Technical University of Denmark

02612 Constrained Optimization 2017

Assignment 2

Authors:
Miguel Suau de Castro (s161333)
Ramiro Mata (s161601)
Michal Baumgartner (s161636)

May 9, 2017

Contents

1 Interior-Point algorithm for LP

As this problem is closely related to the last question from the first assignment, the choice from using the interior-point algorithm with primal-dual framework and the predictor-corrector modification comes naturally. Dealing with LP is less complicated implementation and computation-wise (inverting matrices, LDL vs Cholesky factorization, ...). Main resources for implementation are John's slides (Lecture 7) as well as chapter 14.(2) from Nocedal and Wright.

Pseudo code will be listed initially along with Matlab implementation that is commented in the source code and split into several smaller methods to keep the code DRY and easier to read. To show that the implementation works random LP problem are generated following the description in 14.15 from page 419-420 in N&W. To make sure the results are reproducible for the read a random seed of `12345` is passed to Matlab's `rng` function with a flag to Mersenne twister as PRNG.

Pseudo code

Require: g, A, b and x, λ, μ
1: *Optional:* Calculate the initial point
2: Compute the residuals $\mathbf{r}_{(\cdot)}$
3: Stopped $\leftarrow$ Perform convergence condition check
4: **while** $\neg$ Stopped **do**
5: Compute H and factorize using Cholesky factorization
6: Affine direction
7: Duality gap and the centering parameter
8: Affine-centering-correction direction
9: Update the iteration
10: Stopped $\leftarrow$ Perform convergence condition check
11: **end while**
12: **return** x^*

The aforementioned random LP problems are generated as follows:

1. Randomly choose matrix A of $m \times n$ elements ($m < n$)
2. Choose 1 to m random elements for x, keep the rest until n as zeros
3. Apply the same procedure as above for λ but swap the indexes (there is no intersection between the zeros).

4. Set μ to random vector
5. Compute $g = A'\mu + \lambda$ and $b = Ax$
6. Generate a "starting point" (x, μ, λ) where each element is sufficiently large positive number

n was chosen to be 1000 and $m = 100$. After generating the random problem the implemented LP solver is ran and the error is checked.

The algorithm has 2 main parts: the procedure to determine the step and a measure of the desirability of each point in the search space (N&W 14.1). The primal dual framework uses a Newton-like step towards the point $x_i\lambda_i = \sigma s$ where s is the duality measure and $\sigma \in [0, 1]$ is called the centering parameter (note that the book and the slides don't use the same notation hence the referred terms in this section correspond to the notation used in slides). This gives rise to equation 19 in the Lecture 7 slides or 14.9 in N&W. When $\sigma > 0$ it is possible to take a longer step before violating the bounds given by $(x, \lambda) \geq 0$.

As discussed in the previous assignment the corrector steps (Mehrotra's modifications) is used to deal with the linearizion errors that the affine-scaling causes (see 14.3 or 31-34 in the slides). This approach helps the algorithm to follow a closer trajectory to the primal-dual solution set. One can then change the value of σ to account for the goodness/badness of the affine step (eqns 35-38). Core of the implementation comes from equations 31-45 which are straight forward in Matlab (see the commented code).

The algorithm will keep running until a certain condition is met, the implementation uses the simple stopping criteria given in 48 from the slides (with 2-norm – `norm` in Matlab). To speed up the algorithm the same approach from the last assignment can be used as was hinted during the lectures, that is perform the Cholesky factorization of the Hessian and save it to a variable as it is needed more than once and is computationally expensive.

Matlab implementation is provided in the next subsection along with comments throughout the code that mostly refer to the equations used. Also as mentioned many times before, the logic is very similar to the QP implementation from the previous assignment. Further improvements can be made to the method signature to allow the end-user to pass in an options struct that would possibly contain the maximum number of iterations, tolerance and η to mimic the conventions used in Matlab's Optimization Toolbox.

Matlab implementation

```
function [x,Converged,mu,lambda,iter,tol] = LPSolver(g,A,b,x,
    lambda_init,mu_init)
% Initialize
n = length(A);
lambda = lambda_init;
mu = mu_init;
tol = 1e-9;
tolL = tol;
tolA = tol;
tols = tol;
eta = 0.99;
maxiter = 100;
iter = 0;
% Compute residuals
[rL, rA, rC, s] = computeResiduals(g,A,b,x,mu,lambda,n);
% Convergence check
Converged = checkConvergence(rL,rA,s,tolL,tolA,tols);

while ~Converged && (iter < maxiter)
    iter = iter+1;
    % Hessian (save factorization for speed)
    xdivlambda = x./lambda;
    H = A*diag(xdivlambda)*A';
    L = chol(H,'lower');
    % Affine step -> solve
    [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,rC);
    % Affine step -> step length
    [alpha, beta] = stepLength(x,dx,lambda,dlambda);
    % Center parameter
    xAff = x + alpha*dx;
    lambdaAff = lambda + beta*dlambda;
    sAff = sum(xAff.*lambdaAff)/n;      % duality gap for affine step
    sigma = (sAff/s)^3;                 % centering parameter
    % Center+Corrector step to follow tractory to the primal dual
       solution set
    rC = rC + dx.*dlambda - sigma*s;
    [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,rC);
    % Step length
    [alpha, beta] = stepLength(x,dx,lambda,dlambda);
    % Take step - update from (45)
    x = x + eta*alpha*dx;
    mu = mu + eta*beta*dmu;
    lambda = lambda + eta*beta*dlambda;
    % Compute residuals
    [rL, rA, rC, s] = computeResiduals(g,A,b,x,mu,lambda,n);
    % Check if converged
    Converged = checkConvergence(rL,rA,s,tolL,tolA,tols);
```

```
46 end
47 % Signal the user that the algorithm didn't converge
48 if ~Converged
49     throw(MException('dtu:copt', 'Did not converge!'));
50 end
51
52 end
53
54 function [converged] = checkConvergence(rL,rA,s,tolL,tolA,tols)
55 % Stopping criteria from (48)
56 converged = (norm(rL) <= tolL) && (norm(rA) <= tolA) && (abs(s) <=
       tols);
57 end
58
59 function [rL,rA,rC,s] = computeResiduals(g,A,b,x,mu,lambda,n)
60 % Following eqns (46)
61 rL = g - A'*mu - lambda;
62 rA = A*x - b;
63 rC = x.*lambda;
64 s = sum(rC)/n;
65 end
66
67 function [alpha,beta] = stepLength(x,dx,lambda,dlambda)
68 % Find alpha_aff and beta_aff using (35) and (36)
69 idx_a = find(dx < 0);
70 alpha = min(1, min(-x(idx_a)./dx(idx_a)));
71 idx_b = find(dlambda < 0);
72 beta = min(1, min(-lambda(idx_b)./dlambda(idx_b)));
73 end
74
75 function [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,
       rC)
76 % Solve for dx, dmu, dlambda
77 rhs = -rA + A*((x.*rL + rC)./lambda);
78 dmu = L'\(L\rhs);
79 dx = xdivlambda.*(A'*dmu) - (x.*rL + rC)./lambda;
80 dlambda = -(rC+lambda.*dx)./x;
81 end
```

Results

The solver converges after 19 iterations and the difference between the generated x is around 4.5×10^{-8}, 1.85×10^{-10} for μ and finally 2×10^{-9} for λ while the solver's tolerance is set to be 10^{-9} and $\eta = 0.99$. Overall the results fulfill the expectations. Full code and test suite is included in the appendix.

2 Equality Constrained SQP

In this exercise we are asked to solve optimization problem 18.3 in Nocedal & Wright

$$\begin{aligned} \min \quad & e^{x_1 x_2 x_3 x_4 x_5} - \frac{1}{2}(x_1^3 + x_2^3 + 1)^2 \\ \text{subject to} \quad & x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 10 = 0 \\ & x_2 x_3 - 5 x_4 x_5 = 0 \\ & x_1^3 + x_2^3 + 1 \end{aligned} \tag{1}$$

The first thing one should notice when looking at both the objective function and the constraints is that these are nonlinear equations and hence, the optimization techniques used for solving linear or quadratic optimization problems can not be directly applied in this case.

However, the problem can be solved iteratively, by using Newton's method to obtain a point that satisfies the KKT conditions

$$F(x, \lambda) = \begin{bmatrix} \nabla f(x) - A(x)^T \lambda \\ c(x) \end{bmatrix} = 0 \tag{2}$$

where

$$A(x)^T = \begin{bmatrix} \nabla c_1(x), & \nabla c_2(x), & \dots & \nabla c_m(x) \end{bmatrix} \tag{3}$$

then we define the Jacobian of $F(x, \lambda)$ as

$$J(x, \lambda) = \begin{bmatrix} \nabla^2_{xx} \mathcal{L}(x, \lambda) & -A(x)^T \\ A(x) & 0 \end{bmatrix} \tag{4}$$

which, starting from suitable initial guess x_0, can be used to update the value of x and find the roots of 2 after a few iterations

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \lambda_k \end{bmatrix} + \begin{bmatrix} p_k \\ p_\lambda \end{bmatrix} \tag{5}$$

Where p_k and p_λ represent the Newton's descent direction and are given by

$$\begin{bmatrix} \nabla^2_{xx}\mathcal{L}(x,\lambda) & -A(x)^T \\ A(x) & 0 \end{bmatrix} \begin{bmatrix} p_k \\ p_\lambda \end{bmatrix} = \begin{bmatrix} \nabla f(x) - A(x)^T\lambda \\ c(x) \end{bmatrix} \tag{6}$$

The iteration sequence will converge to a minimizer of the objective function as long as the constraints are independent, that is $A(x)$ must have full row rank, and $\nabla^2_{xx}\mathcal{L}(x,\lambda)$ is positive definite.

An analogous formulation of the method can be obtained by transforming equation 6 into a quadratic optimization program of the form

$$\begin{aligned} \min \quad & f_k + \nabla f_k^T p + \frac{1}{2} p^T \nabla^2_{xx}\mathcal{L}_k p \\ \text{subject to} \quad & A_k p + c_k = 0 \end{aligned} \tag{7}$$

and thus, a solution to 1 can be computed sequentially by solving this optimization problem in every iteration.

2.1 Implementation of the local SQP solver

The algorithm described above has been implemented in MATLAB and can be seen in the code snippet below. The code includes several comments describing the method step by step.

```
function [x,lambda,info] = local_SQP(objfun,const,x,lambda,tol,maxit)
% The following function solves the nonlinear optimization problem in
% exercise 18.3 in N&W.

% initial evaluation of the function and the constraints
[~,g,H] = objfun(x);
[c,A,d2c1,d2c2,d2c3] = const(x);

dxL = g-A'*lambda; % gradient of the Lagrangian
F = [dxL; c]; %% KKT conditions
k = 1;
n = length(x);
x(:,k) = x;
% While the KKT conditions are not satisfied and the number of
    iterations
% is less than the threshold
```

```
16 while ((k < maxit) && (norm(F,'inf') > tol))
17     % compute hessian of the Lagrangian
18     d2L = H - lambda(1)*d2c1 - lambda(2)*d2c2 - lambda(3)*d2c3;
19     % solve the system to get the Newton's descent direction
20     z = [d2L -A'; A zeros(3)]\[-dxL; -c];
21     % obtain next point in the iteration sequence
22     p = z(1:n);
23     x(:,k+1) = x(:,k) + p;
24     % update lambda value
25     lambda = lambda + z(n+1:end);
26     k = k+1;
27     % evaluate function and constraints on the new point
28     [~,g,H] = objfun(x(:,k));
29     [c,A,d2c1,d2c2,d2c3] = const(x(:,k));
30     dxL = g-A'*lambda; % gradient of the Lagrangian
31     % check KKT conditions
32     F = [dxL;
33         c];
34 end
35 info.iter = k; % return number of iterations
36 info.seq = x; % return iteration sequence
37 x = x(:,k); % return last point as solution
38 end
```

The algorithm consists of a while loop where at first the KKT conditions are evaluated on the initial guess. In case these are not satisfied a decrease direction is computed solving the linear system in 5 or the quadratic program in 7. Adding this direction to the current value the next point in the iteration sequence is obtained and the KKT conditions are evaluated again. Figure 1 shows a flow chart of the local SQP algorithm.

Besides that, the function takes as input two functions that allow to evaluate on a given point the objective function and the constraints, as well as their gradient and Hessian using their respective analytical expressions. The two functions are provided in the appendix.

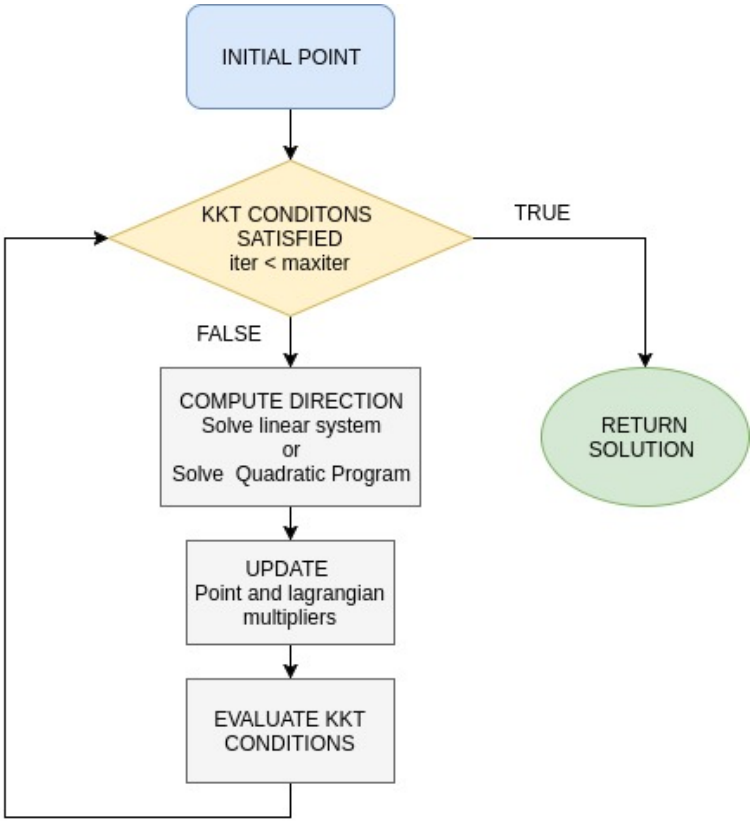

Figure 1: Local SQP algorithm flow chart

x_1	x_2	x_3	x_4	x_5
-1.8000	1.7000	1.9000	-0.8000	-0.8000
-1.8068	1.6983	1.6657	-0.7536	-0.7536
-1.7347	1.6166	1.7997	-0.7643	-0.7643
-1.7091	1.5865	1.8427	-0.7648	-0.7648
-1.7209	1.6001	1.8204	-0.7633	-0.7633
-1.7155	1.5938	1.8303	-0.7638	-0.7638
-1.7179	1.5965	1.8259	-0.7636	-0.7636

Table 1: Iteration sequence for Local SQP algorithm

We tested the algorithm in the given problem and we found out that it requires 7 iterations to converge to a point where the KKT conditions are below the tolerance. The total iteration sequence is shown in table 1.

2.2 Damped BFGS approximation to H

One of the main drawbacks of the local SQP algorithm is the large employ of computer resources, mainly derived from inverting the KKT matrix and computing the Hessian of the Lagrangian $\nabla^2_{xx}\mathcal{L}(x, \lambda)$. As we saw in the previous assignment, the former can be optimized by making use of the symmetry and sparsity properties of the KKT matrix.

On the other hand, there is a family of algorithms for unconstrained optimization known as Quasi-Newton methods that make use of approximations to the Hessian matrix that are faster to compute. It is important to notice that in our particular case, since we have the analytical expression for the Hessian of both the objective function and the constraints, we should not see any speed improvement.

We use the Damped BFGS method to approximate and update the Hessian of the Lagrangian. The Damped BFGS method accomplishes this by obtaining information from the gradients of the Lagrangian ($\nabla\mathcal{L}$). Specifically, assuming the Hessian is positive definite, from Taylor's approximation it can be shown that:

$$\nabla^2\mathcal{L}_{k+1}(x_{k+1} - x_k) \approx \nabla\mathcal{L}_{k+1} - \nabla\mathcal{L}_k \tag{8}$$

Which can be rewritten as

$$B_{k+1}s_k = y_k \tag{9}$$

Where B becomes our approximated $\nabla^2\mathcal{L}$, and where:

$$s_k = x_{k+1} - x_k \tag{10}$$
$$y_k = \nabla\mathcal{L}_{k+1} - \nabla\mathcal{L}_k \tag{11}$$

To overcome the difficulty of cases when the $\nabla^2\mathcal{L}$ has negative eigenvalues, we introduce a condition to handle such situations:

$$s_k^T y_k \geq \theta s_k^T B_k s_k \tag{12}$$

where θ is a positive scalar that we define as follows:

$$\theta_k = \begin{cases} 1, & \text{if } s_k^T y_k \geq 0.2 s_k^T B_k s_k, \\ \frac{0.8 s_k^T B_k s_k}{s_k^T B_k s_k - s_k^T y_k}, & \text{if } s_k^T y_k < 0.2 s_k^T B_k s_k, \end{cases} \tag{13}$$

We introduce the variable r (which replaces the y in the non-damped BFGS algorithm)

$$r_k = \theta y_k + (1 - \theta_k) B_k s_k \tag{14}$$

and conclude the derivation of the damped BFGS updating of the Hessian of the Lagrangian as follows:

$$B_{k+1} = B_k - \frac{B_k s_k s_k' B_k}{s_k' B_k s_k} + \frac{r_k r_k'}{r_k' s_k} \tag{15}$$

In this manner we ensure that the approximations of $\nabla^2 \mathcal{L}$ in new iterates remain positive definite.

Consequently, the local SQP algorithm has been modified so that the Hessian is approximated using the damped BFGS method. A code snippet of the implementation is shown below along with some comments describing the process.

```
% damped BFGS updating
    s = x(:,k) - x(:,k-1);
    y = dxLn - dxL;
    dxL = dxLn;
     % select value of theta to ensure possitive definiteness
    if s'*y >= 0.2*s'*B*s  % curvature condition
        theta = 1; % regular BFGS update
    else
        theta = (0.8*s'*B*s)/(s'*B*s-s'*y); % damped BFGS update
    end
    r = theta*y+(1-theta)*B*s;
    % update B
    B = B - (B*(s*s')*B)/(s'*B*s) + r*r'/(s'*r);
```

As can be seen in the code in case the curvature condition is not satisfied the damped factor θ is selected so as to prevent B from not being positive definite.

The iteration sequence followed by the method when solving 1 with the same starting point $x_0 = (-1.8, 1.7, 1.9, -0.8, -0.8)^T$ can be found in table 2. Where it can be seen that the algorithm just needs 6 iterations to converge.

x_1	x_2	x_3	x_4	x_5
-1.8000	1.7000	1.9000	-0.8000	-0.8000
-1.7269	1.6087	1.8132	-0.7636	-0.7636
-1.7244	1.6041	1.8138	-0.7628	-0.7628
-1.7196	1.5985	1.8228	-0.7634	-0.7634
-1.7172	1.5957	1.8272	-0.7636	-0.7636
-1.7171	1.5957	1.8272	-0.7636	-0.7636

Table 2: Iteration sequence for BFGS approximation

Surprisingly, the number of iterations needed when the damped BFGS approximation is introduced is less than for the ordinary local SQP method. This might be due to the fact that the initial value given to B is the identity matrix which makes the search direction become the gradient or steepest descent direction. From the theory of unconstrained optimization we know that this can speed up the process at the initial steps, that is, when the sequence is still far from the minimizer. In fact, when looking at the values in both tables, it is easy to see that the method seems to move faster towards the solution, specially at the first iteration.

2.3 Damped BFGS approximation to H and line search

As a second modification we are going to look into how large should the step be in every iteration. One of the simplest methods than can help to decide on the step length is called backtracking. This algorithm determines the maximum step length that gives a substantial decrease along the given search direction. Backtracking, which is based in the Goldstein-Armijo condition, starts with a large step length and reduces it in each iteration while the following sufficient decrease condition is not satisfied

$$\phi(x_k + \alpha p_k, \mu_k) \leq \phi(x_k, \mu_k) + \eta \alpha_k D(\phi(x_k; \mu); p_k) \tag{16}$$

In contrast to the algorithms used in unconstrained optimization, the descent condition for nonlinear constrained problems is evaluated on the l_1 merit function $\phi(x_k, \mu_k)$

$$\phi(x; \mu) = f(x) + \mu ||c(x)||_1 \tag{17}$$

Besides, $D(\phi(x_k;\mu_k);p)$, which is the directional derivative of $\phi(x;\mu)$ in the direction of p_k, can be written as

$$D(\phi(x_k;\mu_k);p) = \nabla f_k^T p_k - \mu||c_k||_1 \tag{18}$$

The penalty parameter μ needs to be chosen large enough so that p_k is a descent direction for $\phi(x_k;\mu_k)$. It can be shown that this will be satisfied if

$$\mu \geq \frac{\nabla f_k^T p_k + (\sigma/2) p_k^T \nabla_{xx}^2 \mathcal{L}_k p_k}{(1-\rho)||c_k||_1} \tag{19}$$

with $\rho \in (0,1)$ and $\sigma = 1$ when the Hessian of the Lagrangian is positive definite.

The algorithm along with comments describing all the details is provided in the code snippet below.

```
% Backtraking line search method
    sigma = 1; % Hessian positive definite
    rho = 0.2; % rho in (0,1)
    % choose mu to ensure descent direction
    mu = (g'*p + sigma/2*p'*B*p)/((1-rho)*norm(c,1));
    alpha = 1; % full step
    tau = 0.8; % contraction factor
    % check sufficient decrease condition
    fn = objfun(x(:,k)+alpha*p);
    cn = const(x(:,k)+alpha*p);
    eta = 0.3; % restriction on descent condition
    while fn+mu*norm(cn,1) > f+mu*norm(c,1)+eta*alpha*(g'*p-mu*norm(c
        ,1))
        alpha = tau*alpha; % update alpha
        % evaluate on the new point
        fn = objfun(x(:,k)+alpha*p);
        cn = const(x(:,k)+alpha*p);
    end
    f = fn;
    alphaval(k) = alpha; % Store alfa value in every iteration
    % obtain next point in the iteration sequence
    x(:,k+1) = x(:,k) + alpha*p;
    % update lambda value
    lambda = lambda + alpha*plambda;
```

It is easy to see that the value of η specifies how restrictive the descent condition is. If the value is very large the condition will not be satisfied in most of the iterations, which can lead to an unnecessary waste of computing resources.

x_1	x_2	x_3	x_4	x_5	α
-1.8000	1.7000	1.9000	-0.8000	-0.8000	1.0000
-1.7269	1.6087	1.8132	-0.7636	-0.7636	1.0000
-1.7244	1.6041	1.8138	-0.7628	-0.7628	1.0000
-1.7196	1.5985	1.8228	-0.7634	-0.7634	0.8000
-1.7176	1.5963	1.8263	-0.7636	-0.7636	1.0000
-1.7171	1.5957	1.8272	-0.7636	-0.7636	1.0000
-1.7171	1.5957	1.8272	-0.7636	-0.7636	0.0000

Table 3: Iteration sequence for BFGS approximation and line search

x_1	x_2	x_3	x_4	x_5	α
-1.8000	1.7000	1.9000	-0.8000	-0.8000	0.8000
-1.8055	1.6987	1.7126	-0.7629	-0.7629	0.6400
-1.7178	1.5984	1.8517	-0.7713	-0.7713	1.0000
-1.7160	1.5944	1.8294	-0.7638	-0.7638	1.0000
-1.7177	1.5963	1.8263	-0.7636	-0.7636	0.0000

Table 4: Iteration sequence for local SQP and line search

On the other hand, the contraction factor τ determines how much the step length α is reduced every time the descent condition is not satisfied. This parameter should be kept high enough to prevent the algorithm from taking too short steps.

The results obtained when combining backtracking and the BFGS approximation to the Hessian of the Lagrangian and starting from $x_0 = (-1.8, 1.7, 1.9, -0.8, -0.8)^T$ are collected in table 3 the algorithm requires in this case 7 iterations to converge.

The step length remains unchanged except in the 4th iteration. This means that from the given initial point the full step is most of the times satisfying the sufficient decrease condition. Consequently, we tried increasing the value of η to see if the total number of iterations could be reduced. However putting more weight on the condition did not help and make the algorithm converge more slowly.

We also tried applying line search to the local SQP algorithm with exact Hessian. The results in table 4 reveal that this combination requires just 5 iterations to converge and thus, it happens to be the most efficient for this problem and the given initial point. It can be seen that the algorithm efficiently reduces the step length in the first two iterations looking for a position along the given direction where the sufficient decrease condition is satisfied.

2.4 Convergence rate of the implemented algorithms

We have plotted the convergence rate of the 4 combinations we implemented. Figure 2 shows the 2 norm of the distance between the current iterate and the true solution. Note that since the plot is presented in logarithmic scale and we calculated the error with respect to the solution returned by the solver, the last iteration has 0 error and thus, it is not shown in the graph.

As mentioned in the previous section the BFGS approximation uses the identity matrix as initial guess for B, this means that at the first iteration the step is taken along the steepest descent direction. As shown in the plot this happens to be particularly useful for this problem as the error gets substantially reduced specially at the first and the fifth iterations.

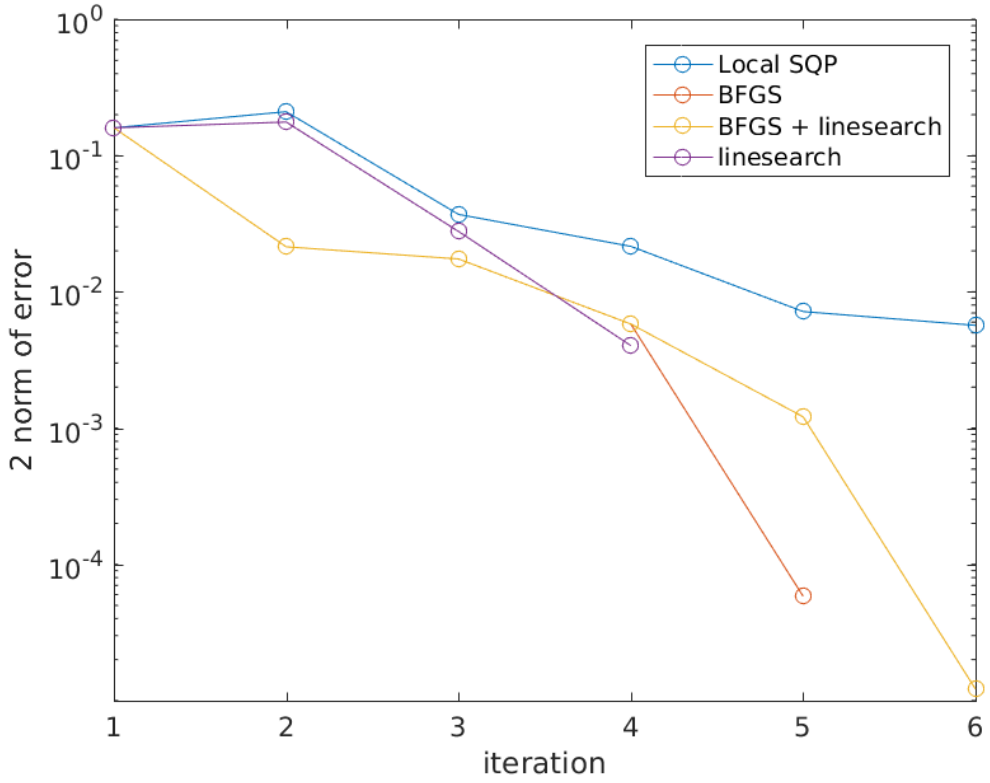

Figure 2: Convergence rate of the 4 different methods

On the other hand we experienced that combining the BFGS approximation and the line search did not give better results. The sufficient decrease condition was

always satisfied when the parameter η was chosen to be low, while the algorithm required more iterations for larger values of η. This last case is shown in the graph, in the fourth iteration the step length $alpha$ is different than 1.

Finally the plot reveals that combining local SQP and backtracking the algorithm can carefully select the step length at every point and find a point where the KKT conditions are below the tolerance in less iterations.

3 Inequality Constrained SQP

The problem in question uses Himmelblau's function which has 4 identical local minima with values of 0. This function is known for it's usage when it comes to testing optimization algorithms. Due to the given constraints the three local minima are not reachable, hence the minimum located at $x_1 = 3$, $x_2 = 2$ should be the output of the algorithms discussed further in this section.

$$\min_{x} \quad (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \tag{20a}$$

$$\text{s.t.} \quad c_1(x) = (x_1 + 2)^2 \geq 0, \tag{20b}$$

$$c_2(x) = -4x_1 + 10x_2 \geq 0 \tag{20c}$$

A figure demonstrating the Himmelblau's function depicted on a contour plot along with the constraints is given:

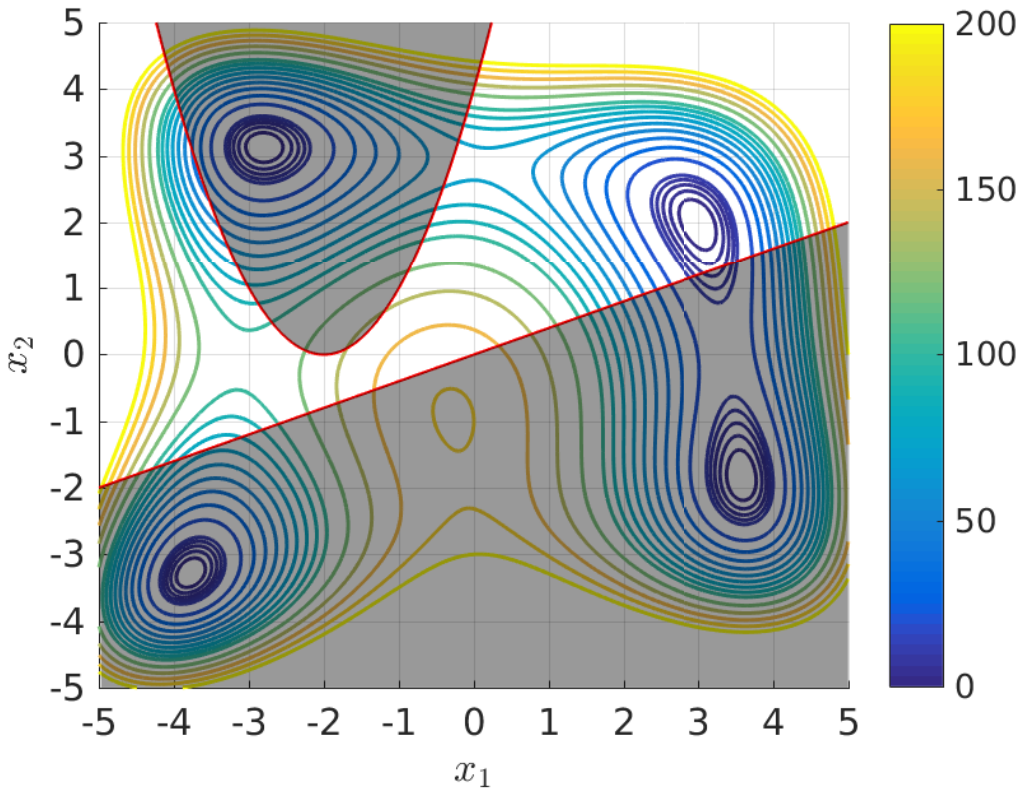

Figure 3: Contour plot of the objective function f and the constraints c_1, c_2. Levels of the contour lines are only shown for values $0 \leq f \leq 200$. Shaded areas indicate the space outside of the constraints. The code for this figure is located in `Ex3.m`

3.1 Damped BFGS approximation to H

The first inequality-constrained QP (IQP) algorithm we implement (which is described in this section) is the most basic of the three. Similar to the EQP algorithms in Section 2 they approach the difficulty of solving a nonlinear objective function (and nonlinear constraints) by using Newton's method to find a point that satisfies the KKT first-order conditions (see eq. 2). That is, instead of directly solving a highly nonlinear function, SQP methods solve a quadratic approximation of the problem at each iteration. This approximation is called the subQP problem. The solution to this subQP problem in IQP methods determines the next step and a guess of the optimal active set. By quadratically approximating a nonlinear objective function (and linearly approximating the constraints), SQP methods bypass highly nonlinear functions and instead solve quadratic functions at each iteration.

For simplicity we generalize the problem as follows.

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c_i(x) \geq 0, i \in I \end{aligned} \tag{21}$$

where I is the set of the inequality indices. Then we formulate the subQP problem by obtaining a quadratic approximation of the objective function and linearize the inequality constraints at the current iterate (x_k,λ_k) as follows.

$$\begin{aligned} \min_{p} \quad & f_k + \nabla f_k^T p + p^T \nabla_{xx}^2 \mathcal{L} p \\ \text{s.t.} \quad & A_k p + c_k \geq 0 \end{aligned} \tag{22}$$

Where p, the incremental, is the solution of the subQP problem; $\nabla_{xx}^2 \mathcal{L}$ is the Hessian of the Lagrangian; and A is the Jacobian of the constraints at the current iterate as shown below:

$$A(k)^T = \begin{bmatrix} \nabla c_1(k), & \nabla c_2(k) \end{bmatrix} \tag{23}$$

We then compute the next iteration step by adding the incrementals (the solutions of the previous subQP problem) to their respective x and λ parts as follows:

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \lambda_k \end{bmatrix} + \begin{bmatrix} p_k \\ p_\lambda \end{bmatrix} \tag{24}$$

Note that the Hessian of the Lagrangian ($\nabla^2_{xx}\mathcal{L}$) requires second order derivations of the objective function and the constraints, which can be computationally demanding. In our case, we have derived them analytically and are found in MATLAB functions *objfun()* and *constraints()*. These take as input any **x** and can output terms necessary to compute the Hessian of the Lagrangian. Nonetheless, an alternative and useful solution is to approximate it. We use the Damped BFGS described in section 2.2 to approximate and update the Hessian of the Lagrangian.

3.1.1 MATLAB Implementation

In our local IQP implementation we used quadprog to solve the subQP problem. Also, we initialized B as the identity matrix, which results in obtaining the steepest descent direction in the 1^{st} iteration before the B is updated. We set the variables for the local IQP as follows:

$$B_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad F(x,\lambda) = \begin{bmatrix} \nabla f(x) - A(x)^T\lambda \\ c(x) \end{bmatrix} \tag{25}$$

where the 1^{st} row above in $F(x,\lambda)$ is the gradient of the Lagrangian ($\nabla\mathcal{L}$). In *quadprog* notation, we implement as follows:

$$H = B \quad f = \nabla f(x) \quad A = -A(x) \quad b = c(x) \tag{26}$$

where $c(x)$ is a vector containing the result of both inequality constraints evaluated at **x**.

The implementation of the local IQP in MATLAB can be seen in the code snippet below. It includes comments referring to the steps of the algorithm (as explained here) and provides report's notation when applicable.

```
function [x,lambda,info] = ineqSQP(x,lambda,tol,maxit)

% Initializing info storage variables
subQPiter = zeros(1,maxit);
subQPfval = zeros(1,maxit);
```

```
k=1;
fcalls=1;
n=length(x);
x(:,k) = x; % storing solution iterates

% Initial evaluation of the problem and constraints
[~,g,~] = objfun(x);                    %gradient of original function
[cineq,dcineq,~,~] = constraint(x);  %constraints and its gradient

B = eye(2);                             %Hessian of L to be aproximated
F = [g - dcineq'*lambda; cineq];     %1st Order KKT conditions

QPoptions = optimoptions(@quadprog, 'Algorithm', 'interior-point-
    convex', 'MaxIterations', 200, 'Display', 'none');

% Checking max iterations and KKT conditions
while ((k < maxit) && (norm(F(1:2),'inf') > tol) )

    [~,g,~] = objfun(x(:,k)); fcalls=fcalls+1;
    [cineq,dcineq,~,~] = constraint(x(:,k)); %c(x) and A(x) in report
        notation

    % Defining variables
    gL = g - dcineq'*lambda;    %gradient of Lagrangian

    % Solving QP subproblem: to get Newton's descent direction
    [z, fval,~,output, lambda] = quadprog(B,g,-1*(dcineq), cineq, [],
        [], [], [], [], QPoptions);
    % Storing SubQP info
    subQPiter(k) = output.iterations;
    subQPfval(k) = fval;

    % Obtain next point in iteration sequence
    incremental = z(1:n); % p in report's notation
    x(:,k+1) = x(:,k) + incremental;

    % Update lambda
    lambda = lambda.ineqlin; %only when using quadprog

    % Evaluate objective function and constraints with new point, x(k
        +1)
    [~,g,~] = objfun(x(:,k+1)); fcalls=fcalls+1;
    [cineq, dcineq] = constraint(x(:,k+1));

    % Compute gradient of lagrangian with new x(k+1)
    gL_new = g - dcineq'*lambda;

    % Update Hessian matrix by damped BFGS procedure
    p = x(:,k+1) - x(:,k);
```

```
51      q = gL_new - gL;
52      B = update_hessian(B, p, q);
53
54      % Update KKT conditions
55      F = [gL_new; cineq];
56      k = k+1;
57
58
59  end
60  % storing convergence stats
61  info.iter    = k;
62  info.seq     = x;
63  info.fcalls = fcalls;
64  info.subQPiter = subQPiter;
65  info.subQPfval = subQPfval;
66  end
```

In essence, the algorithm comprises of a while loop that evaluates the first-order KKT conditions and other stopping criteria regarding maximum iterations and tolerance threshold. When not satisfied (i.e. no convergence), the while loop solves the subQP problem at every iteration using quadprog with the formulation presented in 25 and 26. This solves for the next point in its convergence path and stops until the stopping criteria are satisfied. To update the Hessian of the Lagrangian using the Damped BFGS procedure we use a separate function called *update_hessian.m* which is provided in the appendix.

3.1.2 Discussion

We tested our local IQP implementation on the himmelblau function from three different starting points. We chose the 1^{st} starting point to be near the solution ($x_1 = 1$,$x_2 = 3$); the 2^{nd} was placed such that it would be closer to a local minima than to the optimal solution ($x_1 = -5$,$x_2 = 5$); and the 3^{rd} was placed somewhat distant to the solution as we wanted to see how the algorithms behave when the initial estimate is relatively far from the optimal solution ($x_1 = 8$,$x_2 = 4$).

Figure 4 shows these three starting points along with the convergence paths. When starting from point 1 (in magenta) notice that even though it is very near the solution, the local IQP overshoots its first step moving it farther away from the optimal solution; the same thing occurs in the third iteration. This is potentially due because the local IQP does not have a merit function that can control the step size. However, as the new iterates get closer to the optimal solution, the quadratic approximation (subQP problem) becomes more representative of the

true objective function, and hence the step sizes become smaller as it converges. In this case, the local IQP converges to the optimal solution in 18 iterations. When initiated from starting point 2 (in red), local IQP fails to arrive at the solution and instead is stuck in a local minima. When initiated relatively far away from the solution (starting point 3 in blue) the local IQP behaves favorably and converges in 17 iterations.

Iteration sequence for $x_0 = (1, 3)$

	x_1	x_2
1	1.00	3.00
2	13.28	5.31
3	5.71	2.29
4	5.30	2.12
5	3.76	1.50
6	1.30	4.85
7	2.07	2.83
8	2.36	2.58
9	3.26	1.95
10	2.93	2.13
11	2.99	2.07
12	3.01	2.04
13	3.01	2.01
14	3.00	2.00
15	3.00	2.00
16	3.00	2.00
17	3.00	2.00
18	3.00	2.00

Iteration sequence for $x_0 = (-5, 5)$

	x_1	x_2
1	-5.00	5.00
2	-3.28	-1.31
3	-3.93	-1.57
4	-3.78	-1.51
5	-3.58	-1.43
6	-3.55	-1.42
7	-3.55	-1.42
8	-3.55	-1.42
9	-3.55	-1.42
10	-3.55	-1.42
11	-3.55	-1.42
12	-3.55	-1.42

Iteration sequence for $x_0 = (8, 4)$

	x_1
1	8.00
2	4.00
3	3.06
4	1.22
5	1.18
6	6.56
7	4.24
8	1.70
9	3.81
10	1.95
11	3.55
12	1.92
13	3.16
14	1.89
15	3.06
16	1.89
17	3.03

	$x_0 = (1, 3)$				$x_0 = (-5, 5)$				$x_0 = (8, 4)$			
Method	It	QPit	fcalls	t	It	QPit	fcalls	t	It	QPit	fcalls	t
BFGS	18	3.8	35	116	12	4.3	23	455	17	3.8	33	76
LS	11	3.5	48	53	10	2.8	183	37	16	3.8	217	67
TR	11	3.6	21	78	9	5.2	17	276	14	5.1	27	76

Table 5: Important statistics for the 3 algorithms employed: iterations, mean iterations in subQP problem (QPit), function calls (fcalls), and time (t) in milliseconds.

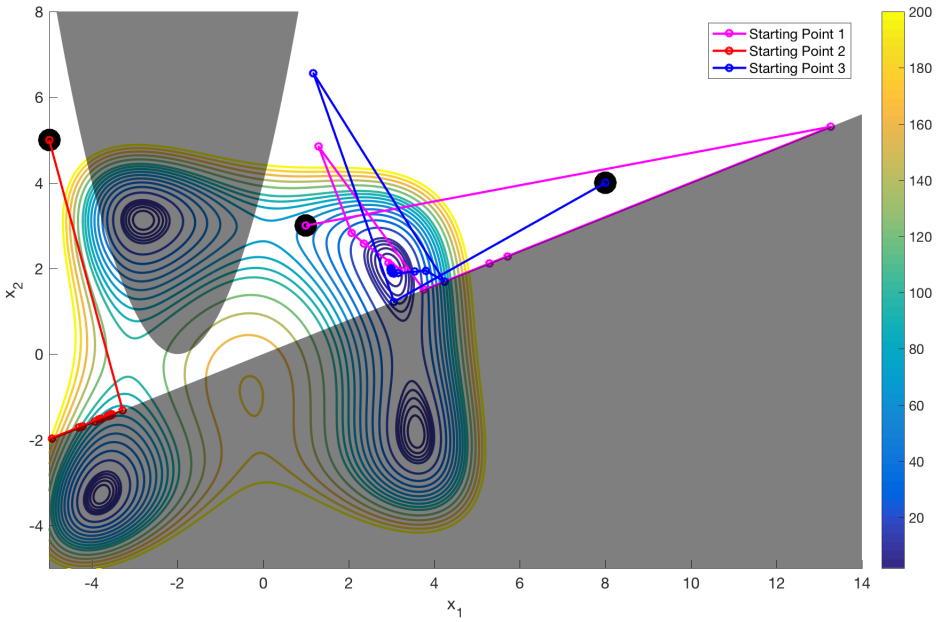

Figure 4: Contour plot showing the convergence steps of our local IQP implementation from three different starting points.

3.2 Damped BFGS approximation to H and line search

We extend the previous algorithm by adding a backtracking line search component. By adding this, we enforce that a step be taken only if a sufficient decrease condition in the step function is obtained (see eq. 16 in Section 2). This is equivalent to the Armijo condition for unconstrained optimization. Similar to the local IQP implementation, this algorithm also computes the next step using Newton's method and thereby solving a subQP problem at every iteration.

3.2.1 MATLAB Implementation

We use quadprog to solve the subQP problem where we set up the variables to accommodate quadprog's formulation exactly the same as we did for the local IQP algorithm and as described in equations 25 and 26. The code is exactly the same as in the local IQP, with the exception that we add this code block to incorporate the backtracking linesearch feature (for the full code of the algorithm, see the appendix):

```
% Backtraking line search method
    sigma = 1; % Hessian positive definite
    rho = 0.2; % rho in (0,1)
    mu = (g'*incremental +      sigma/2*incremental'*B*incremental)
       /((1-rho)*norm(cineq,1));
    alpha = 1; % full step
    tau = 0.8; % contraction factor

    % check sufficient decrease condition
    fn = objfun(x(:,k)+alpha*incremental); fcalls=fcalls+1;
    cn = constraint(x(:,k)+alpha*incremental);
    eta = 0.2;
    while fn+mu*norm(cn,1) > f+mu*norm(cineq,1)+eta*alpha*(g'*
       incremental-mu*norm(cineq,1))
        alpha = tau*alpha; % update alpha
        fn = objfun(x(:,k)+alpha*incremental); fcalls=fcalls+1;
        cn = constraint(x(:,k)+alpha*incremental);
    end
    f = fn;
```

As discussed in Section 2 (EQP), η modulates the descent condition; for high values, a suitable descent direction may not be found. τ, on the other hand, reduces α when the descent condition is not satisfied. We experimented with several values. We found that low values τ (0.1-0.3) would lead to smaller steps (as expected) and consequently would lead to more iterations till convergence. We found a τ of 0.8 to be more suitable.

3.2.2 Discussion

Unlike the local IQP, line search IQP can control the step size as described previously with τ and η, which allows it to converge in less iterations then the basic IQP algorithm. Figure 5 shows the convergence when a backtracking line search feature is added, and it is evident that this feature fixes the issue of large, "over-

shooting" step sizes as exhibited in the local IQP. From the first starting point (in magenta) it is able to converge in 11 iterations (relative to 18 in local IQP). Further, only after two iterations it is already in the vicinity of the optimal solution, so in this case its path is more direct then local IQP. From the 2^{nd} starting point, line search also converges to a local minima albeit a different one, highlighting that it too is vulnerable to local minimas. Finally, from the 3^{rd} starting point it converges as expected: more directly than local IQP albeit in only one iteration less than local IQP (see the table of metrics per method).

Iteration sequence for $x_0 = (1, 3)$

	x_1	x_2
1	1.00	3.00
2	3.57	3.48
3	3.20	2.02
4	2.82	2.23
5	2.98	2.07
6	3.00	2.02
7	3.00	2.00
8	3.00	2.00
9	3.00	2.00
10	3.00	2.00
11	3.00	2.00

Iteration sequence for $x_0 = (-5, 5)$

	x_1	x_2
1	-5.00	5.00
2	-5.00	5.00
3	-4.13	3.79
4	-3.83	3.27
5	-3.71	2.91
6	-3.67	2.78
7	-3.66	2.74
8	-3.65	2.74
9	-3.65	2.74
10	-3.65	2.74

Iteration sequence for $x_0 = (8, 4)$

	x_1	x_2
1	8.00	4.00
2	8.00	4.00
3	5.63	2.25
4	4.83	4.74
5	3.95	3.74
6	3.33	2.98
7	3.02	2.51
8	2.91	2.26
9	2.91	2.13
10	2.94	2.05
11	2.98	2.00
12	3.00	1.99
13	3.00	2.00
14	3.00	2.00
15	3.00	2.00
16	3.00	2.00

	$x_0 = (1, 3)$				$x_0 = (-5, 5)$				$x_0 = (8, 4)$			
Method	It	QPit	fcalls	t	It	QPit	fcalls	t	It	QPit	fcalls	t
BFGS	18	3.8	35	116	12	4.3	23	455	17	3.8	33	76
LS	11	3.5	48	53	10	2.8	183	37	16	3.8	217	67
TR	11	3.6	21	78	9	5.2	17	276	14	5.1	27	76

Table 6: Important statistics for the 3 algorithms employed: iterations, mean iterations in subQP problem (QPit), function calls (fcalls), and time (t) in milliseconds.

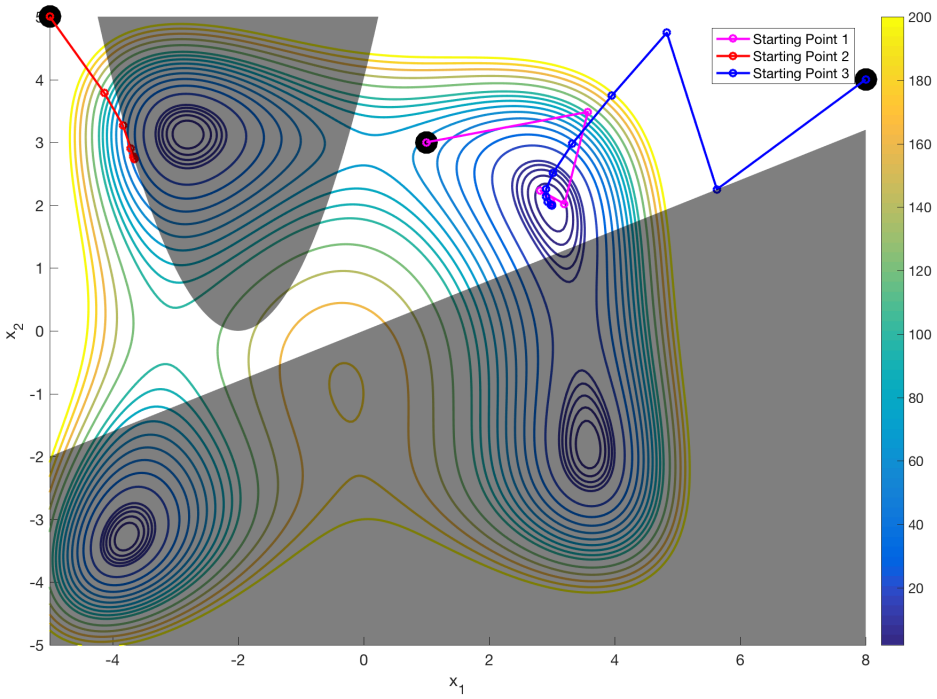

Figure 5: Contour plot showing the convergence steps of our backtracking line search IQP implementation from three different starting points.

3.3 Trust region based approach

We implement a S(L1)QP method as our trust region approach. This approach is similar to our local IQP method. We start with the same subQP problem (eq. 22. However,the S(L1)QP approach differs in that it incorporates the linearized constraints into the objective function in an L1 penalty manner, adds a trust-region constraint, and introduces a slack variable, t. After these modifications, the subQP problem can be formulated as shown in eq. 18.50 in Nocedal and Wright's Numerical Optimization (2nd edition,pg. 550). Note that the trust-region constraint added takes the following form:

$$\|\mathbf{p}\|\infty \leq \Delta_k \tag{27}$$

where p is the solution of the subQP problem and Δ_k is a user specified scalar that determines the radius of the trust region or Neighborhood where the next step is computed. In addition the penalty parameter μ is introduced to modulate the penalty effect during the minimization of the subQP problem. In particular, the μ parameter has to be chosen carefully as the trust region algorithm is sensitive to it. While updating Δ_k in each iteration can lead to improved convergence, in this implementation we choose to keep it fixed. Similarly, the μ parameter can be adjusted at each iteration such that the algorithm can adapt to a range of applications (i.e. objective functions). However, in our specific case we have it fixed.

3.3.1 MATLAB Implementation

Similar to the two previous algorithms, we use quadprog to solve the subQP problem. However, in this case the formulation does change a bit in order to accommodate the trust region constraint with the infinity norm and for the penalty parameter μ. We formulate as follows:

$$B_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \nabla f(x) = \begin{bmatrix} \nabla f(x) \\ \mu \\ \mu \end{bmatrix} \quad A = \begin{bmatrix} A & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} c = \begin{bmatrix} c \\ 0 \\ 0 \end{bmatrix} \tag{28}$$

where **0** is a square matrix of 4 zeros. We initiate our Δ_k and μ parameters as follows:

$$\Delta_k = 1 \quad \mu = 40 \tag{29}$$

Finally, we introduce Lower and Upper Bounds in quadprog to accommodate as mentioned previously for the infinity norm of the trust region constraint of Δ_k as well as for the slack variable ,t, whose constraint is as follows:

$$t_i \geq 0 \tag{30}$$

$$LB = \begin{bmatrix} -\Delta_k \\ -\Delta_k \\ 0 \\ 0 \end{bmatrix} \quad UB = \begin{bmatrix} \Delta_k \\ \Delta_k \\ inf \\ inf \end{bmatrix} \tag{31}$$

We show the code snippet below on what changes in the trustregion IQP implementation from the first IQP (local) that we described.

```
% Padding Hessian approx. with zeros to
% accommodate trust-region formulation
zero = [0 0; 0 0];
B = [eye(2) zero;
    zero   zero];
mu = 40; dk = 1;
```

```
 cineq=[cineq;0;0];
    dcineq=[dcineq zero;
            zero   zero];
    % Defining variables
    gL = g(1:2) - dcineq(1:2,1:2)'*lambda;    %gradient of Lagrangian

    % Solving QP subproblem
    LB = [-dk -dk 0 0]';
    UB = [dk dk inf inf]';
    [z, fval,~,output, lambda] = quadprog(B,g,-1*(dcineq), cineq, [],
        [], LB, UB, [], QPoptions);
    subQPiter(k) = output.iterations;
    subQPfval(k) = fval;

    incremental = z(1:n);
    x(:,k+1) = x(:,k) + incremental;
    t = z(n+1:end);
    lambda = lambda.ineqlin(1:2); %only when using quadprog
```

For the full code of the trustregion algorithm the reader is referred to the appendix (trustregion.m). Note that it is exactly the same as the local IQP implementation (ineqIQP.m) except that we redefine the variables at hand as discussed in 28, 29 and 31.

	$x_0 = (1,3)$				$x_0 = (-5,5)$				$x_0 = (8,4)$			
Method	It	QPit	fcalls	t	It	QPit	fcalls	t	It	QPit	fcalls	t
BFGS	18	3.8	35	116	12	4.3	23	455	17	3.8	33	76
LS	11	3.5	48	53	10	2.8	183	37	16	3.8	217	67
TR	11	3.6	21	78	9	5.2	17	276	14	5.1	27	76

Table 7: Important statistics for the 3 algorithms employed: iterations, mean iterations in subQP problem (QPit), function calls (fcalls), and time (t) in milliseconds.

Iteration sequence for $x_0 = (1,3)$

	x_1	x_2
1	1.00	3.00
2	2.00	2.00
3	3.00	1.72
4	2.84	2.72
5	3.11	1.86
6	2.90	1.94
7	3.00	1.98
8	3.00	2.00
9	3.00	2.00
10	3.00	2.00
11	3.00	2.00

Iteration sequence for $x_0 = (-5,5)$

	x_1	x_2
1	-5.00	5.00
2	-4.17	4.00
3	-3.96	3.00
4	-3.57	2.33
5	-3.73	2.98
6	-3.64	2.66
7	-3.65	2.72
8	-3.65	2.74
9	-3.65	2.74

Iteration sequence for $x_0 = (8,4)$

	x_1	x_2
1	8.00	4.00
2	7.00	3.00
3	6.00	2.40
4	5.00	2.00
5	4.06	1.62
6	3.32	2.62
7	2.32	1.62
8	3.03	1.68
9	2.99	2.03
10	3.01	1.99
11	3.00	2.00
12	3.00	2.00
13	3.00	2.00
14	3.00	2.00

3.3.2 Discussion

In our S(L1)QP trust region approach, rather than controlling the step size as in line search, the L1 penalty determines whether a step is accepted (or not) within a radius of Δ_k (i.e. the trust region). Figure 7 shows how the convergence path deteriorates significantly when Δ_k is changed from 1 to 3. Increasing the radius of the trust region allows it to take larger step sizes, however, in this case it is detrimental as it continuously "jumps over" the solution. The trust region defines the area where the model is considered reliable. Therefore, in this case, by increasing Δ_k and the trust region too much, we are in effect allowing the model to find solutions in regions that are unreliable.

With an appropriate Δ_k, however, we can see in Figure 6 that its convergence is satisfactory. When initiated near the solution (starting point 1) it converges in the same amount of iterations as line search. However, it too is vulnerable as the others to local minima as shown when initiated from starting point 2. When initiated relatively far from the optimal solution (starting point 3), it takes similarly sized steps that satisfy the constraints , and ultimately converges in less iterations than the other two algorithms. All in all , however, the linesearch implementation proved to converge faster in all three starting points. This is despite it having more function calls (evaluating the objective function) in the inner loop when determining whether the descent condition is satisfied.

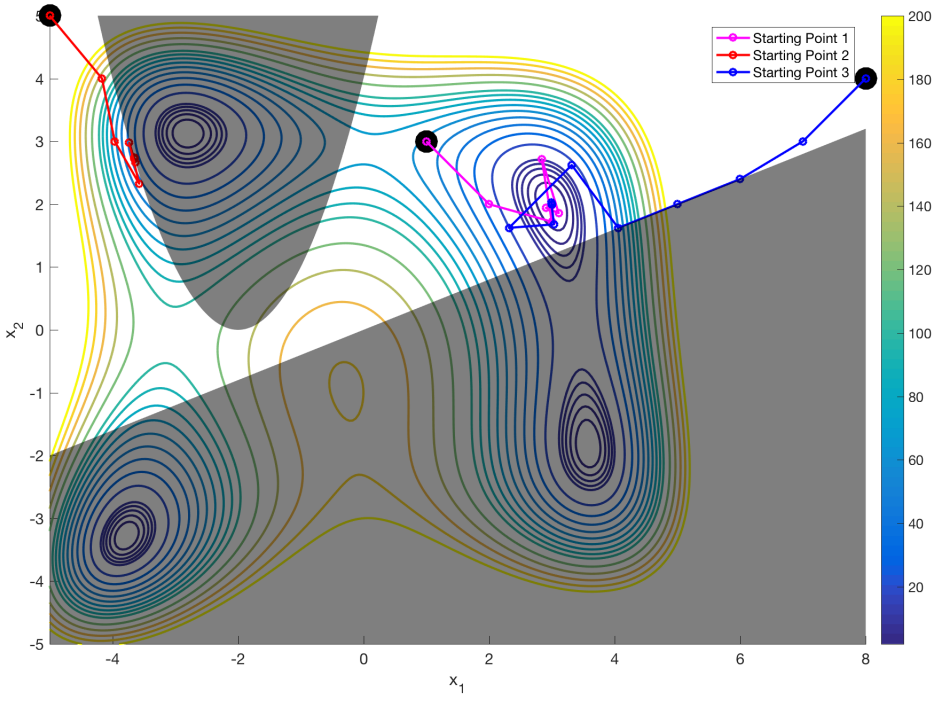

Figure 6: Contour plot showing the convergence steps of our trust region IQP implementation from three different starting points.

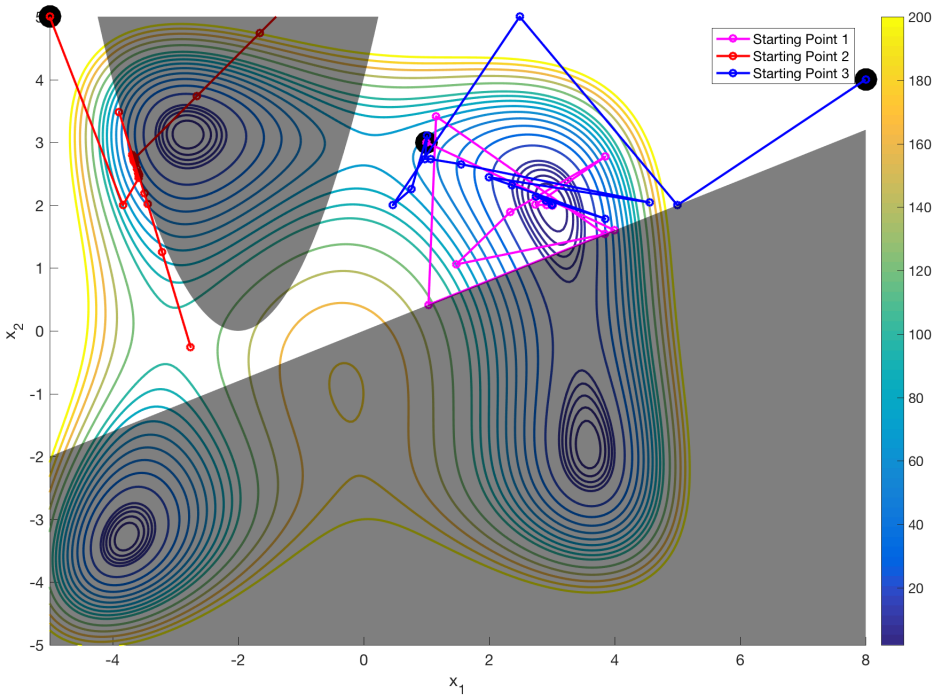

Figure 7: Contour plot showing the convergence steps of our trust region IQP implementation from three different starting points when Δ_k (radius of trust region) is changed from 1 to 3. Evidently, it is very sensitive to this parameter.

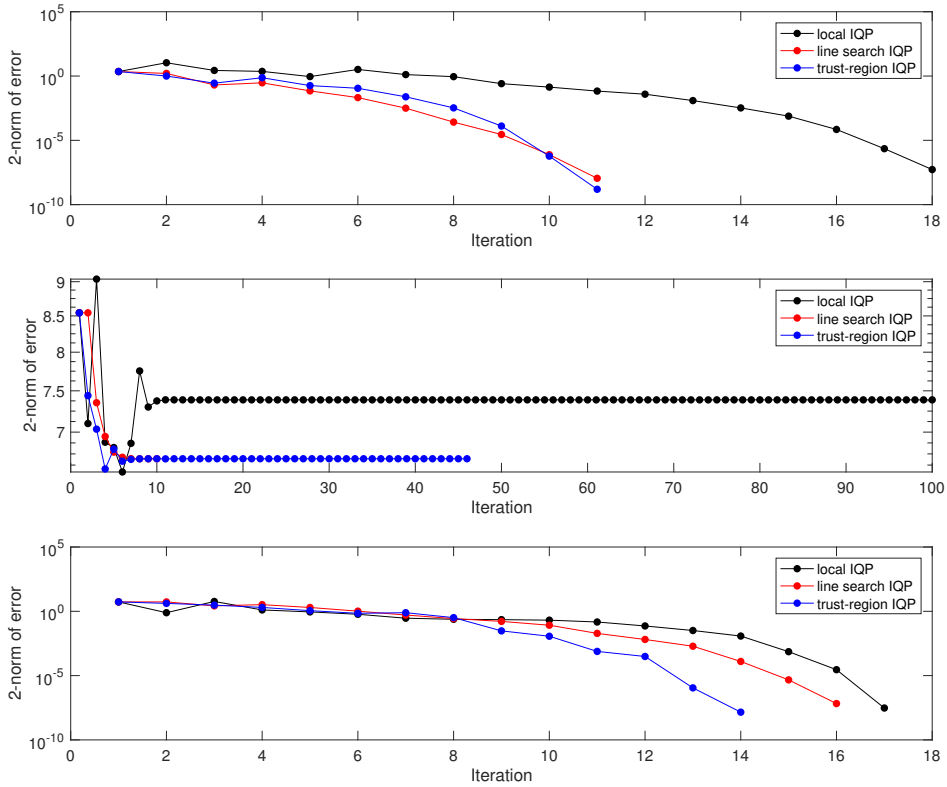

Figure 8: Plot showing the 2-norm error rate of the three algorithms as they converge. The top figure corresponds to the first starting point (1,3); the middle figure to the second starting point (-5,5), and the lower figure corresponds to the third starting point (8,4). Note that there is no convergence to the local minima from neither of the algorithms when initiated from the second starting point.

A Matlab code

A.1 Problem 1

A.1.1 LP solver

```
function [x,Converged,mu,lambda,iter,tol] = LPSolver(g,A,b,x,
    lambda_init,mu_init)
% LPIPPD   Primal-Dual Interior-Point LP Solver
%
%          min  g'*x
%           x
%          s.t. A x  = b      (Lagrange multiplier: mu)
%                 x >= 0      (Lagrange multiplier: lambda)
%
% Syntax: [x,Converged,mu,lambda,iter,tol] = LPsolver(g,A,b,x,
    lambda_init,mu_init)
%

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

% Original modified for Assignment 2 of Constrained Optimization
    (9.5.2017)
% Uses simplified parts from Assignment 1 PDPCIPA QP

% Initialize
n = length(A);
lambda = lambda_init;
mu = mu_init;
tol = 1e-9;
tolL = tol;
tolA = tol;
tols = tol;
eta = 0.99;
maxiter = 100;
iter = 0;
% Compute residuals
[rL, rA, rC, s] = computeResiduals(g,A,b,x,mu,lambda,n);
% Convergence check
Converged = checkConvergence(rL,rA,s,tolL,tolA,tols);

while ~Converged && (iter < maxiter)
```

```
36      iter = iter+1;
37      % Hessian (save factorization for speed)
38      xdivlambda = x./lambda;
39      H = A*diag(xdivlambda)*A';
40      L = chol(H,'lower');
41      % Affine step -> solve
42      [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,rC);
43      % Affine step -> step length
44      [alpha, beta] = stepLength(x,dx,lambda,dlambda);
45      % Center parameter
46      xAff = x + alpha*dx;
47      lambdaAff = lambda + beta*dlambda;
48      sAff = sum(xAff.*lambdaAff)/n;      % duality gap for affine step
49      sigma = (sAff/s)^3;                 % centering parameter
50      % Center+Corrector step to follow tractory to the primal dual
            solution set
51      rC = rC + dx.*dlambda - sigma*s;
52      [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,rC);
53      % Step length
54      [alpha, beta] = stepLength(x,dx,lambda,dlambda);
55      % Take step - update from (45)
56      x = x + eta*alpha*dx;
57      mu = mu + eta*beta*dmu;
58      lambda = lambda + eta*beta*dlambda;
59      % Compute residuals
60      [rL, rA, rC, s] = computeResiduals(g,A,b,x,mu,lambda,n);
61      % Check if converged
62      Converged = checkConvergence(rL,rA,s,tolL,tolA,tols);
63  end
64  % Signal the user that the algorithm didn't converge
65  if ~Converged
66      throw(MException('dtu:copt', 'Did not converge!'));
67  end
68
69  end
70
71  function [converged] = checkConvergence(rL,rA,s,tolL,tolA,tols)
72  % Stopping criteria from (48)
73  converged = (norm(rL) <= tolL) && (norm(rA) <= tolA) && (abs(s) <=
        tols);
74  end
75
76  function [rL,rA,rC,s] = computeResiduals(g,A,b,x,mu,lambda,n)
77  % Following eqns (46)
78  rL = g - A'*mu - lambda;
79  rA = A*x - b;
80  rC = x.*lambda;
81  s = sum(rC)/n;
82  end
```

```
83
84 function [alpha,beta] = stepLength(x,dx,lambda,dlambda)
85 % Find alpha_aff and beta_aff using (35) and (36)
86 idx_a = find(dx < 0);
87 alpha = min(1, min(-x(idx_a)./dx(idx_a)));
88 idx_b = find(dlambda < 0);
89 beta = min(1, min(-lambda(idx_b)./dlambda(idx_b)));
90 end
91
92 function [dx,dmu,dlambda] = stepSolve(x,lambda,xdivlambda,A,L,rA,rL,
       rC)
93 % Solve for dx, dmu, dlambda
94 rhs = -rA + A*((x.*rL + rC)./lambda);
95 dmu = L'\(L\rhs);
96 dx = xdivlambda.*(A'*dmu) - (x.*rL + rC)./lambda;
97 dlambda = -(rC+lambda.*dx)./x;
98 end
```

A.1.2 Test suite

```
1 %
2 % Test LP solver
3 % Original from John,
4 % modified for Assignment 2
5 %
6 m = 100;
7 n = 1000;
8 seed = 12345;
9 rng(seed, 'twister'); % updated so new Matlab versions don't complain
10 % Generate random LP problems
11 fprintf('\nGenerating random LP problem (N&W 14.15 p419-420)\n');
12 A = randn(m,n);
13 x = zeros(n,1);
14 x(1:m) = abs(rand(m,1));
15 lambda = zeros(n,1);
16 lambda(m+1:n) = abs(rand(n-m,1));
17 mu = rand(m,1);
18 g = A'*mu + lambda;
19 b = A*x;
20 % Choose the starting point (x0, lambda0, s0)
21 % with the components set to large positive values.
22 scalef = 100; % "large values" -> scale factor
23 x_init = scalef*ones(n,1);
24 lambda_init = scalef*ones(size(lambda));
25 mu_init = scalef*ones(size(mu));
26 % Run
27 fprintf('\nRunning primal dual interior point LP solver...\n');
28 [xlp,converged,mulp,lambdalp,iter,tol] = LPSolver(g,A,b,x_init,
```

```
    lambda_init,mu_init);
fprintf('Iterations: %d\n', iter);
% Check the computed vs "real" is the algorithm converged
if converged
    fprintf('Converged!\nErrors (solver tolerance %e)\n', tol);
    fprintf('x:      %e\n', max(abs(xlp-x)));
    fprintf('mu:     %e\n', max(abs(mulp-mu)));
    fprintf('lambda: %e\n', max(abs(lambdalp-lambda)));
end
```

A.2 Problem 2

A.2.1 Local SQP solver

```
function [x,lambda,info] = local_SQP(objfun,const,x,lambda,tol,maxit)
% The following function solves the nonlinear optimization problem in
% exercise 18.3 in N&W.

% initial evaluation of the function and the constraints
[~,g,H] = objfun(x);
[c,A,d2c1,d2c2,d2c3] = const(x);

dxL = g-A'*lambda; % gradient of the Lagrangian
F = [dxL; c]; %% KKT conditions
k = 1;
n = length(x);
x(:,k) = x;
% While the KKT conditions are not satisfied and the number of
    iterations
% is less than the threshold
while ((k < maxit) && (norm(F,'inf') > tol))
    % compute hessian of the Lagrangian
    d2L = H - lambda(1)*d2c1 - lambda(2)*d2c2 - lambda(3)*d2c3;
    % solve the system to get the Newton's descent direction
    z = [d2L -A'; A zeros(3)]\[-dxL; -c];
    % obtain next point in the iteration sequence
    p = z(1:n);
    x(:,k+1) = x(:,k) + p;
    % update lambda value
    lambda = lambda + z(n+1:end);
    k = k+1;
    % evaluate function and constraints on the new point
    [~,g,H] = objfun(x(:,k));
    [c,A,d2c1,d2c2,d2c3] = const(x(:,k));
    dxL = g-A'*lambda; % gradient of the Lagrangian
    % check KKT conditions
    F = [dxL;
```

```
        c];
end
info.iter = k; % return number of iterations
info.seq = x; % return iteration sequence
x = x(:,k); % return last point as solution
end
```

```
% The following function evaluates the objective function in exercise 18.3
% in N&W and computes the gradient and the Hessian at a specific point.

function [f,g,H] = objfun(x)

ep = exp(prod(x));
f = ep - 0.5*(x(1)^3+x(2)^3+1)^2;

if (nargout>1)
 g = [ x(2)*x(3)*x(4)*x(5)*ep - 3*x(1)^2*(x(1)^3+x(2)^3+1)
       x(3)*x(4)*x(5)*x(1)*ep - 3*x(2)^2*(x(1)^3+x(2)^3+1)
       x(4)*x(5)*x(1)*x(2)*ep
       x(5)*x(1)*x(2)*x(3)*ep
       x(1)*x(2)*x(3)*x(4)*ep ];
end;

if (nargout>2)
H=...
[ x(2)^2*x(3)^2*x(4)^2*x(5)^2*ep-9*x(1)^4-6*(x(1)^3+x(2)^3+1)*x(1), x(3)*x(4)*x(5)*ep+x(2)*x(3)^2*x(4)^2*x(5)^2*x(1)*ep-9*x(2)^2*x(1)^2, x(2)*x(4)*x(5)*ep+x(2)^2*x(3)*x(4)^2*x(5)^2*x(1)*ep, x(2)*x(3)*x(5)*ep+x(2)^2*x(3)^2*x(4)*x(5)^2*x(1)*ep, x(2)*x(3)*x(4)*ep+x(2)^2*x(3)^2*x(4)^2*x(5)*x(1)*ep;
x(3)*x(4)*x(5)*ep+x(2)*x(3)^2*x(4)^2*x(5)^2*x(1)*ep-9*x(2)^2*x(1)^2, x(1)^2*x(3)^2*x(4)^2*x(5)^2*ep-9*x(2)^4-6*(x(1)^3+x(2)^3+1)*x(2), x(1)*x(4)*x(5)*ep+x(1)^2*x(3)*x(4)^2*x(5)^2*x(2)*ep, x(1)*x(3)*x(5)*ep+x(1)^2*x(3)^2*x(4)*x(5)^2*x(2)*ep, x(1)*x(3)*x(4)*ep+x(1)^2*x(3)^2*x(4)^2*x(5)*x(2)*ep;
x(2)*x(4)*x(5)*ep+x(2)^2*x(3)*x(4)^2*x(5)^2*x(1)*ep,x(1)*x(4)*x(5)*ep+x(1)^2*x(3)*x(4)^2*x(5)^2*x(2)*ep,x(1)^2*x(2)^2*x(4)^2*x(5)^2*ep, x(1)*x(2)*x(5)*ep+x(1)^2*x(2)^2*x(4)*x(5)^2*x(3)*ep,x(1)*x(2)*x(4)*ep+x(1)^2*x(2)^2*x(4)^2*x(5)*x(3)*ep;
x(2)*x(3)*x(5)*ep+x(2)^2*x(3)^2*x(4)*x(5)^2*x(1)*ep,x(1)*x(3)*x(5)*ep+x(1)^2*x(3)^2*x(4)*x(5)^2*x(2)*ep,x(1)*x(2)*x(5)*ep+x(1)^2*x(2)^2*x(4)*x(5)^2*x(3)*ep,x(1)^2*x(2)^2*x(3)^2*x(5)^2*ep,x(1)*x(2)*x(3)*ep+x(1)^2*x(2)^2*x(3)^2*x(5)*x(4)*ep;
x(2)*x(3)*x(4)*ep+x(2)^2*x(3)^2*x(4)^2*x(5)*x(1)*ep,x(1)*x(3)*x(4)*ep+x(1)^2*x(3)^2*x(4)^2*x(5)*x(2)*ep,x(1)*x(2)*x(4)*ep+x(1)^2*x(2)^2*x(4)^2*x(5)*x(3)*ep,x(1)*x(2)*x(3)*ep+x(1)^2*x(2)^2*x(3)^2*x(5)*x(4)*ep, x(1)^2*x(2)^2*x(3)^2*x(4)^2*ep];
```

```
24
25 end;
26
27
28
29 end
```

```
1  % The following function evaluates the constraints in exercise 18.3
2  % in N&W and computes the gradient and the Hessian at a specific
       point.
3
4  function [c,dc,d2c1,d2c2,d2c3] = const(x)
5
6  c = [x(1)^2+x(2)^2+x(3)^2+x(4)^2+x(5)^2-10;
7       x(2)*x(3)-5*x(4)*x(5);
8       x(1)^3+x(2)^3+1];
9
10 dc = [2*x(1) 0 3*x(1)^2;
11       2*x(2) x(3) 3*x(2)^2;
12       2*x(3) x(2) 0;
13       2*x(4) -5*x(5) 0;
14       2*x(5) -5*x(4) 0]';
15
16 d2c1 = diag(2*ones(5));
17 d2c2 = zeros(5);
18 d2c2(2,3) = 1; d2c2(3,2) = 1; d2c2(4,5) = -5; d2c2(5,4) = -5;
19 d2c3 = zeros(5);
20 d2c3(1,1) = 6*x(1);
21 d2c3(2,2) = 6*x(2);
22
23 end
```

A.2.2 Damped BFGS

```
1  function [x,lambda,info] = BFGS_SQP(objfun,const,x,lambda,tol,maxit)
2
3  % The following function solves the nonlinear programming problem in
4  % exercise 18.3 in N&W using the damped BFGS approximation to the
       Hessian
5  % of the Lagrangian.
6
7  % initial evaluation of the function and the constraints
8  [~,g] = objfun(x);
9  [c,A] = const(x);
10
11 dxL = g-A'*lambda; % gradient of the Lagrangian
12 F = [dxL; c];
13
```

```
k = 1;
n = length(x);
x(:,k) = x;

B = eye(n); % Hessian approximation initial value

while ((k < maxit) && (norm(F,'inf') > tol))

    % solve the system to get the search direction
    z = [B -A'; A zeros(3)]\[-dxL; -c];
    % obtain next point in the iteration sequence
    p = z(1:n);
    x(:,k+1) = x(:,k) + p;
    % update lambda value
    lambda = lambda + z(n+1:end);
    k = k+1;
    % evaluate function and constraints on the new point
    [~,g] = objfun(x(:,k));
    [c,A] = const(x(:,k));
    dxLn = g-A'*lambda; % new gradient of the Lagrangian
    % check KKT conditions
    F = [dxLn; c];
    % damped BFGS updating
    s = x(:,k) - x(:,k-1);
    y = dxLn - dxL;
    dxL = dxLn;
     % ensure possitive definiteness
    if s'*y >= 0.2*s'*B*s
        theta = 1;
    else
        theta = (0.8*s'*B*s)/(s'*B*s-s'*y);
    end
    r = theta*y+(1-theta)*B*s;
    % update Bs
    B = B - (B*(s*s')*B)/(s'*B*s) + r*r'/(s'*r);
end
% store number of iterations and iteration sequence
info.iter = k;
info.seq = x;
x = x(:,k); % return last point as solution
end
```

A.2.3 Damped BFGS and line search

```
function [x,lambda,info] = lineSearch_SQP(objfun,const,x,lambda,tol,
    maxit)

% The following function solves the nonlinear programming problem in
```

```
% exercise 18.3 in N&W using the damped BFGS approximation to the
    Hessian
% of the Lagrangian.

% initial evaluation of the function and the constraints
[f,g] = objfun(x);
[c,A] = const(x);

dxL = g-A'*lambda; % gradient of the Lagrangian
F = [dxL; c];

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

B = eye(n); % Hessian approximation initial value

while ((k < maxit) && (norm(F,'inf') > tol))

    % solve the system to get the search direction
    z = [B -A'; A zeros(3)]\[-dxL; -c];
    p = z(1:n);
    plambda = z(n+1:end);

    % Backtraking line search method
    sigma = 1; % Hessian positive definite
    rho = 0.1; % rho in (0,1)
    mu = (g'*p + sigma/2*p'*B*p)/((1-rho)*norm(c,1));
    alpha = 1; % full step
    tau = 0.8; % contraction factor

    % check sufficient decrease condition
    fn = objfun(x(:,k)+alpha*p);
    cn = const(x(:,k)+alpha*p);
    eta = 0.4;
    while fn+mu*norm(cn,1) > f+mu*norm(c,1)+eta*alpha*(g'*p-mu*norm(c
        ,1))
        alpha = tau*alpha; % update alpha
        fn = objfun(x(:,k)+alpha*p);
        cn = const(x(:,k)+alpha*p);
    end
    f = fn;
    alphaval(k) = alpha; % Store alfa value in every iteration
    % obtain next point in the iteration sequence
    x(:,k+1) = x(:,k) + alpha*p;
    % update lambda value
    lambda = lambda + alpha*plambda;

    k = k+1;
```

```
51
52      % evaluate function and constraints on the new point
53      [f,g] = objfun(x(:,k));
54      [c,A] = const(x(:,k));
55      dxLn = g-A'*lambda; % new gradient of the Lagrangian
56      F = [dxLn; c];
57
58      % damped BFGS updating
59      s = x(:,k) - x(:,k-1);
60      y = dxLn - dxL;
61      dxL = dxLn;
62
63      if s'*y >= 0.2*s'*B*s
64          theta = 1;
65      else
66          theta = (0.8*s'*B*s)/(s'*B*s-s'*y);
67      end
68      r = theta*y+(1-theta)*B*s; % ensure possitive definiteness
69
70      B = B - B*(s*s')*B/(s'*B*s) + r*r'/(s'*r); % B updating
71
72      % ensure possitive definiteness
73  end
74  % store number of iterations and iteration sequence
75  info.iter = k;
76  info.seq = x;
77  info.alpha = alphaval; % return step lenght sequence
78  x = x(:,k); % return last point as solution
79  end
```

A.2.4 Problem 2 driver

```
1   clear
2   close all
3   %% Equality Constrained SQP
4   % Initial values
5   x = [-1.8;1.7;1.9;-0.8;-0.8];
6   lambda = [1; 1; 1];
7   tol = 1e-3;
8   maxit = 200;
9   % solve nonlinear optimization problem
10  [x1,lambda1,info1] = local_SQP(@objfun,@const,x,lambda,tol,maxit);
11  % solve using a damped BFGS approximation to the Hessian of the
        Lagrangian
12  [x2,lambda2,info2] = BFGS_SQP(@objfun,@const,x,lambda,tol,maxit);
13  % solve using a damped BFGS approcimation and a line search method
14  [x3,lambda3,info3] = lineSearch_SQP(@objfun,@const,x,lambda,tol,maxit
        );
```

```
% just line search
[x4,lambda4,info4] = lineSearch_SQP2(@objfun,@const,x,lambda,tol,
    maxit);

%% Convergence semilogys
err1 = (info1.seq - x1);
err1_norm = diag(sqrt(err1'*err1));
semilogy(1:info1.iter,err1_norm,'-o')
hold on
err2 = (info2.seq - x2);
err2_norm = diag(sqrt(err2'*err2));
semilogy(1:info2.iter,err2_norm,'-o')
err3 = (info3.seq - x3);
err3_norm = diag(sqrt(err3'*err3));
semilogy(1:info3.iter,err3_norm,'-o')
err4 = (info4.seq - x4);
err4_norm = diag(sqrt(err4'*err4));
semilogy(1:info4.iter,err4_norm,'-o')
xlabel('iteration')
ylabel('2 norm of error')
legend('Local SQP','BFGS','BFGS + linesearch','linesearch')
axis([1 6 1e-5 1])
print('convergence2','-dpng')

%% Iteration sequence
%% Generate INFO 2 iteration sequence table
clear input;
clc;
% Add data, change as needed
input.data = info2.seq'; % make sure to have this correctly formatted
input.tablePositioning = 'h';
input.tableColLabels = {'$x_1$','$x_2$','$x_3$','$x_4$','$x_5$'};
input.tableColumnAlignment = 'l';
input.tableCaption = 'Iteration sequence for BFGS approximation';
input.tableLabel = 'ex2:bfgs'; % prepends "table" -> table:
    MyTableLabel
% Generate tex output
texOut = latexTable(input);

%% Generate INFO 3 iteration sequence table
clear input;
clc;
% Add data, change as needed
input.data = [info4.seq' [info4.alpha 0]']; % make sure to have this
    correctly formatted
input.tablePositioning = 'h';
input.tableColLabels = {'$x_1$','$x_2$','$x_3$','$x_4$','$x_5$','$\
    alpha$'};
input.tableColumnAlignment = 'l';
```

```
input.tableCaption = 'Iteration sequence for local SQP and line
    search';
input.tableLabel = 'ex2:localSQPls'; % prepends "table" -> table:
    MyTableLabel
% Generate tex output
texOut = latexTable(input);
```

A.3 Problem 3

A.3.1 Damped BFGS

```
function [x,lambda,info] = ineqSQP(x,lambda,tol,maxit)

% Initializing info storage variables
subQPiter = zeros(1,maxit);
subQPfval = zeros(1,maxit);
k=1;
fcalls=1;
n=length(x);
x(:,k) = x; % storing solution iterates

% Initial evaluation of the problem and constraints
[~,g,~] = objfun(x);                    %gradient of original function
[cineq,dcineq,~,~] = constraint(x);  %constraints and its gradient

B = eye(2);                             %Hessian of L to be aproximated
F = [g - dcineq'*lambda; cineq];        %1st Order KKT conditions

QPoptions = optimoptions(@quadprog, 'Algorithm', 'interior-point-
    convex', 'MaxIterations', 200, 'Display', 'none');

% Checking max iterations and KKT conditions
while ((k < maxit) && (norm(F(1:2),'inf') > tol) )

    [~,g,~] = objfun(x(:,k)); fcalls=fcalls+1;
    [cineq,dcineq,~,~] = constraint(x(:,k)); %c(x) and A(x) in report
        notation

    % Defining variables
    gL = g - dcineq'*lambda;    %gradient of Lagrangian

    % Solving QP subproblem: to get Newton's descent direction
    [z, fval,~,output, lambda] = quadprog(B,g,-1*(dcineq), cineq, [],
        [], [], [], [], QPoptions);
    % Storing SubQP info
    subQPiter(k) = output.iterations;
    subQPfval(k) = fval;
```

```
34
35      % Obtain next point in iteration sequence
36      incremental = z(1:n); % p in report's notation
37      x(:,k+1) = x(:,k) + incremental;
38
39      % Update lambda
40      lambda = lambda.ineqlin; %only when using quadprog
41
42      % Evaluate objective function and constraints with new point, x(k
            +1)
43      [~,g,~] = objfun(x(:,k+1)); fcalls=fcalls+1;
44      [cineq, dcineq] = constraint(x(:,k+1));
45
46      % Compute gradient of lagrangian with new x(k+1)
47      gL_new = g - dcineq'*lambda;
48
49      % Update Hessian matrix by damped BFGS procedure
50      p = x(:,k+1) - x(:,k);
51      q = gL_new - gL;
52      B = update_hessian(B, p, q);
53
54      % Update KKT conditions
55      F = [gL_new; cineq];
56      k = k+1;
57
58
59  end
60  % storing convergence stats
61  info.iter    = k;
62  info.seq     = x;
63  info.fcalls = fcalls;
64  info.subQPiter = subQPiter;
65  info.subQPfval = subQPfval;
66  end
```

```
1
2   function [f,g,H] = objfun(x)
3
4   %Evaluates the objective function, its first order and second order
5   %derivatives for a supplied x vector.
6
7   x1=x(1);
8   x2=x(2);
9   f = (x1.^2+x2-11).^2 + (x1+x2.^2-7).^2;
10  g = [4*(x1.^2+x2-11).*x1+2*(x1+x2.^2-7); 2*(x1.^2+x2-11)+4*(x1+x2
        .^2-7).*x2];
11  H = [12*x1^2+4*x2-42,       4*(x1+x2);
12        4*(x1+x2),      4*x1+12*x2.^2-26];
13
```

```
14
15 %{
16 With Symbolic Tool
17     % dummy x for testing, and extracting supplied x
18     x = [1;1];
19     x1 = x(1,1); x2 = x(2,1);
20
21     % creating symbolic math: obj function
22     syms x_1 x_2
23     %func = exp(x_1*x_2*x_3*x_4*x_5) - 0.5*(x_1^3 + x_2^3 +1)^2
24     func = (x_1^2 + x_2 - 11)^2 + (x_
25     x_1=x2; x_2=x2;
26
27     % returning solution to obj function
28     f = eval(func);
29
30     % creating symbolic math: objective function
31     syms x_1 x_2 x_3 x_4 x_5
32     df1 = diff(func, x_1);
33     df2 = diff(func, x_2);
34     df3 = diff(func, x_3);
35     df4 = diff(func, x_4);
36     df5 = diff(func, x_5);
37
38     % returning 1st order solution
39     df = zeros(5,1);
40     x_1=x2; x_2=x2; x_3=x3; x_4=x4; x_5=x5;
41     df(1,1) = eval(df1);
42     df(2,1) = eval(df2);
43     df(3,1) = eval(df3);
44     df(4,1) = eval(df4);
45     df(5,1) = eval(df5);
46
47       %}
```

```
1 function [ cineq,dcineq,d2cineq1,d2cineq2 ] = constraint( x )
2 %CONSTRAINT evalutates the equality and inequality constraints for a
      given x.
3
4
5 cineq = [x(1)^2 + 4*x(1) - x(2) + 4;
6           -4*x(1) + 10*x(2)];
7
8 dcineq = [ 2*x(1)+4    -4;
9             -1          10]';
10
11 d2cineq1 = [2  0;
12              0  0];
13
```

```
d2cineq2 = [0  0;
            0  0];
end
```

```
function [ B_new ] = update_hessian(B, p, q )
% Description: Updates Hessian of Lagrangian matrix
% by the modified BFGS procedure

if p'*q >= 0.2*p'*B*p
    theta = 1;

elseif p'*q < 0.2*p'*B*p
    theta = 0.8*p'*B*p / (p'*B*p - p'*q);
end

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

nom = B*(p*p')*B;
denom = (p'*B*p);
B_new = B - (nom / denom) + r*r'/(p'*r);

end
```

A.3.2 Damped BFGS and line search

```
function [x,lambda,info] = ineqSQP(x,lambda,tol,maxit)

% Initializing info storage variables
subQPiter = zeros(1,maxit);
subQPfval = zeros(1,maxit);
fcalls=1;
k=1;
n=length(x);
x(:,k) = x; % storing solution iterates

% Initializing problem
B = eye(2);
[~,g,~] = objfun(x); [cineq,dcineq,~,~] = constraint(x);
F = [g - dcineq'*lambda; cineq];

QPoptions = optimoptions(@quadprog, 'Algorithm', 'interior-point-
    convex', 'MaxIterations', 200, 'Display', 'none');

while ((k < maxit) && (norm(F(1:2),'inf') > tol))   %&& (norm(cineq,'
    inf') < tol))

    [f,g,~] = objfun(x(:,k)); fcalls=fcalls+1;
    [cineq,dcineq,~,~] = constraint(x(:,k));
```

```
22
23      % Defining variables
24      gL = g - dcineq'*lambda;   %gradient of Lagrangian
25
26      % solve system to get search direction
27      [incremental, fval,~,output, lambda_qp] = quadprog(B,g,-dcineq,
            cineq, [], [], [], [], [], QPoptions);
28      subQPiter(k) = output.iterations;
29      subQPfval(k) = fval;
30      plambda = lambda_qp.ineqlin - lambda;
31
32      % Backtraking line search method
33      sigma = 1; % Hessian positive definite
34      rho = 0.2; % rho in (0,1)
35      mu = (g'*incremental + sigma/2*incremental'*B*incremental)/((1-
            rho)*norm(cineq,1));
36      alpha = 1; % full step
37      tau = 0.8; % contraction factor
38
39      % check sufficient decrease condition
40      fn = objfun(x(:,k)+alpha*incremental); fcalls=fcalls+1;
41      cn = constraint(x(:,k)+alpha*incremental);
42      eta = 0.2;
43      while fn+mu*norm(cn,1) > f+mu*norm(cineq,1)+eta*alpha*(g'*
            incremental-mu*norm(cineq,1))
44          alpha = tau*alpha; % update alpha
45          fn = objfun(x(:,k)+alpha*incremental); fcalls=fcalls+1;
46          cn = constraint(x(:,k)+alpha*incremental);
47      end
48      f = fn;
49
50      % update lambda value
51      lambda = lambda + alpha*plambda;
52      gL = g - dcineq'*lambda;   %gradient of Lagrangian
53
54      % evaluate with new x(k+1)
55      x(:,k+1) = x(:,k) + alpha*incremental;
56      [f,g,~] = objfun(x(:,k+1)); fcalls=fcalls+1;
57      [cineq,dcineq,~,~] = constraint(x(:,k+1));
58
59      % compute gradient of lagrangian with new x(k+1)
60      gL_new = g - dcineq'*lambda;
61
62      % update Hessian matrix by modified BFGS procedure
63      p = x(:,k+1) - x(:,k);
64      q = gL_new - gL;
65      B = update_hessian(B, p, q);
66
67      F = [gL_new; cineq];
```

```
68      k = k+1;
69
70  end
71  info.iter = k;
72  info.seq = x;
73  info.fcalls = fcalls;
74  info.subQPiter = subQPiter;
75  info.subQPfval = subQPfval;
76  end
```

A.3.3 Trust region

```
1   function [ x,lambda,info ] = trustregion( x,lambda,tol,maxit )
2   %TRUSTREGION Summary of this function goes here
3   %   Detailed explanation goes here
4
5   % Initializing info storage variables
6   subQPiter = zeros(1,maxit);
7   subQPfval = zeros(1,maxit);
8   fcalls=1;
9   k=1;
10  n=length(x);
11  x(:,k) = x; % storing solution iterates hg
12
13  % Padding Hessian approx. with zeros to
14  % accommodate trust-region formulation
15  zero = [0 0; 0 0];
16  B = [eye(2) zero;
17       zero   zero];
18  mu = 40; dk = 1;
19  [~,g,~] = objfun(x); [cineq,dcineq,~,~] = constraint(x);
20  g = [g; mu; mu];
21  F = [g(1:2) - dcineq'*lambda; cineq];
22
23  QPoptions = optimoptions(@quadprog, 'Algorithm', 'interior-point-
        convex', 'MaxIterations', 200, 'Display', 'none');
24
25  while ((k < maxit) && (norm(F(1:2),'inf') > tol) )   %&& (norm(cineq
        ,'inf') < tol))
26
27      [~,g,~] = objfun(x(:,k)); fcalls=fcalls+1;
28      g = [g; mu; mu];
29      [cineq,dcineq,~,~] = constraint(x(:,k));
30      cineq=[cineq;0;0];
31      dcineq=[dcineq zero;
32              zero   zero];
33      % Defining variables
34      gL = g(1:2) - dcineq(1:2,1:2)'*lambda;    %gradient of Lagrangian
```

```
35
36      % Solving QP subproblem
37      LB = [-dk -dk 0 0]';
38      UB = [dk dk inf inf]';
39      [z, fval,~,output, lambda] = quadprog(B,g,-1*(dcineq), cineq, [],
            [], LB, UB, [], QPoptions);
40      subQPiter(k) = output.iterations;
41      subQPfval(k) = fval;
42
43      incremental = z(1:n);
44      x(:,k+1) = x(:,k) + incremental;
45      t = z(n+1:end);
46      lambda = lambda.ineqlin(1:2); %only when using quadprog
47
48      % evaluate with new x(k+1)
49      [~,g,~] = objfun(x(:,k+1)); fcalls=fcalls+1;
50      [cineq, dcineq] = constraint(x(:,k+1));
51
52      % compute gradient of lagrangian with new x(k+1)
53      gL_new = g - dcineq'*lambda;
54
55      % update Hessian matrix by modified BFGS procedure
56      p = x(:,k+1) - x(:,k);
57      q = gL_new - gL;
58      B(1:2,1:2) = update_hessian(B(1:2,1:2), p, q);
59
60      F = [gL_new; cineq];
61      k = k+1;
62
63  end
64  info.iter = k;
65  info.seq = x;
66  info.fcalls = fcalls;
67  info.subQPiter = subQPiter;
68  info.subQPfval = subQPfval;
69  end
```

A.3.4 Problem 3 driver

```
1  %% Plot the problem given in Ex3
2  x1lim = 5; x1 = -x1lim:0.01:x1lim;
3  x2lim = 5; x2 = -x1lim:0.01:x2lim;
4  [X1,X2] = meshgrid(x1, x2);
5
6  % https://en.wikipedia.org/wiki/Himmelblau%27s_function
7  F = (X1.^2 + X2 - 11).^2 + (X1 + X2.^2 - 7).^2;
8
9  figure(1)
```

```
fprintf('Plotting...\n');
clf
axis([-x1lim x1lim -x1lim x2lim])
hold on
% Objective function
fprintf('Adding contour lines...\n');
[~,h] = contour(X1,X2,F, 'LevelList', ([
    0:2:10 10:10:100 100:20:200
]), 'LineWidth', 1.4); colorbar; % , 'ShowText', 'on'
% Constraints
fprintf('Adding constraints...\n');
plot(x1, x1.^2 + 4.*x1 + 4, 'LineWidth', 1, 'Color', 'r', 'LineStyle'
    , '-')
plot(x1, 0.4*x1, 'LineWidth', 1, 'Color', 'r', 'LineStyle', '-')
x1f = -x1lim:0.01:x1lim;
fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.4, 'EdgeColor'
    , 'none')
patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim 0.4*
    x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.4, '
    EdgeColor', 'none')
hold off
%title('Himmelblau''s function: $f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2
    + (x_1 + x_2^2 - 7)^2$', 'Interpreter', 'latex', 'FontSize', 14)
xlabel('$x_1$', 'FontSize', 14, 'Interpreter', 'latex')
ylabel('$x_2$', 'FontSize', 14, 'Interpreter', 'latex')
set(gca, 'FontSize', 14)
set(gca, 'XTick', -x1lim:x1lim); set(gca, 'YTick', -x2lim:x2lim);
grid on
fprintf('Done! Wait couple of seconds to see the rendered plot.\n');
```

```
clear all; close all;


x0=[1,3 ; -5,5; 8,4]'; %3 starting points
n=size(x0,2);    %number of starting points
lambda=[1 ; 1];  %initial lambda
tol=1e-5;
maxit=100;
rep=30;          %number of repetitions for TicToc
T1=zeros(rep,n); %TicToc stats for ineqSQP
T2=zeros(rep,n); %TicToc stats for linesearch_ineqSQP
T3=zeros(rep,n); %TicToc stats for trustregion_ineqSQP
e1 = cell(1,n); e2 = cell(1,n); e3 = cell(1,n); %store errors for
    each algo
solution = [3;2];

% Structs to store info for each algorithm
ISQP{n}=[]; LS_ISQP{n}=[];  TR_ISQP{n}=[];
X1{n} = []; L1{n} = []; I1{n} = [];
```

```
19 X2{n} = []; L2{n} = []; I2{n} = [];
20 X3{n} = []; L3{n} = []; I3{n} = [];
21
22 %Contour plot
23 fig = figure('Position', [100, 0, 1000,1000]);
24 hold on
25 himmelblauContours
26 plot(x0(1,:),x0(2,:),'k.','markersize',70) %change desired starting
       point
27
28 % Iterating algorithms over 3 starting points
29 for j=1:n
30
31     for i=1:1%rep
32
33         % Algo 1: local IQP
34         T=tic;
35         [x,lambda,info] = ineqSQP(x0(:,j),lambda,tol,maxit);
36         T1(i,j)=toc(T);
37         X1{j} = x;  L1{j} = lambda;  I1{j} = info;
38         ISQP{j} = struct('x',X1(j),'lambda',L1(j),'info',I1(j));
39
40         % Algo 2: linesearch IQP
41         T=tic;
42         [x,lambda,info] = linesearch_ineqSQP(x0(:,j),lambda,tol,20);
43         T2(i,j)=toc(T);
44         X2{j} = x;  L2{j} = lambda;  I2{j} = info;
45         LS_ISQP{j} = struct('x',X2(j),'lambda',L2(j),'info',I2(j));
46
47         % Algo 3: trust-region IQP
48         T=tic;
49         [x,lambda,info] = trustregion(x0(:,j),lambda,tol,maxit);
50         T3(i,j)=toc(T);
51         X3{j} = x;  L3{j} = lambda;  I3{j} = info;
52         TR_ISQP{j} = struct('x',X3(j),'lambda',L3(j),'info',I3(j));
53
54     end%for i (repetitions)
55
56
57     e1{j} = sqrt(sum( (([X1{j}(1,:)-3; X1{j}(2,:)-2]).^2) )); %algo1
           error/iter
58     e2{j} = sqrt(sum( (([X2{j}(1,:)-3; X2{j}(2,:)-2]).^2) )); %algo2
           error/iter
59     e3{j} = sqrt(sum( (([X3{j}(1,:)-3; X3{j}(2,:)-2]).^2) )); %algo3
           error/iter
60
61 end%for j (starting points)
62
63     h1 = plot(X1{1}(1,:),X1{1}(2,:),'k-o','linewidth',2);
```

```
64      h2 = plot(X2{1}(1,:),X2{1}(2,:),'r-o','linewidth',2);
65      h3 = plot(X3{1}(1,:),X3{1}(2,:),'b-o','linewidth',2);
66
67      legend([h1,h2,h3],'local IQP','backtracking line search IQP', '
            trust-region IQP','location','southeast')
68
69      %feeding iterations to error_plot
70      iter1 = [I1{1}.iter;I1{2}.iter;I1{3}.iter];
71      iter2 = [I2{1}.iter;I2{2}.iter;I2{3}.iter];
72      iter3 = [I3{1}.iter;I3{2}.iter;I3{3}.iter];
73      error_plot(e1,e2,e3,iter1,iter2,iter3,n,maxit);
74
75      % Timing stats for each algo
76      T1=mean(T1); T2=mean(T2); T3=mean(T3);
77
78
79      % Clean Workspace
80  %clear i I1 I2 I3 j L1 L2 L3 lambda T x X1 X2 X3 info
81
82  %% Convergence Plots
83
84  % Local IQP Plot
85  fig = figure('Position', [100, 0, 1000,1000]);
86  hold on
87  himmelblauContours
88  plot(x0(1,:),x0(2,:),'k.','markersize',70)
89  h1 = plot(X1{1}(1,:),X1{1}(2,:),'m-o','linewidth',2);
90  h2 = plot(X1{2}(1,:),X1{2}(2,:),'r-o','linewidth',2);
91  h3 = plot(X1{3}(1,:),X1{3}(2,:),'b-o','linewidth',2);
92  legend([h1,h2,h3],'Starting Point 1','Starting Point 2', 'Starting
        Point 3','location','northeast')
93  % show constraints
94  x1lim = 14; x1 = -x1lim:0.01:x1lim;
95  x2lim = 10; x2 = -x1lim:0.01:x2lim;
96  x1f = -10:0.01:14;
97  %plot(x1, x1.^2 + 4.*x1 + 4, 'LineWidth', 1, 'Color', 'r', 'LineStyle
        ', '-')
98  %plot(x1, 0.4*x1, 'LineWidth', 1, 'Color', 'r', 'LineStyle', '-')
99  fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.5, 'EdgeColor'
        , 'none')
100 patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim 0.4*
        x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.5, '
        EdgeColor', 'none')
101 % set axis as needed
102 axis([-5 14 -5 8])
103 hold off
104
105 %% Line search IQP Plot
106 fig = figure('Position', [100, 0, 1000,1000]);
```

```
hold on
himmelblauContours
plot(x0(1,:),x0(2,:),'k.','markersize',70)
h1 = plot(X2{1}(1,:),X2{1}(2,:),'m-o','linewidth',2);
h2 = plot(X2{2}(1,:),X2{2}(2,:),'r-o','linewidth',2);
h3 = plot(X2{3}(1,:),X2{3}(2,:),'b-o','linewidth',2);
legend([h1,h2,h3],'Starting Point 1','Starting Point 2', 'Starting
    Point 3','location','northeast')
% show constraints
x1lim = 10; x1 = -x1lim:0.01:x1lim;
x2lim = 10; x2 = -x1lim:0.01:x2lim;
x1f = -10:0.01:10;
%plot(x1, x1.^2 + 4.*x1 + 4, 'LineWidth', 1, 'Color', 'r', 'LineStyle
    ', '-')
%plot(x1, 0.4*x1, 'LineWidth', 1, 'Color', 'r', 'LineStyle', '-')
fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.5, 'EdgeColor'
    , 'none')
patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim 0.4*
    x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.5, '
    EdgeColor', 'none')
% set axis as needed
axis([-5 8 -5 5])
hold off

%% Trust Region IQP Plot
fig = figure('Position', [100, 0, 1000,1000]);
clf;
hold on
himmelblauContours
plot(x0(1,:),x0(2,:),'k.','markersize',70)
h1 = plot(X3{1}(1,:),X3{1}(2,:),'m-o','linewidth',2);
h2 = plot(X3{2}(1,:),X3{2}(2,:),'r-o','linewidth',2);
h3 = plot(X3{3}(1,:),X3{3}(2,:),'b-o','linewidth',2);
legend([h1,h2,h3],'Starting Point 1','Starting Point 2', 'Starting
    Point 3','location','northeast')
% show constraints
x1lim = 10; x1 = -x1lim:0.01:x1lim;
x2lim = 10; x2 = -x1lim:0.01:x2lim;
x1f = -10:0.01:10;
%plot(x1, x1.^2 + 4.*x1 + 4, 'LineWidth', 1, 'Color', 'r', 'LineStyle
    ', '-')
%plot(x1, 0.4*x1, 'LineWidth', 1, 'Color', 'r', 'LineStyle', '-')
fill(x1f, x1f.^2 + 4.*x1f + 4, [0 0 0], 'FaceAlpha', 0.5, 'EdgeColor'
    , 'none')
patch('Faces',[1 2 3], 'Vertices', [-x1lim 0.4*-x1lim; x1lim 0.4*
    x1lim; x1lim -999], 'FaceColor', 'black', 'FaceAlpha', 0.5, '
    EdgeColor', 'none')
% set axis as needed
axis([-5 8 -5 5])
```

```
146 hold off
147
148
149
150
151 %% LATEX TABLES (for iteration seq of all algorithms)
152
153 % LOCAL IQP
154 %Starting Point 1
155 matrix = I1{1}.seq';
156     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11', ...
157                  '12', '13', '14', '15', '16', '17', '18'};
158     columnLabels = { '$x_1$', '$x_2$'};
159     matrix2latex(matrix, 'localseq1.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
160
161     %%
162 %Starting Point 2
163 matrix = I1{2}.seq(:,1:12)';
164     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11', ...
165                  '12'};
166     columnLabels = { '$x_1$', '$x_2$'};
167     matrix2latex(matrix, 'localseq2.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
168
169     %%
170 %Starting Point 3
171 matrix = I1{3}.seq';
172     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11', ...
173                  '12', '13', '14', '15', '16', '17'};
174     columnLabels = { '$x_1$', '$x_2$'};
175     matrix2latex(matrix, 'localseq13.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
176
177
178 %% LINESEARCH
179 %Starting Point 1
180 matrix = I2{1}.seq';
181     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11'};
182     columnLabels = { '$x_1$', '$x_2$'};
183     matrix2latex(matrix, 'lsseq1.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
```

```
        f', 'size', 'small');
184
185 %Starting Point 2
186 matrix = I2{2}.seq';
187     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10'};
188     columnLabels = { '$x_1$', '$x_2$'};
189     matrix2latex(matrix, 'lsseq2.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
190
191 %Starting Point 3
192 matrix = I2{3}.seq';
193     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11', ...
194                  '12', '13', '14', '15', '16'};
195     columnLabels = { '$x_1$', '$x_2$'};
196     matrix2latex(matrix, 'lsseq3.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
197
198 %% TRUST REGION
199 %Starting Point 1
200 matrix = I3{1}.seq';
201     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11'};
202     columnLabels = { '$x_1$', '$x_2$'};
203     matrix2latex(matrix, 'trseq1.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
204
205 %Starting Point 2
206 matrix = I3{2}.seq(:,1:9)';
207     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9'};
208     columnLabels = { '$x_1$', '$x_2$'};
209     matrix2latex(matrix, 'trseq2.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
210
211 %Starting Point 3
212 matrix = I3{3}.seq';
213     rowLabels = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '
        11', ...
214                  '12', '13', '14'};
215     columnLabels = { '$x_1$', '$x_2$'};
216     matrix2latex(matrix, 'trseq3.tex', 'rowLabels', rowLabels, '
        columnLabels', columnLabels, 'alignment', 'c', 'format', '%-6.2
        f', 'size', 'small');
217
218 %% LATEX Tables with TicToc, Fcalls, etc.
```