Steven T. Fisher
CS 791
MW 2:30 pm - 3:45 pm
HW3

Part I: Basics of convex optimization

Problem (1)**.** ***Gradient Descent Method***

OBJECTIVE FUNCTION:

$$minimize\ f(x) = \frac{1}{2}x^TQx + qx$$

where Q is an $n \times n$ positive definite matrix, and it is not necessarily a diagonal matrix. Note that the solution to this problem is $x^* = -Q^{-1}q$. As an alternative, design a gradient descent algorithm that will solve this unconstrained problem. Please pay attention to the following: with variable $x \in \mathbf{R}$

GRADIENT OF OBJECTIVE FUNCTION

$$\begin{aligned}\nabla f(x) &= \frac{\partial f(x)}{\partial x_i} \forall i = 1, 2, \ldots, n \\ &= Qx + q\end{aligned}$$

HESSIAN OF OBJECTIVE FUNCTION

$$\begin{aligned}\nabla^2 f(x) &= \frac{\partial^2 f(x)}{\partial x_i^2} \forall i = 1, 2, \ldots, n \\ &= Q\end{aligned}$$

(b)Generate figures that look like Fig 9.6 in the textbook with different starting points.

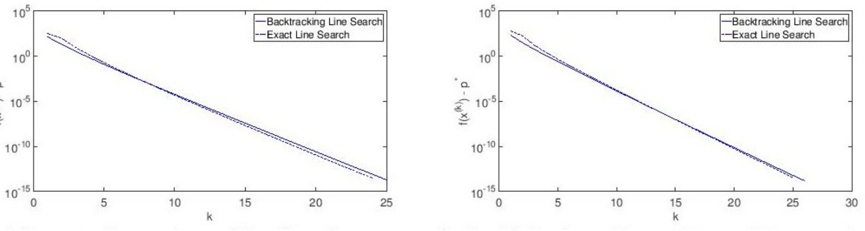

Figure 1: Comparison of Gradient Descent with Backtracking and Exact Line Search for different starting points.

(c) Discuss the effects of different α and β on the convergence of the Gradient Descent method with Backtracking Line search.

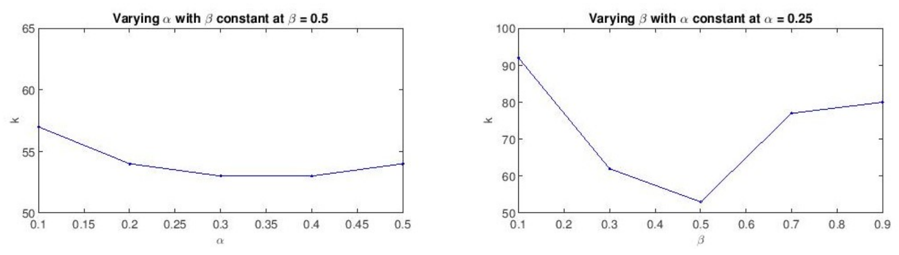

Figure 2: Comparison of the convergence of Backtracking line search with varying α and β

(d) Investigate the effect of problem size n on the convergence behavior of your algorithm

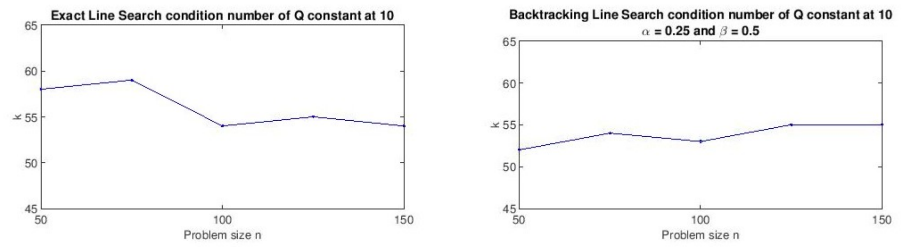

Figure 3: Comparison of the convergence of the convergence of the Gradient Descent method by varying problem size n

(e) Investigate the effect of the condition number of Q on the convergence behavior of your algorithm

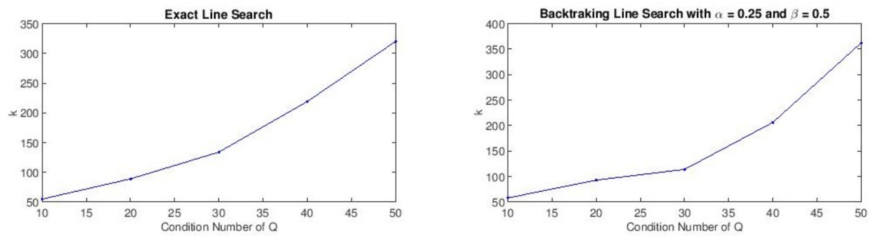

Figure 4: Comparison of the convergence of the Gradient Descent method by varying the condition number of Q

(f)Design a steepest descent algorithm with the choice of norm is given as given at the bottom of pp. 476 of our textbook, where P is selected as a diagonal matrix, whose diagonal elements are the same as those of Q. Is this new algorithm as sensitive to the condition number of Q?

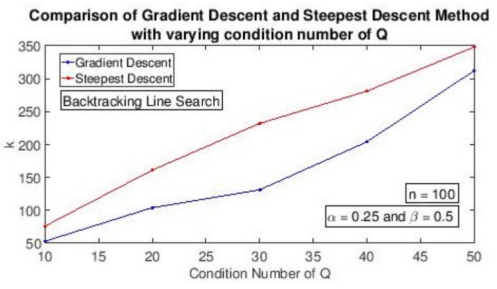

Figure 5: Comparison of the convergence of the Gradient Descent method and the Steepest Descent method varying the condition number of Q

(g) Comment on how the gradient descent algorithm can be used to solve a least-squares problem and give an example of a least-square problem tha tyou solve with this algorithm. Also, comment on the computational advantages of using the gradient descent algorithm as compared with just solving the linear system of equations $x^* = -Qq$. Do these advantages remain when the steepest descent approach described above is used? What are some potential disadvantages to using gradient descent, or steepest descent as compared with just solving the linear system of equations?

The least-squares problem is a special case of the quadratic minimization problem:

$$\text{minimize } ||Ax - b||_2^2 = x^T(A^TA)x - 2(A^Tb)^Tx + b^Tb$$

The optimality conditions for the least square problem is $A^TAx^* = A^Tb$. The gradient descent is a faster method of solving a system of linear equations. Calculating the inverse of a matrix $[Q^{-1}]$ requires $O(n^3)$ steps so for large n values, it is a slow process.

The Steepest descent method also requires calculation of inverse of a matrix $[P^{-1}]$ in order to determine the steepest descent direction so, it should alos be computationally slower than the gradient descent.

The potential disadvantage of using descent methods over just solving the system of linear equations is the phenomenon of descent methods zigzagging when the gradient is nearly orthogonal to the direction to the minimum point. This slow the convergence of the descent methods.

Problem (2). ***Newton's Method for Unconstrained Problems:***

Replicate Fig. 9.20 for the objective funciton in equation(9.20) in the textbook. Please include the expression for the gradient and the Hessian in your report. Also, add a few other initial starting points.

OBJECTIVE FUNCTION:

$$\textit{minimize } f(x) = e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1}$$

GRADIENT OF OBJECTIVE FUNCTION

$$\nabla f(x_1,x_2) = \begin{bmatrix} \dfrac{\partial f(x_1,x_2)}{\partial x_1} \\ \dfrac{\partial f(x_1,x_2)}{\partial x_2} \end{bmatrix}$$

where:

$$\begin{aligned} \frac{\partial f(x_1,x_2)}{\partial x_1} &= e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} - e^{-x_1-0.1} \\ \frac{\partial f(x_1,x_2)}{\partial x_2} &= 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} \end{aligned}$$

HESSIAN OF OBJECTIVE FUNCTION

$$\nabla f(x_1,x_2) = \begin{bmatrix} \dfrac{\partial^2 f(x_1,x_2)}{\partial x_1^2} & \dfrac{\partial^2 f(x_1,x_2)}{\partial x_1 x_2} \\ \dfrac{\partial^2 f(x_1,x_2)}{\partial x_2 x_1} & \dfrac{\partial^2 f(x_1,x_2)}{\partial x_2^2} \end{bmatrix}$$

where:

$$\begin{aligned} \frac{\partial^2 f(x_1,x_2)}{\partial x_1^2} &= e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} - e^{-x_1-0.1} \\ \frac{\partial^2 f(x_1,x_2)}{\partial x_1 x_2} &= 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} \\ \frac{\partial^2 f(x_1,x_2)}{\partial x_2 x_1} &= 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} \\ \frac{\partial^2 f(x_1,x_2)}{\partial x_2^2} &= 9e^{x_1+3x_2-0.1} + 9e^{x_1-3x_2-0.1} \end{aligned}$$

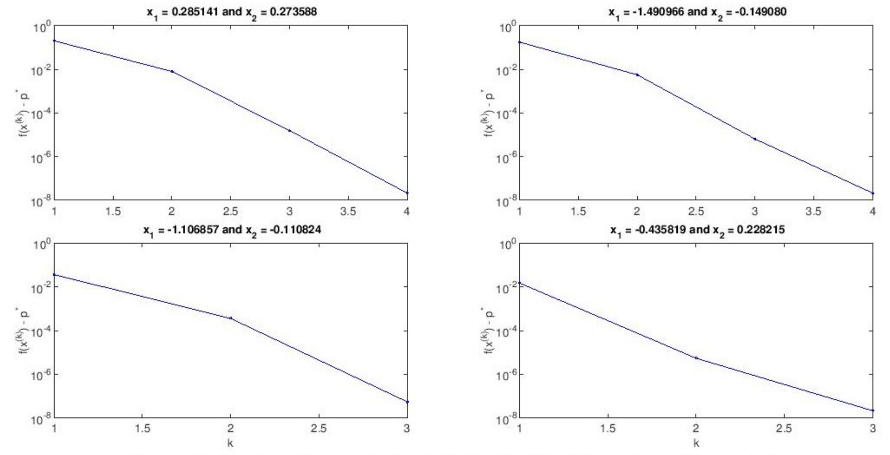

Figure 7: Newton Unconstrained Method with different starting points

Figure 6: Newton Unconstrained Method with different starting points

Problem (3). ***Newton's Method with Equality Constraints:***

OBJECTIVE FUNCTION:

$$\begin{array}{rc} \text{minimize} & f(x) = \sum_{i=1}^{n} x_i \log x_i \\ \text{subject to} & Ax = b \end{array}$$

with $\mathbf{dom} f = \mathbf{R}^n_{++}$ and $A \in \mathbf{R}^{p\times n}$, with $p < n$.

GRADIENT OF OBJECTIVE FUNCTION

$$\begin{aligned} \nabla f(x) &= \frac{\partial f(x)}{\partial x_i} \forall i = 1, 2, \ldots, n \\ &= 1 + \log x_i \end{aligned}$$

HESSIAN OF OBJECTIVE FUNCTION

$$\begin{aligned} \nabla^2 f(x) &= \begin{cases} \dfrac{\partial^2 f(x)}{\partial x_i^2} & \text{if } i = j \\ \dfrac{\partial^2 f(x)}{\partial x_i x_j} & \text{otherwise} \end{cases} \\ &= \mathbf{diag}\left(\frac{1}{x_i}\right) \end{aligned}$$

Generate a problem instance with $n = 100$ and $p = 30$ by choosing A randomly (checking that it has full rank), choosing $\widehat{x}$ as a random positive vector (e.g., with entries uniformly distributed on $[0, 1]$) and the setting $b = A\widehat{x}$. (Thus, $\widehat{x}$ is feasible).

Compute the solution of the problem using Infeasible start Newton Method. You can use initial point $x^{(0)} = \widehat{x}$ (to compare with the standard Newton Method), and also the initial point $x^{(0)} = \mathbf{1}$.

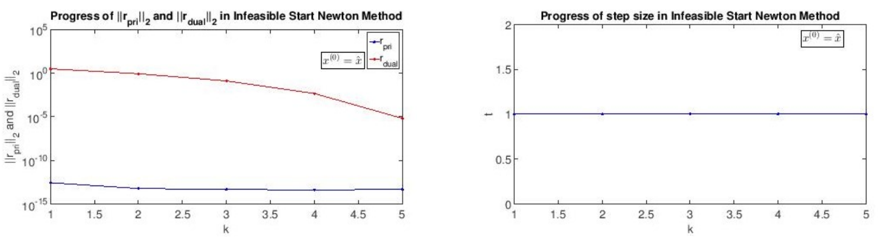

Figure 7: Progress of $||r_{pri}||_2$, $||r_{dual}||_2$ and step size t in the Infeasible Start Newton Method with $x^0 = \widehat{x}$

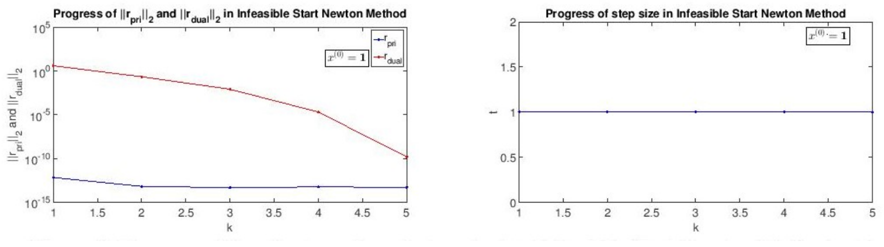

Figure 8: Progress of $||r_{pri}||_2$, $||r_{dual}||_2$ and step size t in the Infeasible Start Newton Method with $x^0 = \mathbf{1}$

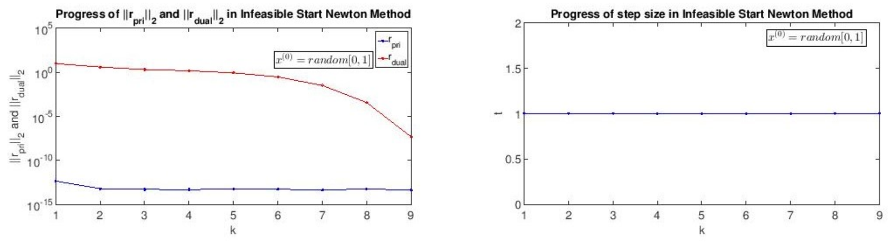

Figure 9: Progress of $||r_{pri}||_2$, $||r_{dual}||_2$ and step size t in the Infeasible Start Newton Method with $x^0 = random[0, 1]$

Problem (4)**. *Feasibility of an LP:***

Develop an algorithm to test the feasiblity of an LP using the approach in (11.19) where $f_i(x)$ are affine functions of x. Express the gradients and the Hessians needed for the algorithm explicitly in your report and describe the algorithm.

OBJECTIVE FUNCTION

$$\begin{gathered}\text{minimize } s\\ \text{subject to } Ax - b \preccurlyeq \mathbf{1}s\end{gathered}$$

where $x \in \mathbf{R^n}$, $s \in \mathbf{R}$, $x \in \mathbf{R^{m \times n}}$, and $b \in \mathbf{R^m}$.
For a basic phase I optimization problem, the problem must be restated in log-barrier form as follows:

$$\text{minimize } ts - \sum \log[\mathbf{1}s + b - Ax]$$

where $t \in \mathbf{R}$ is a scalar value used in the barrier method.

GRADIENT OF OBJECTIVE FUNCTION

$$\nabla f(x,s) = \begin{bmatrix} \dfrac{\partial f(x,s)}{\partial x} \\ \dfrac{\partial f(x,s)}{\partial s} \end{bmatrix}$$

where:

$$\begin{aligned} \frac{\partial f(x,s)}{\partial x} &= \frac{A^T}{\mathbf{1}s + b - Ax} \\ \frac{\partial f(x,s)}{\partial s} &= t - \sum \frac{1}{\mathbf{1}s + b - Ax} \end{aligned}$$

HESSIAN OF OBJECTIVE FUNCTION

$$\nabla^2 f(x,s) = \begin{bmatrix} \dfrac{\partial^2 f(x,s)}{\partial x^2} & \dfrac{\partial^2 f(x,s)}{\partial x \partial s} \\ \dfrac{\partial^2 f(x,s)}{\partial s \partial x} & \dfrac{\partial^2 f(x,s)}{\partial s^2} \end{bmatrix}$$

where:

$$\begin{aligned} \frac{\partial^2 f(x,s)}{\partial x^2} &= \begin{cases} \dfrac{a_{ij}^2}{\mathbf{1}s + b - Ax^2} & \text{if } i = j \\ \dfrac{\prod_{j=1}^n a_{ij}}{\mathbf{1}s + b - Ax} & \text{otherwise} \end{cases} \\ \frac{\partial^2 f(x,s)}{\partial x \partial s} &= \frac{-A^T}{\mathbf{1}s + b - Ax^2} \\ \frac{\partial^2 f(x,s)}{\partial s \partial x} &= \frac{-A^T}{\mathbf{1}s + b - Ax^2} \\ \frac{\partial^2 f(x,s)}{\partial s^2} &= \sum_{i=1}^m \frac{1}{\mathbf{1}s + b - Ax^2} \end{aligned}$$

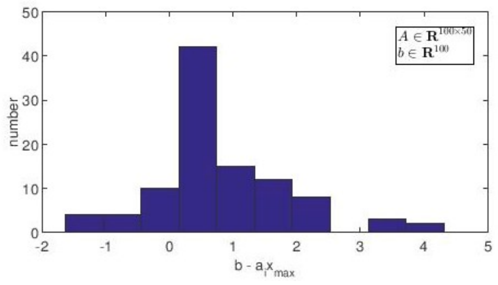

Figure 10: Distribution of $b_i - a_i x_{max}$ for a set of inequalitites $Ax \preccurlyeq b$ where $A \in \mathbf{R^{100\times 50}}$ and $b \in \mathbf{R^{100}}$

EFFECT OF μ

For small μ, the barrier method parameter t increases by a small amount in each iteration; so, the iterations are fairly close to each other, and tend to the central path.

For large μ, the barrier method parameter t increases by a large amount; so, each iteration is not a good approximation of the next iteration. This means that we would require more Newton Method iterations to minimize the log-barrier optimization function.

Problem (5). ***A cvx Experiment in Compressive Sensing:***

Consider the following problem

$$\begin{array}{rl} \text{minimize} & ||x||_1 \\ \text{subject to} & y = \Phi x \end{array}$$

where Φ is a $k \times n$ matrix with $k << n$, and x is a vector with only S nonzero elements. The interpretation is that y constitutes our k measurements of a sparse vector x, through a "measuremnts matrix" Φ. Since Φ is fat, there are infinitely many vectors x that satisfy the equality constraint in the problem, so we cannot determine x uniquely, only from that constraint. But if know that x is sparse (i.e., only S nonzero elements of n), and if k is large enough (i.e., we have enough 'projections' of x), then the optimization problem aboe can uniquely determine x. Note that we do not need to know the sparsity pattern (which elements of x are nonzero).

For the purposes of this problem, we will assume that Φ consists of rows of a DFT matrix which has an element in the l^{th} row and m^{th} column given by $[\Phi]_{l,m} = n^{-1/2} \exp(-j2\pi f_l m/N)$, where $f_l \in \{0, 1, \ldots, n-1\}$ for $l = 1, \ldots, k$. The interpretation is that x is being "snsed" or "sampled" in k different frequencies. Recall that $k < n$, so we have much fewer frequency samples than the length of x, hence the name compressed sampling or compressed sensing. An interesting result in conncetion with the problem above is that if k frequencies are chosedn randomly and uniformly in the set $\{0, 1, \ldots, n-1\}$, and the number of frequency samples are at least

$$k > CS \log n$$

then the above optimization problem has a solution which will reconstruct x perfectly, with high probability. The idea is that for almost all selection of k frequencies out of the n possible, perfect reconstruction of the sparse x is possible with much fewer than n, but more than $CS \log n$ samples, where C is just some constant, independent of n and k. In short, if k is sufficiently large, a random selection of frequencies will with very high probability yield a sampling matrix Φ that will generate a y which can be used to recover x perfectly using the optimization problem above. This probability will approach 1 very rapidly expecially if n is large as well. However, even when $k > CS \log n$ is satisfied, it is possible to select the frequencies such that we cannot reconstruct x (even though this occurs with ever smaller probabiliy when n is large).

(a) Give an example of a set of k frequencies for which we would only be measuring a y vector that would all be zeros, even though k could be high as $n - S$ (which is clearly $> CS \log n$). To find such an example, think of a comb-shaped signal in discrete-time.

The measured vector y will be all zeros if the following conditions are true for the parameters n, S, k, and the vector x.

- The parameter n is square
- The number of non-zero elements in $x, S = \sqrt{n}$
- The number of frequencies selected, $k = n - S = n - \sqrt{n}$
- The set of frequencies selected, $\{f_l \in \{0, 1, \ldots, n-1\} : f_l \neq 0, f_l \neq \frac{pn}{S} \quad \forall \quad p \in \{1, 2, \ldots\}\}$
- The vector x is a comb shaped function with S spikes separated by $\sqrt{n}$

An example would be:

- $n = 64$
- $S = \sqrt{n} = 8$
- $k = n - S = 64 - 8 = 56$
- $f_1 = \{1, 2, \ldots, 63, 64\}/\{8, 16, 24, 32, 40, 48, 56, 64\}$
- $x = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, \ldots]$

(b) Select n, S, and generate Φ randomly using the description above, where the k frequencies are selected randomly, uniformly distributed in $\{1, 1, \ldots, n-1\}$. Use cvx to recover x. For values of $n = 50$, and $n = 100$, determine the smallest value of k that will recover x perfectly from randomly selected frequencies. different starting points.

n	S	k
50	10	26
	15	33
	25	43
	25	46
	30	48
100	20	43
	30	58
	40	73
	50	90
	60	98

Table 1: Empircally determined values of k

Now, table 1 presents empircally determiend values of k for the perfect recovery of x, These k values are able to recover x with a fairly high precision(four positions after our decimal point) after 300 test iterations.

(c) Suppose we now have noisy measurements so that $y = \Phi x + \eta$, where η is an unknown noise vector that satisfies $||\eta||_2 \leq \epsilon$. Use a Φ matrix and k value and set of frequnecies that enable perfect recovery in the noiseless case to generate your data. For this data solve the following problem in cvx.

$$\begin{array}{ll} \text{minimize} & ||x||_1 \\ \text{subject to} & ||y - \Phi x||_2 \leq \epsilon \end{array}$$

Plot $||x^* - x||_2$ versus ϵ for a resonable range of ϵ, keeping Φ, x the same.

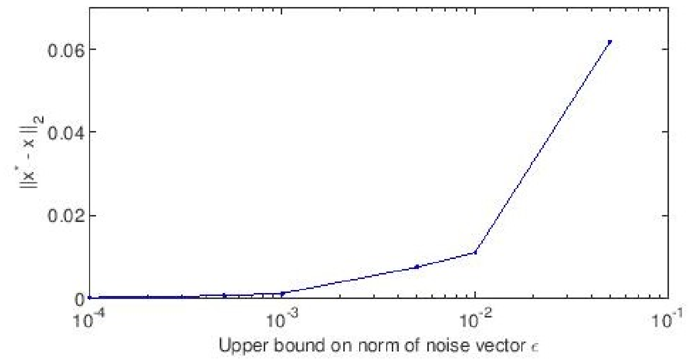

Figure 11: Effet of noise on the recovery of x

Here we can see that as the bound of the nor of the error vecto, ϵ, is increased. Then the difference between the recovered and actual value of x increases.

Part II: Application of convex optimization on your research Please find a research problem in your area that requires optimizaiton techniques to solve the problem. For this problem, you need to explain the problem and formulate it as an optimization problem. Then you need to discuss how this problem is solved.

Note: The problem does not have to be convex and you may use other techniques beyond this course.

For the question for part 2, I decided to consider one of the application problems from the additional excercises for our textbook. The problem is 15.6 and is stated as follows:

Maximizing Algebraic Connectivity of a Graph

Let $G = (V, E)$ be a weighted undirected graph with $n = |V|$ nodes, $m = |E|$ edges, and weights $w_1, \ldots, w_m \in \mathbf{R}_+$ on the edges. If edge k connects nodes i and j, then define $a_k \in \mathbf{R^n}$ as $(a_k)_i = 1, (a_k)_j = -1$, with other entries zero. The weighted Laplacian (matrix) of the graph is defined as

$$L = \sum_{k=1}^{m} w_k a_k a_k^T = A\mathbf{diag}(w)A^T$$

where $A = [a_1 \ldots a_m] \in \mathbf{R^{n\times m}}$ is the incidence matrix of the graph. Nonnegativity of the weights implies $L \succeq 0$.

Denote the eigenvalues of the Laplacian L as

$$\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$$

which are functions of w. The minimum eigenvalue λ_1 is always zero, while the second smallest eigenvalue λ_2 is called the algebraic connectivity of G and is a measure of the connectedness of a graph: The larger λ_2 is, the better connected the graph is. It is often used, for example, in analyzing the robustness of computer networks.

Though not relevant for the rest of the problem, we mention a few other examples of how the algebraic connectivity can be used. These results, which relate graph-theoretic properties of G to properties of the spectrum of L, belong to a field called spectral graph theory. For example, $\lambda_2 > 0$ if and only if the graph is connected. The eigenvector ν_2 associated withλ_2 is often called the Fiedler vector and is widely used in a graph partitioning technique called spectral partitioning, which assigns nodes to one of two groups based on the sign of the relevant component in ν_2 . Finally, λ_2 is also closely related to a quantity called the isoperimetric number or Cheeger constant of G, which measures the degree to which a graph has a bottleneck.

The problem is to choose the edge weights $w \in \mathbf{R^m_+}$, subject to some linear inequalities (and the nonnegativity constraint) so as to maximize the algebraic connectivity:

$$\begin{array}{ll} \text{minimize} & \lambda_2 \\ \text{subject to} & w \succeq 0, Fw \preceq g \end{array}$$

with variable $w \in \mathbf{R^m}$. The problem data are A (which gives the graph topology), and F and g (which describe the constraints on the weights).

(a) Describe how to solve this problem using convex optimization.

Here we are going to first make sure that λ_2 is a concave function of w. So, we will need to construct a matrix so that the smallest eigenvalue is λ_2, since this will help use make sure that λ_2 is concave.

Now, we will note that $\mathbf{1}$ is an eigenvector of L. Now, if we were to restrict our space to $\mathbf{1}^\perp$, then we can express $\lambda_2 = \lambda_{min}(Q^T LQ)$, where $Q \in \mathbf{R^{n\times(n-1)}}$. Now, Q is therefore a matrix whose columns for an orthogonal basis for $N(\mathbf{1}) = \mathbf{1}^\perp$, so that $Q^T LQ$ has eigenvalues $\lambda_2(L), \ldots, \lambda_n(L)$. So, our problem is now

$$\begin{array}{ll} \text{maximize} & \lambda_{min}(Q^T LQ) \\ \text{subject to} & w \succeq, \quad Fw \preceq g \end{array}$$

(b) Numerical example. Solve the problem instance given in max_alg_conn_data.m, which uses $F = \mathbf{1}^T$ and $g = 1$ (so the problem is to allocate a total weight of 1 to the edges of the graph). Compare the algebraic connectivity for the graph obtained with the optimal weights w^* to the one obtained with $w^{unif} = (1/m)\mathbf{1}$ (i.e., a uniform allocation of weight to the edges). Use the function plotgraph(A,xy,w) to visualize the weighted graphs, with weight vectors w^* and w^{unif} . You will find that the optimal weight vector v^* has some zero entries (which due to the finite precision of the solver, will appear as small weight values); you may want to round small values (say, those under 10 4) of w^* to exactly zero. Use the gplot function to visualize the original (given) graph, and the subgraph associated with nonzero weights in w^*. Briefly comment on the following (incorrect) intuition: The more edges a graph has, the more connected it is, so the optimal weight assignment should make use of all available edges.

Appendix Code used in problems Problem 1

```
% Gradient Descent FUnction with Backtracking Line Search
% Input
%   param_Q : nxn positive definite matrix
%   param_q : nx1 column vector
% Output
%   ret_x : optimal minimized x
%   ret_itr : number of iterations
```

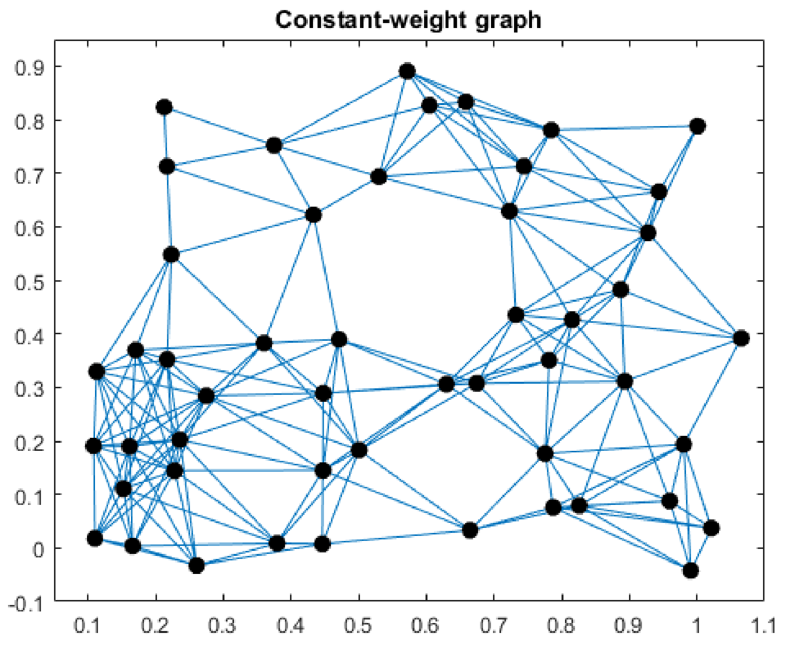

Figure 12: Comparison of the convergence of Backtracking line search with varying α and β

```
%   ret_diff_f : difference in approximated objective
%                function value from actual value for
%                each iteration
% Optimation Problem
%    minimize f(x) = (1/2)x'Qx + q'x

function [ret_x, ret_itr, ret_diff_f] = gradientDescent1( param_Q, param_q, x )

% Correct solution for x
x_act = -param_Q \ param_q;
f_act = (1/2) * x_act' * param_Q * x_act + param_q' * x_act;

% Backtracking lien search constants
alpha = 0.25;
beta = 0.5;

% Stopping Condition Limit
inta = 0.000001;
```

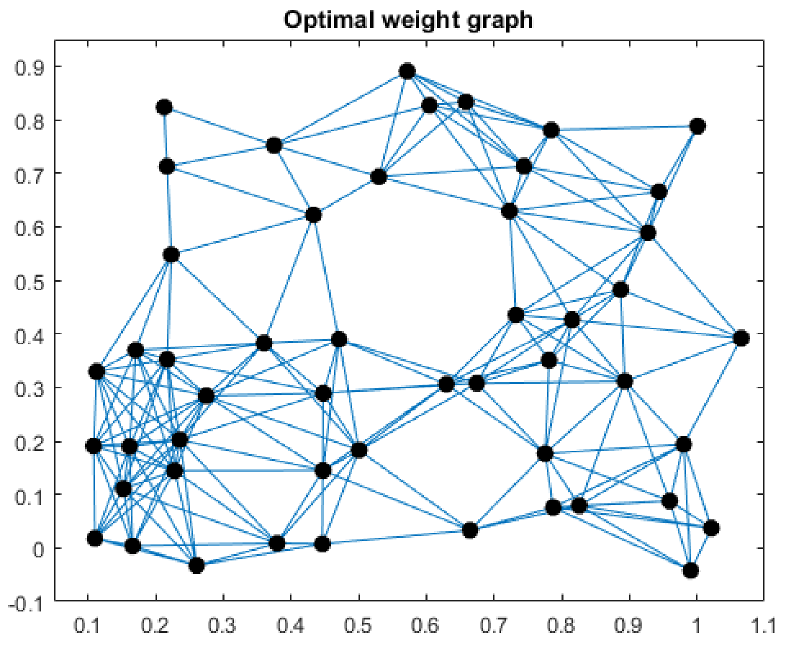

Figure 13: Comparison of the convergence of Backtracking line search with varying α and β

```
%Size of the input parameters
size_Q = size( param_Q);
disp( 'Size of matrix Q')
disp(size_Q)


size_q = size( param_q);
disp( 'Size of vector q')
disp(size_q)

% Make sure param_Q is a positive definite matrix
eigen_Q = eig( param_Q );
if ~( all(eigen_Q) > 0)
    disp('Parameter Q is not a positive definite matrix')
end

% Condition number of positive definite matrix param_Q
eigen_max_Q = max( eigen_Q );
eigen_min_Q = min( eigen_Q );
```

```
cond_num_ub = eigen_max_Q / eigen_min_Q;

disp('Condition number of the positive definite matrix Q is ')
disp(cond_num_ub);

% Make sure param_Q and param_q have correct dimensions
if ( size_Q(1) ~= size_Q(2) || size_Q(1) ~= size_q(1) )
    disp('Parameter Q and q have incorrect dimensions')
end

% Pick a random starting poitn x in dom(f)
%x = rand( size_q(1), 1);

% Gradient of objective function
grad_f = param_Q * x + param_q;

itr = 0;

while norm( grad_f ) > inta

    % Gradient of objective function
    grad_f = param_Q * x + param_q;

    % Determine the search direction
    search_dir = -grad_f;

    % Choose step-size using backtracking line search
    t = 1;
    track_x = x + t * search_dir;
    track_f = (1/2) * track_x' * param_Q * track_x + param_q' * track_x;
    track_f_min = (1/2) * x' * param_Q * x + param_q' * x + alpha * t * grad_f'

    while track_f > track_f_min
        t = beta * t;
        track_x = x + t * search_dir;
        track_f = (1/2) * track_x' * param_Q * track_x + param_q' * track_x;
        track_f_min = (1/2) * x' * param_Q * x + param_q' * x + alpha * t * grad
    end

    %Update x with new value
    x = x + t * search_dir;

    %Update iteration count
    itr = itr + 1;

    % Difference in the objective funtion value fro the ideal value
    f_itr = (1/2) * x' * param_Q * x + param_q' * x;
    diff_f( itr ) = f_itr - f_act;
end

%Return from function
ret_x = x;
ret_itr = itr;
ret_diff_f = diff_f;

% Gradient Descent FUnction with Exact Line Search
% Input
%   param_Q : nxn positive definite matrix
%   param_q : nx1 column vector
% Output
%   ret_x : optimal minimized x
%   ret_itr : number of iterations
%   ret_diff_f : difference in approximated objective
%                function value from actual value for
%                each iteration
% Optimation Problem
%    minimize f(x) = (1/2)x'Qx + q'x

function [ret_x, ret_itr, ret_diff_f] = gradientDescent2( param_Q, param_q, x )

% Correct solution for x
x_act = -param_Q \ param_q;
f_act = (1/2) * x_act' * param_Q * x_act + param_q' * x_act;

% Stopping Condition Limit
inta = 0.000001;

% Exact Line Search Limit
epsilon = 0.000001;

%Size of the input parameters
size_Q = size( param_Q);
disp( 'Size of matrix Q')
disp(size_Q)


size_q = size( param_q);
disp( 'Size of vector q')
disp(size_q)

% Make sure param_Q is a positive definite matrix
eigen_Q = eig( param_Q );
```

```
if ~( all(eigen_Q) > 0)
    disp('Parameter Q is not a positive definite matrix')
end

% Condition number of positive definite matrix param_Q
eigen_max_Q = max( eigen_Q );
eigen_min_Q = min( eigen_Q );
cond_num_ub = eigen_max_Q / eigen_min_Q;

disp('Condition number of the positive definite matrix Q is ')
disp(cond_num_ub);

% Make sure param_Q and param_q have correct dimensions
if ( size_Q(1) ~= size_Q(2) || size_Q(1) ~= size_q(1) )
    disp('Parameter Q and q have incorrect dimensions')
end

% Pick a random starting poitn x in dom(f)
%x = rand( size_q(1), 1);

% Gradient of objective function
grad_f = param_Q * x + param_q;

itr = 0;

while norm( grad_f ) > inta

    % Gradient of objective function
    grad_f = param_Q * x + param_q;

    % Determine the search direction
    search_dir = -grad_f;

    % Choose step-size using exact line search
    t_low = 0;
    t_up = 1;

    track_f = param_Q * ( x + t_up * search_dir ) + param_q;

    while( ( search_dir' * track_f ) < 0)
        t_low = t_up;
        t_up = t_up * 2;
    end

    while( ( t_up - t_low) > epsilon )
        t_mid = (1/2) * (t_low + t_up);
```

```
            track_f = param_Q * (x + t_up * search_dir) + param_q;
            if((search_dir' * track_f) < 0)
                t_low = t_mid;
            else
                t_up = t_mid;
            end
            t = (1/2) * (t_low + t_up);
        end

        %Update x with new value
        x = x + t * search_dir;

        %Update iteration count
        itr = itr + 1;

        % Difference in the objective funtion value fro the ideal value
        f_itr = (1/2) * x' * param_Q * x + param_q' * x;
        diff_f( itr ) = f_itr - f_act;
    end

    %Return from function
    ret_x = x;
    ret_itr = itr;
    ret_diff_f = diff_f;

    % Steepest Descent FUnction with Backtracking Line Search
    % Input
    %   param_Q : nxn positive definite matrix
    %   param_q : nx1 column vector
    % Output
    %   ret_x : optimal minimized x
    %   ret_itr : number of iterations
    %   ret_diff_f : difference in approximated objective
    %                function value from actual value for
    %                each iteration
    % Optimation Problem
    %    minimize f(x) = (1/2)x'Qx + q'x

    function [ret_x, ret_itr, ret_diff_f] = steepestDescent( param_Q, param_q, x )

    % Correct solution for x
    x_act = -param_Q \ param_q;
    f_act = (1/2) * x_act' * param_Q * x_act + param_q' * x_act;

    % Symmetric positive definite matrix P from matrix Q
    P = diag(diag(param_Q));
```

```
% Backtracking line search constants
alpha = 0.25;
beta = 0.5;

% Stopping Condition Limit
inta = 0.000001;

% Exact Line Search Limit
epsilon = 0.000001;

%Size of the input parameters
size_Q = size( param_Q);
disp( 'Size of matrix Q')
disp(size_Q)


size_q = size( param_q);
disp( 'Size of vector q')
disp(size_q)

% Make sure param_Q is a positive definite matrix
eigen_Q = eig( param_Q );
if ~( all(eigen_Q) > 0)
    disp('Parameter Q is not a positive definite matrix')
end

% Condition number of positive definite matrix param_Q
eigen_max_Q = max( eigen_Q );
eigen_min_Q = min( eigen_Q );
cond_num_ub = eigen_max_Q / eigen_min_Q;

disp('Condition number of the positive definite matrix Q is ')
disp(cond_num_ub);

% Make sure param_Q and param_q have correct dimensions
if ( size_Q(1) ~= size_Q(2) || size_Q(1) ~= size_q(1) )
    disp('Parameter Q and q have incorrect dimensions')
end

% Pick a random starting poitn x in dom(f)
%x = rand( size_q(1), 1);

% Gradient of objective function
grad_f = param_Q * x + param_q;
```

```
itr = 0;

while norm( grad_f ) > inta

    % Gradient of objective function
    grad_f = param_Q * x + param_q;

    % Determine the search direction
    search_dir = -P \ grad_f;

    % Choose step-size using backtracking line search
    t = 1;
    track_x = x + t * search_dir;
    track_f = (1/2) * track_x' * param_Q * track_x + param_q' * track_x;
    track_f_min = (1/2) * x' * param_Q * x + param_q' * x + alpha * t * grad_f'

    while track_f > track_min
        t = beta * t;
        track_x = x + t * search_dir;
        track_f = (1/2) * track_x' * param_Q * track_x + param_q' * track_x;
        track_f_min = (1/2) * x' * param_Q * x + param_q' * x + alpha * t * grad
    end

    %Update x with new value
    x = x + t * search_dir;

    %Update iteration count
    itr = itr + 1;

    % Difference in the objective funtion value fro the ideal value
    f_itr = (1/2) * x' * param_Q * x + param_q' * x;
    diff_f( itr ) = f_itr - f_act;
end

%Return from function
ret_x = x;
ret_itr = itr;
ret_diff_f = diff_f;
```

Problem 2

```
% Newton's Method For Unconstrained Problems
% Optimization Problem
%    minimize f(x1, x2) = exp( x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) +
%    exp(-x1 - 0.1)

% Solve the system with cvx
```

```
cvx_begin
    variable cvx_x1
    variable cvx_x2
    minimize( exp( x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) + exp(-x1 - 0.1))
cvx_end

f_act = cvx_optval;

% Choose a random startign point for x1 and x2
startx1 = randn;
startx2 = randn;

x1 = startx1;
x2 = startx2;

% Stopping condition for Newton's Method
epsilon - 0.00001;

% Backtracking line search constants
alpha = 0.1;
beta = 0.7;

% Calculate the gradient of the objective function
grad_f_1 = exp( x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) + exp(-x1 - 0.1);
grad_f_2 = 3 * exp(x1 + 3 * x2 - 0.1) - 3 * exp(x1 - 3 * x2 - 0.1);

grad_f = [ grad_f_1 ; grad_f_2 ];

% Calculate the hessian of the objective function
hess_f_11 = exp ( x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) + exp( -x1 - 0.1);
hess_f_12 = 3 * exp( x1 + 3 * x2 - 0.1) - 3 * exp(x1 - 3 * x2 - 0.1);
hess_f_21 = 3 * exp( x1 + 3 * x2 - 0.1) - 3 * exp(x1 - 3 * x2 - 0.1);
hess_f_22 = 9 * exp( x1 + 3 * x2 - 0.1) + 9 * exp(x1 - 3 * x2 - 0.1);

hess_f = [hess_f_11 hess_f_12 ; hess_f_21 hess_f_22];

% Calculate the Newton search direction and the stopping condition
n_search_dir = - ( hess_f \ grad_f );
stop_cond = - (grad_f' * n_search_dir);

itr = 0;

while stop_cond > epsilon
    %Choose the step-size using backtracking line search
    t = 1;
    track_x1 = x1 + t * n_search_dir(1);
    track_x2 = x2 + t * n_search_dir(2);
    track_f = exp( track_x1 + 3 * track_x2 - 0.1) + exp(track_x1 - 3 * track_x2 
    track_f_min = exp( x1 + 3 * x2 - 0.1) + exp( x1 - e * x2 - 0.1) + exp(-x1 - 

    while track_f > track_f_min
        t = beta * t;
        track_x1 = x1 + t * n_search_dir(1);
        track_x2 = x2 + t * n_search_dir(2);
        track_f = exp( track_x1 + 3 * track_x2 - 0.1) + exp(track_x1 - 3 * track
        track_f_min = exp( x1 + 3 * x2 - 0.1) + exp( x1 - e * x2 - 0.1) + exp(-x
    end

    % Update x with the new value
    x1 = x1 + t * n_search_dir(1);
    x2 = x2 + t * n_search_dir(2);

    % Calculate the gradient with the new value of x
    grad_f_1 = exp( x1 + 3 * x2 - 0.1) + exp( x1 - 3 * x2 - 0.1) - exp(-x1 - 0.1
    grad_f_2 = 3 * exp(x1 + 3 * x2 - 0.1) - 3*exp(x1 - 3 * x2 - 01);

    grad_f = [ grad_f_1 ; grad_ f_2 ];
    % Calculate the hessian of the objective function
    hess_f_11 = exp ( x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) + exp( -x1 - 0
    hess_f_12 = 3 * exp( x1 + 3 * x2 - 0.1) - 3 * exp(x1 - 3 * x2 - 0.1);
    hess_f_21 = 3 * exp( x1 + 3 * x2 - 0.1) - 3 * exp(x1 - 3 * x2 - 0.1);
    hess_f_22 = 9 * exp( x1 + 3 * x2 - 0.1) + 9 * exp(x1 - 3 * x2 - 0.1);

    hess_f = [hess_f_11 hess_f_12 ; hess_f_21 hess_f_22];

    % Calculate the Newton search direction and stopping condition with new
    % value of x
    n_search_dir = - (hess_f \ grad_f);
    stop_cond = -(grad_f' * n_search_dir);

    % Update iteration count
    itr = itr + 1;

    % Difference in objective function value from ideal value for iteration
    f_itr = exp(x1 + 3 * x2 - 0.1) + exp(x1 - 3 * x2 - 0.1) + exp(-x1 - 0.1);
    diff_f(itr) = f_itr - f_act;
end

% Plot the progression of difference of optimal value obtained from
% Newton's Method with actual optimal value
semilogy(1 : itr, diff_f , 'b-o');
set( gcam 'FontSize', 36 );
```

```
titlestr = sprintf(' x_1 = %f and x_2 = %f', startx1, startx2);
title(titlestr, 'FontSize', 36);
xlabel('k', 'FontSize', 36);
ylabel('f(x^{(k)}) - p^*', 'FontSize', 36);
```

Problem 3

```
% Newton's method with Equality Constraints
% Optimization Problem
%   minimize f(x) = Sum( x_i * log x_i ) for i = 1 to n
%         s. t. Ax = b
% where A = [pxn] matrix with p < n and full rank
function [ ret_x, ret_itr, ret_rp, ret_rd, ret_stp ] = newtonEqConstrProblem( A,

% Stopping Condition Limit
epsilon = 0.00001;

% Backtracking line search constants
alpha = 0.1;
beta = 0.7;

% Size of the Coefficient Matrix
size_n = 100;
size_p = 30;

% Gradient of objective function for x
grad_f = 1 + log(x);

% RHS of the KKT system
r_dual = grad_f + (A' * v);
r_primal = (A * x) - b;
r = [r_dual ; r_primal];

% A * x = b stop flag
stop = 0;

% Number of iterations
itr = 0;

% Hessian matrix definition
Hessian_f = zeros( size_n, size_n );

% Stopping conditions for Infeasible Start Newton Method
while ( ~stop && norm(r) > epsilon)

    % Hessian of objective function for x
    for index1 = 1 : size_n
    for index2 : size_n
        if index1 == index2
            Hessian_f( index1, index2) = 1/ x( index1 );
        else
            Hessian_f( index1, index2) = 0;
        end
    end
end

% Schur Complement
Schur_Cmpl = -A * ( Hessian_f \ A');

% Primal and Dual Newton Search Directions
dual_n_search_dir = Schur_Cmp1 \ ( ( A * ( Hessian_f \ r_dual ) ) - r_primal
prime_n_search_dir = Hessian_f \ ( ~ ( A' * dual_n_search_dir ) - r_dual );

% Backtracking Line Search
t = 1;
track_x = x + t * prime_n_search_dir;
track_v = v + t * dual_n_search_dir;

track_grad_f = 1 + log( track_x );

track_r_dual = track_grad_f + A' * track_v;
track_r_primal = A * track_x - b;

track_r = [ track_r_dual ; track_r_primal ];
track_r_min = r;

while norm( track_r ) > ( 1 - alpha * t ) * norm( track_r_min )
    t = beta * t;
    track_x = x + t * prime_n_search_dir;
    track_v = v + t * dual_n_search_dir;

    track_grad_f = 1 + log( track_x );

    track_r_dual  = track_grad_f + A' * track_v;
    track_r_primal = A * track_x - b;

    track_r = [ track_r_dual ; track_r_primal ];
    track_r_min = r;
end

% Update new values for x and v
x = x * t * prime_n_search_dir;
v = v * t * dual_n_search_dir;
```

```
    % Gradient of objective function for new value of x
    grad_f = 1 + log( x );

    %Update RHS of the KKT system with new values for x and v
    r_dual = grad_f + A' * v;
    r_primal = A * x - b;
    r = [ r_dual ; r_primal ];

    % A * x = b stop flag
    if ((A * x ) == b )
        stop = 1;
    end

    itr = itr + 1;

    % Save the progression of step-size primal and dual variables
    rp(:, itr) = r_primal;
    rd(:, irt) = r_dual;
    stp(itr) = t;
end

ret_x = x;
ret_itr = itr;
ret_rp = rp;
ret_rd = rd;
ret_stp = stp;
```

Problem 4

```
% Newton's Unconstrained Method for Log-Barrier problem
% Optimization Problem
%   minimize f(x,s) = ts - sum( log (-A*x + b + 1s))
% where A = [mxn] matrix with n < m
function[ ret_x , ret_s , ret_itr] = newtonLP( t, A, b, start_x , start_s)

% Maximum number of iterations
max_itr = 100;

% Stopping condition for Newton Method
epsilon = 0.00001;

% Backtracking line search constants
alpha = 0.25;
beta = 0.5;

% Size of the coefficient matrix A
```

```
[size_m, size_n] = size( A );

% Set the initial valuels of the starting points
x = start_x;
s = start_s;

% Calculate the value of -f(x,s)
d = -A * x + b + s * ones( size_m, 1);

% Calculate the gradient of the objective function
grad_fx = A' * (1./d);
grad_fs = t - sum(1./d);

grad_f = [ grad_fx ; grad_fs ];

% Calcuate the hessian of the objective function
hess_fxx = zeros( size_n, size_n );

for index1 = 1 : size_n
    for index2 = 1 : size_n
        if index1 == index2
            hess_fxx( index1, index2 ) = ((A( :, index2 )).^2) * (1./d.^2);
        else
            hess_fxx( index1, index2 ) = prob(A,2)' * (1./d.^2);
        end
    end
end

hess_fxs = -A' * (1./d.^2);
hess_fss = sum(1./d.^2);

hess_f = [hess_fxx hess_fxs; hess_fxs' hess_fss];

% Calculate search direction and stopping condition for Newton's Method
n_search_dir = -(hess_f\grad_f);
stop_cond = -(grad_f' * n_search_dir);

% Iteration counter for number of Newton Steps
itr = 0;

while(stop_cond > epsilon && itr < max_itr)
    % Choose step-size using the backtracking line search
    track = 1;
    track_x = x * n_search_dir(1 : size_n, 1);
    track_s = s + track * n_search_dir(size_n + 1, 1);
    track_d = -A * track_x + b + track_s * ones( size_m, 1);
```

```
track_f = t*track_s - sum(log(track_d));
track_f_min = t * s - sum(log(d)) + alpha * track * grad_f' * n_search_dir;

while track_f > track_f_min
    track = beta * track;
    track_x = x * n_search_dir(1 : size_n, 1);
    track_s = s + track * n_search_dir(size_n + 1, 1);
    track_d = -A * track_x + b + track_s * ones( size_m, 1);
    track_f = t*track_s - sum(log(track_d));
    track_f_min = t * s - sum(log(d)) + alpha * track * grad_f' * n_search_d
end

% Update x and s with new values
x = x + track * n_search_dir(1 : size_n, 1);
s = s + track * n_search_dir(size_n + 1, 1);

% Calcuate the value of -f(x,s) with the new x and s
d = -A * x + b + s * ones(size_m, 1);

% Calculate the gradient of the objective function with new x and s
grad_fx = A' * (1./d);
grad_fs = t - sum(1./d);

grad_f = [grad_fx ; grad_fs];

% Calcuate the hessian of the objective function
for index1 = 1 : size_n
    for index2 = 1 : size_n
        if index1 == index2
            hess_fxx( index1, index2 ) = ( ( A( :, index2 )).^2) * (1./d.^2)
        else
            hess_fxx( index1, index2 ) = prob(A,2)' * (1./d.^2);
        end
    end
end

hess_fxs = -A' * (1./d.^2);
hess_fss = sum(1./d.^2);

hess_f = [hess_fxx hess_fxs; hess_fxs' hess_fss];

% Calculate search direction adn stopping condition for the Newton's
% Method with new x and s
n_search_dir = -(hess_f \ grad_f);
stop_cond = -(grad_f' * n_search_dir);
```

```
    % Update iteration count
    itr = itr + 1
end

ret_x = x;
ret_s = s;
ret_itr = itr;

% Log-Barrier Method for Testing Feasiblility of LP (Phase I Optimization)

% Problem size variables
size_m = 100;
size_n = 50;

% Generate random coefficient matrix and RHS
A = randn(size_m, size_n);
b = randn(size_m, 1);

% Random initial value for x
start_x = randn(size_n, 1);

% Initial value for s based on constraints
start_s = 2 * max(A * start_x - b);

% Barrier method variables
t = 1;
mu = 10;
epsilon = 0.000001;

% Counter for indexing
index = 0;

% Flag to end iterations
stop = 0;

% Counter for number of inequalities satisfied by x
numIneqSat = 0;

% Optimal values from previous iteration
x_1 = start_x;
s_1 = start_s;

while(~stop)
    %Centering Step
    [x, s, itr] = newtonLP(t, A, b, x_1, s_1);
    index = index + 1;

    % Test the number of inequalities satisfied by x
    numIneqSat(index) = sum(A * x < b);

    % Tracking progression of x and s through each iteration
    x_t(:, index) = x;
    s_t(:, index) = s;
    itr_str(index) = itr;

    %Set the optimal values of the previous iteration as starting points
    %for the next iteration
    x_1 = x;
    s_1 + s;

    % Test the stopping condition
    if( x< 0 || abs(size_m/t) < epsilon)
        stop = 1;
    else
        t - mu * t;
    end
end
```

Problem 5

```
% Compressive Sensing
% Optimization Problem
%    minimize ||x||_1
%        s.t. y = phi * x
% where phi = [k,n] matrix with k << n
%         x = [n,1] vector with S < n non-zero elements

% Size of x
size_n = 64;

% Number of non-zero elements in x
nz_S = 8;

% Vector x as a comb shape of size nz_S
x = zeros(size_n , 1);
for index1 = 1 : nz_S;
    x(index1 * (size_n / nz_S)) = 1;
end

% Iteration counter
itr = 0;
```

```
% Set of selectable frequencies
freq = (0 : size_n - 1);

%Pick k using constraints
size_k = size_n - nz_S;

% Non uniformly distributed frequency choices
freq_k = freq;
freq_k(~mod(freq_k, (size_n/nz_S))) = [];

% Form the DFT matrix
phi = zeros( size_k, size_n);
for index1 = 1 : size_k
    for index2 = 1 : size_n
        phi(index1, index2) = size_n ^(-0.5) * exp(-1i * 2 * pi * freq_k(index1)
    end
end

% Form the Sample Matrix
y = phi * x;

% Solve the system using CVX
cvx_begin
    variable cvx_x(size_n)
    minimize(norm(cvx_x, 1))
    subject to
        phi * cvx_x == y
cvx_end

% Compressive Sensing
% Optimization Problem
%    minimize ||x||_1
%        s.t. y = phi * x
% where phi = [k,n] matrix with k << n
%          x = [n,1] vector with S < n non-zero elements

% Size of x
size_n = 50;

% Number of non-zero elements in x
nz_S = 10;

% Vector x with nz_S number of uniformly distributed values
x = sprand(size_n, 1, (nz_S/size_n));
opt_f = norm(x,1)
```

```
% Set of selectable frequencies
freq = (0 : size_n - 1);

% Pick k using constraints
size_k = 25;

% Set of frequency randomly chosen from set of selectable frequencies
freq_k = sort(randsample(freq, size_k));

% Form the DFT matrix
phi = zeros( size_k, size_n);
for index1 = 1 : size_k
    for index2 = 1 : size_n
        phi(index1, index2) = size_n ^(-0.5) * exp(-1i * 2 * pi * freq_k(index1)
    end
end

% Form the Sample Matrix
y = phi * x;

% Solve the system using CVX
cvx_begin
    variable cvx_x(size_n)
    minimize(norm(cvx_x, 1))
    subject to
        phi * cvx_x == y
cvx_end

% Compressive Sensing
% Optimization Problem
%    minimize ||x||_1
%        s.t. y = phi * x
% where phi = [k,n] matrix with k << n
%         x = [n,1] vector with S < n non-zero elements
%         e = upper bound of norm of noise vector

% Size of x
size_n = 100;

% Number of non-zero elements in x
nz_S = 30;

% Noise levels
e = [0.0001 0.0005 0.001 0.005 0.01 0.05];

% Vector x with nz_S number of uniformly distributed values
```

```
x = sprand(size_n , 1, (nz_S./size_n));
opt_f = norm(x,1);

% Set of selectable frequencies
freq = (0 : size_n - 1);

% Pick k using constraints
size_k = 58;

% Set of frequency randomly chosen from set of selectable frequencies
freq_k = sort(randsample(freq , size_k));

% Form the DFT matrix
phi = zeros( size_k , size_n);
for index1 = 1 : size_k
    for index2 = 1 : size_n
        phi(index1 , index2) = size_n ^(-0.5) * exp(-1i * 2 * pi * freq_k(index1)
    end
end

% Randomly generated Noise Vectors
n1 = rand(size_k , 1);
while(norm(n1) > e(1))
    n1 = n1.*rand(size_k , 1);
end

n2 = rand(size_k , 1);
while(norm(n2) > e(2))
    n2 = n2.*rand(size_k , 1);
end

n3 = rand(size_k , 1);
while(norm(n3) > e(3))
    n3 = n3.*rand(size_k , 1);
end

n4 = rand(size_k , 1);
while(norm(n4) > e(4))
    n4 = n4.*rand(size_k , 1);
end

n5 = rand(size_k , 1);
while(norm(n5) > e(5))
    n5 = n5.*rand(size_k , 1);
end
```

```
n6 = rand(size_k, 1);
while(norm(n6) > e(6))
    n6 = n6.*rand(size_k, 1);
end

% Form the Sample Matrix
y1 = phi * x + n1;
y2 = phi * x + n2;
y3 = phi * x + n3;
y4 = phi * x + n4;
y5 = phi * x + n5;
y6 = phi * x + n6;

% Solve the system using CVX
cvx_begin
    variable cvx_x1(size_n)
    minimize(norm(cvx_x1, 1))
    subject to
        norm(y1 - phi * cvx_x1) <= e(1);
cvx_end

cvx_begin
    variable cvx_x2(size_n)
    minimize(norm(cvx_x2, 1))
    subject to
        norm(y2 - phi * cvx_x2) <= e(2);
cvx_end

cvx_begin
    variable cvx_x3(size_n)
    minimize(norm(cvx_x3, 1))
    subject to
        norm(y3 - phi * cvx_x3) <= e(3);
cvx_end

cvx_begin
    variable cvx_x4(size_n)
    minimize(norm(cvx_x4, 1))
    subject to
        norm(y4 - phi * cvx_x4) <= e(4);
cvx_end

cvx_begin
    variable cvx_x5(size_n)
    minimize(norm(cvx_x5, 1))
    subject to
```

```
        norm(y5 - phi * cvx_x5) <= e(5);
cvx_end

cvx_begin
    variable cvx_x6(size_n)
    minimize(norm(cvx_x6, 1))
    subject to
        norm(y6 - phi * cvx_x6) <= e(6);
cvx_end

% Calculate the norm of differences between noiseless recovery
% and noisy recovery
nrm(1) = norm(x - cvx_x1);
nrm(2) = norm(x - cvx_x2);
nrm(3) = norm(x - cvx_x3);
nrm(4) = norm(x - cvx_x4);
nrm(5) = norm(x - cvx_x5);
nrm(6) = norm(x - cvx_x6);

% Plot the norm vs. error bound
semilogx(e, nrm, 'b-o');
set(gca, 'FontSize', 12);
xlabel('Upper bound on norm of noise vector \epsilon', 'FontSize', 12);
ylabel('||x^* - x||_x', 'FontSize', 12);
```

Part 2

```
%% maximizing algebraic connectivity of a graph
max_alg_conn_data;
Q = null(ones(1,n)); % columns of Q are orthonormal basis for 1^\perp
cvx_begin
    variable w(m)
    L = A*diag(w)*A';
    maximize (lambda_min(Q'*L*Q))
    subject to
        w >= 0;
        F*w <= g;
cvx_end
w(abs(w) < 1e-4) = 0;

% compare algebraic connectivities
L_unif = (1/m)*A*A';
dunif = eig(L_unif);
dopt = eig(L);
fprintf(1, 'Algebraic connectivity of L_unif: %f\n', dunif(2));
fprintf(1, 'Algebraic connectivity of L_opt: %f\n', dopt(2));
```

```
% plot topology of constant-weight graph
figure(1), clf
gplot(L_unif,xy);
hold on;
plot(xy(:,1), xy(:,2), 'ko','LineWidth',4, 'MarkerSize',4);
axis([0.05 1.1 -0.1 0.95]);
title('Constant-weight graph')
hold off;
print -deps graph_plot1.eps;

% plot topology of optimal weight graph
figure(2), clf
gplot(L,xy);
hold on;
plot(xy(:,1), xy(:,2), 'ko','LineWidth',4, 'MarkerSize',4);
axis([0.05 1.1 -0.1 0.95]);
title('Optimal weight graph')
hold off;
print -deps graph_plot2.eps;
```