EE263 Prof. S. Boyd
Oct. 27 – 28 or Oct. 28 – 29, 2006.

Midterm exam solutions

1. *Point of closest convergence of a set of lines.* We have m lines in $\mathbf{R}^n$, described as

$$\mathcal{L}_i = \{p_i + tv_i \mid t \in \mathbf{R}\}, \quad i = 1, \ldots, m,$$

where $p_i \in \mathbf{R}^n$, and $v_i \in \mathbf{R}^n$, with $\|v_i\| = 1$, for $i = 1, \ldots, m$. We define the distance of a point $z \in \mathbf{R}^n$ to a line $\mathcal{L}$ as

$$\mathbf{dist}(z, \mathcal{L}) = \min\{\|z - u\| \mid u \in \mathcal{L}\}.$$

(In other words, $\mathbf{dist}(z, \mathcal{L})$ gives the closest distance between the point z and the line $\mathcal{L}$.)

We seek a point $z^\star \in \mathbf{R}^n$ that minimizes the sum of the squares of the distances to the lines,

$$\sum_{i=1}^{m} \mathbf{dist}(z, \mathcal{L}_i)^2.$$

The point $z^\star$ that minimizes this quantity is called the *point of closest convergence*.

(a) Explain how to find the point of closest convergence, given the lines (*i.e.*, given $p_1, \ldots, p_m$ and $v_1, \ldots, v_m$). If your method works provided some condition holds (such as some matrix being full rank), say so. If you can relate this condition to a simple one involving the lines, please do so.

(b) Find the point $z^\star$ of closest convergence for the lines with data given in the Matlab file `line_conv_data.m`. This file contains $n \times m$ matrices `P` and `V` whose columns are the vectors $p_1, \ldots, p_m$, and $v_1, \ldots, v_m$, respectively. The file also contains commands to plot the lines and the point of closest convergence (once you have found it). Please include this plot with your solution.

Solution.

(a) There are several ways to solve this problem. Our first solution starts by working out an explicit expression for $\mathbf{dist}(z, \mathcal{L}_i)$. To find this distance we need to solve the simple least-squares problem of minimizing $\|z - p_i - tv_i\|^2$ over $t \in \mathbf{R}$. The optimal t is given by $t^\star = v_i^T(z - p_i)$, so we have

$$\mathbf{dist}(z, \mathcal{L}_i) = \|z - p_i - t^\star v_i\| = \|(I - v_i v_i^T)(z - p_i)\|.$$

This makes sense: we recognize $I - v_i v_i^T$ as projection onto the orthogonal complement of the line through the origin in the direction v_i, *i.e.*, projection onto the plane with normal vector v_i.

We can now set up our problem as a standard least-squares problem. We define

$$A = \begin{bmatrix} I - v_1 v_1^T \\ \vdots \\ I - v_m v_m^T \end{bmatrix}, \quad b = \begin{bmatrix} (I - v_1 v_1^T) p_1 \\ \vdots \\ (I - v_m v_m^T) p_m \end{bmatrix},$$

so we can write

$$\sum_{i=1}^{m} \mathbf{dist}(z, \mathcal{L}_i)^2 = \|Az - b\|^2.$$

Now we can solve the problem, assuming A is full rank (we'll come back to this). The solution is

$$z^\star = (A^T A)^{-1} A^T b = \left(mI - \sum_{i=1}^{m} v_i v_i^T \right)^{-1} \sum_{i=1}^{m} (p_i - v_i v_i^T p_i).$$

Finally, let's look at the conditions under which A is not full rank. Each $n \times n$ block of A, *i.e.*, $I - v_i v_i^T$, has rank exactly $n - 1$, with nullspace $\mathrm{span}(v_i)$. So unless all the v_i are aligned (*i.e.*, $v_i = v_j$ or $v_i = -v_j$ for all i, j), A is full rank. Geometrically, this means that the lines are all parallel. So we can say that A above is full rank, unless all the lines are parallel.

Here is another solution of the problem (or really, a variation on the solution given above). If we define

$$C = \begin{bmatrix} -v_1 & 0 & \cdots & 0 & I \\ 0 & -v_2 & \cdots & 0 & I \\ \vdots & \vdots & \ddots & \vdots & I \\ 0 & 0 & \cdots & -v_m & I \end{bmatrix}, \quad d = \begin{bmatrix} p_1 \\ \vdots \\ p_m \end{bmatrix}, \quad u = \begin{bmatrix} t_1 \\ \vdots \\ t_m \\ z \end{bmatrix},$$

we have

$$\sum_{i=1}^{m} \mathbf{dist}(z, \mathcal{L}_i)^2 = \min_{t_1, \ldots, t_m} \|Cu - d\|,$$

and

$$\min_{z} \sum_{i=1}^{m} \mathbf{dist}(z, \mathcal{L}_i)^2 = \min_{u} \|Cu - d\|.$$

In the last expression, we are optimizing over the line parameters t_i and the point z at the same time.

Therefore, assuming C is full rank, we have

$$z^\star = \begin{bmatrix} 0 & 0 \\ 0 & I \end{bmatrix} (C^T C)^{-1} C^T d,$$

which expands to the same solution we have above. And of course, C is full rank if and only if A is, which occurs exactly when the lines are not all parallel.

(b) The following code solves for the point of closest convergence using the two different approaches and checks that the solutions are identical.

```
% first solution
A=[];
b=[];
for i=1:m
    A=[A;eye(n)-V(:,i)*V(:,i)'];
    b=[b;(eye(n)-V(:,i)*V(:,i)')*P(:,i)];
end
zstar=A\b;

% second solution
C=zeros(n*m,m);
E=[];
d=[];
for i=1:m
    E=[E;eye(n)];
    C(n*(i-1)+1:n*i,i)=-V(:,i);
    d=[d;P(:,i)];
end
C=[C E];

zstar=A\b;
f=C\d;
zstar2=f(m+1:m+n);

% check that two solutions give (almost) same answer
zstar2-zstar
```

The result is $z^\star = (1.9157, 3.3951)$ and figure 1 shows the lines together with the point of closest convergence.

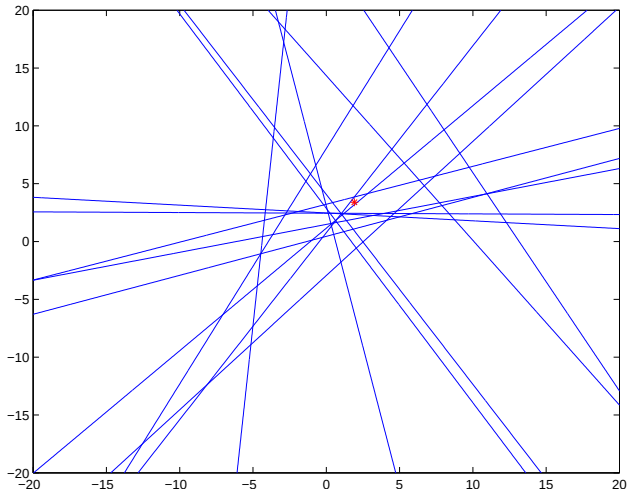


Figure 1: Point of closest convergence.

2. *Estimating direction and amplitude of a light beam.* A light beam with (nonnegative) amplitude a comes from a direction $d \in \mathbf{R}^3$, where $\|d\| = 1$. (This means the beam travels in the direction $-d$.) The beam falls on $m \geq 3$ photodetectors, each of which generates a scalar signal that depends on the beam amplitude and direction, and the direction in which the photodetector is pointed. Specifically, photodetector i generates an output signal p_i, with

$$p_i = a\alpha \cos\theta_i + v_i,$$

where θ_i is the angle between the beam direction d and the outward normal vector q_i of the surface of the ith photodetector, and α is the photodetector sensitivity. You can interpret $q_i \in \mathbf{R}^3$, which we assume has norm one, as the direction the ith photodetector is pointed. We assume that $|\theta_i| < 90^\circ$, *i.e.*, the beam illuminates the top of the photodetectors. The numbers v_i are small measurement errors.

You are given the photodetector direction vectors $q_1, \ldots, q_m \in \mathbf{R}^3$, the photodetector sensitivity α, and the noisy photodetector outputs, $p_1, \ldots, p_m \in \mathbf{R}$. Your job is to estimate the beam direction $d \in \mathbf{R}^3$ (which is a unit vector), and a, the beam amplitude.

To describe unit vectors $q_1, \ldots, q_m$ and d in $\mathbf{R}^3$ we will use azimuth and elevation, defined as follows:

$$q = \begin{bmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ \sin\phi \end{bmatrix}.$$

Here ϕ is the elevation (which will be between 0° and 90°, since all unit vectors in this problem have positive 3rd component, *i.e.*, point upward). The azimuth angle θ, which varies from 0° to 360°, gives the direction in the plane spanned by the first and second coordinates. If $q = e_3$ (*i.e.*, the direction is directly up), the azimuth is undefined.

(a) Explain how to do this, using a method or methods from this class. The simpler the method the better. If some matrix (or matrices) needs to be full rank for your method to work, say so.

(b) Carry out your method on the data given in `beam_estim_data.m`. This mfile defines `p`, the vector of photodetector outputs, a vector `det_az`, which gives the azimuth angles of the photodetector directions, and a vector `det_el`, which gives the elevation angles of the photodetector directions. Note that both of these are given in *degrees*, not radians. Give your final estimate of the beam amplitude a and beam direction d (in azimuth and elevation, in degrees).

Solution.

(a) Since $\cos\theta_i = q_i^T d/(\|q_i\|\|d\|) = q_i^T d$ (using $\|q_i\| = \|d\| = 1$), we have

$$p_i = a\alpha q_i^T d + v_i.$$

In this equation we are given p_i, α, and q_i; we are to estimate $a \in \mathbf{R}$ and $d \in \mathbf{R}^3$, using the given information that v_i is small. At first glance it looks like a nonlinear problem, since two of the variables we need to estimate, a and d, are multiplied together in this formula.

But a little thought reveals that things are actually much simpler. Let's define $x \in \mathbf{R}^3$ as $x = ad$. We can just as well work with x since given any nonzero $x \in \mathbf{R}^3$, we have $a = \|x\|$ and $d = x/\|x\|$. (Conversely, given any a and d, we have $x = ad$ by definition.)

We can therefore express the problem in terms of the variable x as

$$p = \alpha \begin{bmatrix} q_1^T \\ \vdots \\ q_m^T \end{bmatrix} x + v = \alpha Q x + v,$$

where $p = (p_1, \ldots, p_m)$, $v = (v_1, \ldots, v_m)$, and Q is the matrix with rows q_i^T.

Now we can get a reasonable guess of x using least-squares. Assuming Q is full rank, we have the least-squares estimate

$$\hat{x} = (1/\alpha)(Q^T Q)^{-1} Q^T p.$$

We then form estimates of a and d using $\hat{a} = \|\hat{x}\|$, $\hat{d} = \hat{x}/\|\hat{x}\|$.

The matrix Q is full rank (*i.e.*, rank 3), if and only if the vectors $\{q_1, \ldots, q_m\}$ span $\mathbf{R}^3$. In other words, we cannot have all photodetectors pointing in a common plane.

(b) The following code solves the problem for the given data.

```
beam_estim_data

for i=1:m
    Q(i,:)=[  cosd(det_el(i))*cosd(det_az(i)),...
              cosd(det_el(i))*sind(det_az(i)),...
              sind(det_el(i))  ];
end

xhat=(1/alpha)*(Q\p);
ahat=norm(xhat);
dhat=xhat/norm(xhat);

elevation=asind(dhat(3))
azimuth=acosd(dhat(1)/cosd(elevation))
```

The result is $\hat{a} = 5.0107$, $\hat{\phi}_d = 38.7174$, and $\hat{\theta}_d = 77.6623$.

3. *Minimum energy input with way-point constraints.* We consider a vehicle that moves in $\mathbf{R}^2$ due to an applied force input. We will use a discrete-time model, with time index $k = 1, 2, \ldots$; time index k corresponds to time $t = kh$, where $h > 0$ is the sample interval. The position at time index k is denoted by $p(k) \in \mathbf{R}^2$, and the velocity by $v(k) \in \mathbf{R}^2$, for $k = 1, \ldots, K+1$. These are related by the equations

$$p(k+1) = p(k) + hv(k), \quad v(k+1) = (1-\alpha)v(k) + (h/m)f(k), \quad k = 1, \ldots, K,$$

where $f(k) \in \mathbf{R}^2$ is the force applied to the vehicle at time index k, $m > 0$ is the vehicle mass, and $\alpha \in (0, 1)$ models drag on the vehicle: In the absence of any other force, the vehicle velocity decreases by the factor $1 - \alpha$ in each time index. (These formulas are approximations of more accurate formulas that we will see soon, but for the purposes of this problem, we consider them exact.) The vehicle starts at the origin, at rest, *i.e.*, we have $p(1) = 0$, $v(1) = 0$. (We take $k = 1$ as the initial time, to simplify indexing.)

The problem is to find forces $f(1), \ldots, f(K) \in \mathbf{R}^2$ that minimize the cost function

$$J = \sum_{k=1}^{K} \|f(k)\|^2,$$

subject to *way-point constraints*

$$p(k_i) = w_i, \quad i = 1, \ldots, M,$$

where k_i are integers between 1 and K. (These state that at the time $t_i = hk_i$, the vehicle must pass through the location $w_i \in \mathbf{R}^2$.) Note that there is no requirement on the vehicle velocity at the way-points.

(a) Explain how to solve this problem, given all the problem data (*i.e.*, h, α, m, K, the way-points $w_1, \ldots, w_M$, and the way-point indices $k_1, \ldots, k_M$).

(b) Carry out your method on the specific problem instance with data $h = 0.1$, $m = 1$, $\alpha = 0.1$, $K = 100$, and the $M = 4$ way-points

$$w_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \quad w_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}, \quad w_3 = \begin{bmatrix} 4 \\ -3 \end{bmatrix}, \quad w_4 = \begin{bmatrix} -4 \\ -2 \end{bmatrix},$$

with way-point indices $k_1 = 10$, $k_2 = 30$, $k_3 = 40$, and $k_4 = 80$.

Give the optimal value of J.

Plot $f_1(k)$ and $f_2(k)$ versus k, using

```
subplot(211); plot(f(1,:));
subplot(212); plot(f(2,:));
```

We assume here that `f` is a $2 \times K$ matrix, with columns $f(1), \ldots, f(K)$.

Plot the vehicle trajectory, using `plot(p(1,:),p(2,:))`. Here `p` is a $2 \times (K+1)$ matrix with columns $p(1), \ldots, p(K+1)$.

(a) The equations of motion can be written as the discrete-time linear dynamical system

$$x(k+1) = Ax(k) + Bf(k), \quad p(k) = Cx(k), \quad x(1) = 0,$$

where

$$x(k) = \begin{bmatrix} p(k) \\ v(k) \end{bmatrix}, \quad A = \begin{bmatrix} I & hI \\ 0 & (1-\alpha)I \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ (h/m)I \end{bmatrix}, \quad C = \begin{bmatrix} I & 0 \end{bmatrix}.$$

We can solve these state equations to get $x(k)$ in terms of the input forces $f(1), \ldots, f(k-1)$:

$$x(k) = \begin{bmatrix} A^{k-2}B & A^{k-3}B & \cdots & B \end{bmatrix} \begin{bmatrix} f(1) \\ f(2) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

The position of the vehicle at way-point index k_i is therefore

$$p(k_i) = Cx(k_i) = C\begin{bmatrix} A^{k_i-2}B & A^{k_i-3}B & \cdots & B \end{bmatrix} \begin{bmatrix} f(1) \\ f(2) \\ \vdots \\ f(k_i-1) \end{bmatrix}.$$

We can write the way-point constraint $p(k_i) = w_{k_i}$ as

$$w_i = C\begin{bmatrix} A^{k_i-2}B & A^{k_i-3}B & \cdots & B & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} f(1) \\ \vdots \\ f(K) \end{bmatrix},$$

or equivalently ,

$$w_i = G_i u,$$

where

$$u = \begin{bmatrix} f(1) \\ \vdots \\ f(K) \end{bmatrix} \in \mathbf{R}^{2K}, \quad G_i = C\begin{bmatrix} A^{k_i-2}B & A^{k_i-3}B & \cdots & B & 0 & \cdots & 0 \end{bmatrix} \in \mathbf{R}^{2\times 2K}.$$

Using notation

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_M \end{bmatrix}, \quad G = \begin{bmatrix} G_1 \\ \vdots \\ G_M \end{bmatrix},$$

and noting that $J = \|u\|^2$, the problem becomes

$$\begin{array}{ll} \text{minimize} & \|u\|^2 \\ \text{subject to} & Gu = w. \end{array}$$

This is just a least-norm problem and the optimal u is given by

$$u = G^{\dagger}w = G^T(GG^T)^{-1}w.$$

(b) The following Matlab script computes the minimum norm input, and plots it and the associated trajectory.

```
% problem parameters
h = .1;
m = 1;
M=4;
alpha=0.1;
K = 100;
% way-points
k1=10; w1=[ 2; 2];
k2=30; w2=[ -2; 3];
k3=40; w3=[ 4; -3];
k4=80; w4=[-4; -2];

A = [eye(2) h*eye(2); zeros(2) (1-alpha)*eye(2)];
B = [zeros(2); h/m*eye(2)];
C = [eye(2) zeros(2)];
[n, nn] = size(B);

k = [k1 k2 k3 k4];
G = [];
for i = 1:M
    ABmatrix = [];
    temp = B;
    for j=1:k(i)-1
        ABmatrix = [temp ABmatrix];
        temp = A*temp;
    end
    Gi = C*[ABmatrix zeros(n, nn*(K-k(i)+1))];
    G = [G; Gi];
end
w = [w1; w2; w3; w4];
u = pinv(G)*w;

% plotting the input
f = [u(1:2:end)'; u(2:2:end)'];
figure;
subplot(211); plot(f(1,:));
subplot(212); plot(f(2,:));
```

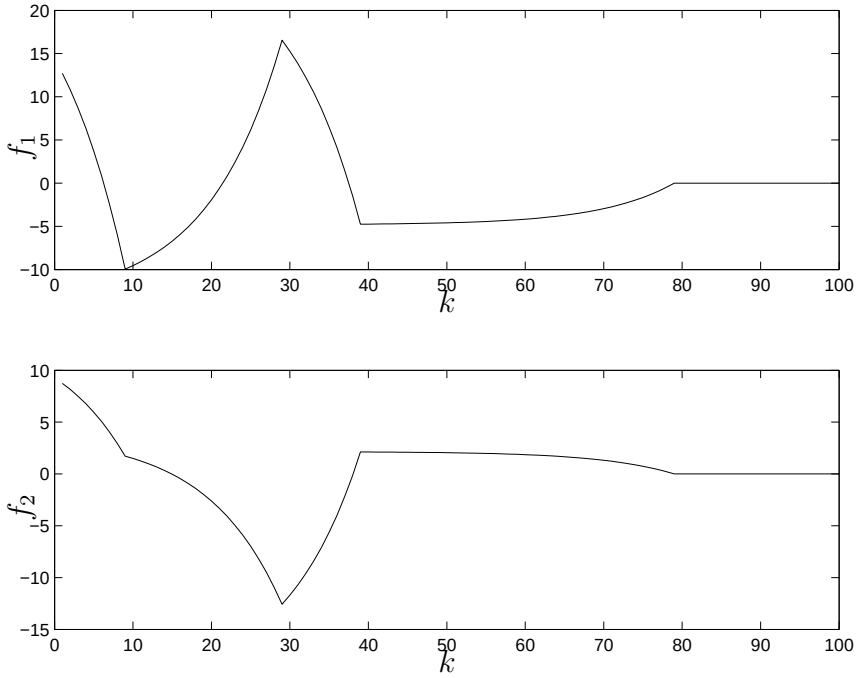


Figure 2: f versus k.

```
% simulating the system
p = zeros(2,K+1);
v = zeros(2,K+1);
for i=1:K
    p(:,i+1) = p(:,i) + h*v(:,i);
    v(:,i+1) = (1-alpha)*v(:,i) + h*f(:,i)/m;
end

% Optimal value of J
J = norm(u)^2

figure;
plot(p(1,:),p(2,:));
hold on
ps = [w1 w2 w3 w4];
plot(ps(1,:),ps(2,:),'*');
```

Figure (2) shows the minimum norm input forces. We see that for $k \geq 80$, the optimal force is zero. This makes perfect sense: for $k \geq 80$, the force $f(k)$ does not affect the vehicle position at any of the way-points, so using any force on the vehicle for $k \geq 80$ just increases the cost J.

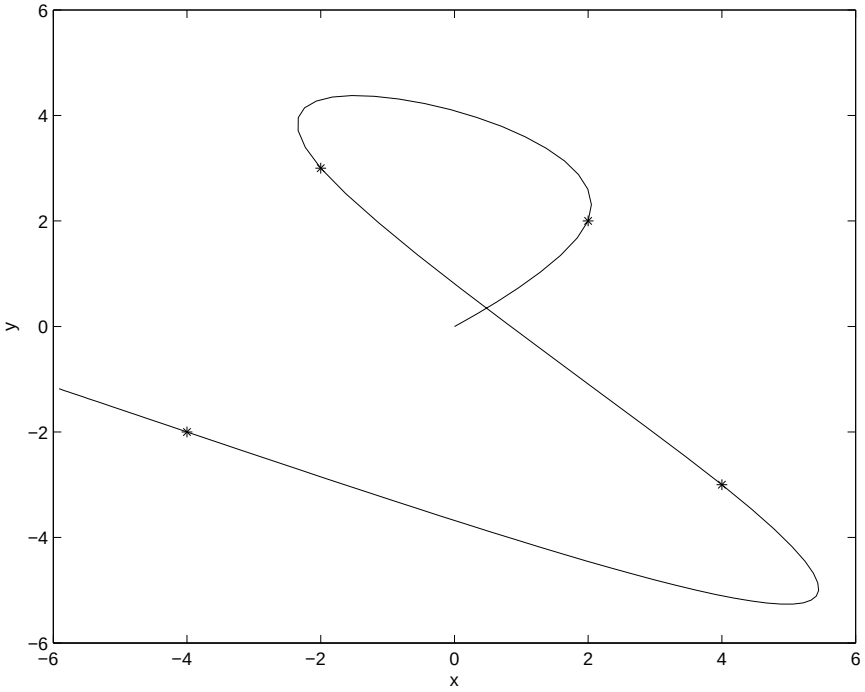


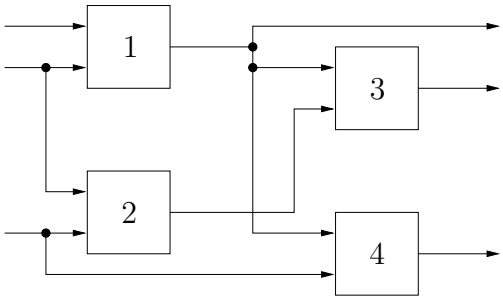
Figure 3: Trajectory in $\mathbf{R}^2$.

The optimal value of J is found to be 4770.5.
Figure (3) shows the resulting trajectory.

4. *Digital circuit gate sizing.* A digital circuit consists of a set of n (logic) gates, interconnected by wires. Each gate has one or more inputs (typically between one and four), and one output, which is connected via the wires to other gate inputs and possibly to some external circuitry. When the output of gate i is connected to an input of gate j, we say that gate i *drives* gate j, or that gate j is in the *fan-out* of gate i. We describe the topology of the circuit by the *fan-out list* for each gate, which tells us which other gates the output of a gate connects to. We denote the fan-out list of gate i as $\mathrm{FO}(i) \subseteq \{1, \ldots, n\}$. We can have $\mathrm{FO}(i) = \emptyset$, which means that the output of gate i does not connect to the inputs of any of the gates $1, \ldots, n$ (presumably the output of gate i connects to some external circuitry). It's common to order the gates in such a way that each gate only drives gates with higher indices, *i.e.*, we have $\mathrm{FO}(i) \subseteq \{i+1, \ldots, n\}$. We'll assume that's the case here. (This means that the gate interconnections form a directed acyclic graph.)

   To illustrate the notation, a simple digital circuit with $n = 4$ gates, each with 2 inputs, is shown below. For this circuit we have

$$\mathrm{FO}(1) = \{3, 4\}, \quad \mathrm{FO}(2) = \{3\}, \quad \mathrm{FO}(3) = \emptyset, \quad \mathrm{FO}(4) = \emptyset.$$



   The 3 input signals arriving from the left are called *primary inputs*, and the 3 output signals emerging from the right are called *primary outputs* of the circuit. (You don't need to know this, however, to solve this problem.)

   Each gate has a (real) *scale factor* or *size* x_i. These scale factors are the design variables in the gate sizing problem. They must satisfy $1 \leq x_i \leq x^{\max}$, where $x^{\max}$ is a given maximum allowed gate scale factor (typically on the order of 100). The total area of the circuit has the form

$$A = \sum_{i=1}^{n} a_i x_i,$$

   where a_i are positive constants.

   Each gate has an *input capacitance* C_i^{in}, which depends on the scale factor x_i as

$$C_i^{\mathrm{in}} = \alpha_i x_i,$$

   where α_i are positive constants.

Each gate has a *delay* d_i, which is given by

$$d_i = \beta_i + \gamma_i C_i^{\text{load}}/x_i,$$

where β_i and γ_i are positive constants, and C_i^{load} is the *load capacitance* of gate i. Note that the gate delay d_i is always larger than β_i, which can be intepreted as the minimum possible delay of gate i, achieved only in the limit as the gate scale factor becomes large.

The load capacitance of gate i is given by

$$C_i^{\text{load}} = C_i^{\text{ext}} + \sum_{j \in \text{FO}(i)} C_j^{\text{in}},$$

where C_i^{ext} is a positive constant that accounts for the capacitance of the interconnect wires and external circuitry.

We will follow a simple design method, which assigns an equal delay T to all gates in the circuit, *i.e.*, we have $d_i = T$, where $T > 0$ is given. For a given value of T, there may or may not exist a feasible design (*i.e.*, a choice of the x_i, with $1 \leq x_i \leq x^{\max}$) that yields $d_i = T$ for $i = 1, \ldots, n$. We can assume, of course, that $T > \max_i \beta_i$, *i.e.*, T is larger than the largest minimum delay of the gates.

Finally, we get to the problem.

(a) Explain how to find a design $x^\star \in \mathbf{R}^n$ that minimizes T, subject to a given area constraint $A \leq A^{\max}$. You can assume the fanout lists, and all constants in the problem description are known; your job is to find the scale factors x_i. Be sure to explain how you determine if the design problem is feasible, *i.e.*, whether or not there is an x that gives $d_i = T$, with $1 \leq x_i \leq x^{\max}$, and $A \leq A^{\max}$.

Your method can involve any of the methods or concepts we have seen so far in the course. It can also involve a simple search procedure, *e.g.*, trying (many) different values of T over a range.

Note: this problem concerns the general case, and not the simple example shown above.

(b) Carry out your method on the particular circuit with data given in the file `gate_sizing_data.m`. The fan-out lists are given as an $n \times n$ matrix `F`, with i, j entry one if $j \in \text{FO}(i)$, and zero otherwise. In other words, the ith row of `F` gives the fanout of gate i. The jth entry in the ith row is 1 if gate j is in the fan-out of gate i, and 0 otherwise.

Comments and hints.

- You do not need to know anything about digital circuits; *everything* you need to know is stated above.
- Yes, this problem *does* belong on the EE263 midterm.

Solution.

(a) We define the fanout matrix F as $F_{ij} = 1$, if $j \in \mathrm{FO}(i)$, and $F_{ij} = 0$ otherwise. The matrix F is *strictly upper triangular*, since $\mathrm{FO}(i) \subseteq \{i+1, \ldots, n\}$.

Using the formulas given above, and $d_i = T$, we have

$$\begin{aligned} T &= d_i \\ &= \beta_i + \gamma_i \frac{C_i^{\mathrm{load}}}{x_i} \\ &= \beta_i + \gamma_i \frac{C_i^{\mathrm{ext}} + \sum_{j \in \mathrm{FO}(i)} C_j^{\mathrm{in}}}{x_i} \\ &= \beta_i + \gamma_i \frac{C_i^{\mathrm{ext}} + \sum_{j \in \mathrm{FO}(i)} \alpha_j x_j}{x_i}. \end{aligned}$$

Multiplying by x_i we get the equivalent equations

$$Tx_i = \beta_i x_i + \gamma_i \left(C_i^{\mathrm{ext}} + \sum_{j \in \mathrm{FO}(i)} \alpha_j x_j \right),$$

which we can express in matrix form as

$$Tx = \mathbf{diag}(\beta)x + \mathbf{diag}(\gamma)C^{\mathrm{ext}} + \mathbf{diag}(\gamma)F\,\mathbf{diag}(\alpha)x.$$

Defining

$$K = \mathbf{diag}(\beta) + \mathbf{diag}(\gamma)F\,\mathbf{diag}(\alpha),$$

we can write the equations as

$$(TI - K)x = \mathbf{diag}(\gamma)C^{\mathrm{ext}},$$

a set of n linear equations in n unknowns. So this problem really does belong in EE263, after all.

For choices of T for which $TI - K$ is nonsingular, there is only one solution of this set of linear equations,

$$x = (TI - K)^{-1}\,\mathbf{diag}(\gamma)C^{\mathrm{ext}}.$$

If this x happens to satisfy $1 \leq x_i \leq x^{\max}$, and $A = a^T x \leq A^{\max}$, then it is a feasible design. Our job, then, is to find the smallest T for which this occurs. If it occurs for no T, then the problem is infeasible.

Let's analyze the issue of singularity of $TI - K$. The matrix K is upper triangular, with diagonal elements β_i. So $TI - K$ is upper triangular, with diagonal elements $T - \beta_i$. But these are all positive, by our assumption. So the matrix $TI - K$ is nonsingular.

Thus, for each value of T (larger than $\max_i \beta_i$) there is exactly one possible choice of gate sizes. Among the ones that are feasible, we have to choose the one corresponding to the smallest value of T.

We can solve this problem by examing a reasonable range of values of T, and for each value, finding x. We check whether x is feasible, by looking at $\min_i x_i$, $\max_i x_i$, and A. We take our final design as the one which is feasible, and has smallest value of T. Alternatively, we can start with a value of T just a little bit larger than $\max_i \beta_i$, then increase T until we find our first feasible x, which we take as our solution.

(b) The following code generatea x for a range of value of T, and plots $\min_i x_i$, $\max_i x_i$, and A, versus T.

```
gate_sizing_data

deltaT=0.001;
Trange=max(beta)+deltaT:deltaT:6;
i=1;
for T=Trange
    K=diag(beta)+diag(gamma)*F*diag(alpha);
    x=(T*eye(n)-K)\diag(gamma)*Cext;
    maxX(i)=max(x);
    minX(i)=min(x);
    Area(i)=a'*x;
    i=i+1;
end

res=Area<=Amax & minX>=1 & (maxX<=xmax);
index=find(res);
T=Trange(index(1))

subplot(3,1,1)
plot(Trange,minX)
ylabel('minx')
axis([2 6 0 4])
line([2,6],[1,1],'Color','r')
grid on
subplot(3,1,2)
plot(Trange,maxX)
ylabel('maxx')
axis([2 6 0 150])
line([2,6],[100,100],'Color','r')
grid on
subplot(3,1,3)
```

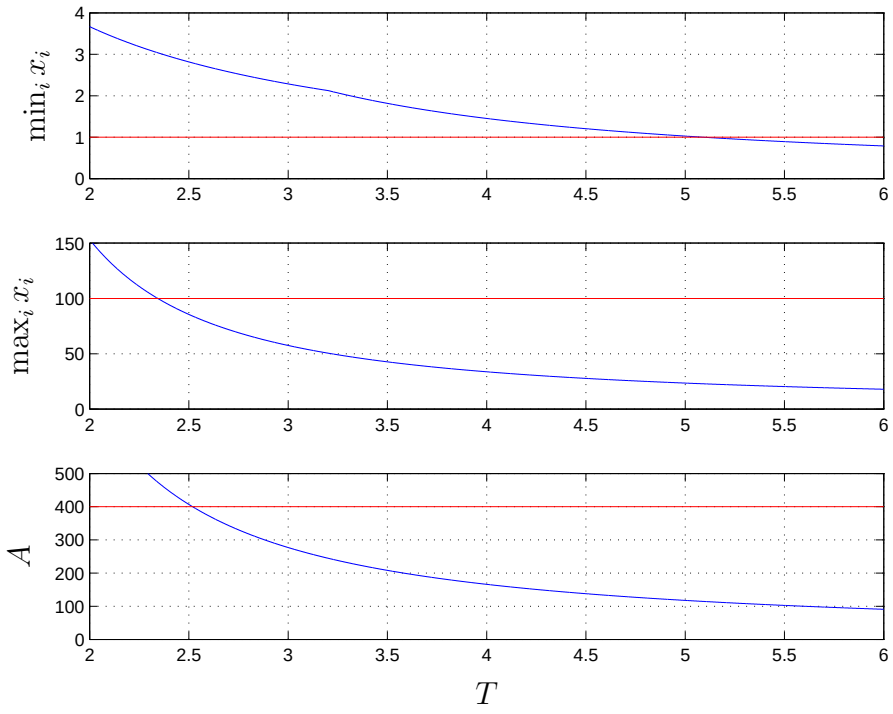


Figure 4: $\max_i x_i, \min_i x_i$, and A versus T.

```
plot(Trange,Area)
xlabel('T')
ylabel('A')
axis([2 6 0 500])
line([2,6],[400,400],'Color','r')
grid on
```

The output of the code is

```
T= 2.5194
```

Figure 4 shows how the minimum and maximum gate sizes, and the total area, vary with T, with the blue lines showing the limits. This shows that the feasible designs correspond to $2.5194 \leq T \leq 5.088$.

A few more comments about this problem:

- Since the matrix $TI - K$ is upper triangular, we can solve for x very, very quickly. In fact, if we use sparse matrix operations, we can easily compute x very quickly (seconds or less) for a problem with $n = 10^5$ gates or more. You didn't need to know this; we're just pointing it out for fun.

- The plots above show that as T increases, all of gate sizes decrease. This implies that $\min_i x_i$, $\max_i x_i$, and A all decrease as T increases. This means you can use a more efficient bisection search to find the optimal T. Again, you didn't need to know this; we're just pointing it out.

5. *Oh no. It's the dreaded theory problem.* In the list below there are 11 statements about two square matrices A and B in $\mathbf{R}^{n\times n}$.

   (a) $\mathcal{R}(B) \subseteq \mathcal{R}(A)$.

   (b) there exists a matrix $Y \in \mathbf{R}^{n\times n}$ such that $B = YA$.

   (c) $AB = 0$.

   (d) $BA = 0$.

   (e) $\mathbf{rank}([\ A \quad B\ ]) = \mathbf{rank}(A)$.

   (f) $\mathcal{R}(A) \perp \mathcal{N}(B^T)$.

   (g) $\mathbf{rank}(\begin{bmatrix} A \\ B \end{bmatrix}) = \mathbf{rank}(A)$.

   (h) $\mathcal{R}(A) \subseteq \mathcal{N}(B)$.

   (i) there exists a matrix $Z \in \mathbf{R}^{n\times n}$ such that $B = AZ$.

   (j) $\mathbf{rank}([\ A \quad B\ ]) = \mathbf{rank}(B)$.

   (k) $\mathcal{N}(A) \subseteq \mathcal{N}(B)$.

   Your job is to collect them into (the largest possible) groups of equivalent statements. Two statements are equivalent if each one implies the other. For example, the statement 'A is onto' is equivalent to '$\mathcal{N}(A) = \{0\}$' (when A is square, which we assume here), because every square matrix that is onto has zero nullspace, and vice versa. Two statements are not equivalent if there exist (real) square matrices A and B for which one holds, but the other does not. A group of statements is equivalent if any pair of statements in the group is equivalent.

   We want *just* your answer, which will consist of lists of mutually equivalent statements. We will not read any justification. If you add any text to your answer, as in 'c and e are equivalent, provided A is nonsingular', we will mark your response as wrong.

   Put your answer in the following specific form. List each group of equivalent statements on a line, in (alphabetic) order. Each new line should start with the first letter not listed above. For example, you might give your answer as

   a, c, d, h
   b, i
   e
   f, g, j, k.

   This means you believe that statements a, c, d, and h are equivalent; statements b and i are equivalent; and statements f, g, j, and k are equivalent. You also believe that the first group of statements is not equivalent to the second, or the third, and so on.

   We will take points off for false groupings (*i.e.*, listing statements in the same line when they are not equivalent) as well as for missed groupings (*i.e.*, when you list equivalent statements in different lines).

Solution. Let b_i be the ith column of B.

$$\begin{aligned}
\mathcal{R}(B) \subseteq \mathcal{R}(A) &\Leftrightarrow \text{every column of } B \text{ is in the range of } A \\
&\Leftrightarrow \text{there exists a vector } z_i \text{ such that } b_i = Az_i \\
&\Leftrightarrow \text{there exists a matrix } Z \in \mathbf{R}^{n\times n} \text{ such that } B = AZ \\
&\Leftrightarrow \mathbf{rank}([\ A \quad B\ ]) = \mathbf{rank}(A). \qquad (1)
\end{aligned}$$

This shows that statements a, e and i are equivalent.

$$\begin{aligned}
\mathcal{N}(A) \subseteq \mathcal{N}(B) &\Leftrightarrow \mathcal{N}(A)^\perp \supseteq \mathcal{N}(B)^\perp \\
&\Leftrightarrow \mathcal{R}(B^T) \subseteq \mathcal{R}(A^T) \\
&\Leftrightarrow \text{there exists a matrix } \tilde{Y} \in \mathbf{R}^{n\times n} \text{ such that } B^T = A^T\tilde{Y} \\
&\Leftrightarrow \text{there exists a matrix } Y \in \mathbf{R}^{n\times n} \text{ such that } B = YA \\
&\Leftrightarrow \mathbf{rank}([\ A^T \quad B^T\ ]) = \mathbf{rank}(A^T) \\
&\Leftrightarrow \mathbf{rank}(\begin{bmatrix} A \\ B \end{bmatrix}) = \mathbf{rank}(A). \qquad (2)
\end{aligned}$$

This shows that statements b, g and k are equivalent.

$$\begin{aligned}
\mathcal{R}(A) \subseteq \mathcal{N}(B) &\Leftrightarrow \text{for all } z \in \mathbf{R}^n,\ B(Az) = 0 \\
&\Leftrightarrow BA = 0. \qquad (3)
\end{aligned}$$

This shows that statements d and h are equivalent.

$$\begin{aligned}
\mathcal{R}(A) \perp \mathcal{N}(B^T) &\Leftrightarrow \mathcal{R}(A) \subseteq \mathcal{N}(B^T)^\perp \\
&\Leftrightarrow \mathcal{R}(A) \subseteq \mathcal{R}(B) \\
&\Leftrightarrow \mathbf{rank}([\ A \quad B\ ]) = \mathbf{rank}(B). \qquad (4)
\end{aligned}$$

This shows that statements f and j are equivalent.

None of these groups of statements is equivalent to any other, or to c. This is demonstrated by the following counterexamples.

Take

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

Since $AB = 0$ but $BA \neq 0$, then group (3) and statement c are not equivalent. Furthermore since

$$\mathbf{rank}(\begin{bmatrix} A \\ B \end{bmatrix}) = \mathbf{rank}(A) = \mathbf{rank}(B) = 1$$

but $\mathbf{rank}([A \quad B]) = 2$, groups (2) and (1) are not equivalent. Groups (2) and (4) are not either.

When $A = B \neq 0$, $\mathcal{N}(A) = \mathcal{N}(B)$ but $AB = BA = A^2 \neq 0$. Hence groups (2) and (3) are not equivalent. Group (2) and statement c are not equivalent either.

Take

$$A = I, \quad B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

Since $\mathbf{rank}([AB]) = \mathbf{rank}(A) = 2$ but $\mathbf{rank}(B) = 1$, groups (1) and (4) are not equivalent. Furthermore since $BA \neq 0$ groups (1) and (3) are not equivalent. Since $AB \neq 0$, group (1) and statement c aren't either.

In a similar fashion, taking

$$A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad B = I,$$

shows that groups (3) and (4) are not equivalent and that statement c and group (4) aren't either.

Thus, the final answer is

a, e, i
b, g, k
c
d, h
f, j.

6. *Smooth interpolation on a 2D grid.* This problem concerns arrays of real numbers on an $m \times n$ grid. Such as array can represent an image, or a sampled description of a function defined on a rectangle. We can describe such an array by a matrix $U \in \mathbf{R}^{m \times n}$, where U_{ij} gives the real number at location i, j, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. We will think of the index i as associated with the y axis, and the index j as associated with the x axis.

   It will also be convenient to describe such an array by a vector $u = \mathbf{vec}(U) \in \mathbf{R}^{mn}$. Here **vec** is the function that stacks the columns of a matrix on top of each other:

$$\mathbf{vec}(U) = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix},$$

   where $U = [u_1 \cdots u_n]$. To go back to the array representation, from the vector, we have $U = \mathbf{vec}^{-1}(u)$. (This looks complicated, but isn't; $\mathbf{vec}^{-1}$ just arranges the elements in a vector into an array.)

   We will need two linear functions that operate on $m \times n$ arrays. These are simple approximations of partial differentiation with respect to the x and y axes, respectively. The first function takes as argument an $m \times n$ array U and returns an $m \times (n-1)$ array V of forward (rightward) differences:

$$V_{ij} = U_{i,j+1} - U_{ij}, \quad i = 1, \ldots, m, \quad j = 1, \ldots, n-1.$$

   We can represent this linear mapping as multiplication by a matrix $D_x \in \mathbf{R}^{m(n-1) \times mn}$, which satisfies

$$\mathbf{vec}(V) = D_x \mathbf{vec}(U).$$

   (This looks scarier than it is—each row of the matrix D_x has exactly one $+1$ and one -1 entry in it.)

   The other linear function, which is a simple approximation of partial differentiation with respect to the y axis, maps an $m \times n$ array U into an $(m-1) \times n$ array W, is defined as

$$W_{ij} = U_{i+1,j} - U_{ij}, \quad i = 1, \ldots, m-1, \quad j = 1, \ldots, n.$$

   We define the matrix $D_y \in \mathbf{R}^{(m-1)n \times mn}$, which satisfies $\mathbf{vec}(W) = D_y \mathbf{vec}(U)$.

   We define the *roughness* of an array U as

$$R = \|D_x \mathbf{vec}(U)\|^2 + \|D_y \mathbf{vec}(U)\|^2.$$

   The roughness measure R is the sum of the squares of the differences of each element in the array and its neighbors. Small R corresponds to smooth, or smoothly varying, U. The roughness measure R is zero precisely for constant arrays, *i.e.*, when U_{ij} are all equal.

Now we get to the problem, which is to interpolate some unknown values in an array in the smoothest possible way, given the known values in the array. To define this precisely, we partition the set of indices $\{1, \ldots, mn\}$ into two sets: I_{known} and I_{unknown}. We let $k \geq 1$ denote the number of known values (*i.e.*, the number of elements in I_{known}), and $mn - k$ the number of unknown values (the number of elements in I_{unknown}). We are given the values u_i for $i \in I_{\text{known}}$; the goal is to guess (or estimate or assign) values for u_i for $i \in I_{\text{unknown}}$. We'll choose the values for u_i, with $i \in I_{\text{unknown}}$, so that the resulting U is as smooth as possible, *i.e.*, so it minimizes R. Thus, the goal is to fill in or interpolate missing data in a 2D array (an image, say), so the reconstructed array is as smooth as possible.

We give the k known values in a vector $w_{\text{known}} \in \mathbf{R}^k$, and the $mn - k$ unknown values in a vector $w_{\text{unknown}} \in \mathbf{R}^{mn-k}$. The complete array is obtained by putting the entries of w_{known} and w_{unknown} into the correct positions of the array. We describe these operations using two matrices $Z_{\text{known}} \in \mathbf{R}^{mn \times k}$ and $Z_{\text{unknown}} \in \mathbf{R}^{mn \times (mn-k)}$, that satisfy

$$\mathbf{vec}(U) = Z_{\text{known}} w_{\text{known}} + Z_{\text{unknown}} w_{\text{unknown}}.$$

(This looks complicated, but isn't: Each row of these matrices is a unit vector, so multiplication with either matrix just stuffs the entries of the w vectors into particular locations in $\mathbf{vec}(U)$. In fact, the matrix $[Z_{\text{known}}\ Z_{\text{unknown}}]$ is an $mn \times mn$ permutation matrix.)

In summary, you are given the problem data w_{known} (which gives the known array values), Z_{known} (which gives the locations of the known values), and Z_{unknown} (which gives the locations of the unknown array values, in some specific order). Your job is to find w_{unknown} that minimizes R.

(a) Explain how to solve this problem. You are welcome to use any of the operations, matrices, and vectors defined above in your solution (*e.g.*, $\mathbf{vec}$, $\mathbf{vec}^{-1}$, D_x, D_y, Z_{known}, Z_{unknown}, w_{known}, ...). If your solution is valid provided some matrix is (or some matrices are) full rank, say so.

(b) Carry out your method using the data created by `smooth_interpolation.m`. The file gives m, n, w_{known}, Z_{known} and Z_{unknown}. This file also creates the matrices D_x and D_y, which you are welcome to use. (This was *very* nice of us, by the way.) You are welcome to look at the code that generates these matrices, but you do not need to understand it. For this problem instance, around 50% of the array elements are known, and around 50% are unknown.

The mfile also includes the original array `Uorig` from which we removed elements to create the problem. This is just so you can see how well your smooth reconstruction method does in reconstructing the original array. Of course, you cannot use `Uorig` to create your interpolated array `U`.

To visualize the arrays use the Matlab command `imagesc()`, with matrix argument. If you prefer a grayscale image, or don't have a color printer, you can

issue the command `colormap gray`. The mfile that gives the problem data will plot the original image `Uorig`, as well as an image containing the known values, with zeros substituted for the unknown locations. This will allow you to see the pattern of known and unknown array values.

Compare `Uorig` (the original array) and `U` (the interpolated array found by your method), using `imagesc()`. Hand in complete source code, as well as the plots. Be sure to give the value of roughness R of U.

Hints:

- In Matlab, $\mathbf{vec}(U)$ can be computed as `U(:)`;
- $\mathbf{vec}^{-1}(u)$ can be computed as `reshape(u,m,n)`.

Solution.

(a) We can express our roughness measure directly in terms of the vector of known values w_{known} and unknown values w_{unknown} as

$$\begin{aligned} R &= \|D_x(Z_{\text{known}}w_{\text{known}} + Z_{\text{unknown}}w_{\text{unknown}})\|^2 \\ &\quad +\|D_y(Z_{\text{known}}w_{\text{known}} + Z_{\text{unknown}}w_{\text{unknown}})\|^2 \\ &= \left\| \begin{bmatrix} D_x \\ D_y \end{bmatrix} Z_{\text{known}}w_{\text{known}} + \begin{bmatrix} D_x \\ D_y \end{bmatrix} Z_{\text{unknown}}w_{\text{unknown}} \right\|^2. \end{aligned}$$

Defining

$$A = \begin{bmatrix} D_x \\ D_y \end{bmatrix} Z_{\text{unknown}}, \quad b = -\begin{bmatrix} D_x \\ D_y \end{bmatrix} Z_{\text{known}}w_{\text{known}},$$

we can express the problem in the familiar form

$$\text{minimize} \quad \|Aw_{\text{unknown}} - b\|^2.$$

Provided A is skinny and full rank, the solution is

$$\begin{aligned} w_{\text{unknown}} &= A^\dagger b \\ &= (A^TA)^{-1}A^Tb \\ &= -\left(Z_{\text{unknown}}^T(D_x^TD_x + D_y^TD_y)Z_{\text{unknown}}\right)^{-1} \cdot \\ &\quad \cdot \left(Z_{\text{unknown}}^T(D_x^TD_x + D_y^TD_y)Z_{\text{known}}\right) w_{\text{known}}. \end{aligned}$$

When is $A \in \mathbf{R}^{(2mn-m-n)\times(mn-k)}$ skinny and full rank? It's always skinny, since $2mn - m - n \geq mn - k$. If A were not full rank, then there would exist some nonzero w with $Aw = 0$. This means that $Z_{\text{unknown}}w$ is in the nullspace of both D_x and D_y, which means that $Z_{\text{unknown}}w$ is a constant (*i.e.*, its entries are all the same). This means that we have to have $w = 0$, assuming there is at least one known array value. In other words, A is *always* full rank and skinny!

(b) w_{unknown} is easily found in Matlab with the command

```
wunkown = [Dx; Dy]*Zunknown \ -[Dx; Dy]*Zknown*wknown;
```

Yes, that really is the solution, in just one line.

Next we need to create our complete array by putting the entries of w_{known} and w_{unknown} in the correct positions of the array. We use Matlab again:

```
U = reshape([Zknown Zunknown]*[wknown; wunknown], m, n);
```

We calculate the roughness of our final array U as

```
R = norm(Dx*U(:))^2 + norm(Dy*U(:))^2
```

which for our example is $R = 12.8794$.

Finally, we graph `Uorig`, `Uobscured` and `U`, with the results shown in Figure (5).

```
subplot(221);
imagesc(Uorig)
title('Original image');

subplot(222);
imagesc(Uobscured);
title('Obscured image');

subplot(223);
imagesc(U);
title('Reconstructed image');
```

One thing you notice about the reconstructed image is, it's a really, really good approximation of the orginal image. It's very impressive; we've guessed (very well) half the entries of a (smooth) image, from the remaining half.

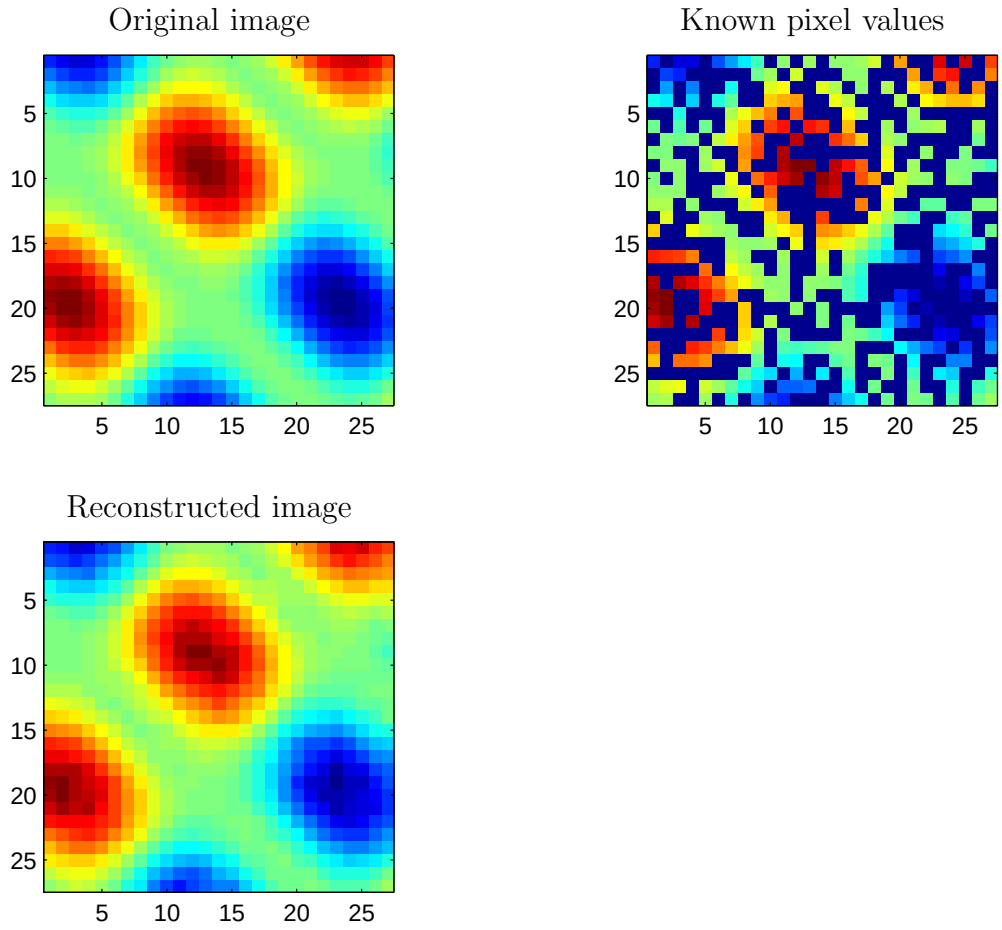
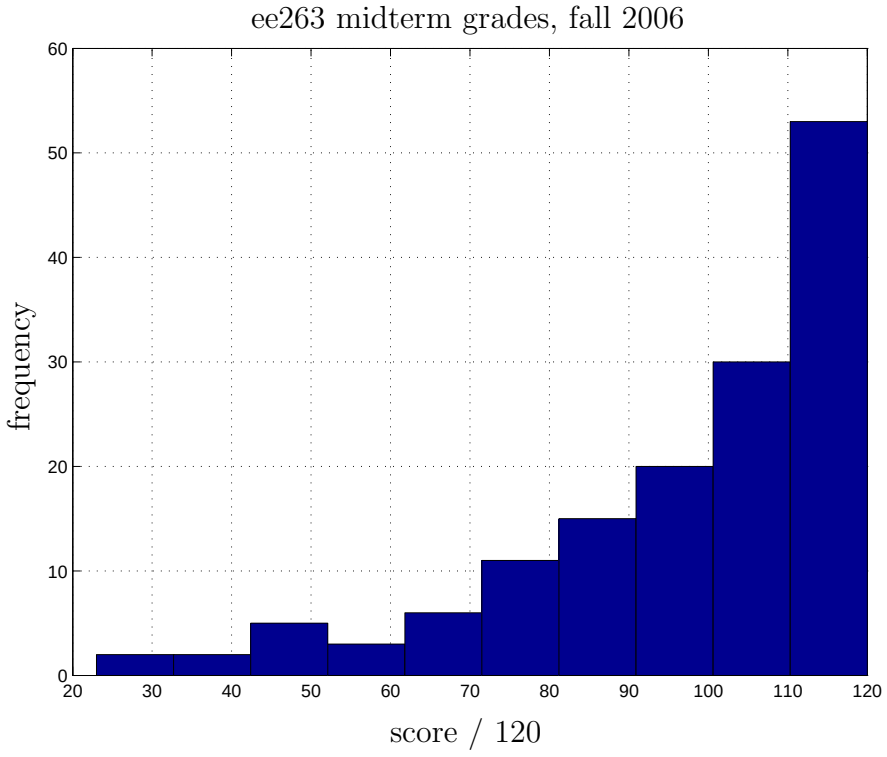


Figure 5: Original, obscured and reconstructed image arrays


ee263 midterm grades, fall 2006
frequency
score / 120
0
10
20
30
40
50
60
20
30
40
50
60
70
80
90
100
110
120