

Homework 5 Solutions

Due Thursday 10/29

1. *Projection matrices.*

A matrix $P \in \mathbb{R}^{n \times n}$ is called a *projection matrix* if $P = P^T$ and $P^2 = P$.

- Show that if P is a projection matrix then so is $I - P$.
- Suppose that the columns of $U \in \mathbb{R}^{n \times k}$ are orthonormal. Show that UU^T is a projection matrix. (Later we will show that the converse is true: every projection matrix can be expressed as UU^T for some U with orthonormal columns.)
- Suppose $A \in \mathbb{R}^{n \times k}$ is full rank, with $k \leq n$. Show that $A(A^T A)^{-1}A^T$ is a projection matrix.
- If $S \subseteq \mathbb{R}^n$ and $x \in \mathbb{R}^n$, the point y in S closest to x is called the *projection of x on S* . Show that if P is a projection matrix, then $y = Px$ is the projection of x on $\text{range}(P)$. (Which is why such matrices are called projection matrices . . .)

Solution.

- To show that $I - P$ is a projection matrix we need to check two properties:

- $I - P = (I - P)^T$
- $(I - P)^2 = I - P$.

The first one is easy: $(I - P)^T = I - P^T = I - P$ because $P = P^T$ (P is a projection matrix.) To show the second property we have

$$\begin{aligned} (I - P)^2 &= I - 2P + P^2 \\ &= I - 2P + P && \text{(since } P = P^2\text{)} \\ &= I - P \end{aligned}$$

and we are done.

- Since the columns of U are orthonormal we have $U^T U = I$. Using this fact it is easy to prove that UU^T is a projection matrix, *i.e.*, $(UU^T)^T = UU^T$ and $(UU^T)^2 = UU^T$. Clearly, $(UU^T)^T = (U^T)^T U^T = UU^T$ and

$$\begin{aligned} (UU^T)^2 &= (UU^T)(UU^T) \\ &= U(U^T U)U^T \\ &= UU^T && \text{(since } U^T U = I\text{)}. \end{aligned}$$

- First note that $(A(A^T A)^{-1}A^T)^T = A(A^T A)^{-1}A^T$ because

$$\begin{aligned} (A(A^T A)^{-1}A^T)^T &= (A^T)^T ((A^T A)^{-1})^T A^T \\ &= A ((A^T A)^T)^{-1} A^T \\ &= A(A^T A)^{-1}A^T. \end{aligned}$$

Also $(A(A^T A)^{-1}A^T)^2 = A(A^T A)^{-1}A^T$ because

$$\begin{aligned} (A(A^T A)^{-1}A^T)^2 &= (A(A^T A)^{-1}A^T) (A(A^T A)^{-1}A^T) \\ &= A ((A^T A)^{-1}A^T A) (A^T A)^{-1}A^T \\ &= A(A^T A)^{-1}A^T && \text{(since } (A^T A)^{-1}A^T A = I\text{)}. \end{aligned}$$

- (d) To show that Px is the projection of x on $\text{range}(P)$ we verify that the “error” $x - Px$ is orthogonal to *any* vector in $\text{range}(P)$. Since $\text{range}(P)$ is nothing but the span of the columns of P we only need to show that $x - Px$ is orthogonal to the columns of P , or in other words, $P^T(x - Px) = 0$. But

$$\begin{aligned} P^T(x - Px) &= P(x - Px) && \text{(since } P = P^T\text{)} \\ &= Px - P^2x \\ &= 0 && \text{(since } P^2 = P\text{)} \end{aligned}$$

and we are done.

2. Gradient of some common functions.

Recall that the gradient of a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, at a point $x \in \mathbb{R}^n$, is defined as the vector

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix},$$

where the partial derivatives are evaluated at the point x . The first order Taylor approximation of f , near x , is given by

$$\hat{f}_{\text{tay}}(z) = f(x) + \nabla f(x)^T(z - x).$$

This function is affine, *i.e.*, a linear function plus a constant. For z near x , the Taylor approximation \hat{f}_{tay} is very near f . Find the gradient of the following functions. Express the gradients using matrix notation.

- (a) $f(x) = a^T x + b$, where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$.
 (b) $f(x) = x^T A x$, for $A \in \mathbb{R}^{n \times n}$.
 (c) $f(x) = x^T A x$, where $A = A^T \in \mathbb{R}^{n \times n}$. (Yes, this is a special case of the previous one.)

Solution.

- (a) We can write $f(x)$ as

$$f(x) = \sum_{i=1}^n a_i x_i + b,$$

where a_i and x_i are the i -th components of the vector a and x respectively. Thus, we have

$$\frac{\partial f}{\partial x_i} = a_i,$$

which gives that $\nabla f(x) = a$.

- (b) For $f(x) = x^T A x$, we have

$$f(x) = \sum_{i=1}^n \sum_{j=1}^n x_i a_{ij} x_j.$$

Thus, we have

$$\frac{\partial f}{\partial x_k} = \sum_{j=1}^n a_{kj} x_j + \sum_{i=1}^n a_{ik} x_i.$$

Define a_k to be the k -th column and b_k to be the k -th row of matrix A . Then we have

$$\frac{\partial f}{\partial x_k} = b_k^T x + a_k^T x$$

Writing this in matrix form, we get that $\nabla f(x) = (A + A^T)x$.

(c) For the case when $A = A^T$, we get that $\nabla f(x) = 2Ax$

3. Least-squares deconvolution.

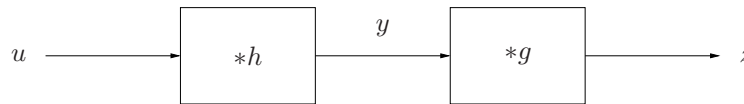
A communications channel is modeled by a finite-impulse-response (FIR) filter:

$$y(t) = \sum_{\tau=0}^{n-1} u(t-\tau)h(\tau),$$

where $u : \mathbb{Z} \rightarrow \mathbb{R}$ is the channel input sequence, $y : \mathbb{Z} \rightarrow \mathbb{R}$ is the channel output, and $h(0), \dots, h(n-1)$ is the impulse response of the channel. In terms of discrete-time convolution we write this as $y = h * u$. You will design a *deconvolution filter* or *equalizer* which also has FIR form:

$$z(t) = \sum_{\tau=0}^{m-1} y(t-\tau)g(\tau),$$

where $z : \mathbb{Z} \rightarrow \mathbb{R}$ is the filter output, y is the channel output, and $g(0), \dots, g(m-1)$ is the impulse response of the filter, which we are to design. This is shown in the block diagram below.



The goal is to choose $g = (g(0), \dots, g(m-1))$ so that the filter output is approximately the channel input delayed by D samples, *i.e.*, $z(t) \approx u(t-D)$. Since $z = g * h * u$ (discrete-time convolution), this means that we'd like

$$(g * h)(t) \approx \begin{cases} 0 & t \neq D, \\ 1 & t = D \end{cases}$$

We will refer to $g * h$ as the *equalized impulse response*; the goal is to make it as close as possible to a D -sample delay. Specifically, we want the *least-squares* equalizer is g that minimizes the sum-of-squares error

$$\sum_{t \neq D} (g * h)(t)^2,$$

subject to the constraint

$$(g * h)(D) = 1.$$

To solve the problem below you'll need to get the file `deconv_data.m` from the class web page in the *Matlab files* section. It will define the channel impulse response h as a Matlab vector `h`. (Indices in Matlab run from 1 to n , while the argument of the channel impulse response runs from $t = 0$ to $t = n - 1$, so `h(3)` in Matlab corresponds to $h(2)$.)

- Find the least-squares equalizer g , of length $m = 20$, with delay $D = 12$. Plot the impulse responses of the channel (h) and the equalizer (g). Plot the equalized impulse response ($g * h$).
- The vector y (also defined in `deconv_data.m`) contains the channel output corresponding to a signal u passed through the channel (*i.e.*, $y = h * u$). The signal u is binary, *i.e.*, $u(t) \in \{-1, 1\}$, and starts at $t = 0$ (*i.e.*, $u(t) = 0$ for $t < 0$). Pass y through the least-squares equalizer found in part a, to form the signal z . Give a histogram plot of the amplitude distribution of both y and z . (You can remove the first and last D samples of z before making the histogram plot.) Comment on what you find.

Matlab hints: The command `conv` convolves two vectors; the command `hist` plots a histogram (of the amplitude distribution).

Solution.

We are to solve the following minimization problem:

$$\text{Minimize } \sum_{t \neq D} (g * h)(t)^2 \quad \text{subject to } (g * h)(D) = 1$$

i.e.,

$$\text{Minimize } \sum_{t \neq D} \left(\sum_{\tau=0}^{m-1} h(t-\tau)g(\tau) \right)^2 \quad \text{subject to } \sum_{\tau=0}^{m-1} h(D-\tau)g(\tau) = 1$$

The outermost sum looks infinite, but since h is nonzero only on $\{0, \dots, n-1\}$ g is nonzero only on $\{0, \dots, m-1\}$, it's enough to sum over $t \in \{0, \dots, n+m-1\} \setminus \{D\}$. The convolution can be expressed as $(g * h)(t) = c_t^T g$ where (assuming $m > n$):

$$c_t^T = \begin{cases} \begin{bmatrix} h(t) & h(t-1) & \dots & h(0) & 0 & \dots & \dots & \dots & 0 \end{bmatrix} & \text{if } 0 \leq t < n \\ \begin{bmatrix} 0 & \dots & 0 & h(n-1) & h(n-2) & \dots & h(0) & 0 & \dots & 0 \end{bmatrix} & \text{if } n \leq t < m \\ \begin{bmatrix} 0 & \dots & \dots & \dots & 0 & h(n-1) & \dots & h(t-m+1) \end{bmatrix} & \text{if } m \leq t < n+m \\ 0 & \text{otherwise} \end{cases}$$

It seems like a good idea to restate this in matrix notation. The vector $(g * h)$ consists of $(g * h)(0)$ to $(g * h)(n+m-1)$ and the vector g is $g(0)$ to $g(m-1)$. Then:

$$(g * h) = \begin{bmatrix} h(0) & 0 & & & \dots & & & 0 \\ \vdots & \ddots & \ddots & & & & & \\ h(n-1) & \dots & h(0) & 0 & \ddots & & & \\ 0 & h(n-1) & \dots & h(0) & 0 & \dots & \vdots & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 0 & h(n-1) & \dots & h(0) & 0 & \\ \vdots & & & 0 & h(n-1) & \dots & h(0) & \\ & & & & \ddots & \ddots & \vdots & \\ 0 & & \dots & & & 0 & h(n-1) & \end{bmatrix} g$$

Now, what we want to minimize is the sum of the squared elements of $(g * h)$, except for the $(D+1)$ 'th one, while fulfilling the *exact* constraint that the $(D+1)$ 'th element be equal to 1. This is just a least-squares problem with an equality constraint, as discussed in the previous problem! We can write it as:

$$\text{Minimize } \|Hg - 0\| \quad \text{subject to } d^T g = 1$$

where H is the above matrix (called a Toeplitz matrix) with the $(D+1)$ 'th row removed, and d^T is that row. The solution is then

$$g_{\text{ls}} = Q((HQ)^T(HQ))^{-1}(HQ)^T(0 - Hr) + r$$

where the columns of Q form a basis for $\text{null}(d^T)$, and r is a solution of $d^T g = 1$. For r we arbitrarily take the least-norm solution

$$r = d(d^T d)^{-1} = \frac{d}{\|d\|^2}$$

and we compute Q by QR-factorizing $(d^T)^T = d$ as explained in the previous problem.

(a) Here's a Matlab implementation.

```

deconv_data; % defines impulse response h and binary data y
n = length(h);
m = 20; % length of equalizer
conv_length = m+n-1; % length of (g*h) (equalized channel)
D = 12; % desired delay
preH = toeplitz([h;zeros(m-1,1)], [h(1);zeros(m-1,1)]);
dT = preH(D+1,:); % (D+1)'th row
H = preH([1:D (D+2):conv_length],:); % all rows except (D+1)
imp_eq_desired = zeros(conv_length-1,1);
[preQ,R]=qr(dT');
Q=preQ(:,2:end); % Q2 part is all columns except first one
r=pinv(dT)*1;
g_ls = Q*pinv(H*Q)*(imp_eq_desired-H*r)+r;

```

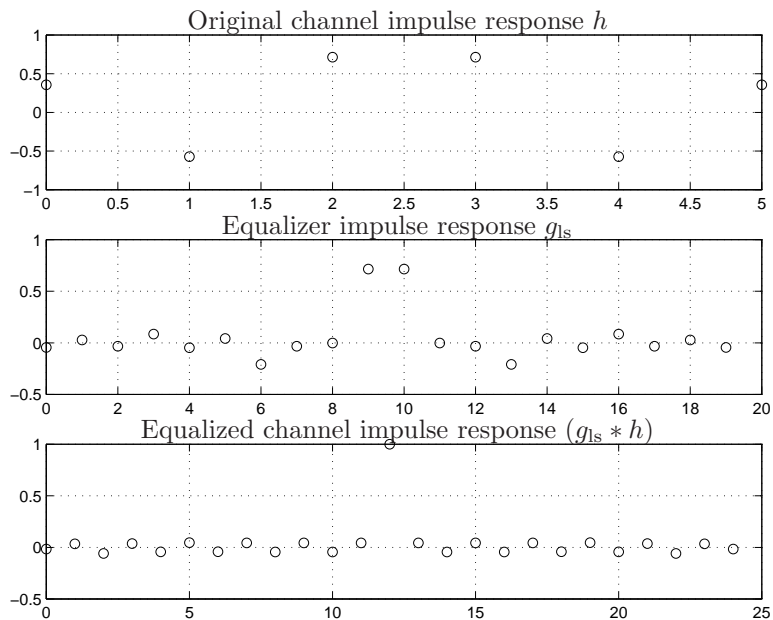
The optimal equalizer g is found to be:

```

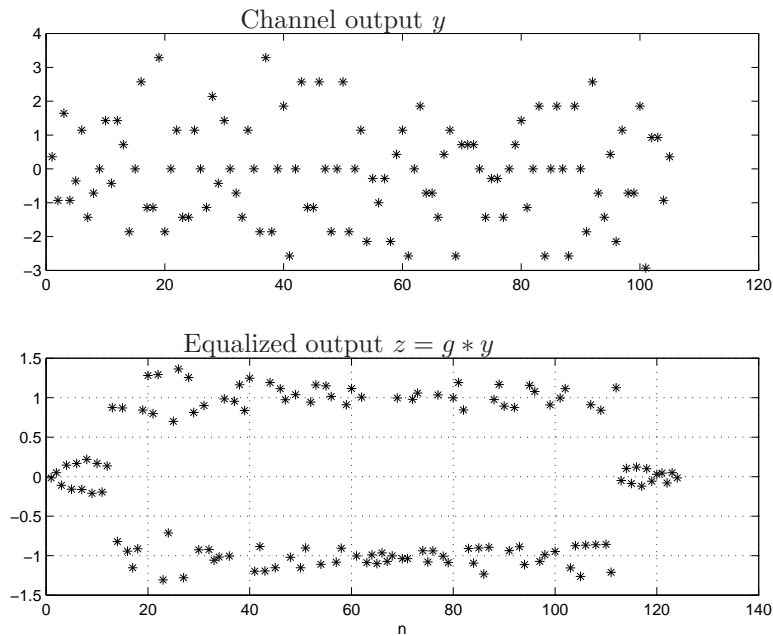
g_ls =
-0.0424
0.0264
-0.0316
0.0809
-0.0444
0.0401
-0.1997
-0.0314
-0.0021
0.6846
0.6846
-0.0021
-0.0314
-0.1997
0.0401
-0.0444
0.0809
-0.0316
0.0264
-0.0424

```

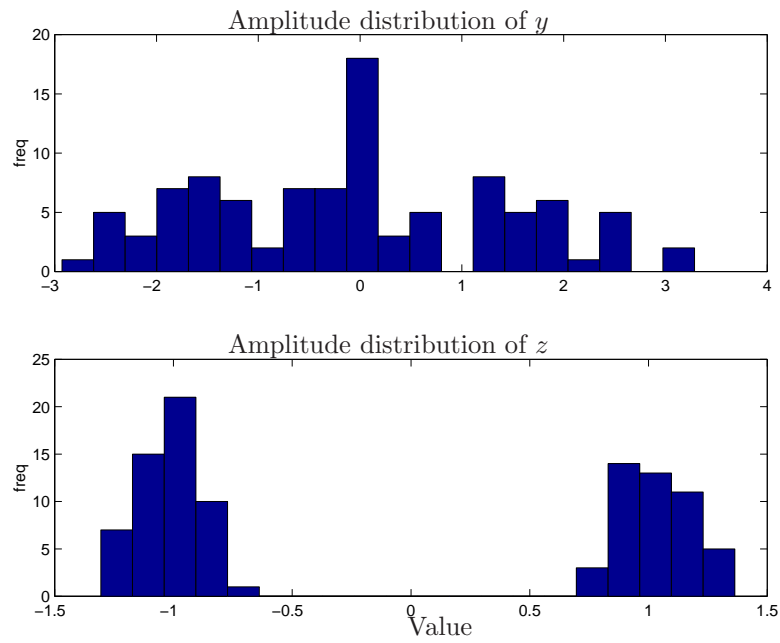
The plots below show the channel and equalizer impulse response, and their convolution, the equalized response. It seems like we succeeded in our task: the equalized impulse response is exactly 1 at $t = D$ (easily verified in Matlab) and pretty small everywhere else (RMS value outside of $t = D$ is 0.0427).



(b) Computing the equalized output is a simple matter of typing `z=conv(y,g_ls)`; in Matlab. Below are plotted the channel output and the equalized output.



As the top plot shows, the binary signal u has been ‘smeared’ in passing through the channel, resulting in y which looks anything but binary. In communications this is called *inter-symbol interference*. Looking at y , you can see that it would be very difficult to guess whether $x(t)$ was $+1$ or -1 . After we pass y through the deconvolution filter, the ‘smearing’ has been pretty much undone. In communications, we’d say that the equalizer has removed much of the intersymbol interference. By examining the (deconvolved) signal $g * y$, which is concentrated near the values $+1$ and -1 , it is clear we can make a very good guess as to whether the input signal was $+1$ or -1 (for example, by just taking the sign of $g * h$). Finally the amplitude distributions (histograms) of y and z are shown below. Before making the histogram of z , we removed the first and last D values, because the equalizer does not work correctly on them (due to convolution edge effect).



The distribution of y looks ‘smeared’ between -3 and $+3$ (roughly), but the distribution of the deconvolved (equalized) signal z looks fairly well separated and concentrated around -1 and $+1$, hence close to a binary signal.

4. Estimation with sensor offset and drift.

We consider the usual estimation setup:

$$y_i = a_i^T x + v_i, \quad i = 1, \dots, m,$$

where

- y_i is the i th (scalar) measurement
- $x \in \mathbb{R}^n$ is the vector of parameters we wish to estimate from the measurements
- v_i is the sensor or measurement error of the i th measurement

In this problem we assume the measurements y_i are taken at times evenly spaced, T seconds apart, starting at time $t = T$. Thus, y_i , the i th measurement, is taken at time $t = iT$. (This isn’t really material; it just makes the interpretation simpler.) You can assume that $m \geq n$ and the measurement matrix

$$A = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix}$$

is full rank (*i.e.*, has rank n). Usually we assume (often implicitly) that the measurement errors v_i are random, unpredictable, small, and centered around zero. (You don’t need to worry about how to make this idea precise.) In such cases, least-squares estimation of x works well. In some cases, however, the measurement error includes some *predictable* terms. For example, each sensor measurement might include a (common) *offset* or *bias*, as well as a term that grows linearly with time (called a *drift*). We model this situation as

$$v_i = \alpha + \beta iT + w_i$$

where α is the sensor bias (which is unknown but the *same* for all sensor measurements), β is the drift term (again the same for all measurements), and w_i is part of the sensor

error that is unpredictable, small, and centered around 0. If we knew the offset α and the drift term β we could just subtract the predictable part of the sensor signal, *i.e.*, $\alpha + \beta iT$ from the sensor signal. But we're interested in the case where we don't know the offset α or the drift coefficient β . Show how to use least-squares to *simultaneously* estimate the parameter vector $x \in \mathbb{R}^n$, the offset $\alpha \in \mathbb{R}$, and the drift coefficient $\beta \in \mathbb{R}$. Clearly explain your method. If your method always works, say so. Otherwise describe the conditions (on the matrix A) that must hold for your method to work, and give a simple example where the conditions don't hold.

Solution.

Substituting the expression for the noise into the measurement equation gives

$$y_i = a_i^T x + \alpha + \beta iT + w_i \quad i = 1, \dots, m.$$

In matrix form we can write this as

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_y = \underbrace{\begin{bmatrix} a_1^T & 1 & T \\ a_2^T & 1 & 2T \\ \vdots & \vdots & \vdots \\ a_m^T & 1 & mT \end{bmatrix}}_{\tilde{A}} \underbrace{\begin{bmatrix} x \\ \alpha \\ \beta \end{bmatrix}}_{\tilde{x}}.$$

If \tilde{A} is skinny ($m \geq n + 2$) and full-rank, least-squares can be used to estimate x , α , and β . In that case,

$$\tilde{x}_{\text{ls}} = \begin{bmatrix} x_{\text{ls}} \\ \alpha_{\text{ls}} \\ \beta_{\text{ls}} \end{bmatrix} = (\tilde{A}^T \tilde{A})^{-1} \tilde{A}^T y.$$

The requirement that \tilde{A} be skinny (or at least, square) makes perfect sense: you can't extract $n + 2$ parameters (*i.e.*, x , α , β) from fewer than $n + 2$ measurements. Even if A is skinny and full-rank, \tilde{A} may not be. For example, with

$$A = \begin{bmatrix} 2 & 0 \\ 2 & 1 \\ 2 & 0 \\ 2 & 1 \end{bmatrix},$$

we have

$$\tilde{A} = \begin{bmatrix} 2 & 0 & 1 & T \\ 2 & 1 & 1 & 2T \\ 2 & 0 & 1 & 3T \\ 2 & 1 & 1 & 4T \end{bmatrix},$$

which is not full-rank. In this example we can understand exactly what happened. The first column of A , which tells us how the sensors respond to the first component of x , has exactly the same form as an offset, so it is not possible to separate the offset from the signal induced by x_1 . In the general case, \tilde{A} is not full rank only if some linear combinations of the sensor signals looks exactly like an offset or drift, or some linear combination of offset and drift. Some people asserted that \tilde{A} is full rank if the vector of ones and the vector $(T, 2T, \dots, mT)$ are not in the span of the columns of A . This is false. Several people went into the case when \tilde{A} is not full rank in great detail, suggesting regularization and other things. We weren't really expecting you to go into such detail about this case. Most people who went down this route, however, failed to mention the *most important thing* about what happens. When \tilde{A} is not full rank, *you cannot separate the offset and drift parameters from the parameter x by any means at all*. Regularization means that the numerics will work, but the result will be quite meaningless. Some people pointed out that the normal equations always have a solution, even when \tilde{A} is not full rank. Again, this is true, but the most important thing here is that even if you solve the

normal equations, the results are meaningless, since you cannot separate out the offset and drift terms from the parameter x . Accounting for known noise characteristics (like offset and drift) can greatly improve estimation accuracy. The following matlab code shows an example.

```
T = 60;           % Time between samples
alpha = 3;       % sensor offset
beta = .05;      % sensor drift constant
num_x = 3;
num_y = 8;
A = [ 1     4     0
      2     0     1
     -2    -2     3
     -1     1    -4
     -3     1     1
      0    -2     2
      3     2     3
      0    -4    -6 ]; % matrix whose rows are a_i^T
x = [-8; 20; 5];
% with v a (Gaussian, random) noise
y = A*x;
for i = 1:num_y;
y(i) = y(i)+alpha+beta*T*i+randn;
end;
x_ls = A\y;
for i = 1:num_y
last_col(i) = T*i;
end
A = [A ones(num_y,1) last_col'];
x_ls_with_noise_model = A\y;
norm(x-x_ls)
norm(x-x_ls_with_noise_model(1:3))
```

Additional comments. Many people correctly stated that \tilde{A} needed to be full rank and then presented a condition they claimed was equivalent. Unfortunately, many of these statements were incorrect. The most common error was to claim that if neither of the two column vectors that were appended to A in creating \tilde{A} was in $\text{range}(A)$, then \tilde{A} was full rank. As a counter-example, take

$$A = \begin{bmatrix} 2 \\ 3 \\ \vdots \\ m+1 \end{bmatrix},$$

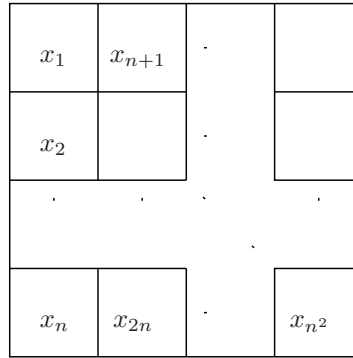
and

$$\tilde{A} = \begin{bmatrix} 2 & 1 & 1 \\ 3 & 1 & 2 \\ \vdots & \vdots & \vdots \\ m+1 & 1 & m \end{bmatrix}.$$

Since the first column of \tilde{A} is the sum of the last two, \tilde{A} has rank 2, not 3.

5. *Image reconstruction from line integrals.*

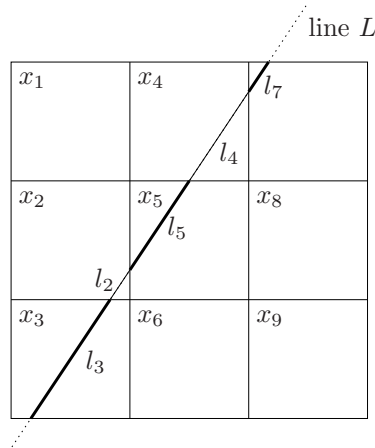
In this problem we explore a simple version of a tomography problem. We consider a square region, which we divide into an $n \times n$ array of square pixels, as shown below.



The pixels are indexed column first, by a single index i ranging from 1 to n^2 , as shown above. We are interested in some physical property such as density (say) which varies over the region. To simplify things, we'll assume that the density is constant inside each pixel, and we denote by x_i the density in pixel i , $i = 1, \dots, n^2$. Thus, $x \in \mathbb{R}^{n^2}$ is a vector that describes the density across the rectangular array of pixels. The problem is to estimate the vector of densities x , from a set of sensor measurements that we now describe. Each sensor measurement is a *line integral* of the density over a line L . In addition, each measurement is corrupted by a (small) noise term. In other words, the sensor measurement for line L is given by

$$\sum_{i=1}^{n^2} l_i x_i + v,$$

where l_i is the length of the intersection of line L with pixel i (or zero if they don't intersect), and v is a (small) measurement noise. This is illustrated below for a problem with $n = 3$. In this example, we have $l_1 = l_6 = l_8 = l_9 = 0$.



Now suppose we have N line integral measurements, associated with lines L_1, \dots, L_N . From these measurements, we want to estimate the vector of densities x . The lines are characterized by the intersection lengths

$$l_{ij}, \quad i = 1, \dots, n^2, \quad j = 1, \dots, N,$$

where l_{ij} gives the length of the intersection of line L_j with pixel i . Then, the whole set of measurements forms a vector $y \in \mathbb{R}^N$ whose elements are given by

$$y_j = \sum_{i=1}^{n^2} l_{ij} x_i + v_j, \quad j = 1, \dots, N.$$

And now the problem: you will reconstruct the pixel densities x from the line integral measurements y . The class webpage contains the M-file `tomodata.m`, which you should download and run in Matlab. It creates the following variables:

- `N`, the number of measurements (N),
- `n_pixels`, the side length in pixels of the square region (n),
- `y`, a vector with the line integrals y_j , $j = 1, \dots, N$,
- `lines_d`, a vector containing the displacement d_j , $j = 1, \dots, N$, (distance from the center of the region in pixels lengths) of each line, and
- `lines_theta`, a vector containing the angles θ_j , $j = 1, \dots, N$, of each line.

The file `tmeasure.m`, on the same webpage, shows how the measurements were computed, in case you're curious. You should take a look, but you don't need to understand it to solve the problem. We also provide the function `line_pixel_length.m` on the webpage, which you do need to use in order to solve the problem. This function computes the pixel intersection lengths for a given line. That is, given d_j and θ_j (and the side length n), `line_pixel_length.m` returns a $n \times n$ matrix, whose i, j th element corresponds to the intersection length for pixel i, j on the image. Use this information to find x , and display it as an image (of n by n pixels). You'll know you have it right when the image of x forms a familiar acronym... *Matlab hints:* Here are a few functions that you'll find useful to display an image:

- `A=reshape(v,n,m)`, converts the vector v (which must have $n*m$ elements) into an $n \times m$ matrix (the first column of A is the first n elements of v , etc.),
- `imagesc(A)`, displays the matrix A as an image, scaled so that its lowest value is black and its highest value is white,
- `colormap gray`, changes Matlab's image display mode to grayscale (you'll want to do this to view the pixel patch),
- `axis image`, redefines the axes of a plot to make the pixels square.

Note: While irrelevant to your solution, this is actually a simple version of *tomography*, best known for its application in medical imaging as the CAT scan. If an *x-ray* gets attenuated at rate x_i in pixel i (a little piece of a cross-section of your body), the j -th measurement is

$$z_j = \prod_{i=1}^{n^2} e^{-x_i l_{ij}},$$

with the l_{ij} as before. Now define $y_j = -\log z_j$, and we get

$$y_j = \sum_{i=1}^{n^2} x_i l_{ij}.$$

Solution.

The first thing to do is to restate the problem in the familiar form $y = Ax + v$. Here, $y \in \mathbb{R}^N$ is the measurement (given), $x \in \mathbb{R}^{n^2}$ is the physical quantity we are interested in, $A \in \mathbb{R}^{N \times n^2}$ is the relation between them, and $v \in \mathbb{R}^N$ is the noise, or measurement error (unknown, and we'll not worry about it). So we need to find the elements of A ... how do we do that?

$$\text{Comparing } y = Ax, \quad \text{i.e., } y_j = \sum_{i=1}^{n^2} A_{ji} x_i \quad \text{with our model } y_j = \sum_{i=1}^{n^2} l_{ij} x_i + v_j$$

it is clear that $A_{ji} = l_{ij}$, $j = 1 \dots N$, $i = 1 \dots n^2$. We have thus determined a standard linear model that we want to "invert" to find x . On running `tomodata.m`, we find that

$n = 8$ and $N = 121$, so $N > n^2$, *i.e.*, we have more rows than columns – a skinny matrix. If A is full-rank the problem is overdetermined. We can find a unique (but not exact) solution x_{ls} – the least-squares solution that minimizes $\|Ax - y\|$ – which, due to its noise-reducing properties, provides a good estimate of x (it is the *best linear unbiased estimate*). So here’s what we do: we construct A element for element by finding the length of the intersection of each line with each pixel. That’s done using the function `line_pixel_length.m` provided. A turns out to be full-rank (`rank(A)` returns 64), so we can compute a unique x_{ls} . We go ahead and solve the least-squares problem, and then display the result.

Here comes a translation of the above paragraph into Matlab code:

```
% Load the problem data.
tomodata

%% Create the matrix A, then compute its elements.
A=zeros(N,n_pixels^2); for i=1:N
    % Compute the intersection lengths of line i with all pixels
    l=line_pixel_length(lines_d(i),lines_theta(i),n_pixels);
    A(i,:)=l'; % Those lengths become the i'th row of A
end

%% Check the rank of A
r=rank(A) % Returns 900, i.e. n_pixels^2, i.e. number of
columns

%% Solve the least-squares problem: choose x to minimize
||y-Ax||
x_ls=A\y;

%% Reorder elements of vector x into a matrix X, for display.
X_ls=reshape(x_ls,n_pixels,n_pixels);

%% Display the reconstructed grayscale image.
figure(1) colormap gray imagesc(X_ls) axis image
```

And here’s the end result, the reconstructed image

