02612

Constrained Optimization

Assignment 2

Phillip Brinck Vetter (s144097)
Ali Saleem (s154675)
Raja Shan Zaker Mahmood (s144102)

May 2, 2018
Technical University of Denmark

Contents

Introduction

This assignment presents answers to Assignment 2 in the course. The general topic of the assignment is Sequential Quadratic Programs. Information are gathered from course slides or the book [Nocedal and Wright, 2006] which in both cases will be evident in the text. Code will be included directly in the text where it is deemed to provide a better overview, while larger functions and main scripts will be provided in the Appendix. It is our hope that this improves the reading of the report and removes needless flicking back and forth, as we know that the report will have to be read in a very short time.

Problem 1 - Interior-Point Algorithm for Linear Programming

For this part of the assignment we will demonstrate an primal-dual-Interier-Point algorithm for solving of linear programs on the standard form:

$$\min c^T x, \quad \text{subject to } Ax = b, \quad x \geq 0 \tag{1}$$

with the dual problem:

$$\min b^T \lambda, \quad \text{subject to } A^T\lambda + s = c, \quad s \geq 0 \tag{2}$$

where $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $\lambda \in \mathbb{R}^m$ and $s \in \mathbb{R}^n$. The KKT-conditions that has to be fulfilled for a solution of (1) and (2) are the following:

$$\begin{aligned} A^T\lambda + s &= c, && \text{(3a)}\\ Ax &= b, && \text{(3b)}\\ x_i s_i &= 0, \;\; i = 1, 2, \ldots, n, && \text{(3c)}\\ (x, s) &\geq 0. && \text{(3d)} \end{aligned}$$

These optimality conditions can be restated into a mapping: $F : \mathbb{R}^{2n+m} \to \mathbb{R}^{2n+m}$,

$$F(x, \lambda, s) = \begin{bmatrix} A^T\lambda + s - c \\ Ax - b \\ XSe \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{4a}$$

$$x, s \geq 0, \tag{4b}$$

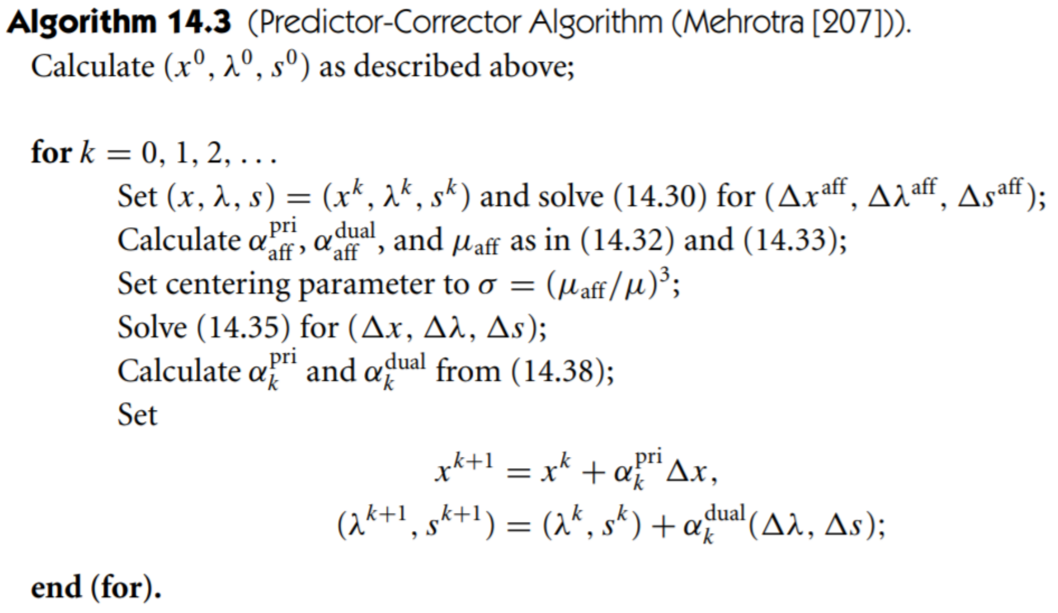
where $X = \text{diag}(x_1, x_2, \ldots, x_n)$ and $S = \text{diag}(s_1, s_2, \ldots, s_n)$. To determine a search direction for the nonlinear system in (4) we use a Newton's method. We use Newton's method to solve the following system of equations:

$$J(x, \lambda, s) \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = -F(x, \lambda, s) \tag{5}$$

where $J(x,\lambda,s) = \begin{bmatrix} \nabla_x F & \nabla_\lambda F & \nabla_s F \end{bmatrix}$. This is also the jacobian of F. Writing (5) out gives:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} -(A^T\lambda + s - c) \\ -(Ax - b) \\ -XSe \end{bmatrix} \tag{6}$$

The primal-dual methods solves the linear system of equations seen on (6) to get a search direction, but an important feature of these methods, is that they consider the nonnegativity bounds in (3d) to be strictly positive, hence the name *Interior-point methods*. However, by simply setting the search direction as the solution of (6) would usually violate the condition of $(x, s) > 0$. This pure Newton step is usually denoted the *affine scaling direction* and rarely allows much progress towards a solution. We will therefore demonstrate an algorithm known as the *Mehrotra Predictor-Corrector Algorithm* which is an primal-dual interior point method for solving such linear programs. An outline of the algorithm can be seen on Figure 1:



```
**Algorithm 14.3** (Predictor-Corrector Algorithm (Mehrotra [207])).
  Calculate $(x^0, \lambda^0, s^0)$ as described above;

  for $k = 0, 1, 2, \ldots$
      Set $(x, \lambda, s) = (x^k, \lambda^k, s^k)$ and solve (14.30) for $(\Delta x^{\text{aff}}, \Delta\lambda^{\text{aff}}, \Delta s^{\text{aff}})$;
      Calculate $\alpha^{\text{pri}}_{\text{aff}}, \alpha^{\text{dual}}_{\text{aff}}$, and $\mu_{\text{aff}}$ as in (14.32) and (14.33);
      Set centering parameter to $\sigma = (\mu_{\text{aff}}/\mu)^3$;
      Solve (14.35) for $(\Delta x, \Delta\lambda, \Delta s)$;
      Calculate $\alpha^{\text{pri}}_k$ and $\alpha^{\text{dual}}_k$ from (14.38);
      Set
                    $x^{k+1} = x^k + \alpha^{\text{pri}}_k \Delta x,$
          $(\lambda^{k+1}, s^{k+1}) = (\lambda^k, s^k) + \alpha^{\text{dual}}_k (\Delta\lambda, \Delta s);$

  end (for).
```

Figure 1: Primal-dual-Interior-Point algorithm as illustrated in [Nocedal and Wright, 2006] (p.411)

As most algorithms, we need an initial guess of the optimal solution (x^*, λ^*, s^*). This initial guess is denoted (x^0, λ^0, s^0). The choice of this initial guess is an important one as it plays a significant role on the robustness of this algorithm. Apparently, a poor choice of an initial guess which only satisfies the minimal condition, that is $x^0 > 0$ and $s^0 > 0$ can result in a failure of convergence. A method to find a good starting point is described on [Nocedal and Wright, 2006] (p.410). It goes as follows:
You find a vector $\tilde{x}$ of minimum norm which satisfies the primal constraint $Ax = b$ and vectors $\tilde{\lambda}$ and $\tilde{s}$ satisfying the dual constraint $A^T\lambda + s = c$ such that $\tilde{s}$ also has minimum norm. This

corresponds to solving the following QPs:

$$\min_{x} \frac{1}{2}x^T x \quad \text{subject to} Ax = b \tag{7a}$$

$$\min_{(\lambda,s)} \frac{1}{2}s^T s \quad \text{subject to} A^T\lambda + s = c \tag{7b}$$

Apparently, $\tilde{x}$ and $\tilde{\lambda}$ and $\tilde{s}$ can be written explicitly:

$$\tilde{x} = A^T(AA^T)^{-1}b, \quad \tilde{\lambda} = (AA^T)^{-1}Ac, \tilde{s} = c - A^T\tilde{\lambda} \tag{8}$$

Often, the solution to these QPs will include negative components for $\tilde{s}$ and $\tilde{x}$ which violates the nonnegativity constraint we have on x and s. We therefore need to adjust $\tilde{x}$ and $\tilde{s}$ as follows:

$$\tilde{x} = \tilde{x} + \delta_x e, \quad \tilde{s} = \tilde{s} + \delta_s e,$$

where e denotes the usual $e = (1, 1, \ldots, 1)^T$ and δ_x and δ_s are defined as follows:

$$\delta_x = \max\left(-\frac{3}{2} \cdot \min_i \tilde{x}_i, 0\right), \quad \delta_s = \max\left(-\frac{3}{2} \cdot \min_i \tilde{s}_i, 0\right)$$

To ensure now that all components in $\tilde{x}$ and $\tilde{s}$ are not too close to zero and not to dissimilar, we define 2 new quantities:

$$\tilde{\delta_x} = \frac{1}{2}\frac{\tilde{x}^T\tilde{s}}{e^T\tilde{s}}, \quad \tilde{\delta_s} = \frac{1}{2}\frac{\tilde{x}^T\tilde{s}}{e^T\tilde{x}}$$

We can now finally define the starting points as follows:

$$x^0 = \tilde{x} + \tilde{\delta}_x e, \quad \lambda^0 = \tilde{\lambda}, \quad s^0 = \tilde{s} + \tilde{\delta}_s e$$

The computational cost of producing such an initial guess with this method is approximately the same as one step of the primal-dual method.[Nocedal and Wright, 2006](p.410)
With the initial guess you solve the system of equations seen in (6) to get the affine scaling direction. This is also known as the *Predictor step.* Afterwards you compute a step length for the primal and dual variables. By computing the step length with the following scheme, you're attaining the maximum allowable step length along the affine-direction:

$$\alpha_{\text{aff}}^{\text{prim}} = \min\left(1, \min_{i:\Delta x_i^{\text{aff}}<0} -\frac{x_i}{\Delta x_i^{\text{aff}}}\right) \tag{9a}$$

$$\alpha_{\text{aff}}^{\text{dual}} = \min\left(1, \min_{i:\Delta s_i^{\text{aff}}<0} -\frac{x_i}{\Delta s_i^{\text{aff}}}\right) \tag{9b}$$

This scheme ensures that if the step along the affine direction reduces the duality measure μ, which is defined as the following:

$$\mu = \frac{\sum_{i=1}^{n} x_i s_i}{n}, \tag{10}$$

the centering parameter σ is set to a smaller value and a higher value, close to 1, if it doesn't make much progress with reducing the duality measure. You furthermore calculate the duality measure along the affine-direction:

$$\mu_{\text{aff}} = \frac{\left(x + \alpha_{\text{aff}}^{\text{prim}} \Delta x^{\text{aff}}\right)^T \left(s + \alpha_{\text{aff}}^{\text{dual}} \Delta s^{\text{aff}}\right)}{n} \tag{11}$$

This is used to set the centering parameter σ which is defined as follows:

$$\sigma = \left(\frac{\mu_{\text{aff}}}{\mu}\right)^3 \tag{12}$$

There is no analytic reasoning behind this choice of centering parameter but has been deemed to work well in practice. Now with the centering parameter, we are able to compute the *Corrector step* by solving the following system:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cor}} \\ \Delta \lambda^{\text{cor}} \\ \Delta s^{\text{cor}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\Delta X^{\text{aff}} \Delta S^{\text{aff}} e \end{bmatrix} \tag{13}$$

Summing the Predictor and Corrector step makes typically better progress with reducing the duality measure than the affine-step alone. However, we also make use of a centering parameter σ. We can furthermore aggregate the predictor, centering and corrector contributions into a single system:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_c \\ -r_b \\ -XSe - \Delta X^{\text{aff}} \Delta S^{\text{aff}} e + \sigma \mu e \end{bmatrix} \tag{14}$$

where $r_c = A^T\lambda + s - c$ and $r_b = Ax - b$. The solution to this system is the corrected search direction where the centering parameter has been taking into consideration. Notice that the coefficient matrix on the left hand side of (6) and (14) are the same. This means we only need to factorize it once and the cost of solving the second system (14) is quite small. A test script has been made to test the implemented algorithm. The test script can be seen here:

```
%% Test script for Algorithm 14.3

%Initialize some random variables

%%% REMEMBER THAT A NEEDS TO HAVE FULL ROW RANK %%%
n = 3;     %Variable
```

```
m = 2;     %Constraints


A = 3*randi(4,m,n);   %Generate random A matrix

x = 4*rand(n,1);    %Generate random "solution"
x(m+1:end) = 0;

s = 10*rand(n,1);   %Generate random slack variables
s(1:m) = 0;

b = A*x;            %Calculate equalities from random A and "solution"
lambda = 10*rand(m,1);  %Generate random lagranian multipliers
g = A'*lambda+s;        %Generate obj function

%Check if we get the same as the "solution"
[x_opt,lambda_opt,s_opt] = PredictorCorrectorV2(g,A,b);
```

For the primal-dual interior point algorithm, we will refer to take a look at **Appendix A**

Problem 2 - Equality Constrained SQP

1)

For this section, we want to apply the local SQP Algorithm to solve an equality constrained problem. We sometimes denote the vector $[x_1, x_2, x_3, x_4, x_5]^T \in \mathcal{R}^5$ as x as a shorthand. The problem has the following objective function.

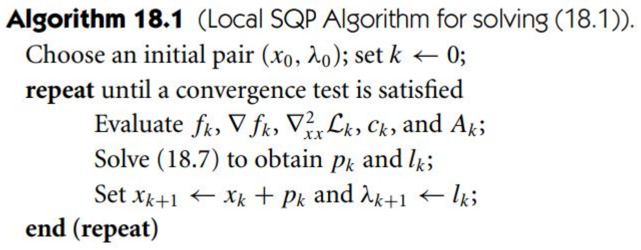
$$\min_x \quad f(x) = \exp(x_1x_2x_3x_4x_5) - \frac{1}{2}\left(x_1^3 + x_2^3 + 1\right)^2 \tag{15}$$

$$\text{s.t} \quad c_1(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 10 = 0 \tag{16}$$

$$c_2(x) = x_2x_3 - 5x_4x_5 = 0 \tag{17}$$

$$c_3(x) = x_1^3 + x_2^3 + 1 = 0 \tag{18}$$

The algorithm used to solve the problem, can be found in [Nocedal and Wright, 2006], as Algorithm 18.1.



Algorithm 18.1 (Local SQP Algorithm for solving (18.1)).

```
Choose an initial pair $(x_0, \lambda_0)$; set $k \leftarrow 0$;
repeat until a convergence test is satisfied
    Evaluate $f_k, \nabla f_k, \nabla^2_{xx}\mathcal{L}_k, c_k$, and $A_k$;
    Solve (18.7) to obtain $p_k$ and $l_k$;
    Set $x_{k+1} \leftarrow x_k + p_k$ and $\lambda_{k+1} \leftarrow l_k$;
end (repeat)
```

Figure 2: Local SQP Algorithm pseudocode.

A flowchart can be constructed for this algorithm. This is shown informally in figure 3.

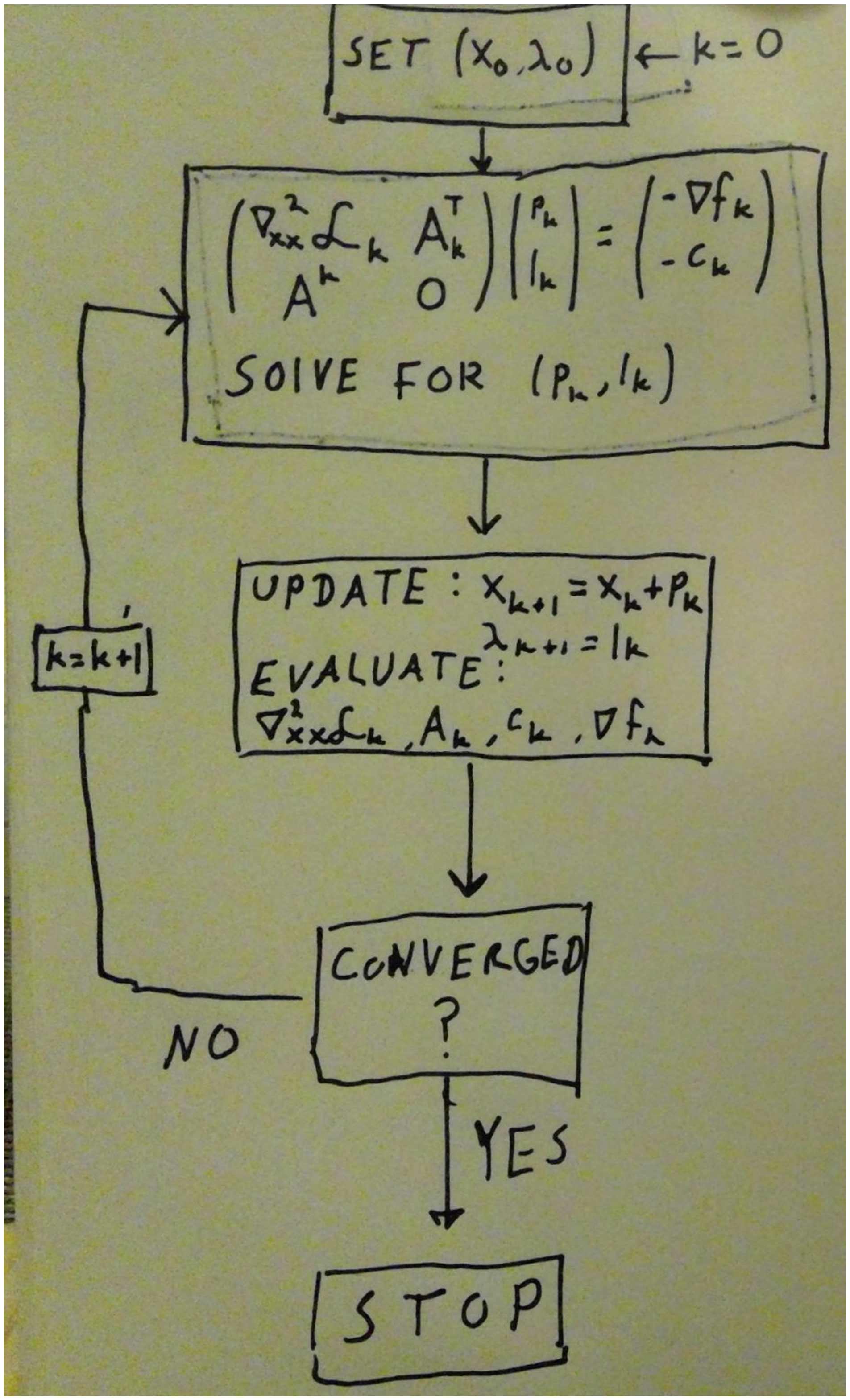

SET (x_0, λ_0)
$\leftarrow k=0$
$\begin{pmatrix} \nabla^2_{xx}\mathcal{L}_k & A_k^T \\ A^k & 0 \end{pmatrix} \begin{pmatrix} p_k \\ l_k \end{pmatrix} = \begin{pmatrix} -\nabla f_k \\ -c_k \end{pmatrix}$
SOlVE FOR (p_k, l_k)
UPDATE : $x_{k+1} = x_k + p_k$
$\lambda_{k+1} = l_k$
EVALUATE:
$\nabla^2_{xx}\mathcal{L}_k, A_k, c_k, \nabla f_k$
$k=k+1$
CONVERGED ?
NO
YES
STOP

Figure 3: A flow chart for the local SQP algorithm, known as algorithm 18.1. This shows an informal overview of the control structure. For a more detailed implementation, we refer to the code later in this section.

An initial guess for x is set as $x_0 = [-1.8, 1.7, 1.9, -0.8, -0.8]^T$, and for λ as $\lambda_0 = [1, 1, 1]^T$. As a first step, we implement functions for evaluating the function value of the objective function, the constraints and their respective Gradients and Hessians. Here the objective function, as $ObjFun.m$

```
function [f,df,d2f] = ObjFun(x)
%Computation of the function its derivative and its hessian

%Variables
x1 = x(1); x2 = x(2); x3 = x(3); x4 = x(4); x5 = x(5);

%Useful expressions
e = exp(x1*x2*x3*x4*x5);
k = (x1^3 + x2^3 + 1);

%Function value
f = e - 0.5*k^2;

%Derivative
df = [ x(2)*x(3)*x(4)*x(5)*e - 3*k*x(1)^2 ; x(1)*x(3)*x(4)*x(5)*e...
    - 3*k*x(2)^2 ; x(1)*x(2)*x(4)*x(5)*e ; x(1)*x(2)*x(3)*x(5)*e ;...
    x(1)*x(2)*x(3)*x(4)*e ];

%Hessian

H1 = [ x2^2*x3^2*x4^2*x5^2*e - 9*x1^4 - 6*k*x1 , ...
    x3*x4*x5*e + x1*x2*x3^2*x4^2*x5^2*e - 9*x2^2*x1^2 ,...
    x2*x4*x5*e + x1*x2^2*x3*x4^2*x5^2*e , x2*x3*x5*e ...
    + x1*x2^2*x3^2*x4*x5*e , x2*x3*x4*e + x1*x2^2*x3^2*x4^2*x5*e ];
H2 = [ x3*x4*x5*e + x1*x2*x3^2*x4^2*x5^2*e - 9*x1^2*x2^2 ,...
    (x1*x2*x3*x4)^2*e - 9*x2^4 - 6*k*x2 , x1*x4*x5*e + ...
    x1^2*x2*x3*x4^2*x5^2*e , x1*x3*x5*e + x1^2*x2*x3^2*x4*x5^2*e ,...
    x1*x3*x4*e + x1^2*x2*x3^2*x4^2*x5*e ];
H3 = [ x2*x4*x5*e + x1*x2^2*x3*x4^2*x5^2*e , x1*x4*x5*e...
    + x1^2*x2*x3*x4^2*x5^2*e , x1^2*x2^2*x4^2*x5^2*e ,...
    x1*x2*x5*e + x1^2*x2^2*x3*x4^2*x5^2*e , x1*x2*x4*e...
    + x1^2*x2^2*x3*x4^2*x5*e ];
H4 = [ x2*x3*x5*e + x1*x2^2*x3^2*x4*x5^2*e , x1*x3*x5*e ...
    + x1^2*x2*x3^2*x4*x5^2*e , x1*x2*x5*e + x1^2*x2^2*x3*x4*x5^2*e...
    , (x1*x2*x3*x5)^2*e , x1*x2*x3*e + x1^2*x2^2*x3^2*x4*x5*e ];
H5 = [ x2*x3*x4*e + x1*x2^2*x3^2*x4^2*x5*e , x1*x3*x4*e +...
    x1^2*x2*x3^2*x4^2*x5*e , x1*x2*x4*e + x1^2*x2^2*x3*x4^2*x5*e...
    , x1*x2*x3*e + x1^2*x2^2*x3^2*x4*x5*e , (x1*x2*x3*x4)^2*e ];
d2f = [H1 ; H2 ; H3 ; H4 ; H5];


end
```

The evaluations of the constraints are each implemented in their own files, numbered as in the beginning of the section.

```
function [c,dc,d2c] = ConFun1(x)

c = x(1)^2 + x(2)^2 + x(3)^2 + x(4)^2 + x(5)^2 - 10;

dc = [2*x(1) ; 2*x(2) ; 2*x(3) ; 2*x(4) ; 2*x(5)];

d2c = 2*eye(5);

end
```

```
function [c,dc,d2c] = ConFun2(x)

c = x(2)*x(3) - 5*x(4)*x(5);

dc = [0 ; x(3) ; x(2) ; -5*x(5) ; -5*x(4)];

d2c = zeros(5);
d2c(3,2) = 1; d2c(2,3) = 1; d2c(5,4) = -5; d2c(4,5) = -5;

end
```

```
function [c,dc,d2c] = ConFun3(x)

c = x(1)^3 + x(2)^3 + 1;

dc = [3*x(1)^2 ; 3*x(2)^2 ; 0 ; 0 ; 0];

d2c = zeros(5,5,1);
d2c(1,1,1) = 6*x(1); d2c(2,2) = 6*x(2);

end
```

The program works by considering the equality constrained version of a nonlinear program.

$$\min_{x\in\mathcal{R}^n} \quad f(x) \tag{19}$$
$$\text{s.t} \quad c(x) = 0 \tag{20}$$

The KKT, first order optimality conditions are then stated, in terms of the Lagrangian function $L(x,y) = f(x) - y^T c(x)$, y being a vector of Lagrange multipliers. The condition that the gradient of the Lagrangian must equal zero at an optimal point, can be expressed in matrix form.

$$F(x,y) = \begin{bmatrix} \nabla_x L(x,y) \\ \nabla_y L(x,y) \end{bmatrix} = \begin{bmatrix} \nabla L(x,y) - y^T c(x) \\ -c(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{21}$$

We call the left hand side of the above matrix $F(x,y)$. The nonlinear problem can be solved by the use of an iterative method similar to that of Newton. Take a Taylor expansion of $F(x,y)$ to first order around the iterate (x^k, y^k) and evaluate it in the next iteration (x^{k+1}, y^{k+1}) to obtain

$$F(x,y) = F(x^k,y^k) + \left[\nabla F(x^k,y^k)\right]^T \left(x^{k+1} - x^k\right) = F(x^k,y^k) + J(x^k,y^k)\Delta x = 0 \tag{22}$$

The last equation can then be rewritten to obtain the equation that yields the step direction Δx by

$$J(x^k,y^k)\Delta x = -F(x^k,y^k) \tag{23}$$

Where as shown in the above equations the Jacobian is the transposed gradient of $F(x^k,y^k)$ i.e

$$J(x^k,y^k) = \left[\nabla F(x^k,y^k)\right]^T = \begin{bmatrix} \nabla^2_{xx}L(x^k,y^k) & -\nabla c(x^k) \\ -\nabla c(x^k)^T & 0 \end{bmatrix} \tag{24}$$

Thus in a closed form the local equality-constrained quadratic sub-problem is given by

$$\begin{bmatrix} \nabla^2_{xx}L(x^k,y^k) & -\nabla c(x^k) \\ -\nabla c(x^k)^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \nabla_x L(x^k,y^k) \\ -c(x^k) \end{bmatrix} \tag{25}$$

The system of equations can be further reduced by splitting the right hand side into the two contributions from the Lagrangian $\nabla_x L(x^k, y^k = \nabla_x f(x^k) - \nabla_x c(x^k) y^k$ and furthermore using that $[\Delta x\,,\,\Delta y] = \left[\Delta x\,,\,y^{k+1}\right] - \left[0\,,\,y^k\right]$. Insertion then yields the final closed form

$$\begin{bmatrix} \nabla^2_{xx}L(x^k,y^k) & -\nabla c(x^k) \\ -\nabla c(x^k)^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ y^{k+1} \end{bmatrix} = -\begin{bmatrix} \nabla_x f(x^k) \\ -c(x^k) \end{bmatrix} \tag{26}$$

The above problem is the one which is solved in the *Problem2a - Equality Constrained SQP* script provided below. In *Problem2b - Equality Constrained SQP* the hessian of the Lagrangian $\nabla^2_{xx}L(x^k,y^k)$ which provides the main computation power is simply substituted using the BFGS approximation as described in detail later.
As suggested in section 18.1 in [Nocedal and Wright, 2006], the procedure can also be viewed as modelling the nonlinear program as a quadratic function and using the usual procedure for the solution to an equality constrained quadratic program. The algorithm is then evaluation of objective function, constraint functions, respective gradients and Hessians, followed by the solution to an equality constrained quadratic problem. A step is taken taken based on the solution to the QP. There is no modification to the step length and the full Netwon step is taken at each iteration. The function evaluations are done again and convergence is tested. If the problem has converged, an acceptable solution has been found, otherwise the procedure starts over. This can be illustrated in a flow diagram.

```
%% Problem 2a - Equality Constrained SQP
clear variables; close all; clc

%Starting Guess and 3 Starting Lagrange Multipliers
x = [-1.8 , 1.7 , 1.9 , -0.8 , -0.8]';
y = [ 1 , 1 , 1]';

%Initialization
```

```matlab
maxit = 100;
tol = 10^(-8);
k = 0;

%Initial Computations
[f, df, d2f] = ObjFun(x);
[c(1,1), dc(:,1), d2c(:,:,1)] = ConFun1(x);
[c(2,1), dc(:,2), d2c(:,:,2)] = ConFun2(x);
[c(3,1), dc(:,3), d2c(:,:,3)] = ConFun3(x);

%Stats
stat.k = [];
stat.x = x;
stat.y = y;
stat.df = df;
stat.d2f = d2f;
stat.time = [];

%Main Algorithm
Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
while ~Converged && k < maxit
    tic
    k = k + 1;
    %Compute Hessian
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    G = 0;
    for i = 1:3
        G = G - y(i)*d2c(:,:,i);
    end

    H = d2f + G;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Solve Quadratic Problem using LDL Factorization
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    KKT = [H , -dc ; -dc' , zeros(3,3)];
    RHS = -[df ; -c];

    [L,D,p] = ldl(KKT,'lower','vector');
    d(p) = L' \ ( D \ ( L \ RHS(p) ) ); d = reshape(d,8,1);

    dx = d(1:5);
    y = d(6:8);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %New Step and Function Evaluations
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    x = x + dx;
    [f, df, d2f] = ObjFun(x);
    [c(1,1), dc(:,1), d2c(:,:,1)] = ConFun1(x);
    [c(2,1), dc(:,2), d2c(:,:,2)] = ConFun2(x);
    [c(3,1), dc(:,3), d2c(:,:,3)] = ConFun3(x);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Convergence Check
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    stat.k = [stat.k k];
    stat.x = [stat.x x];
    stat.y = [stat.y y];
    stat.df = [stat.df df];
    stat.d2f = [stat.d2f d2f];
    stat.time = [stat.time toc];
end

if Converged && k <= maxit
    disp('A solution was found and is given by x = ')
    disp(x)
end
```

The iteration sequence for the problem is given to 4 decimals. The tolerance is set at 10^{-8}, so the variation in the last digits is invisible here, but the convergence happened at $k = 7$.

$$\left[\begin{array}{c|cccccccc}
k & 0 & 1 & 1 & 3 & 4 & 5 & 6 & 7 \\
\hline
x_1 & -1.8 & -1.7567 & -1.5979 & -1.7386 & -1.7138 & -1.717 & -1.717 & -1.7171 \\
x_2 & 1.7 & 1.6422 & 1.4595 & 1.6236 & 1.5920 & 1.5958 & 1.5957 & 1.5957 \\
x_3 & 1.9 & 1.7580 & 2.0595 & 1.8062 & 1.8338 & 1.8271 & 1.8272 & 1.8272 \\
x_4 & -0.8 & -0.7599 & -0.7828 & -0.7713 & -0.7643 & -0.7636 & -0.7636 & -0.7636 \\
x_5 & -0.8 & -0.7599 & -0.7828 & -0.7714 & -0.7643 & -0.7636 & -0.7636 & -0.7636
\end{array}\right] \tag{27}$$

2

In this section we modify the previous algorithm, by introducing the BFGS-approximation to the Hessian. There are situations where the information represented by the Hessian is impractical to obtain, and we would therefore like to have a method for approximating that matrix, preserving as much information as necessity dictates. The approximation to the Hessian, B, is defined iteratively, starting with an identity matrix. In keeping with the book [Nocedal and Wright, 2006], using the notation used in the course slides, we have that we use the gradient information. There is no z variable, as we have no inequality constraints.

$$p = x^{k+1} - x^k \tag{28}$$
$$q = \nabla_x L(x^{k+1}, y^{k+1}) - \nabla_x L(x^k, y^{k+1}) \tag{29}$$

We want iteraties to fulfill the curvature condition $p^T q > 0$. To ensure that this is the case, we use the damped BFGS-approximation, where we define a parameter θ, so that.

$$\theta = \begin{cases} 1 & p^T q \geq 0.2 p^T (Bp) \\ \dfrac{0.8 p^T (Bp)}{p^T (Bp) - p^T q} & p^T q \leq 0.2 p^T (Bp) \end{cases} \tag{30}$$

This value then interpolates the following expression, replacing q.

$$r = \theta q + (1 - \theta)(Bq) \tag{31}$$

The iteration of B is then, finally.

$$B \leftarrow B + \frac{r r^T}{p^T r} - \frac{(Bp)(Bp)^T}{p^T (Bp)} \tag{32}$$

This update is implemented in the script below.

```matlab
%% Problem 2b - Equality Constrained SQP
clear variables; close all; clc

%Starting Guess and 3 Starting Lagrange Multipliers
x = [-1.8 , 1.7 , 1.9 , -0.8 , -0.8]';
y = [ 1 , 1 , 1]';

%Initialization
maxit = 100;
tol = 10^(-8);
k = 0;
B = eye(5);

%Initial Computations
[f df] = ObjFun(x);
[c(1,1) dc(:,1)] = ConFun1(x);
[c(2,1) dc(:,2)] = ConFun2(x);
[c(3,1) dc(:,3)] = ConFun3(x);

%Stats
stat.k = [];
stat.x = x;
stat.y = y;
stat.df = df;
stat.B = B;
stat.theta = [];
stat.time = [];

%Main Algorithm
Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
while ~Converged && k < maxit
    tic
    k = k + 1;

    %Solve BFGS Problem using LDL Factorization
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    KKT = [B , -dc ; -dc' , zeros(3,3)];
    RHS = -[df ; -c];

    [L,D,s] = ldl(KKT,'lower','vector');
    d(s) = L' \ ( D \ ( L \ RHS(s) ) ); d = reshape(d,8,1);

    p = d(1:5);
    y_new = d(6:8);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %New Step and New Function Evaluations
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    x_new = x + p;
    [f_new df_new] = ObjFun(x_new);
    [c_new(1,1) dc_new(:,1)] = ConFun1(x_new);
    [c_new(2,1) dc_new(:,2)] = ConFun2(x_new);
    [c_new(3,1) dc_new(:,3)] = ConFun3(x_new);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %BFGS Update Step
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    q = ( df_new - dc_new*y_new ) - ( df - dc*y_new ) ;

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

    r = theta * q + (1-theta)*(B*p);

    B = B + (r * r') ./ ( p'*r ) - (B*p)*(B*p)' / ( p'*(B*p) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Rename Variables
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    f = f_new; df = df_new;
    c = c_new; dc = dc_new;
    y = y_new;
    x = x_new;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Convergence Check
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Stats Update
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    stat.k = [stat.k k];
    stat.x = [stat.x x];
    stat.y = [stat.y y];
    stat.df = [stat.df df];
    stat.B = [stat.B B];
    stat.theta = [stat.theta theta];
    stat.time = [stat.time toc];
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
end

The iteration sequence for the BFGS implementation follows. We see that it converges at a
if Converged && k <= maxit
disp('A solution was found and is given by x = ')
disp(x)
end
```

$$\left[\begin{array}{c|ccccccccc}
k & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
x_1 & -1.8 & -1.7269 & -1.7244 & -1.7239 & -1.7218 & -1.7171 & -1.7171 & -1.7171 & -1.7171 \\
x_2 & 1.7 & 1.6087 & 1.6041 & 1.6035 & 1.6011 & 1.5957 & 1.5957 & 1.5957 & 1.5957 \\
x_3 & 1.9 & 1.8132 & 1.8137 & 1.8146 & 1.8186 & 1.8273 & 1.8272 & 1.82725 & 1.8272 \\
x_4 & -0.8 & -0.7636 & -0.76281 & -0.7629 & -0.7631 & -0.7637 & -0.7636 & -0.7636 & -0.7636 \\
x_5 & -0.8 & -0.7636 & -0.7628 & -0.7629 & -0.7631 & -0.7636 & -0.7636 & -0.7636 & -0.7636
\end{array}\right] \tag{33}$$

3

The implementation of the line search functionality in the above algorithms is based on the Powell l_1 merit function. We would like to choose a step length that gives a sufficient decrease in the objective (here merit) function. We can state this for the Powell l_1-merit function, using the so-called Armijo Condition. These can be found in [Nocedal and Wright, 2006] and the specific version we have implemented, is found in the course slides and in figure 4.

```
Require: $f(x^k)$, $\nabla f(x^k)$, $h(x^k)$, $g(x^k)$, $\lambda$, $\mu$, $\Delta x^k$
  $\alpha = 1$, $i = 1$, STOP = false
  $c = \phi(0) = f(x^k) + \lambda|h(x^k)| + \mu|\min\{0, g(x^k)\}|$
  $b = \phi'(0) = \nabla f(x^k)'\Delta x^k - \lambda'|h(x^k)| - \mu'|\min\{0, g(x^k)\}|$
  while not STOP do
    $x = x^k + \alpha\Delta x^k$
    Evaluate $f(x)$, $h(x)$, $g(x)$
    Compute $\phi(\alpha) = f(x) + \lambda'|h(x)| + \mu'|\min\{0, g(x)\}|$
    if $\phi(\alpha) \leq \phi(0) + 0.1\phi'(0)\alpha$ then
      STOP = true
    else
      Compute $a = \frac{\phi(\alpha)-(c+b\alpha)}{\alpha^2}$ and $\alpha_{\min} = \frac{-b}{2a}$
                  $\alpha = \min\{0.9\alpha, \max\{\alpha_{\min}, 0.1\alpha\}\}$
    end if
  end while
```

Figure 4: The line search algorithm used in the project. It follows from applying the Armijo conditions to the Powell l_1 merit function.

```matlab
%% Problem 2c - Equality Constrained SQP
clear variables; close all; clc

%Starting Guess and 3 Starting Lagrange Multipliers
x = [-1.8 , 1.7 , 1.9 , -0.8 , -0.8]';
y = [ 1 , 1 , 1]';

%Initialization
maxit = 100;
tol = 10^(-8);
k = 0;
B = eye(5);

%Initial Computations
[f df] = ObjFun(x);
[c(1,1) dc(:,1)] = ConFun1(x);
[c(2,1) dc(:,2)] = ConFun2(x);
[c(3,1) dc(:,3)] = ConFun3(x);

%Stats
stat.alpha = [];
stat.k = k;
stat.x = x;
stat.y = y;
stat.df = df;
stat.B = B;
stat.theta = nan;
stat.time = nan;
stat.conv = [];

%Main Algorithm
Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
while ~Converged && k < maxit
    tic
    k = k + 1;

    %Solve BFGS Problem using LDL Factorization
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    KKT = [B , -dc ; -dc' , zeros(3,3)];
    RHS = -[df ; -c];

    [L,D,s] = ldl(KKT,'lower','vector');
    d(s) = L' \ ( D \ ( L \ RHS(s) ) ); d = reshape(d,8,1);

    p = d(1:5);
    y_new = d(6:8);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %New Step and New Function Evaluations
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    [x_new] = LineSearch2c(x,y,p,f,df,c);
    %x_new = x + p;

    [f_new df_new] = ObjFun(x_new);
    [c_new(1,1) dc_new(:,1)] = ConFun1(x_new);
    [c_new(2,1) dc_new(:,2)] = ConFun2(x_new);
    [c_new(3,1) dc_new(:,3)] = ConFun3(x_new);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %BFGS Update Step
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    q = ( df_new - dc_new*y_new ) - ( df - dc*y_new ) ;

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

    r = theta * q + (1-theta)*(B*p);

    B = B + (r * r') ./ ( p'*r ) - (B*p)*(B*p)' / ( p'*(B*p) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Rename Variables
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    f = f_new; df = df_new;
    c = c_new; dc = dc_new;
    y = y_new;
    x = x_new;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Convergence Check
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    Converged = ( norm(df - dc*y,'inf') <= tol && ( norm(c,'inf') <= tol ) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    dL = norm(df - dc*y,'inf');
    %Stats Update
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    stat.k = [stat.k k];
    stat.x = [stat.x x];
    stat.y = [stat.y y];
    stat.df = [stat.df df];
    stat.B = [stat.B B];
    stat.theta = [stat.theta theta];
    stat.time = [stat.time toc];
    %stat.alpha = [stat.alpha alpha];
    stat.conv = [stat.conv dL];
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
end

if Converged && k <= maxit
disp('A solution was found and is given by x = ')
disp(x)
disp('The stats are given by')
disp(stat)
end
```

Unfortunately, the line search implementation does not behave as nicely as the previous two. We do not see convergence to a high tolerance.

$$\left[\begin{array}{c|ccccccccc}
k & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
x_1 & -1.8 & -1.7269 & -1.7269 & -1.7269 & -1.7269 & -1.7269 & -1.5690 & -1.7177 & -1.7177 \\
x_2 & 1.7 & 1.6087 & 1.6087 & 1.6087 & 1.6087 & 1.6087 & 1.4250 & 1.6002 & 1.6002 \\
x_3 & 1.9 & 1.813 & 1.8132 & 1.8132 & 1.8132 & 1.8132 & 2.1047 & 1.8462 & 1.8462 \\
x_4 & -0.84 & -0.7636 & -0.7636 & -0.7636 & -0.7636 & -0.7636 & -0.7816 & -0.7746 & -0.7746 \\
x_5 & -0.8 & -0.7636 & -0.7636 & -0.7636 & -0.7636 & -0.7636 & -0.7816 & -0.7746 & -0.77458
\end{array}\right] \tag{34}$$

So yes, an unsuccessful implementation, which is never nice, but it does get to converge, when we have a very lenient tolerance of 10^-2. This would suggest that it overshoots the target in some way, due to an incorrect update of step length α. The implemented line search can be found in **Appendix B**

4

As we have tested for convergence with the error measure $\nabla f(x^k) - \nabla c(x^k)y = \nabla \mathcal{L}_k$, here abbreviated as dL. The result is plotted with a logarithmic y-axis in figure 5.

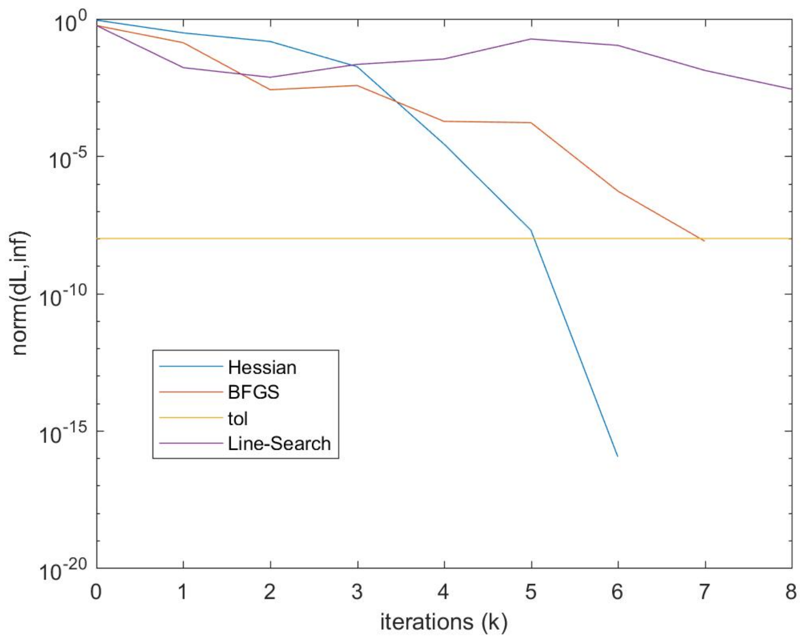


Figure 5: Convergence plot for the implementations of the 3 algorithms considered in this section. The vertical line is the value of the tolerance used. The measure of error used is the inf-norm of the qauntity dL

We see that the method using the full Hessian converges faster than the others. This is to be expected, as this method retains the most information about the nonlinear problem. A surprise, however, is that the BFGS performs worse with the added line search. This would suggest that our implementation is wrong. The BFGS performs slightly worse than when using the Hessian and this makes sense as we are discarding some information, when forming the approximate hessian matrix used in the BFGS.

Problem 3 - Inequality Constrained SQP

For this section we consider the following problem:

$$\begin{aligned}
\min_x \quad & f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 && \text{(35a)}\\
s.t \quad & c_1(x) = (x_1+2)^2 - x_2 \geq 0 && \text{(35b)}\\
& c_2(x) = -4x_1 + 10x_2 \geq 0 && \text{(35c)}
\end{aligned}$$

The gradients of both the objective function and all constraint functions will be necessary in the following algorithms. They are found to be

$$\nabla f(x_1, x_2) = \begin{bmatrix} 4x_1^3 + 2x_2^2 + 4x_1x_2 - 42x_1 - 14 \\ 4x_2^3 + 2x_1^2 + 4x_1x_2 - 26x_2 - 22 \end{bmatrix} \tag{36}$$

$$\nabla c(x_1, x_2) = \begin{bmatrix} \nabla c_1(x) & \nabla c_2(x) \end{bmatrix} = \begin{bmatrix} 2x_1 + 4 & -4 \\ -1 & 10 \end{bmatrix} \tag{37}$$

1

The SQP procedure for the inequality-constrained nonlinear problem given in (36) bear close resemblance to that provided in Problem 2 for the equality constrained SQP. The difference is that the local SQP method which yields the next iteration step is an inequality-constrained quadratic program. The solution can thus not be written in the closed form as it provided in (26) for the equality-constrained case. It has previously in the course been demonstrated that the KKT system associated with the quadratic program given by

$$\begin{aligned}
\min_x \quad & f(x) = \frac{1}{2}x^T Hx + g^T x + \gamma && \text{(38a)}\\
s.t \quad & A^T x - b \geq 0 && \text{(38b)}
\end{aligned}$$

can be expressed as

$$\begin{bmatrix} H & -A \\ -A^T & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{39}$$

Thus we infer that the system provided in (26) is equivalent to the KKT-system for a local quadratic program which is retrieved from a second order Taylor expansion of the objective function around some iterate including a first order expansion of the constraints. This program can then be written as the following:

$$\begin{aligned}
\min_x \quad & f(x) = \frac{1}{2}\left[\Delta x\right]^T \left[\nabla^2_{xx} L(x^k, y^k)\right] \Delta x + \left[\nabla_x L(x^k, y^k)\right] \Delta x && \text{(40a)}\\
s.t \quad & \left[\nabla_x c(x^k)\right]^T \Delta x + c(x^k) \geq 0 && \text{(40b)}
\end{aligned}$$

This problem is the essence of the inequality-constrained SQP. The solution is obtained through either the use of an Active-Set or an Interior-Point algorithm. The former was implemented in the first assignment of the course while the latter was implemented just as a linear-program solver in Problem 1 of this paper. While it was our intention to apply the self-developed Active Set algorithm its application required changes that was deemed too time consuming. For this reason the implementation provided in the following will be using the MATLAB function *quadprog*. This solver uses an Interior-Point method. The implementation is provided in Appendix C. In contrast to usual implementation the initial BFGS matrix B was set to 100 times the identity matrix as this was seen to provide a more stable convergence pattern. In most cases the algorithm would converge towards the global minimizer located at $(x_1, x_2) = (3, 2)$ however often it would appear to become stuck close to one of the local minimizers that coincides with the constrain-boundaries without converging properly. This issue could not be solved. The two cases are illustrated on Figure (6) and Figure (7) respectively. In the former case of convergence five different points and their iteration sequence have been provided in Table (41)

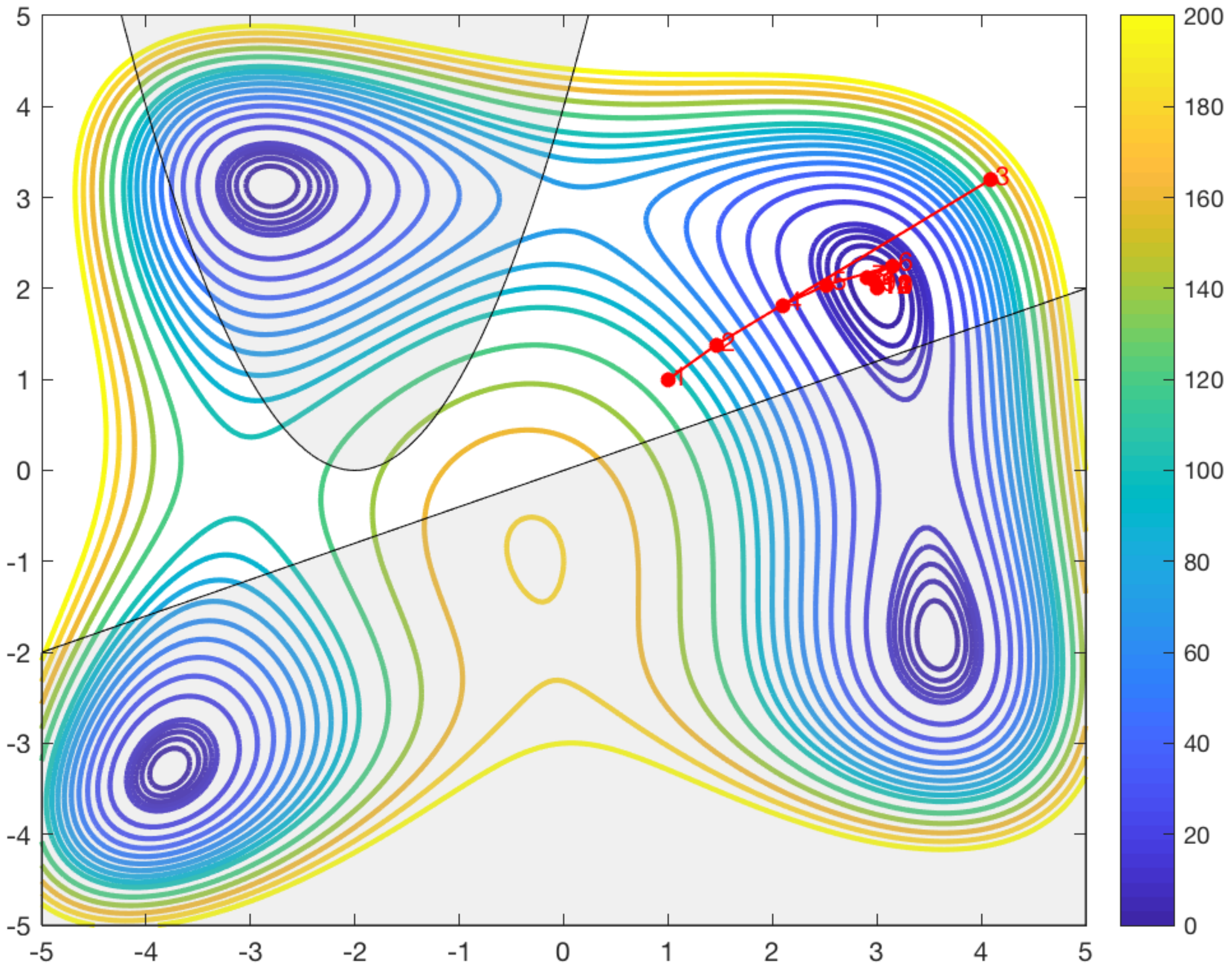


Figure 6: Convergence towards the global minimizer $(x_1, x_2) = (3, 2)$ from the initial starting point $x_0 = (1, 1)$. The convergence is achieved in 14 iterations. Similar convergence towards this point was seen for the starting points $x_0 = (0, 0)$, $x_0 = (3, -2)$ and $x_0 = (-0.5, 2)$.

x_0	k	1	3	5	7	9	11	13	15	17
(1,1)	x_1	1	4.0849	2.5186	2.9023	2.998	3.0004	3	–	–
	x_2	1	3.194	2.0405	2.122	2.0382	2.0002	2	–	–
(0,0)	x_1	0	1.1328	3.0326	2.9887	2.9469	2.994	3.000	3	–
	x_2	0	1.6120	3.3638	2.6621	2.1143	2.0012	1.9996	2	–
(3,-2)	x_1	3	3.1436	3.0241	3.0000	3	–	–	–	–
	x_2	-2	1.8010	1.9734	2.0001	2	–	–	–	–
(-0.5,2)	x_1	-0.5	0.6656	0.7066	2.0568	2.2385	2.9249	3.0092	3.0006	3
	x_2	2	3.1187	2.9735	3.1733	2.6958	1.9205	1.9253	1.9997	2

(41)

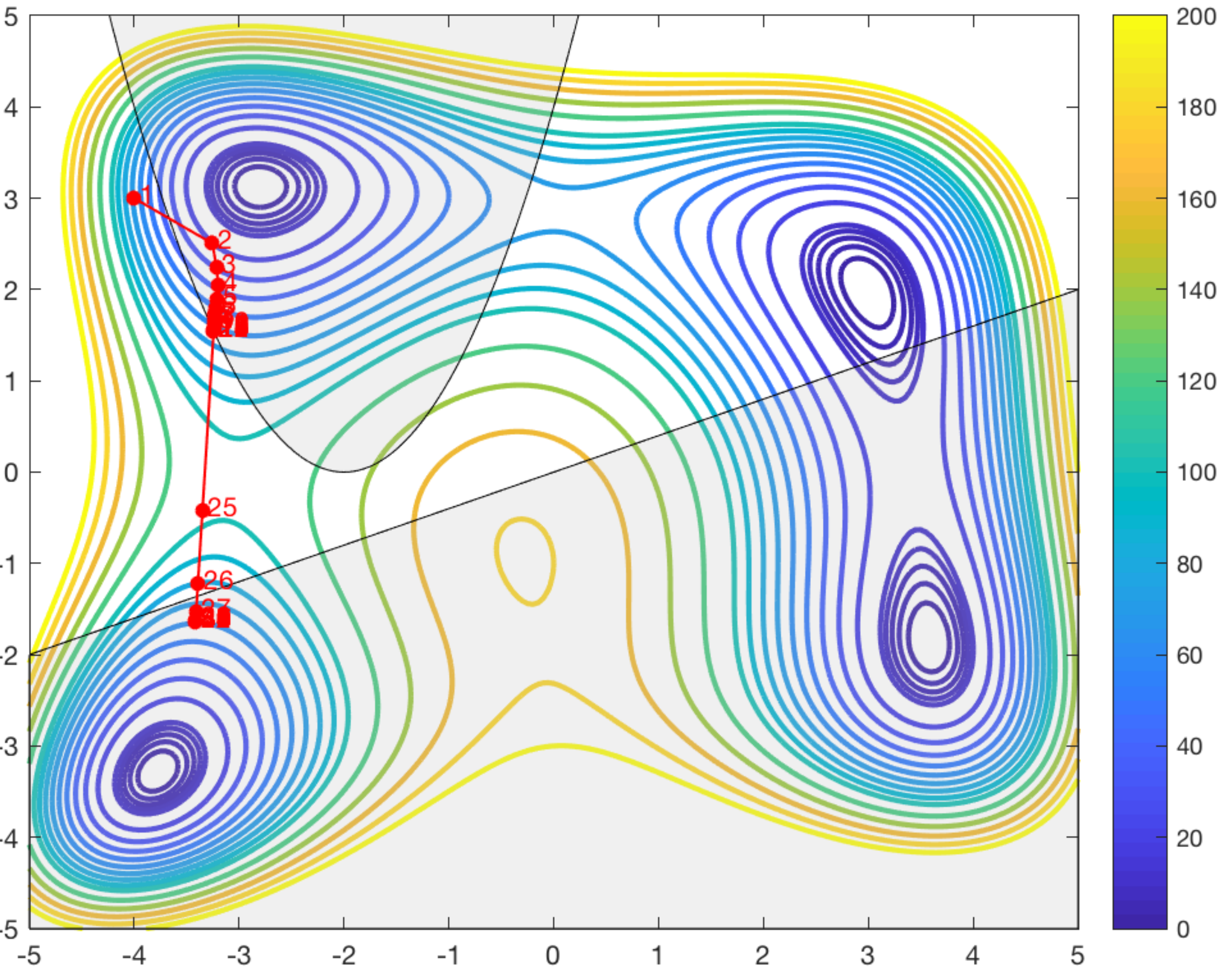


Figure 7: Contour plot illustrating issues with certain starting points. In this case it is $(x_1, x_2) = (-4, 3)$ which behaves oddly. It is seen to initially converge towards the local minimizer that is located at the upper left side of the first constraint, while it then decides to try to converge towards the local minimizer at the second constraint.

It is clear from inspection of Figure (7) below that the algorithm behaves odd for some particular initial values. Both $(x_1, x_2) = (-4, 3)$ and $(x_1, x_2) = (-2, 3)$ behaved as seen. While it is certainly close the local minimizers it refuses to converge towards them. Inspection of the value of the Lagrange at these points also reveal though that there should no be reason for it to equal zero. The seen behavior can not be theoretically explained at the time of writing.

2

The implementation of Line Search is very much identical to the algorithm used in Problem 2 (provided in Appendix B). The only noteworthy difference is the computation of the coefficients in the quadratic expansion around the initial point which changes to

$$\phi(0) = c = f(x^k) + \left[y^k\right]^T |\min(0, c(x^k)| \tag{42}$$

$$\phi'(0) = \left[\nabla f(x^k)\right]^T \Delta x^k - \left[y^k\right]^T |\min(0, c(x^k)| \tag{43}$$

In general the addition of the Line Search algorithm on the developed SQP BFGS inequality-constrained algorithm proved helpful. While iteration sequences shall not be printed as they are very much identical to those provided in Table (41) it was seen that the number of iterations for the converging points presented dropped roughly two iterations. As can be seen from Figure (8) in the case of the non-converging points a less chaotic path is now iterated though the issues mentioned previously remain unsolved with this addition and the convergence is still off on these boundaries. Notice that sequence converges within the boundary zone. For the initial point in question which was given as $(x_1, x_2) = (-4, 3)$ the convergence is towards the point $x_0 = (-3.2550, 2.5050)$. It's remarkably close to the result obtained from the Trust-Region method as shown later in Figure (11) which seems to be right. The coordinates of this point are $x_{TR} = (-3.6546, 2.7377)$. It is evident that something is causing a slight perturbation in the problem or similar though no errors were found when investigated. We were unable to fix the problem and thus the results presented here are final despite the fact that they do not work optimally as we would except and desire.

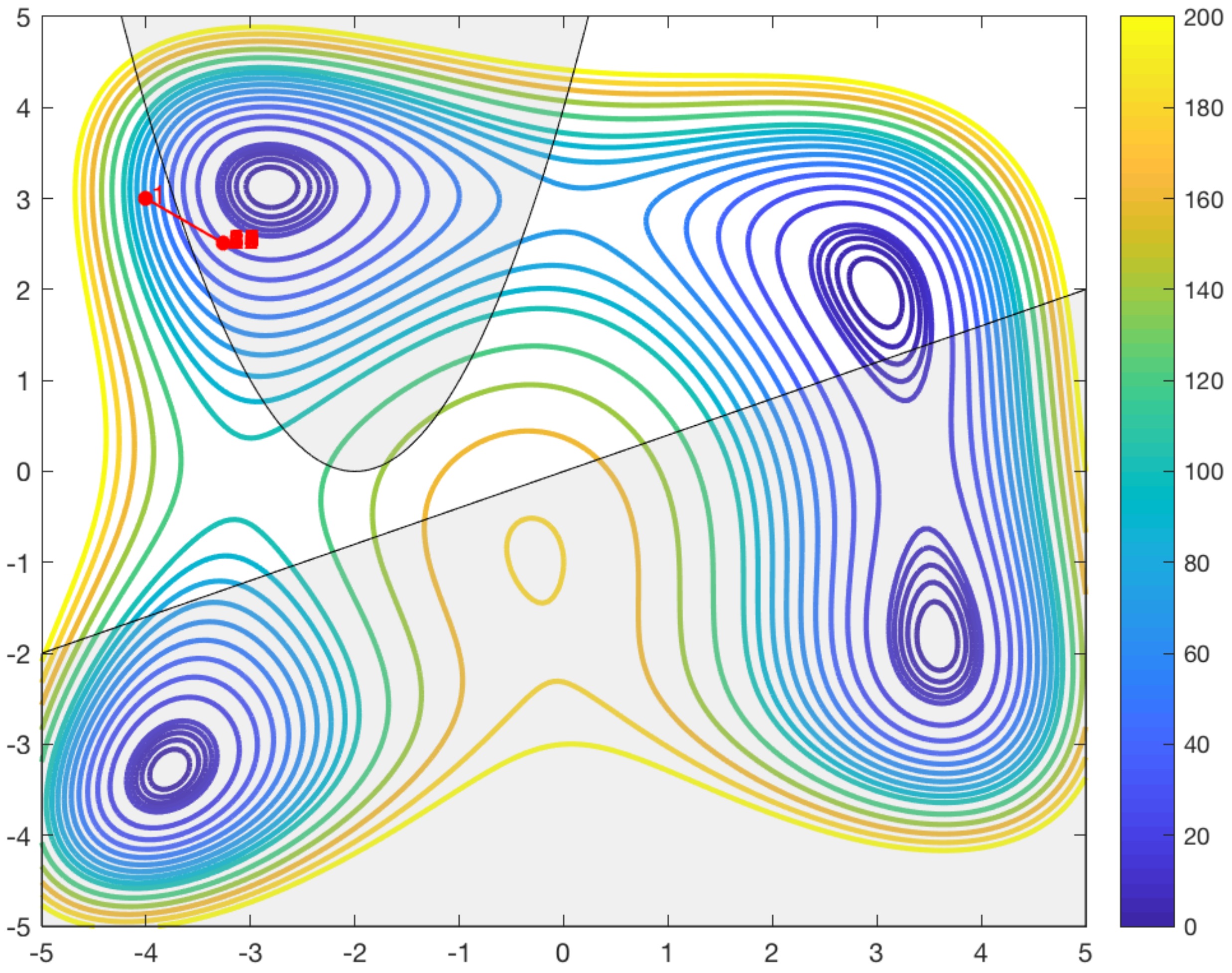
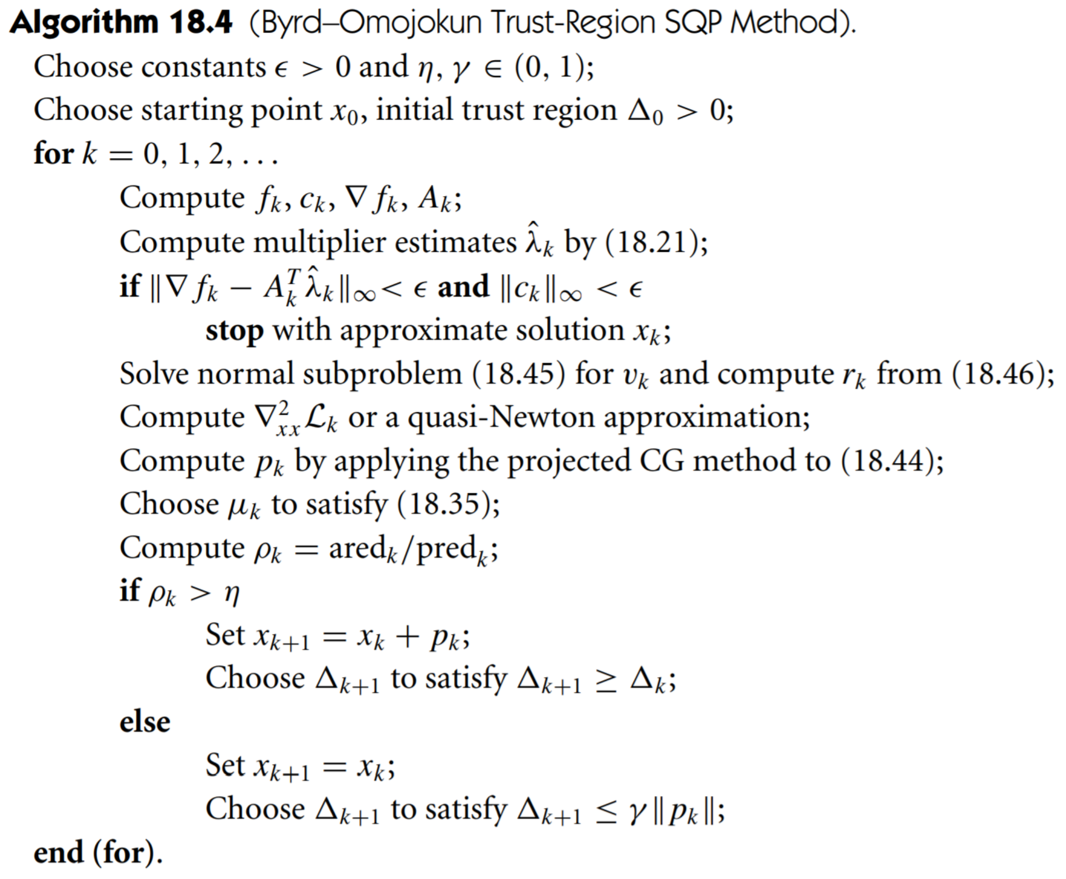


Figure 8: Contour for the initial point $(x_1, x_2) = (-4, 3)$ using a Line Search algorithm n top of the BFGS. Notice the difference between this iteration sequence and that displayed on Figure (7). The convergence is more stable however still behaves odd as it converges within the constrained region. The final point obtained in which the algorithm stays is given by $x_f = (-3.2550, 2.5050))$

3

For this section we will implement an SQP solver with a trust-region based approach. The algorithm follows the following outline with a few modifications:



Algorithm 18.4 (Byrd–Omojokun Trust-Region SQP Method).

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

Figure 9: Outline of an trust-region SQP solver

The key difference between this and the implemented algorithm, is that the subproblem that is solved, is actually subproblem (18.50) in [Nocedal and Wright, 2006] as Algorithm 18.4 originally solves equality-imposed problems only. By solving (18.50) we're actually computing the step p_k. An in-depth explanation of the modified algorithm goes as follows:
We first chose a tolerance $\epsilon > 0$ and $\eta, \gamma \in (0, 1)$. These constants has been chosen to be:

$$\epsilon = 10^{-6} \quad \eta = 1/4 \quad \gamma = 1/2$$

These parameter choices are purely arbitrarily and has therefore no analytical justification. We then initiate an inital trust region Δ_0 and the starting point x_0 is given as a function input. The iteration then starts by evaluating the objective function value f_k, gradient of the objective function ∇f_k, the constraints c_k and the Jacobian of the constraints at x_k. These are defined as follows for this problem:

$$f_k = \left[(x_1^2)^k + (x_2)^k - 11\right]^2 + \left[(x_1)^k + (x_2^2)^k - 7\right]^2 \qquad c_k = \begin{bmatrix} [(x_1)^k + 2]^2 - (x_2)^k \\ -4(x_1)^k + 10(x_2)^k \end{bmatrix} \tag{44a}$$

$$\nabla f_k = \begin{bmatrix} 4[(x_1^2)^k + (x_2)^k - 11](x_1)^k + 2(x_2^2)^k + 2(x_1)^k - 14 \\ 2(x_1^2)^k + 2(x_2)^k - 22 + 4\left[(x_2^2)^k + (x_1)^k - 7\right](x_2)^k \end{bmatrix} \qquad A_k = \left[\nabla c_1(x_k)^T \; \nabla c_2(x_k)^T\right] \tag{44b}$$

After the evaluation, we skip the estimation of the lagrange multipliers as this estimation will usually return lagrange multipliers that satisfies the convergence criterion, even if aren't at an optimum. This means that we also need an inital guess to the lagrange multipliers as the algorithm isn't using a quasi-Newton approximation for the hessian of the Lagrangian. We now compute the step by solving (18.50) which is defined as the following:

$$\begin{aligned}
\min_{p,v,w,t} \quad & f_k + \nabla f_k^T p + \frac{1}{2} p^T \nabla_{xx}^2 \mathcal{L} p + \mu \sum_{i \in \mathcal{E}} (v_i + w_i) + \mu \sum_{i \in \mathcal{I}} t_i && \text{(45a)} \\
\text{s.t} \quad & \nabla c_i(x_k)^T p + c_i(x_k) = v_i - w_i, \quad i \in \mathcal{E}, && \text{(45b)} \\
& \nabla c_i(x_k)^T p + c_i(x_k) \geq -t_i, \quad i \in \mathcal{I}, && \text{(45c)} \\
& v, w, t \geq 0, && \text{(45d)} \\
& \|p\|_\infty \leq \Delta_k && \text{(45e)}
\end{aligned}$$

However, since we have no equality conditions, the problem can be reduced into the following:

$$\begin{aligned}
\min_{p,t} \quad & f_k + \nabla f_k^T p + \frac{1}{2} p^T \nabla_{xx}^2 \mathcal{L} p + \mu \sum_{i \in \mathcal{I}} t_i && \text{(46a)} \\
& \nabla c_i(x_k)^T p + c_i(x_k) \geq -t_i, \quad i \in \mathcal{I}, && \text{(46b)} \\
& t \geq 0, && \text{(46c)} \\
& \|p\|_\infty \leq \Delta_k && \text{(46d)}
\end{aligned}$$

After computing the step by solving (46), we update μ_k to be equal to the right if it less than the right hand side of (18.36) which is:

$$\mu \geq \frac{\nabla f_k^T p_k + (\sigma/2) p_k^T \nabla_{xx}^2 \mathcal{L}_k p_k}{(1-\rho)\|c_k\|_1} \tag{47}$$

where $\sigma = 1$ if $\nabla_{xx}^2 \mathcal{L}_k$ is positive definite and 0 otherwise. If μ from the previous iteration satisfies (47), then μ is unchanged. Next up we compute the following ratio:

$$\rho_k = \frac{ared_k}{pred_k} = \frac{\phi_1(x_k;\mu) - \phi_1(x_k + p_k;\mu)}{q_\mu(0) - q_\mu(p_k)}$$

where $\phi_1(x;\mu)$ is the ℓ_1 merit function:

$$\phi_1(x;\mu) = f(x) + \mu \sum_{i \in \mathcal{E}} |c_i(x)| + \mu \sum_{i \in \mathcal{I}} [c_i(x)]^- \tag{48}$$

where the notation $y^- = \max\{0, -y\}$ and q_μ is defined as:

$$q_\mu(p) = f_k + \nabla f_k^T + \frac{1}{2} p^T \nabla_{xx}^2 \mathcal{L}_k p + \mu \sum_{i \in \mathcal{E}} |c_i(x_k) + \nabla c_i(x_k)^T p| + \mu \sum_{i \in \mathcal{I}} \left[c_i(x_k) + \nabla c_i(x_k)^T p\right]^- \tag{49}$$

We remind ourselves that this problem doesn't contain any equality conditions so we can reduce (48) and (49) to the following:

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

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

The ratio ρ_k is then used to whether we accept or reject the found step. This is checked by seeing if $\rho_k > \eta$ then:

$$x_{k+1} = x_k + p_k$$
$$\text{Choose } \Delta_{k+1} \text{ to satisfy } \Delta_{k+1} \geq \Delta_k.$$

If it is the other case, that $\rho_k <= \eta$, then the following holds:

$$x_{k+1} = x_k$$
$$\text{Choose } \Delta_{k+1} \text{ to satisfy } \Delta_{k+1} \leq \gamma \|p_k\|$$

An actual implementation of this algorithm can be seen in Appendix D.
We tested the code for several starting points and could conclude that it didn't converge for all of them. This can be due to multiple factors, where the computation of the new μ can be one of them. In algorithm 18.5 in [Nocedal and Wright, 2006], they demonstrate a rather effective method to update μ. However, the method we chose to update μ with is also valid and a bit more simple. One of the points where the algorithm converged was $x_0 = [1.2, 4]^T$. The iteration sequence can be seen on Figure 10:

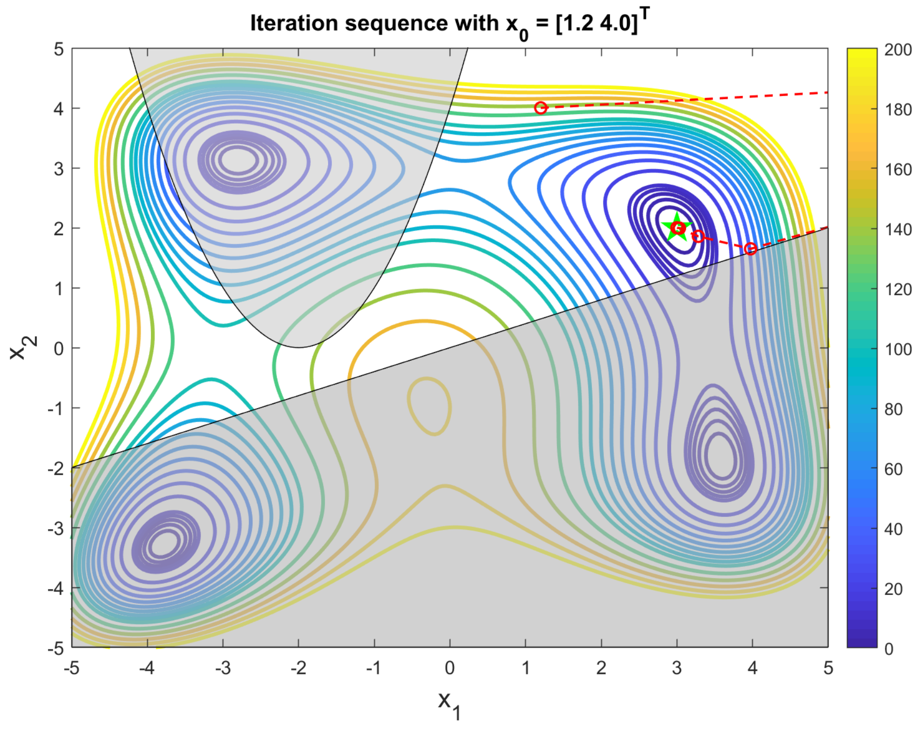


Figure 10: Contour plot of iteration sequence with $x_0 = [1.2, 4]^T$

On Figure 10 we see how we reach the global optimum which is $x^* = [3,2]^T$. The green star indicates x^*. On the following table we're able to see the different steps the algorithm decided on and the resulting x values:

k-th Iteration	1	2	3	4	5	6	7	8	9	10
x_1	1.2	11.129	7.5069	5.2779	3.9736	3.283	3.0236	3.006	3	3
x_2	4	4.6721	3.0028	2.1112	1.6486	1.8569	1.9928	1.9997	2	2

Table 1: Table of the iteration sequence for $x_0 = [1.2, 4]^T$

We see that it took 10 iterations for the algorithm to converge. Another point that converged was when $x_0 = [0,0]^T$. This resulted in the following iteration sequence:

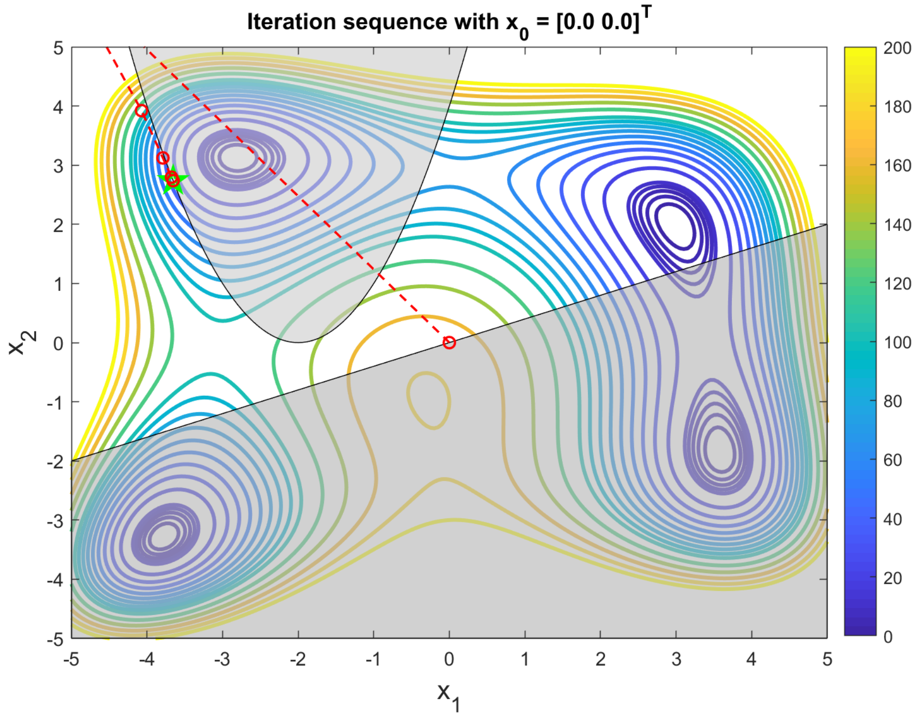


Figure 11: Contour plot of iteration sequence with $x_0 = [0,0]^T$

And the table of the iteration sequence points:

k	1	2	3	4	5	6	7	8	9	10
x_1	0	-9	-6.0444	-4.683	-4.0723	-3.7894	-3.6762	-3.6553	-3.6546	-3.6546
x_2	0	11.143	7.6218	5.3451	3.9213	3.1218	2.7968	2.7394	2.7377	2.7377

Table 2: Table of the iteration sequence for $x_0 = [0,0]^T$

Appendix A - Primal-dual interior point for LPs

```
function [x,lambda,s] = PredictorCorrectorV2(g,A,b)


[m,n] = size(A);

eta = 0.995;
k = 0;
kmax = 100;
tolL = 1.0e-9;
tolA = 1.0e-9;
tols = 1.0e-9;

%Calculate x_0, lambda_0 and s_0 as described on page 410 N&W
len = length(A*A');
invA = (A*A')\eye(len);
x_tilde = A'*invA*b;
lambda_tilde = invA*A*g;
s_tilde = g - A'*lambda_tilde;

delta_x = max(-3/2,min(min(x_tilde,0)));
delta_s = max(-3/2,min(min(s_tilde,0)));

x_tilde = x_tilde + delta_x*ones(length(x_tilde),1);
s_tilde = s_tilde + delta_s*ones(length(s_tilde),1);

delta_x2 = 0.5 * (x_tilde'*s_tilde)/(ones(1,length(s_tilde))*s_tilde);
delta_s2 = 0.5 * (x_tilde'*s_tilde)/(ones(1,length(x_tilde))*x_tilde);

x = x_tilde + delta_x2*ones(length(x_tilde),1);
s = s_tilde + delta_s2*ones(length(s_tilde),1);
lambda = lambda_tilde;


%Check residuals
r_c = -(A'*lambda + s - g); %Dual constraints
r_b = A*x - b;              %Primal constraints
XSe = x.*s;                 %Complementarity conditions
mu  = sum(XSe)/n;           %Duality measure

% Check if initial guess is optimal Converged
Converged = (norm(r_c,inf) <= tolL) && ...
            (norm(r_b,inf) <= tolA) && ...
            (abs(mu) <= tols);

```

```matlab


while(k < kmax && ~Converged)

    k = k + 1;

    %%%%   AFFINE STEP %%%%%%%
    %Define left hand side for computing affine direction as in Lecture 8
    lhs_affine = [zeros(n), A', eye(n); A, zeros(m,m), zeros(m,n);...
                  diag(s), zeros(n,m), diag(x)];

    %%% FACTORIZATION %%%%
    [L,U] = lu(lhs_affine);

    %Define right hand side for computing affine direction as in Lecture 8
    rhs_affine  = [-r_c; -r_b; -XSe];

    %Solve system for affine direction
    affine_direction = U\(L\rhs_affine);

    %Extract the different values for each parameter
    x_affine(:,1)      = affine_direction(1:n);
    lambda_affine(:,1) = affine_direction(n+1:end-n);
    s_affine(:,1)      = affine_direction(end-n+1:end);

    %%%%% STEP LENGTH %%%%%
    idx = x_affine < 0;
    alpha_primal_aff   = min([1; -x(idx,1)./x_affine(idx,1)]);

    idx = s_affine < 0;
    alpha_dual_aff     = min([1; -s(idx,1)./s_affine(idx,1)]);

    %%%% CENTERING PARAMETER %%%%%
    xAff_step = x + alpha_primal_aff * x_affine;
    sAff_step = s + alpha_dual_aff * s_affine;

    mu_Aff = (xAff_step' * sAff_step)/n;

    sigma = (mu_Aff/mu)^3;


    %%%% SEARCH DIRECTION (14.35) in N&W %%%%

    rhs = [-r_c; -r_b; (-XSe -(x_affine .* s_affine)) + sigma*mu];

    search_Direction = U\(L\rhs);
    dx(:,1)       = search_Direction(1:n);
    dLambda(:,1)  = search_Direction(n+1 : end-n);
    ds(:,1)       = search_Direction(end-n+1 : end);

    %%%%% TAKE THE FINAL STEP %%%%%%

    %%% Find step length as in (14.38) %%%
    idx = dx < 0;
    alpha_pri = min([1; eta * (-x(idx,1)./dx(idx,1))]);

    idx = ds < 0;
    alpha_dual = min([1; eta * (-s(idx,1)./ds(idx,1))]);

    %%% Compute the step %%%

    x = x + alpha_pri * dx;
    lambda = lambda + alpha_dual * dLambda;
    s = s + alpha_dual * ds;

    %%%% UPDATE RESIDUALS  %%%%%
    r_c = A'*lambda + s - g;
    r_b = A*x - b;
    XSe = x.*s;
    mu  = sum(XSe)/n;

% Converged
Converged = (norm(r_c,inf) <= tolL) && ...
            (norm(r_b,inf) <= tolA) && ...
            (abs(mu) <= tols);


end

end
```

Appendix B - Line search for Equality Constrained NLP

```
function [x_out,alpha] = LineSearch2c(x,y,p,f,df,c)
%Line Search algorithm for Equality Constrained SQP

%Initial Alpha Guess
alpha = 1;

%Quadratic Approximation Coefficients
```

```
b = df'*p - y'*abs(c);
c = f + y'*abs(c);

Converged = false;
while ~Converged
    %Attempted New Step
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    x_new = x + alpha*p;
    %%%%%%%%%%%%%%%%%%%%%%%%%%

    %Retrieve New Function Evaluations
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    [f_new] = ObjFun(x_new);
    [c_new(1,1)] = ConFun1(x_new);
    [c_new(2,1)] = ConFun2(x_new);
    [c_new(3,1)] = ConFun3(x_new);
    %%%%%%%%%%%%%%%%%%%%%%%%%%

    %Compute q
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    q = f_new + y'*abs(c_new);
    %%%%%%%%%%%%%%%%%%%%%%%%%%

    %Stop or Update Alpha
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    if q <= c + 10^(-1)*b*alpha
        x_out = x_new;
        Converged = true;
        break
    else
        a = ( q - (c + b*alpha) ) / alpha^2;
        alpha_min = -b/(2*a);

        alpha = min( 0.9*alpha , max( alpha_min,0.1*alpha) );
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%
end
end
```

Appendix C - Inequality Constrained SQP using BFGS Approximation

```
%% Problem 3a - Inequality Constrained SQP
clear variables; close all; clc
options = optimoptions('quadprog','Display','off');

%Starting Guess
x = [-4 3]';     %The Two Left Boundary Mins
%x = [1 1]';     %Converges to Global Min
%x = [0,0]';      %Converges to Global Min
%x = [3 -2]';     %%Converges to Global Min
%x = [-2 3]';     %The Two Left Boundary Mins
%x = [-0.5 2]';  %Converges to Global Min
y = [0 0]';

%Initialization
maxit = 50;
tol = 10^(-6);
k = 0;
B = 10^2*eye(2);

%Initial Computations
[f df] = ObjFun(x);
[c(1,1) dc(:,1)] = ConFun1(x);
[c(2,1) dc(:,2)] = ConFun2(x);
L = df - dc*y;

%Stats
stat.k = [];
stat.x = x;
stat.y = y;
stat.c = c;
stat.df = df;
stat.dc = dc;
stat.L = L;
stat.B = B;
stat.theta = [];
stat.time = [];

%Main Algorithm
Converged = ( norm(L,'inf') <= tol && ( min(c) >= 0 ) );
while ~Converged && k < maxit
    tic
    k = k + 1

    %Solve Inequality Quadratic Program for Step Direction using BFGS
    %Approx by Interior-Point Algorithm (quadprog)
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    [p,~,~,~,lambda] = quadprog(B,df,-dc,c,[],[],[],[],[],options);
    y_new = lambda.ineqlin;
    %%%%%%%%%%%%%%%%%%%%%%%%%%

    %New Step and New Function Evaluations
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    x_new = x + p;
    [f_new df_new] = ObjFun(x_new);
    [c_new(1,1) dc_new(:,1)] = ConFun1(x_new);
    [c_new(2,1) dc_new(:,2)] = ConFun2(x_new);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %BFGS Update Step
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    q = ( df_new - dc_new*y_new ) - ( df - dc*y_new ) ;

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

    r = theta * q + (1-theta)*(B*p);

    B = B + (r * r') ./ ( p'*r ) - (B*p)*(B*p)' / ( p'*(B*p) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Rename Variables
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    f = f_new; df = df_new;
    c = c_new; dc = dc_new;
    y = y_new;
    x = x_new;
    L = df - dc*y;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Convergence Check
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    Converged = ( norm(L,'inf') <= tol && ( min(c) >= 0 ) );
    %%%%%%%%%%%%%%%%%%%%%%%%%%%

    %Stats Update
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    stat.k = [stat.k k];
    stat.x = [stat.x x];
    stat.y = [stat.y y];
    stat.c = [stat.c c];
    stat.df = [stat.df df];
    stat.dc = [stat.dc dc];
    stat.L = [stat.L L];
    stat.B = [stat.B B];
    stat.theta = [stat.theta theta];
    stat.time = [stat.time toc];
    %%%%%%%%%%%%%%%%%%%%%%%%%%
end

if Converged && k <= maxit
    disp('A solution was found and is given by x = ')
    disp(x)
    disp('The stats are given by')
    disp(stat)
elseif k >= maxit
    disp('The maximum nubmer of iterations has been reached and no solution was found')
end

%Contour Plot and Iteration Sequence
x = -5:0.05:5;
y = -5:0.05:5;
[X,Y] = meshgrid(x,y);
F = (X.^2+Y-11).^2 + (X + Y.^2 - 7).^2;
v = [0:2:10 10:10:100 100:20:200];
[c,h] = contour(X,Y,F,v,'linewidth',2);
colorbar
yc1 = (x+2).^2;
yc2 = (4*x)/10;
hold on
fill([x(yc1<=5.1)],[yc1(yc1<=5.1)],[0.7 0.7 0.7],'facealpha',0.2)
fill([x x(end) x(1)],[yc2 -5 -5],[0.7 0.7 0.7],'facealpha',0.2)
xlim([-5 5])
ylim([-5 5])

for i = 1:k
    plot(stat.x(1,i),stat.x(2,i),'o','MarkerSize',5,'MarkerEdgeColor','Red','MarkerFaceCol
    plot(stat.x(1,i:i+1),stat.x(2,i:i+1),'-','Linewidth',1,'Color','Red')
    Numb = sprintf('%d',stat.k(i));
    text(stat.x(1,i)+0.05,stat.x(2,i)+0.05,Numb,'Color','Red','FontSize',10)
end
```

Appendix D - Inequality Constrained SQP using Trust-Region

```
function [x_opt,lambda_opt,info] = SQP_TrustRegionv2(fun,nonLinCons,x)


tol = 1e-6;
eta = 1/4;
gamma = 1/2;
mu = 100;


Trust_reg = 10;

maxIter = 100;
k = 0;
Converged = 0;
lam(:,1) = [1;1];
x(:,1) = x;


while(k < maxIter && ~Converged)
    k = k + 1;



    %%% COMPUTE f, c, GRADIENT OF f %%%

    [f_k,g_k,H_k] = fun(x(:,k));
    [c,dc,d2c] = nonLinCons(x(:,k));
    df_k = g_k;


    %%% SET UP THE JACOBIAN %%%

    A = dc';

    %%% CHECK FOR CONVERGENCE %%%
    if(norm(g_k - A'*lam(:,k),'inf') < tol)
        Converged = 1;
        x_opt = x(:,k);
        lambda_opt = lam(:,k);
        info.x = x;
        info.Iter = k;
        info.lamda = lam;
        info.Steps = p(1:2,:);
    end


    %%% SOLVE SUBPROBLEM 18.50 IN N&W %%%
    A_ineq = [A, eye(2)];
    g_k(end+1:end+2) = mu;

    H_k = H_k - lam(1,k)*d2c(:,:,1) - lam(2,k)*d2c(:,:,2);
    xx_L = H_k;
    H_k(end+1:end+2,end+1:end+2) = 0;

    lb = [-Trust_reg, -Trust_reg, 0, 0];
    ub = [Trust_reg,Trust_reg, Inf, Inf];

    opt = optimoptions('quadprog','Display','off');
    [p(:,k),~,~,~,lambda] = quadprog(H_k,g_k,-A_ineq,c,[],[],lb,ub,[],opt);
    lam(:,k+1) = lambda.ineqlin;


    %%% COMPUTE MU %%%

    if(p(:,k)' * H_k * p(:,k) > 0)
        sigma = 1;
    else
        sigma = 0;
    end


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

    %Define merit functions from (18.51)
    phi_1 = f_k + mu * sum(max(0,-c));
    phi_12 = fun(x(:,k)+p(1:2,k)) + mu * sum(max(0,-nonLinCons(x(:,k)+p(1:2,k)) ) );

    %Define q_mu as in (18.49)
    q_mu = f_k + df_k'*p(1:2,k) + 0.5* (p(1:2,k)' * xx_L * p(1:2,k) ) + mu * sum(max(0,-(c
    q_0 = f_k + mu * sum(max(0,-nonLinCons([0;0])) );

    %Calculate rho_k
    rho_k = (phi_1 - phi_12) / (q_0 - q_mu);
```

```matlab


    %%% CHECK IF WE INCREASE REGION OR NOT %%%
    if(rho_k < eta)
        x(:,k+1) = x(:,k) + p(1:2,k);
        Trust_reg = Trust_reg+10;
    else
        x(:,k+1) = x(:,k);
        Trust_reg = gamma * norm(p(1:2,k));
    end
end
```

References

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer.