EE364a Homework 7 solutions

8.16 *Maximum volume rectangle inside a polyhedron.* Formulate the following problem as a convex optimization problem. Find the rectangle

$$\mathcal{R} = \{x \in \mathbf{R}^n \mid l \preceq x \preceq u\}$$

of maximum volume, enclosed in a polyhedron $\mathcal{P} = \{x \mid Ax \preceq b\}$. The variables are $l, u \in \mathbf{R}^n$. Your formulation should not involve an exponential number of constraints.

Solution. A straightforward, but very inefficient, way to express the constraint $\mathcal{R} \subseteq \mathcal{P}$ is to use the set of $m2^n$ inequalities $Av^i \preceq b$, where v^i are the (2^n) corners of $\mathcal{R}$. (If the corners of a box lie inside a polyhedron, then the box does.) Fortunately it is possible to express the constraint in a far more efficient way. Define

$$a_{ij}^+ = \max\{a_{ij}, 0\}, \qquad a_{ij}^- = \max\{-a_{ij}, 0\}.$$

Then we have $\mathcal{R} \subseteq \mathcal{P}$ if and only if

$$\sum_{i=1}^{n} (a_{ij}^+ u_j - a_{ij}^- l_j) \leq b_i, \quad i = 1, \ldots, m,$$

The maximum volume rectangle is the solution of

$$\begin{array}{ll} \text{maximize} & (\prod_{i=1}^n (u_i - l_i))^{1/n} \\ \text{subject to} & \sum_{i=1}^n (a_{ij}^+ u_j - a_{ij}^- l_j) \leq b_i, \quad i = 1, \ldots, m, \end{array}$$

with implicit constraint $u \succeq l$. Another formulation can be found by taking the log of the objective, which yields

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^n \log(u_i - l_i) \\ \text{subject to} & \sum_{i=1}^n (a_{ij}^+ u_j - a_{ij}^- l_j) \leq b_i, \quad i = 1, \ldots, m. \end{array}$$

9.30 *Gradient and Newton methods.* Consider the unconstrained problem

$$\text{minimize} \quad f(x) = -\sum_{i=1}^m \log(1 - a_i^T x) - \sum_{i=1}^n \log(1 - x_i^2),$$

with variable $x \in \mathbf{R}^n$, and $\mathbf{dom}\, f = \{x \mid a_i^T x < 1,\ i = 1, \ldots, m,\ |x_i| < 1,\ i = 1, \ldots, n\}$. This is the problem of computing the analytic center of the set of linear inequalities

$$a_i^T x \leq 1, \quad i = 1, \ldots, m, \qquad |x_i| \leq 1, \quad i = 1, \ldots, n.$$

Note that we can choose $x^{(0)} = 0$ as our initial point. You can generate instances of this problem by choosing a_i from some distribution on $\mathbf{R}^n$.

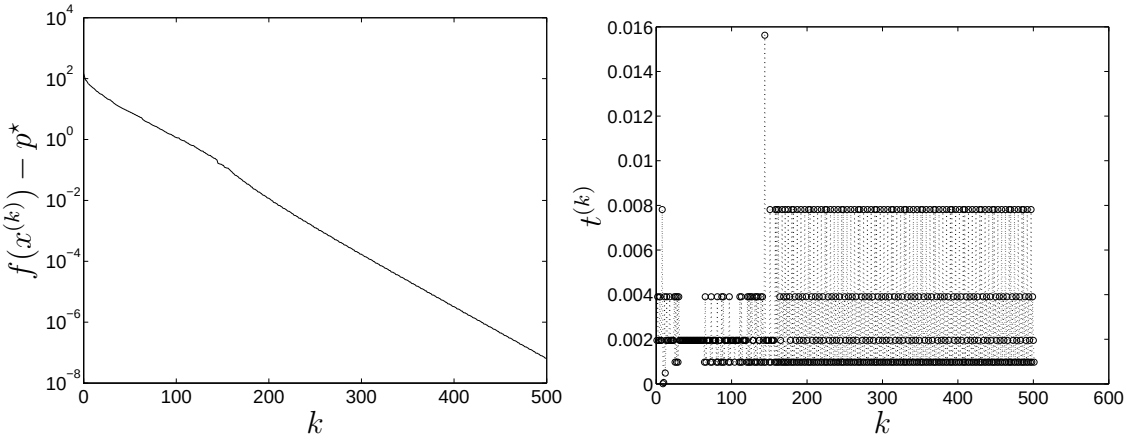
(a) Use the gradient method to solve the problem, using reasonable choices for the backtracking parameters, and a stopping criterion of the form $\|\nabla f(x)\|_2 \leq \eta$. Plot the objective function and step length versus iteration number. (Once you have determined $p^\star$ to high accuracy, you can also plot $f - p^\star$ versus iteration.) Experiment with the backtracking parameters α and β to see their effect on the total number of iterations required. Carry these experiments out for several instances of the problem, of different sizes.

(b) Repeat using Newton's method, with stopping criterion based on the Newton decrement λ^2. Look for quadratic convergence. You do not have to use an efficient method to compute the Newton step, as in exercise 9.27; you can use a general purpose dense solver, although it is better to use one that is based on a Cholesky factorization.

Hint. Use the chain rule to find expressions for $\nabla f(x)$ and $\nabla^2 f(x)$.

Solution.

(a) *Gradient method.* The figures show the function values and step lengths versus iteration number for an example with $m = 200$, $n = 100$. We used $\alpha = 0.01$, $\beta = 0.5$, and exit condition $\|\nabla f(x^{(k)})\|_2 \leq 10^{-3}$.



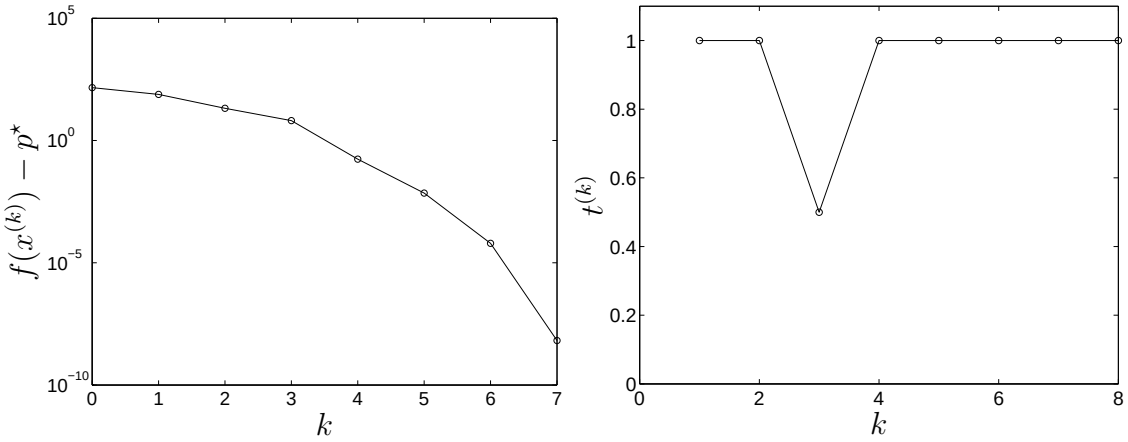
The following is a Matlab implementation.

```
randn('state',1);
m=200;
n=100;

ALPHA = 0.01;
BETA = 0.5;
MAXITERS = 1000;
NTTOL  = 1e-8;
GRADTOL  = 1e-3;
```

```
% generate random problem
A = randn(m,n);

% gradient method

vals = [];  steps = [];

x = zeros(n,1);
for iter = 1:MAXITERS

   val = -sum(log(1-A*x)) - sum(log(1+x)) - sum(log(1-x));
   vals = [vals, val];
   d = 1./(1-A*x);
   grad =  A'*d - 1./(1+x) + 1./(1-x);
   v = -grad;

   fprime = grad'*v;
   norm(grad)
   if norm(grad) < GRADTOL, break; end;

   t = 1;
   while ((max(A*(x+t*v)) >= 1) | (max(abs(x+t*v)) >= 1)),
      t = BETA*t;
   end;
   while ( -sum(log(1-A*(x+t*v))) - sum(log(1-(x+t*v).^2)) >  ...
           val + ALPHA*t*fprime )
      t = BETA*t;
   end;

   x = x+t*v;
   steps = [steps,t];

end;

figure(1)
semilogy([0:(length(vals)-2)], vals(1:length(vals)-1)-optval, '-');
xlabel('x');  ylabel('z');

figure(2)
plot([1:length(steps)], steps, ':',[1:length(steps)], steps, 'o');
xlabel('x');  ylabel('z');
```

(b) *Newton method.* The figures show the function values and step lengths versus

iteration number for the same example. We used $\alpha = 0.01$, $\beta = 0.5$, and exit condition $\lambda(x^{(k)})^2 \leq 10^{-8}$.



The following is a Matlab implementation.

```
% Newton method

vals = [];  steps = [];

x = zeros(n,1);
for iter = 1:MAXITERS

   val = -sum(log(1-A*x)) - sum(log(1+x)) - sum(log(1-x));
   vals = [vals, val];
   d = 1./(1-A*x);
   grad =  A'*d - 1./(1+x) + 1./(1-x);
   hess =  A'*diag(d.^2)*A + diag(1./(1+x).^2 + 1./(1-x).^2);
   v = -hess\grad;

   fprime = grad'*v
   if abs(fprime) < NTTOL, break; end;

   t = 1;
   while ((max(A*(x+t*v)) >= 1) | (max(abs(x+t*v)) >= 1)),
      t = BETA*t;
   end;
   while ( -sum(log(1-A*(x+t*v))) - sum(log(1-(x+t*v).^2)) >  ...
           val + ALPHA*t*fprime )
      t = BETA*t;
   end;

   x = x+t*v;
```

```
    steps = [steps,t];

end;
optval = vals(length(vals));

figure(3)
semilogy([0:(length(vals)-2)], vals(1:length(vals)-1)-optval, '-', ...
         [0:(length(vals)-2)], vals(1:length(vals)-1)-optval, 'o');
xlabel('x');  ylabel('z');

figure(4)
plot([1:length(steps)], steps, '-', [1:length(steps)], steps, 'o');
axis([0, length(steps), 0, 1.1]);
xlabel('x');  ylabel('z');
```

9.31 *Some approximate Newton methods.* The cost of Newton's method is dominated by the cost of evaluating the Hessian $\nabla^2 f(x)$ and the cost of solving the Newton system. For large problems, it is sometimes useful to replace the Hessian by a positive definite approximation that makes it easier to form and solve for the search step. In this problem we explore some common examples of this idea.

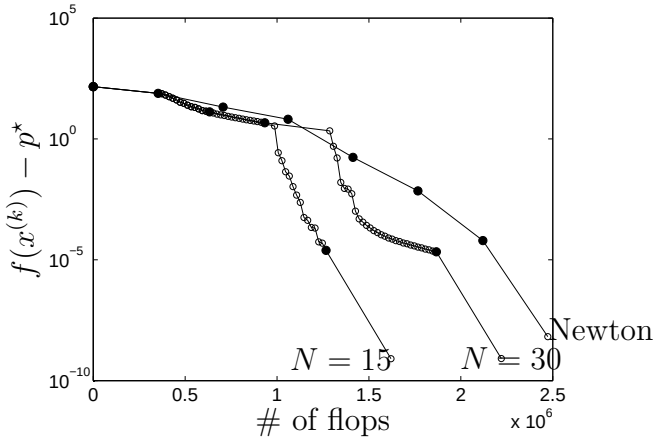
For each of the approximate Newton methods described below, test the method on some instances of the analytic centering problem described in exercise 9.30, and compare the results to those obtained using the Newton method and gradient method.

(a) *Re-using the Hessian.* We evaluate and factor the Hessian only every N iterations, where $N > 1$, and use the search step $\Delta x = -H^{-1}\nabla f(x)$, where H is the last Hessian evaluated. (We need to evaluate and factor the Hessian once every N steps; for the other steps, we compute the search direction using back and forward substitution.)

(b) *Diagonal approximation.* We replace the Hessian by its diagonal, so we only have to evaluate the n second derivatives $\partial^2 f(x)/\partial x_i^2$, and computing the search step is very easy.

Solution.

(a) The figure shows the function value versus approxmate total number of flops required (for the same example as in the solution of exercise 9.30), for $N = 1$ (*i.e.*, Newton's method), $N = 15$, and $N = 30$.



We see that the speed of convergence is increased using the method of using a factorized Hessian for several steps, as measured by true effort (*i.e.*, number of flops required). Of course in terms of iterations, the method is worse than the basic Newton method.

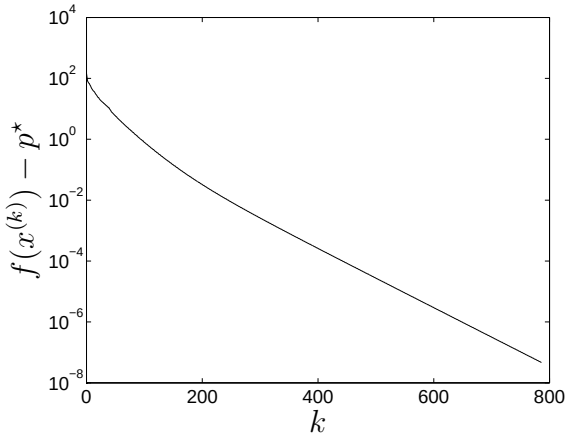
The following is a Matlab implementation.

```
randn('state',1);
m=200;
n=100;

ALPHA    = 0.01;
BETA     = 0.5;
MAXITERS = 1000;
NTTOL    = 1e-9;
GRADTOL  = 1e-3;

% generate random problem
A = randn(m,n);

% Newton method with periodically updated Hessian

for N = [1,15,30];  % re-compute Hessian every N iterations

    vals  = [];
    flops = [];
    flop  = 0;
    x = zeros(n,1);
    for iter = 1:MAXITERS

        val = -sum(log(1-A*x))-sum(log(1+x))-sum(log(1-x));
        vals  = [vals,val];
```

```
        flops = [flops,flop];

        d = 1./(1-A*x);
        grad =  A'*d-1./(1+x)+1./(1-x);

        if (rem(iter-1,N) == 0)
            H = A'*diag(d.^2)*A+diag(1./(1+x).^2+1./(1-x).^2);
            L = chol(H,'lower');
            flop  =  (1/3)*n^3; % add flop for Cholesky factorization
        else
            flop  = 0;
        end
        v = -L'\(L\grad);
        flop = flop+2*n^2; % add flop for fwd/bwd substitution

        fprime = grad'*v
        if (abs(fprime) < NTTOL) break; end

        t = 1;
        while ((max(A*(x+t*v))>=1) | (max(abs(x+t*v))> 1)),
            t = BETA*t;
        end
        while (-sum(log(1-A*(x+t*v)))-sum(log(1-(x+t*v).^2)) >  ...
            val + ALPHA*t*fprime )
            t = BETA*t;
        end
        x = x+t*v;
    end

    if (N==1), optval = vals(length(vals)); end

    figure(1)
    cflops = cumsum(flops(1:end-1));
    perror = vals(1:end-1)-optval;
    semilogy(cflops,perror,'-',cflops,perror,'o');
    hold on;
    semilogy(cflops(1:N:end-1),perror(1:N:end-1),...
         'mo','MarkerEdgeColor','k','MarkerFaceColor','b','MarkerSize',8);
    text(cflops(end),perror(end),['N',num2str(N)]);
    hold on;

end
```

```
xlabel('x');  ylabel('z');
```

(b) The figure shows the function value versus iteration number (for the same example as in the solution of exercise 9.30), for a diagonal approximation of the Hessian. The experiment shows that the algorithm converges very much like the gradient method.



The following is a Matlab implementation.

```
% Newton method with diagonal approximation of Hessian

vals = [];
x = zeros(n,1);
for iter = 1:MAXITERS

    val = -sum(log(1-A*x)) - sum(log(1+x)) - sum(log(1-x));
    vals = [vals, val];
    d = 1./(1-A*x);
    grad =  A'*d - 1./(1+x) + 1./(1-x);
    hess =  A'*diag(d.^2)*A + diag(1./(1+x).^2 + 1./(1-x).^2);
    H = diag(diag(hess));
    norm(grad)
    if norm(grad) < GRADTOL, break; end;
    v = -H\grad;  fprime = grad'*v;

    if (fprime > 0), keyboard; end;

    t = 1;
    while ((max(A*(x+t*v)) >= 1) | (max(abs(x+t*v)) >= 1)),
        t = BETA*t;
    end
    while ( -sum(log(1-A*(x+t*v))) - sum(log(1-(x+t*v).^2)) >  ...
            val + ALPHA*t*fprime )
```

```
        t = BETA*t;
    end
   x = x+t*v;
end

figure(2)
semilogy([0:(length(vals)-2)], vals(1:length(vals)-1)-optval, '-');
xlabel('x');  ylabel('z');
```

Solutions to additional exercises

Suggestions for exercise 9.30

We recommend the following to generate a problem instance:

```
n = 100;
m = 200;
randn('state',1);
A=randn(m,n);
```

Of course, you should try out your code with different dimensions, and different data as well.

In all cases, be sure that your line search *first* finds a step length for which the tentative point is in $\mathbf{dom}\, f$; if you attempt to evaluate f outside its domain, you'll get complex numbers, and you'll never recover.

To find expressions for $\nabla f(x)$ and $\nabla^2 f(x)$, use the chain rule (see Appendix A.4); if you attempt to compute $\partial^2 f(x)/\partial x_i \partial x_j$, you will be sorry.

To compute the Newton step, you can use `vnt=-H\g`.

Suggestions for exercise 9.31

For 9.31a, you should try out $N = 1$, $N = 15$, and $N = 30$. You might as well compute and store the Cholesky factorization of the Hessian, and then back solve to get the search directions, even though you won't really see any speedup in Matlab for such a small problem. After you evaluate the Hessian, you can find the Cholesky factorization as `L=chol(H,'lower')`. You can then compute a search step as `-L'\(L\g)`, where `g` is the gradient at the current point. Matlab will do the right thing, *i.e.*, it will first solve `L\g` using forward substitution, and then it will solve `-L'\(L\g)` using backward substitution. Each substitution is order n^2.

To fairly compare the convergence of the three methods (*i.e.*, $N = 1$, $N = 15$, $N = 30$), the horizontal axis should show the approximate total number of flops required, and not the number of iterations. You can compute the approximate number of flops using $n^3/3$ for each factorization, and $2n^2$ for each solve (where each 'solve' involves a forward substitution step and a backward substitution step).

Additional exercises

1. *Three-way linear classification.* We are given data

$$x^{(1)}, \ldots, x^{(N)}, \quad y^{(1)}, \ldots, y^{(M)}, \quad z^{(1)}, \ldots, z^{(P)},$$

three nonempty sets of vectors in $\mathbf{R}^n$. We wish to find three affine functions on $\mathbf{R}^n$,

$$f_i(z) = a_i^T z - b_i, \quad i = 1, 2, 3,$$

that satisfy the following properties:

$$\begin{array}{l} f_1(x^{(j)}) > \max\{f_2(x^{(j)}), f_3(x^{(j)})\}, \quad j = 1, \ldots, N, \\ f_2(y^{(j)}) > \max\{f_1(y^{(j)}), f_3(y^{(j)})\}, \quad j = 1, \ldots, M, \\ f_3(z^{(j)}) > \max\{f_1(z^{(j)}), f_2(z^{(j)})\}, \quad j = 1, \ldots, P. \end{array}$$

In words: f_1 is the largest of the three functions on the x data points, f_2 is the largest of the three functions on the y data points, f_3 is the largest of the three functions on the z data points. We can give a simple geometric interpretation: The functions f_1, f_2, and f_3 partition $\mathbf{R}^n$ into three regions,

$$\begin{array}{rcl} R_1 & = & \{z \mid f_1(z) > \max\{f_2(z), f_3(z)\}\}, \\ R_2 & = & \{z \mid f_2(z) > \max\{f_1(z), f_3(z)\}\}, \\ R_3 & = & \{z \mid f_3(z) > \max\{f_1(z), f_2(z)\}\}, \end{array}$$

defined by where each function is the largest of the three. Our goal is to find functions with $x^{(j)} \in R_1$, $y^{(j)} \in R_2$, and $z^{(j)} \in R_3$.

Pose this as a convex optimization problem. You may not use strict inequalities in your formulation.

Solve the specific instance of the 3-way separation problem given in `sep3way_data.m`, with the columns of the matrices `X`, `Y` and `Z` giving the $x^{(j)},\ j = 1, \ldots, N$, $y^{(j)},\ j = 1, \ldots, M$ and $z^{(j)},\ j = 1, \ldots, P$. To save you the trouble of plotting data points and separation boundaries, we have included the plotting code in `sep3way_data.m`. (Note that `a1`, `a2`, `a3`, `b1` and `b2` contain arbitrary numbers; you should compute the correct values using `cvx`.)

Solution. The inequalities

$$\begin{array}{l} f_1(x^{(j)}) > \max\{f_2(x^{(j)}), f_3(x^{(j)})\}, \quad j = 1, \ldots, N, \\ f_2(y^{(j)}) > \max\{f_1(y^{(j)}), f_3(y^{(j)})\}, \quad j = 1, \ldots, M, \\ f_3(z^{(j)}) > \max\{f_1(z^{(j)}), f_2(z^{(j)})\}, \quad j = 1, \ldots, P. \end{array}$$

are homogeneous in a_i and b_i so we can express them as

$$\begin{array}{l} f_1(x^{(j)}) \geq \max\{f_2(x^{(j)}), f_3(x^{(j)})\} + 1, \quad j = 1, \ldots, N, \\ f_2(y^{(j)}) \geq \max\{f_1(y^{(j)}), f_3(y^{(j)})\} + 1, \quad j = 1, \ldots, M, \\ f_3(z^{(j)}) \geq \max\{f_1(z^{(j)}), f_2(z^{(j)})\} + 1, \quad j = 1, \ldots, P. \end{array}$$

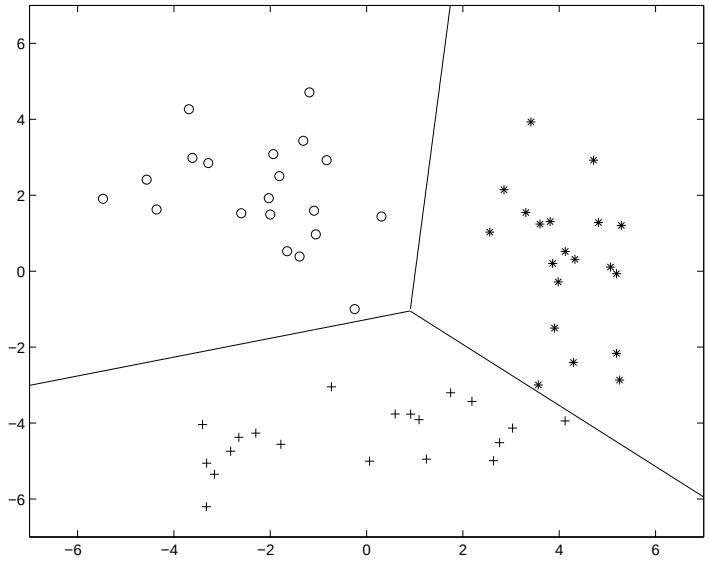
Note that we can add any vector α to each of the a_i, without affecting these inequalities (which only refer to difference between a_i's), and we can add any number β to each of the b_i's for the same reason. We can use this observation to normalize or simplify the a_i and b_i. For example, we can assume without loss of generality that $a_1 + a_2 + a_3 = 0$ and $b_1 + b_2 + b_3 = 0$.

The following script implements this method for 3-way classification and tests it on a small separable data set

```
clear all; close all;
% data for problem instance
M = 20;
N = 20;
P = 20;

X = [

    3.5674    4.1253    2.8535    5.1892    4.3273    3.8133    3.4117 ...
    3.8636    5.0668    3.9044    4.2944    4.7143    3.3082    5.2540 ...
    2.5590    3.6001    4.8156    5.2902    5.1908    3.9802 ;...
   -2.9981    0.5178    2.1436   -0.0677    0.3144    1.3064    3.9297 ...
    0.2051    0.1067   -1.4982   -2.4051    2.9224    1.5444   -2.8687 ...
    1.0281    1.2420    1.2814    1.2035   -2.1644   -0.2821];

Y = [
   -4.5665   -3.6904   -3.2881   -1.6491   -5.4731   -3.6170   -1.1876 ...
   -1.0539   -1.3915   -2.0312   -1.9999   -0.2480   -1.3149   -0.8305 ...
   -1.9355   -1.0898   -2.6040   -4.3602   -1.8105    0.3096; ...
    2.4117    4.2642    2.8460    0.5250    1.9053    2.9831    4.7079 ...
    0.9702    0.3854    1.9228    1.4914   -0.9984    3.4330    2.9246 ...
    3.0833    1.5910    1.5266    1.6256    2.5037    1.4384];

Z = [
    1.7451    2.6345    0.5937   -2.8217    3.0304    1.0917   -1.7793 ...
    1.2422    2.1873   -2.3008   -3.3258    2.7617    0.9166    0.0601 ...
   -2.6520   -3.3205    4.1229   -3.4085   -3.1594   -0.7311; ...
   -3.2010   -4.9921   -3.7621   -4.7420   -4.1315   -3.9120   -4.5596 ...
   -4.9499   -3.4310   -4.2656   -6.2023   -4.5186   -3.7659   -5.0039 ...
   -4.3744   -5.0559   -3.9443   -4.0412   -5.3493   -3.0465];

cvx_begin
variables a1(2) a2(2) a3(2) b1 b2 b3
    a1'*X-b1 >= max(a2'*X-b2,a3'*X-b3)+1;
    a2'*Y-b2 >= max(a1'*Y-b1,a3'*Y-b3)+1;
    a3'*Z-b3 >= max(a1'*Z-b1,a2'*Z-b2)+1;
    a1 + a2 + a3 == 0
    b1 + b2 + b3 == 0
cvx_end

% now let's plot the three-way separation induced by
% a1,a2,a3,b1,b2,b3
```

```
% find maximally confusing point
p = [(a1-a2)';(a1-a3)']\[(b1-b2);(b1-b3)];

% plot
t = [-7:0.01:7];
u1 = a1-a2; u2 = a2-a3; u3 = a3-a1;
v1 = b1-b2; v2 = b2-b3; v3 = b3-b1;
line1 = (-t*u1(1)+v1)/u1(2); idx1 = find(u2'*[t;line1]-v2>0);
line2 = (-t*u2(1)+v2)/u2(2); idx2 = find(u3'*[t;line2]-v3>0);
line3 = (-t*u3(1)+v3)/u3(2); idx3 = find(u1'*[t;line3]-v1>0);
plot(X(1,:),X(2,:),'*',Y(1,:),Y(2,:),'ro',Z(1,:),Z(2,:),'g+',...
     t(idx1),line1(idx1),'k',t(idx2),line2(idx2),'k',t(idx3),line3(idx3),'k');
axis([-7 7 -7 7]);
```

The following figure is generated.



2. *Efficient numerical method for a regularized least-squares problem.* We consider a regularized least squares problem with smoothing,

$$\text{minimize} \quad \sum_{i=1}^{k}(a_i^T x - b_i)^2 + \delta \sum_{i=1}^{n-1}(x_i - x_{i+1})^2 + \eta \sum_{i=1}^{n} x_i^2,$$

where $x \in \mathbf{R}^n$ is the variable, and $\delta, \eta > 0$ are parameters.

(a) Express the optimality conditions for this problem as a set of linear equations involving x. (These are called the normal equations.)

(b) Now assume that $k \ll n$. Describe an efficient method to solve the normal equations found in (2a). Give an approximate flop count for a general method that does not exploit structure, and also for your efficient method.

(c) *A numerical instance.* In this part you will try out your efficient method. We'll choose $k = 100$ and $n = 2000$, and $\delta = \eta = 1$. First, randomly generate A and b with these dimensions. Form the normal equations as in (2a), and solve them using a generic method. Next, write (short) code implementing your efficient method, and run it on your problem instance. Verify that the solutions found by the two methods are nearly the same, and also that your efficient method is much faster than the generic one.

Note: You'll need to know some things about Matlab to be sure you get the speedup from the efficient method. Your method should involve solving linear equations with tridiagonal coefficient matrix. In this case, both the factorization and the back substitution can be carried out very efficiently. The Matlab documentation says that banded matrices are recognized and exploited, when solving equations, but we found this wasn't always the case. To be sure Matlab knows your matrix is tridiagonal, you can declare the matrix as sparse, using `spdiags`, which can be used to create a tridiagonal matrix. You could also create the tridiagonal matrix conventionally, and then convert the resulting matrix to a sparse one using `sparse`.

One other thing you need to know. Suppose you need to solve a group of linear equations with the same coefficient matrix, *i.e.*, you need to compute $F^{-1}a_1, ..., F^{-1}a_m$, where F is invertible and a_i are column vectors. By concatenating columns, this can be expressed as a single matrix

$$\left[F^{-1}a_1 \ \cdots \ F^{-1}a_m\right] = F^{-1}\left[a_1 \ \cdots \ a_m\right].$$

To compute this matrix using Matlab, you should collect the righthand sides into one matrix (as above) and use Matlab's backslash operator: `F\A`. This will do the right thing: factor the matrix F once, and carry out multiple back substitutions for the righthand sides.

Solution.

(a) The objective function is

$$x^T(A^TA + \delta\Delta + \eta I)x - 2b^TAx + b^Tb,$$

where $A \in \mathbf{R}^{k\times n}$ is the matrix with rows a_i, and $\Delta \in \mathbf{R}^{n\times n}$ is the tridiagonal

matrix

$$\Delta = \begin{bmatrix} 1 & -1 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 1 \end{bmatrix}.$$

Since the problem is unconstrained, the optimality conditions are

$$(A^TA + \delta\Delta + \eta I)x^\star = A^Tb. \tag{1}$$

(b) If no structure is exploited, then solving (1) costs approximately $(1/3)n^3$ flops. If $k \ll n$, we need to solve a system $Fx = g$ where F is the sum of a tridiagonal and a (relatively) low-rank matrix. We can use the Sherman-Morrison-Woodbury formula

$$x^\star = (\delta\Delta + \eta I)^{-1}g - (\delta\Delta + \eta I)^{-1}A^T(I + A(\delta\Delta + \eta I)^{-1}A^T)^{-1}A(\delta\Delta + \eta I)^{-1}g$$

to efficiently solve (1) as follows:

i. Solve $(\delta\Delta + \eta I)z_1 = g$ and $(\delta\Delta + \eta I)Z_2 = A^T$ for z_1 and Z_2. Since $\delta\Delta + \eta I$ is tridiagonal, the total cost for this is approximately $4nk + 5n$ flops (n for factorization and $4n(k + 1)$ for the solves).

ii. Form Az_1 and AZ_2 ($2nk + 2nk^2$ flops).

iii. Solve $(I + AZ_2)z_3 = Az_1$ for z_3 ($(1/3)k^3$ flops).

iv. Form $x^\star = z_1 - Z_2z_3$ ($2nk$ flops).

The total flop count, keeping only leading terms, is $2nk^2$ flops, which is much smaller than $(1/3)n^3$ when $k \ll n$.

(c) Here's the Matlab code:

```
clear all; close all;

n = 2000;
k = 100;
delta = 1;
eta = 1;

A = rand(k,n);
b = rand(k,1);

e = ones(n,1);
D = spdiags([-e 2*e -e],[-1 0 1], n,n);
```

```
D(1,1) = 1; D(n,n) = 1;
I = speye(n);

F = A'*A + eta*I + delta*D;
P = eta*I + delta*D; %P is cheap to invert since it's tridiagonal
g = A'*b;

%Directly computing optimal solution
fprintf('\nComputing solution directly\n');
s1 = cputime;
x_gen = F\g;
s2 = cputime;
fprintf('Done (in %g sec)\n',s2-s1);

fprintf('\nComputing solution using efficient method\n');
%x_eff = P^{-1}g - P^{-1}A'(I +AP^{-1}A')^{-1}AP^{-1}g.

t1= cputime;
Z_0 = P\[g A'];
z_1 = Z_0(:,1);
%z_2 = A*z_1;
Z_2 = Z_0(:,2:k+1);
z_3 = (sparse(1:k,1:k,1) +A*Z_2)\(A*z_1);
x_eff = z_1 - Z_2*z_3;
t2 = cputime;
fprintf('Done (in %g sec)\n',t2-t1);

fprintf('\nrelative error = %e\n',norm(x_eff-x_gen)/norm(x_gen) );
```