0;
108 ub=3;
109 con1 = x(1)+x(2)-1;
110 con2 = x(1)+4*x(2)-2;
111 c = zeros(4,1);
112 c(1) = con1-ub;
113 c(2) = con2-ub;
114 c(3) = -con1+lb;
115 c(4) =  -con2+lb;
116
117 ceq = 0;
118 end
119
120
121 %% Constraint functions for Rosenbrock's problem
122
123 function [c,ceq] = confunRosenbrock_fmincon(x)
124
```

```
lb=-1;
ub=3;

c = zeros(4,1);
c(1) = (x(1)+1)^2+x(2)-ub;
c(2) = (x(1)-1)+x(2)-ub;
c(3) = -(x(1)+1)^2+x(2)+lb;
c(4) = -(x(1)-1)+x(2)+lb;
ceq = 0;

end

function [c,dc] = confunRosenbrock(x)

c = zeros(2,1);

% Inequality constraints c(x) >= 0

c(1,1) = (x(1)+1)^2-x(2);
c(2,1) = (x(1)-1)-x(2);

if nargout >1
    dc = zeros(2,2);
    dc(1,1)=2*(x(1)+1);
    dc(1,2)=-1;
    dc(2,1)=1;
    dc(2,2)=-1;
end

end

%% Function to plot the results for Himmelblau's problem
function [ab,fig] = plot_iteration_sequence(xk,pathinfo,addtocontour)
% plot_iteration_sequence A function to plot the iteration sequences from
    the
% SQP algorithms to a contour plot of Himmelblau's problem
%
% Input variables:
% xk            | A list containing x elements
% pathinfo      | Info for the path: [{pathcolor and line style},{path
    legend}]
% addtocontour  | Add path to contour plot (do not generate new plot)
%
% Output variables:
% ab            | legend information
% fig           | Variable containing the contour plot
%
% Syntax:
% [ab,fig] = plot_iteration_sequence(xk,pathinfo,addtocontour,save_fig,
    varargin)

if  addtocontour
    x = -7:0.005:7;
    y = -7:0.005:7;
    [X,Y] = meshgrid(x,y);
```

```
177     F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
178     v = [0:2:10 10:10:100 100:20:200];
179
180     fig = figure;
181     [c,h]=contour(X,Y,F,v,'linewidth',2);
182     colorbar
183
184     hold on
185
186     yc1 = (x+2).^2;
187     yc2 = (4.*x)/10;
188
189     %With gl and gu:
190     yc1_b = yc1-10^8;
191     yc2_b = yc2+10^7;
192
193     lb =-5;
194     ub = 5;
195
196     hold on
197         fill(x,yc1,[0.7 0.7 0.7],'facealpha',0.2)
198         fill([x x(end) x(1)],[yc2 y(1) y(1)],[0.7 0.7 0.7],'facealpha',0.3)
199         fill([x x(end) x(1)],[yc1_b y(1) y(1)],[0.7 0.7 0.7],'facealpha',0.3
                )
200         fill([x x(end) x(1)],[yc2_b y(end) y(end)],[0.7 0.7 0.7],'facealpha'
                ,0.3)
201         % plot the x1 and x2 bounds
202         fill([x(1) lb lb x(1)],[y(end) y(end) y(1) y(1)],[0.7 0.7 0.7],'
                facealpha',0.3)
203         fill([ub x(end) x(end) ub],[y(end) y(end) y(1) y(1)],[0.7 0.7 0.7],'
                facealpha',0.3)
204         fill([x(end) x(end) x(1) x(1)],[y(1) lb lb y(1)],[0.7 0.7 0.7],'
                facealpha',0.3)
205         fill([x(end) x(end) x(1) x(1)],[ub y(end) y(end) ub],[0.7 0.7 0.7],'
                facealpha',0.3)
206
207     hold off
208     ylim([-7, 7])
209     title('Himmelblau''s Problem','fontsize',28)
210     xlabel('x_1','fontsize',20)
211     ylabel('x_2','fontsize',20),
212     set(gca, 'fontsize',16)
213     hold on
214 end
215
216 a=size(xk);
217
218 if a(2)==1
219    aa =  plot(xk(1),xk(2),'p');
220 else
221     for i=1:(length(xk)-1)
222         ab = plot([xk(1,i),xk(1,i+1)],[xk(2,i),xk(2,i+1)],pathinfo{1},'
                LineWidth',1.2);
223         hold on
224         %pause(0.5)
```

```
    end
    aa = plot(xk(1,end),xk(2,end),'p','LineWidth',2.5);
end

aa.MarkerEdgeColor = 'k';%[0.5 0 0.8];
aa.MarkerFaceColor = 'k';%[0.5 0 0.8];

end
```

Listing A.11: Functions for problem 4 driver.

A.2.6 Problem 5 Driver

```
%% Problem 5: Markovitz Portfolio Optimization
close all; clear all; clc;


%% ----- Optimal Solution as funtion of return -----

%% Exercise 5.3: Compute a portfolio with return 12 and minimal risk
Covariance = [2.50, 0.93, 0.62, 0.74, -0.23;...
       0.93, 1.50, 0.22, 0.56, 0.26;...
       0.62, 0.22, 1.90, 0.78, -0.27;...
       0.74, 0.56, 0.78, 3.60, -0.56;...
       -0.23, 0.26, -0.27, -0.56, 3.90];
Security = [1; 2; 3; 4; 5];
nAssets = 5; % Number of assets
r = 12.0; % Desired return
R = [16.10;8.50;15.70;10.02;18.68]; % Returns

% Formulate problem
H = Covariance;
f = [];                          % Objective has no linear term
Aeq = [ones(1,nAssets); -R'];    % Equality Aeq*x = beq
beq = [1;-r];
Aineq = -eye(nAssets);           % Inequality Aeq*x <= beq
bineq = zeros(nAssets,1);

options = optimoptions('quadprog','Display','off','TolFun',1e-10);


[x] = quadprog(H,f,Aineq,bineq,Aeq,beq,[],[],[],options);
risk = x'*H*x;

fprintf('Optimal Portfolio:\n')
disp(x)
fprintf('Expected return: %.2f\n', r)
fprintf('Minimal risk: %.4f\n', risk)

%% Exercise 5.4: Compute the efficient frontier

```

```
39
40 iTRet = linspace(min(R),max(R),10000); %min(R):0.001:max(R);
41 risks = zeros(length(iTRet),1);
42 port_weights = zeros(length(iTRet),nAssets);
43
44 options = optimset('display','off');
45
46 % Compute the risks given the returns
47 for i = 1:length(iTRet) % from lowest to highest
48
49     beq = [1;-iTRet(i)];
50     [x] = quadprog(H,f,Aineq,bineq,Aeq,beq,[],[],[],options);
51     risks(i) =  x'*H*x; % Var(Return)
52     port_weights(i,:) = x; % Weights of assets for given return
53
54 end
55
56 % Obtain parameters for efficient frontier
57 optimalRet = iTRet(risks == min(risks));
58 effidx = find(iTRet==optimalRet);
59 optimalrisk = min(risks);
60 effRet = iTRet(effidx:end);
61 effrisk = risks(effidx:end);
62
63 fprintf('Optimal Portfolio:\n')
64 disp(port_weights(effidx,:)')
65
66 % Plotting the efficient frontier (risk as funciton of return)
67 figure,
68 h = gcf;
69 plot(optimalRet,optimalrisk,'ko','MarkerFaceColor','k'), hold on
70 plot(iTRet, risks,'b--'),
71 plot(effRet,effrisk,'b')
72 plot(R, diag(H),'ko'),  % Overall risks and returns
73 title('The Efficient Frontier','FontSize', 17),
74 xlabel('Expected Return [%]','FontSize', 14),
75 ylabel('Risk [Var[R]]','FontSize', 14),
76 legend('Minimum Risk','Minimum Variance Frontier',...
77     'Efficient Frontier','Individual Security','Location','northwest')
78 set(gca, 'FontSize',13)
79
80
81 % Plot the optimal portfolio as function of return
82 figure,
83 h2 = gcf;
84 plot(iTRet, port_weights,'LineWidth',0.95)
85 legend('Security 1','Security 2','Security 3','Security 4','Security 5','
       Location','best'),
86 xlabel('Return [%]','FontSize',14),ylabel('Portfolio Weights [%]','FontSize'
       ,14),
87 title('Optimal Portfolio vs. Return','FontSize', 17)
88 set(gca, 'FontSize',13)
89
90
91 %% ----- Bi-criterion optimization -----
```

```
92
93 %% Exercise 5.1-3.: Formulate the opt problem as a bi-criterion of
94 % the variance and the return
95
96
97 Covariance = [2.50, 0.93, 0.62, 0.74, -0.23;...
98         0.93, 1.50, 0.22, 0.56, 0.26;...
99         0.62, 0.22, 1.90, 0.78, -0.27;...
100        0.74, 0.56, 0.78, 3.60, -0.56;...
101        -0.23, 0.26, -0.27, -0.56, 3.90];
102 R = [16.10;8.50;15.70;10.02;18.68]; % Returns
103 nAssets = 5;
104
105 Aeq = ones(1,nAssets);   % Equality Aeq*x = beq
106 beq = 1;
107 Aineq = -eye(nAssets);
108 bineq = zeros(nAssets,1);
109
110 f = -R;
111
112 % Change to false to only compare the implemented solvers
113 CompareOtherSolvers = true;
114
115
116 n = 20; % Number of different values for alpha
117 options = optimset('display','off');
118
119 % Storage variables
120 Time = zeros(n,5,2);
121 Err = zeros(n,4,2);
122 alphas = linspace(0.001,1,n);
123 i = 1;
124
125 % Comparing with EQP solver (no lower bounds on the weights)
126 for alpha = linspace(0.001,1,n)
127
128     % Formulate problem as a bi-criterion
129     H = alpha.*Covariance;
130     g = (1-alpha).*f;
131
132     % EQP solver
133     tStart = cputime;
134     xEQP(i,:,1) = EqualityQPSolver(H,g,Aeq',beq,'rangespace');
135     Time(i,1,1) = cputime - tStart;
136
137
138     if CompareOtherSolvers
139         % Matlab's quadprog
140         tStart = cputime;
141         [xEQP(i,:,2),fval,exitflag,output] = quadprog(H,g,[],[],Aeq,beq
                ,[],[],[],options);
142         Time(i,2,1) = cputime - tStart;
143
144         % Compute error between our solver and the other solver
145
```

```
146             Err(i,1,1) = immse(xEQP(i,:,1),xEQP(i,:,2)); %norm(xEQP(i,:,1)-xEQP
                    (i,:,j));
147         end
148
149         % Compute return and risk for efficient frontier allowing short-selling
150         risk_eqp(i,1) = xEQP(i,:,1)*Covariance*xEQP(i,:,1)';
151         return_eqp(i,1) = -f'*xEQP(i,:,1)';
152
153         i = i+1;
154
155     end
156
157
158     % Setup parameters for primal-dual IP QP solver
159     x0 = zeros(nAssets,1);
160     s0 = ones(2*nAssets,1) ;
161     y0 = ones(length(beq),1);
162     z0 = ones(2*nAssets,1);
163
164     % Set lower and upper bounds
165     l  = zeros(nAssets,1);
166     u = ones(nAssets,1);
167
168     % Inequality constraints
169     C = [eye(nAssets);-eye(nAssets)];
170     d = [l;-u];
171
172     i = 1;
173     % Comparing with prima-dual IP QP solver
174     for alpha = linspace(0.001,1,n)
175
176         H = alpha*Covariance;
177         g = (1-alpha)*f;
178
179         % QP IPOPT solver
180         tStart = cputime;
181         xIPOPT(i,:,1) = PrimalDualInteriorPoint_QP(H,g,Aeq',beq,C,d,x0,y0,z0,s0)
                ;
182         Time(i,1,2) = cputime -tStart;
183
184         if CompareOtherSolvers
185             % Matlab's quadprog
186             tStart = cputime;
187             [xIPOPT(i,:,2), , ,output] = quadprog(H,g,Aineq,bineq,Aeq,beq
                    ,[],[],[],options);
188             Time(i,2,2) = cputime - tStart;
189
190             % MOSEK
191             cvx_solver mosek
192             tStart = cputime;
193             cvx_begin quiet
194                 variable x(nAssets)
195                 minimize(1/2*x'*H*x + g'*x)
196                 subject to
197                     Aeq*x == beq;
```

```
                    l <= x;
                     x <= u;
            cvx_end
            Time(i,3,2) = cputime - tStart;
            xIPOPT(i,:,3) = x;

            % Gurobi
            model.modelsense = 'min';
            model.sense = '=';
            model.Q = 0.5*alpha.*sparse(Covariance); %sparse(H);
            model.A = sparse(Aeq);
            model.rhs = beq;            % vector b or d in the linear constraints
            model.obj = g;
            model.lb = l;%zeros(nAssets,1);
            model.ub = u;
            params.OutputFlag = 0;
            tstart = cputime;
            result = gurobi(model, params);
            Time(i,4,2) = cputime - tstart;
            xIPOPT(i,:,4) = result.x;

            % CVX
            cvx_solver sdpt3 % Set back to default cvx solver
            tStart = cputime;
            cvx_begin quiet
                variable x(nAssets)
                minimize(1/2*x'*H*x + g'*x)
                subject to
                    Aeq*x == beq;
                    l <= x;
                     x <= u;
            cvx_end
            Time(i,5,2) = cputime - tStart;
            xIPOPT(i,:,5) = x;

             % Compute error between our solver and the other solvers
            for j = 2:5
                Err(i,j-1,2) = immse(xIPOPT(i,:,1),xIPOPT(i,:,j)); %norm(xIPOPT(i
                    ,:,1)-xIPOPT(i,:,j));
            end
        end

        risk_ipopt(i,1) = xIPOPT(i,:,1)*Covariance*xIPOPT(i,:,1)';
        return_ipopt(i,1) = -f'*xIPOPT(i,:,1)';

        i = i+1;
    end


    % Plot efficient frontier (short-selling allowed) with overall risks and
        returns
    figure; h = gcf;
    plot(return_eqp(:,1),risk_eqp(:,1),'b','LineWidth',1)
    title('Bi-Criterion - The Efficient Frontier: Short-Selling Allowed',...
        'FontSize', 27),
```

```
xlabel('Expected Return [%]','FontSize', 24),
ylabel('Risk [Var[R]]','FontSize', 24),
legend('Efficient Frontier','Location','northwest')
set(gca, 'FontSize',18)

% Plot efficient frontier (short-selling not allowed) with overall risks and returns
figure; h2 = gcf;
plot(return_ipopt(:,1),risk_ipopt(:,1),'b','LineWidth',1) % , R, diag(Covariance),'ko'
title('Bi-Criterion - The Efficient Frontier: Short-Selling not Allowed',...
    'FontSize', 27),
xlabel('Expected Return [%]','FontSize', 24),
ylabel('Risk [Var[R]]','FontSize', 24),
legend('Efficient Frontier','Location','north')
set(gca, 'FontSize',18)



if CompareOtherSolvers
    color = ['b','m','g','r',{'#7E2F8E'}];
    % Plot comparison of optimal portfolios for the implemented solver vs the other solvers
    figure; h3=gcf;
    semilogy(alphas, Err(:,1,1),'b','LineWidth',1)
    title('Comparison of Optimal Portfolios: Short-Selling Allowed',...
        'FontSize',32)
    legend('Matlab''s quadprog vs. EqualityQPSolver',...
        'Location','best')
    ylabel('log10(MSE)','FontSize',24),
    xlabel('\alpha','FontSize',24)
    set(gca, 'FontSize',18)

    figure; h4=gcf;
    for k=1:4
       semilogy(alphas, Err(:,k,2),'color',color{k+1},'LineWidth',1), hold on
    end
    title('Comparison of Optimal Portfolios: Short-Selling not Allowed',...
        'FontSize',30)
    legend('Matlab''s quadprog',...
        'MOSEK',...
        'Gurobi',...
        'CVX',...
         'Location','northwest')
    ylabel('log10(MSE)','FontSize',24),
    xlabel('\alpha','FontSize',24)
    set(gca, 'FontSize',19)

    % Plot CPUtime for the solvers
    figure, h5=gcf;
    plot(alphas, Time(:,1,1),'b','LineWidth',1), hold on,
    plot(alphas, Time(:,2,1),'m','LineWidth',1)
    title('CPU Time: Short-selling Allowed','FontSize',30)
    legend('EqualityQPSolver (Range Space)','Matlab''s quadprog')
```

```
    ylabel('Time [s]','FontSize',24),
    xlabel('\alpha','FontSize',24)
    set(gca, 'FontSize',18)

    figure, h6=gcf;

    for i=1:5
        plot(alphas, Time(:,i,2),'color',color{i},'LineWidth',1), hold on
    end
    title('CPU Time: Short-selling not Allowed','FontSize',30)
    legend('PrimalDualInteriorPoint\_QP','Matlab''s quadprog',...
        'MOSEK','Gurobi','CVX')
    ylabel('Time [s]','FontSize',24),xlabel('\alpha','FontSize',24)
    set(gca, 'FontSize',18)
end



%% ----- Risk-free Asset -----

%% Exercise 5.1-3

Covariance = [2.50, 0.93, 0.62, 0.74, -0.23;...
       0.93, 1.50, 0.22, 0.56, 0.26;...
       0.62, 0.22, 1.90, 0.78, -0.27;...
       0.74, 0.56, 0.78, 3.60, -0.56;...
       -0.23, 0.26, -0.27, -0.56, 3.90];
Rpre = [16.10;8.50;15.70;10.02;18.68]; % Returns
iTRet = linspace(min(Rpre),max(Rpre),10000);
nAssets = 6;

% Add a risk free security with return rf=0.0
R = [Rpre;0];
covariance = [Covariance, zeros(nAssets-1,1); zeros(1,nAssets)];
H = covariance;
f = [];

Aeq = [ones(1,nAssets); -R'];
beq = [1; -min(R)];

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

iTRet_rf = linspace(min(R),max(R),10000);
risks_rf = zeros(length(iTRet_rf),1);
port_weights_rf = zeros(length(iTRet_rf),nAssets);

options = optimset('display','off');

% Compute efficient frontier
for i = 1:length(iTRet_rf) % from lowest to highest

    beq = [1;-iTRet_rf(i)];
    [x] = quadprog(H,f,Aineq,bineq,Aeq,beq,[],[],[],options);
    risks_rf(i) =  x'*H*x; % Var(Return)
```

```
357     port_weights_rf(i,:) = x; % Weights of assets for given return
358
359 end
360
361 % Get the optimal portfolio
362 optimalRet = iTRet_rf(risks_rf == min(risks_rf));
363 effidx = find(iTRet_rf==optimalRet);
364 optimalrisk = min(risks_rf);
365 effRet = iTRet_rf(effidx:end);
366 effrisk = risks_rf(effidx:end);
367
368
369 % Plotting the efficient frontier (risk as function of return)
370 figure; fig = gcf;
371 plot(iTRet_rf, risks_rf,'b'), hold on
372 plot(R, diag(H),'ko'),  % Overall risks and returns
373 title('The Efficient Frontier: Risk-Free Security','FontSize',17),
374 xlabel('Expected Return [%]','FontSize',14), ylabel('Risk [Var]','FontSize'
      ,14),
375 legend('The Efficient Frontier: Risk-Free Security','Individual Security','
      Location','northwest')
376 set(gca, 'FontSize',13)
377
378
379 % Plotting the optimal portfolio as function of return
380 figure; fig2 = gcf;
381 plot(iTRet_rf, port_weights_rf,'LineWidth',0.95);
382 legend('Security 1','Security 2','Security 3','Security 4','Security 5',...
383     'Risk Free Security','Location','best'),
384 xlabel('Return [%]','FontSize',14),ylabel('Portfolio Weights [%]','FontSize'
      ,14),
385 title('Optimal Portfolio vs. Return: Risk-Free Security','FontSize',17)
386 set(gca, 'FontSize',13)
387
388
389 %% Exercise 5.3: Minimal risk and optimal portfolio given a return of 14.00
390
391 r = 14.00; % Desired return
392
393 options = optimoptions('quadprog','Display','off','TolFun',1e-10);
394
395 % Change beq to include the desired return
396 beq = [1;-r];
397
398 [x] = quadprog(H,f,Aineq,bineq,Aeq,beq,[],[],[],options);
399 optrisk = x'*H*x;
400
401 fprintf('Optimal Portfolio:\n')
402 disp(x)
403 fprintf('Expected return: %.f\n', r)
404 fprintf('Minimal risk: %.4f\n', optrisk)
405
406
407 % Optimal without risk-free security
```

```
[x_norf] = quadprog(Covariance,f,-eye(nAssets-1),zeros(nAssets-1,1),[ones(1,
    nAssets-1);-Rpre'],[1;-r],[],[],[],options);
optriskwithoutrf = x_norf'*Covariance*x_norf;


% Plot optimal point on efficient frontier along with optimal point without

% Risk-free security
figure; fig3 = gcf;
plot(iTRet_rf, risks_rf,'b'), hold on
plot(R, diag(H),'ko'),   % Overall risks and returns
plot(r,optrisk,'bo','MarkerFaceColor','b'),hold on

% Without Risk-free security
plot(r, optriskwithoutrf,'mo','MarkerFaceColor','m'),
plot(effRet,effrisk,'m')
plot(iTRet,risks,'m--'),
legend('Efficient Frontier with Risk-Free Security',...
    'Individual Securities', 'Miminum Risk with Risk-Free Security',...
    'Minimum Risk without Risk-Free Security',...
    'Efficient Frontier without Risk-Free Security',...
    'Location','northwest'),
xlabel('Expected Return [%]','FontSize',14),ylabel('Risk [Var]','FontSize'
    ,14),
title('The Efficient Frontier with and without Risk-Free Security','FontSize
    ',17)
set(gca, 'FontSize',13)
```

Listing A.12: Driver for problem 5, Chapter 5.

A.3 Test Problem Functions

A.3.1 Recycling System Test Problem

```
function [H,g,A,b] = objfun_RecycleSystem(n,ubar,d0)
% objfun_RecycleSystem  Generates parameters for solving a
% quadratic program based on the recycling system from week 5
% in the course 02612 Constrainted Optimization:
%
%          min 0.5*x'H*x + g'x
%            x
%           s.t. A'x = b
%
% Syntax: [H,A,b,g] = objfun_RecycleSystem(n,ubar,d0)

n = n-1;
H = eye(n+1); % n+1 variables

A = [zeros(1, n-1); eye(n-1)];
A = [A zeros(n, 1)] -eye(n);
```

```
A(:,n+1) = [zeros(n-1,1); -1]'; % Add row with -1 in the end
A(1, n) = 1;
A = A'; % such that A'*x = b

g = -1 * ubar * ones(n+1,1);
b = [-d0; zeros(n-1, 1)];
```

Listing A.13: A function to generate parameters for the Recycling System test problem provided in week 5 exercises.

A.3.2 Random Convex EQP Generator

```
function [H,g,A,b,x,lambda] = GenerateRandomConvexQP(n,varargin)
% GenerateRandomConvexQP Generates a random convex quadratic program
% with linear constraints
%
%               min 1/2*x'Hx + g'x
%               subject to A'x = b
%
% Syntax: [H,g,A,b,x,lambda] = GenerateRandomConvexQP(n,varargin)
%
% Varargin can be:
%                   m | number of constraints


%% Define dimensions
if  isempty(varargin)
    x_dim = n;
    lambda_dim = varargin{1};
    if x_dim<lambda_dim % cannot have more constraints than variables
        error('Number of variables must be larger than number of constraints
            ')
    end
else
    x_dim = n;
    % Randomly select nr of equality constraints
    lambda_dim = randi(x_dim,1);
end

%% Main
x = rand(x_dim, 1);

lambda = rand(lambda_dim,1);

% Generate matrix A for linear constraints
A = rand(x_dim, lambda_dim);

% Make sure H is positive definite
X = rand(x_dim);
H = X*X' + x_dim*eye(x_dim);

```

```
39 % Generate output matrices g and b
40 gb = [H, -A; -A' zeros(lambda_dim)]*[x;lambda];
41 g= -gb(1:x_dim);
42 b=-gb(x_dim+1:end);
43
44
45 end
```

Listing A.14: A function to generate a random convex QP problem.

A.4 Exam Assignment Description

02612 Constrained Optimization 2022
Exam Assignment
Hand-in deadline: May 30, 2022, 13:30

John Bagterp Jørgensen

April 5, 2022

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`, MOSEK, Gurobi, 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. Test your LP algorithm on the following test problem with the data

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

A =
1.0000
1.0000
1.0000
1.0000
1.0000

b =
1
```

   `l=zeros(5,1)` and `u=ones(5,1)`.

6. 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`, MOSEK, Gurobi, 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 and other nonlinear optimization problems you choose using a library function for nonlinear programs, e.g. `fmincon` in Matlab, IPOPT, and `CasADi (IPOPT called from 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
9. Solve a number of test problems you choose using library optimization algorithms and your own nonlinear optimization algorithms. Use the test to document that your algorithm works and to document its efficiency compared to library functions.

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 own algorithms (e.g. Problem 1-3) 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`, MOSEK, Guroi, 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 Learn: 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.

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 May 30, 2022 at 13:30.

Bibliography

[1] J. B. Jørgensen, *Numerical Methods for Constrained Optimization*. Springer, 2021.

[2] J. B. Jørgensen, "02612 constrained optimization." Online. `http://www.imm.dtu.dk/~jbjo/02612.html`, [Accessed: April 30, 2022].

[3] M. J. Osborne, "Mathematical methods for economic theory: 3.1 concave and convex functions of a single variable." Online. `https://mjo.osborne.economics.utoronto.ca/index.php/tutorial/index/1/cv1/t`, [Accessed: April 25, 2022].

[4] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering, Springer New York, 2006.

[5] E. L. Hansen and C. Völcker, *Numerical Algorithms for Sequential Quadratic Optimization*. Technical University of Denmark, 2007.

[6] M. S. Gockenbach, *Introduction to inequality-constrained optimization: The logarithmic barrier method*. Michigan Technological University, 2003.

[7] "Harry markowitz's modern portfolio theory [the efficient frontier]." Online. `https://www.guidedchoice.com/video/dr-harry-markowitz-father-of-modern-portfolio-theory/`, [Accessed: May 18, 2022].

[8] A. Hayes, "Short selling." Online. `https://www.investopedia.com/terms/s/shortselling.asp`, [Accessed: May 18, 2022].

[9] J. Chen, "What is a risk-free asset?." Online. `https://www.investopedia.com/terms/r/riskfreeasset.asp`, [Accessed: May 18, 2022].

{2.7182818284}ϑφερτυθιοπσδφγηξκλ