ECE-210-B Homework The Last

On Nearly Everything

Cat Van West, Spring 2024

Contents

This is the last assignment for MATLAB seminar. You've done it! Or, well, you've *almost* done it…

Each problem in this homework counts for some number of points, adding up to 3.1 total. The sections don't correspond to individual problems – they're just an approximate attempt at organizing this document. Pick as many of the problems to do as you need to. Some of these are tricky, some of these are useful, some of these are just fun. Good luck! I will, of course, be available to help.

One note: for this homework, I will **not** accept resubmissions. You can always submit new stuff, but not redo problems you've already done.

1 Linear Algebra

This is, as far as I can tell, a classic MATLAB seminar assignment, and shall remain so. At least I'm not making you all use classes. No line counts this time, but see how short you can get your functions while keeping style.

Shooting the Schmidt (+0.6) I won't pick on S. Mintchev as has been done in earlier years. I think he would consider writing a program to compute orthonormal bases good practice. Since your MATLAB skills are so sharp, you should have no trouble doing so… ;)

1. Create a function called `gram_schmidt`. The input to the function should be a 2D matrix, each column of which is assumed to be linearly independent. Implement Gram-Schmidt to create an orthonormal set of vectors from these, and store them in columns in an output matrix (same idea as the input). MATLAB's `norm` function might be helpful here.
2. Writing test code is excellent practice in programming, so let's do so. Create another function called `is_orthonormal`, which takes a single matrix as its input and returns logical 1 (true) if the columns of that matrix are orthonormal and logical 0 (false) otherwise.

   Be careful with this – direct comparison of floating-point numbers with `==` is a bad idea. Instead, apply a threshold to the difference of the two numbers like so: if $|x - \hat{x}| < \varepsilon$ then... The `eps` function might be useful for deriving ε – with a nice big fudge factor to make sure that it actually works.
3. Finally, we'd like to estimate another vector as a linear combination of these orthonormal vectors (i.e., project a vector onto the space they generate). Implement a function called `ortho_proj` which takes a (column) vector to be estimated and a matrix with orthonormal columns to project onto, and outputs the estimated (column) vector.

   Note: this function might be useful inside `gram_schmidt`, and using it there could decrease code duplication. If you like, rewrite accordingly.
4. Test all of the above on some random complex vectors (created using `rand` – these vectors will very likely be linearly independent if you make them large enough). Test the following cases:

   (a) There are more elements in a vector than there are vectors.

(b) There are as many elements in a vector as there are vectors.
(c) There are more vectors than there are elements in a vector.

Compare the results – in particular, compare the Euclidean distance (as a measure of error) between your test vectors and their projections onto an orthonormal basis.

Bonus Rewrite `gram_schmidt` so that it also works correctly on a linearly dependent set of vectors. Test it on such a set to demonstrate this.

De-Gauss, Re-Gauss (+0.2) Let's see what projecting an actual signal vector looks like.

1. Uniformly sample $\sin(x)$ on $[0, 2\pi]$ with 1000 points. Also generate five Gaussians by sampling

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-(x-\mu)^2}{2\sigma^2}$$

over the same interval, with $\sigma = 1$ and $\mu \in \{0, \pi/2, \pi, 3\pi/2, 2\pi\}$. MATLAB's `ndgrid` might be helpful for doing this compactly.

2. Use `gram_schmidt` to create an orthonormal set of vectors from the Gaussians, and `ortho_proj` to estimate the sinusoid from that set.

   Create a 2×1 subplot (using `subplot` or `tiledlayout`). Plot the original and the estimated sinusoid together on the upper plot, and the orthonormal basis function on the lower plot. Give all plots meaningful labels, titles, and legends.

My Head is Spinning (+0.3) It is a somewhat remarkable fact that most sets of five points uniquely determine a conic section. It is perhaps a *more* remarkable fact that there is a simple closed-form solution for the section in question! (Well, I say "simple", but…)

The idea is this: given a conic section with equation

$$a_1x^2 + a_2y^2 + a_3xy + a_4x + a_5y = 1,$$

and five points on it, this equation may be used to derive a system of five *linear* equations yielding the parameters in question. (See here for a relevant question and a few links.) So let's do this.

1. Create a function `generate_points` which creates a set of five random points in some (bounded!) region of the plane.

2. Create another function called `fit_conic` which takes five points in the plane, (x_1, y_1) through (x_5, y_5), and solves the linear system

$$\begin{pmatrix} x_1^2 & y_1^2 & x_1y_1 & x_1 & y_1 \\ x_2^2 & y_2^2 & x_2y_2 & x_2 & y_2 \\ x_3^2 & y_3^2 & x_3y_3 & x_3 & y_3 \\ x_4^2 & y_4^2 & x_4y_4 & x_4 & y_4 \\ x_5^2 & y_5^2 & x_5y_5 & x_5 & y_5 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

   (a perfect candidate for matrix left-divide!) for $a_1, \ldots, a_5$.

3. Test this out on some random sets of points, and plot both the points and the ellipses. There are several ways to plot this conic – `fimplicit` or `contour` both come to mind.

   Note: it is possible for the coefficient matrix in the conic fit to be singular (i.e., not invertible). Come up with a set of points that you *think* will cause this and see what happens if you try to fit a conic to them.

Bonus (A big one!) Use `appdesigner` to make an interactive app that live-fits this curve to five moveable points! I will really appreciate this. I am also very easily entertained.

Bonus (Because why not?) Extend this concept to a curve of your choosing in 3-space.

2 Symbolic Manipulation

Many of these problems give results as…well, not plots. Don't worry about printing them (it will not be readable if you're actually using MATLAB), just keep them stored somewhere convenient. And *definitely* discuss them in the comments.

Cheating at Calculus (+0.6) Solve the following problems using the symbolic toolbox. Revel in the ease your computer exhibits in munching on these equations.

1. Find all analytic solutions (see `dsolve`) to the differential equation

$$\frac{dy}{dx} = y^2 x^3.$$

   Plot a few of these solutions using `fplot` for different values of the parameter (one of the solutions should have a parameter). You can use `subs` to set said parameter.

   Note: you'll have to declare the constant parameter as a symbol using `syms` before you can refer to it!

2. Find the Laplace transform of

$$h(t) = 5\delta(t) + U(t) + e^{-t/2} \sin 3t.$$

   ($U(t)$ represents the unit step; either use `heaviside` or, if you're feeling adventurous, define it yourself with `piecewise`). Also compute it by hand (or look it up) and compare. Take the inverse Laplace transform using `ilaplace` and explain why it's *not* quite $y(t)$!

3. Given the vector field

$$\mathbf{F} = \begin{pmatrix} (3 + a^2)xz \\ ze^y \\ e^y - x^2 e^{\pi a} \end{pmatrix}$$

   where a is a free parameter, and the closed path in $\mathbb{R}^3$

$$\mathcal{C} : \mathbf{r}(t) = \begin{pmatrix} (1 - 2\cos t)\cos 3t \\ (1 - 2\cos t)\sin 3t \\ \sin t \end{pmatrix}, \quad -\pi \leqslant t \leqslant \pi,$$

find

$$\oint_C \mathbf{F} \cdot d\mathbf{r} = \int_{-\pi}^{\pi} \mathbf{F}(\mathbf{r}(t)) \cdot \mathbf{r}'(t)\, dt$$

and plot it as a function of $a \in [-3, 1]$ using `fplot`.

Note: you may want to declare your symbols as `F(x, y, z) r` rather than `F(x, y, z) r(t)`, as this makes indexing $r(t)$ easier.

Bonus Is there a value of a for which $\mathbf{F}$ is conservative (i.e., its line integral about any closed path is zero)?

It Ain't Magic (+0.2) You're either really happy or really pissed off at this point. Hopefully an exposition of the shortcomings of the symbolic toolbox doesn't make things worse.

1. Create the symbolic function

$$f(x, y) = 1 - x^2 - y^2.$$

Use `matlabFunction` to create its numeric counterpart function. (Remember how I said I wouldn't tell you how to do this? I lied.)

2. Integrate $f(x, y)$ over the unit disk using both
   (a) `int` and symbolic bounds (e.g., $\pm\sqrt{1 - x^2}$, …), and
   (b) numerical approximation (e.g, `trapz` and a 100×100 grid).

   Time both operations using `tic` and `toc`. Compare the answers (using `double` to first convert the symbolic answer into a numeric one, if you like).

3 Probability

No one really tried this last year for what should be obvious reasons – it's a bear. This deals with probability and stochastic processes both symbolically and numerically… which is fun for some people (i.e., me). Kudos to you if you attempt it!

Random Thoughts (+0.7) A Poisson process is (as I'm sure you definitely recall) a process in which discrete "events" occur in continuous time, at some mean rate λ of events per unit time. A Poisson random variable measures how many of these events are likely to occur in that unit of time, and has the p.m.f.

$$f_{\text{Poisson}}(x) = \frac{\lambda^x e^{-\lambda}}{x!},\ x = 0, 1, 2, \ldots$$

The exponential distribution measures the wait time to the first change, and has the p.d.f.

$$f_{\text{exponential}}(x) = \lambda e^{-\lambda x},\ x > 0.$$

Both of these distributions are useful in communications theory (for example, bit errors are approximately Poisson distributed if you have a good channel). Let's play with them.

1. Speaking of bit errors: suppose you have a channel with a bit error rate of about 1 in 10,000. Simulate this channel by simulating several (read: very many several) transmissions through this channel. (One way to do this would be to sample from the uniform distribution and threshold against $^{9{,}999}/_{10{,}000}$ to yield 1s where bit errors occur.)
2. Measure both
   (a) the number of errors in the first 30,000 bits for each transmission, as well as
   (b) the time until the first error (might not be one – choose a sensible method for handling this).

   (These are both vectorizable.) Plot histograms of both of these in separate subplots (normalized to represent a p.m.f/p.d.f – see the documentation for `histogram`) and overlay the appropriate (analytical) distribution functions over them.

Note: you can either define the distributions yourself or use a built-in function (`poisspdf` `exppdf`). Read the documentation *carefully* if you do.

3. Denote the wait time to the first error as W. Suppose we want to characterize this channel based on its time between errors. But, since we're engineers and a bit lazy, we realize that excessive optimization yields diminishing returns – thus, we assign the quality metric $Q = \sqrt{W}$. So: how high-quality is it?

   Recalling from probability that, for random variables W, Q such that W has p.d.f. $f(w)$ and $Q = u(W),\ u^{-1} = v$, we have

   $$g(q) = f(v(q))v'(q),\ v'(q) \geqslant 0,$$

   find the *analytical* p.d.f. (using symbolics!) of Q, along with its analytical mean and variance.

4. You know what's next, right? Based on the wait times obtained earlier, numerically compute the mean and variance of the channel quality. Do these match the analytical results?

Bonus Suppose we take several measurements (a random sample) of Q and average them in an effort to get a better estimate $\overline{Q}$. How will $\overline{Q}$ be distributed? Give a plot (or some other justification besides a comment).

4 A Game?

One Last Thing (+0.5) I came across something for PH429 that I really want to share with more than just the three others in that class. We were discussing variations on Conway's Game of Life, and I found Lenia, which extended cellular automatons to much more than I thought possible. So we're going to implement the Game of Life as the Lenia project did it: using convolution!

If you've never seen it before (or as a reminder if you have), the Game of Life is played on a grid of square cells. Each cell is either alive or dead. Each cell has eight neighbors (intuitively defined). The rules of the Game are as follows:

a. Any living cell with one neighbor or fewer dies of loneliness.
b. Any living cell with four neighbors or more dies of overcrowding.
c. Any living cell with two or three neighbors lives on to the next generation.
d. Any dead cell with exactly *three* neighbors comes back to life!

Let's see if we can make this happen.

1. We're going to have MATLAB's `conv2` do the work of adding up the neighbors. Define a kernel K to be the matrix

$$K = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

   Note that convolving this with a 2D grid of zeroes and ones will leave each element with a count of how many of its eight neighbors are alive. Test this by creating a random 9×9 array of zeroes or ones G and calling `conv2(G, K, 'same')`.

2. Create a *growth function:* a function that takes in the matrix of counts obtained above and decides whether each cell should grow (here, come to life) or shrink (here, die). For us, the function should take an input matrix U of counts and return the following values for each element:

$$\text{growth}(U) = \begin{cases} 1, & \text{where } u_{ij} = 3 \text{ exactly} \\ -1, & \text{where } u_{ij} < 2 \text{ or } u_{ij} > 3 \\ 0, & \text{else} \end{cases}$$

This can be done very efficiently with a few vectorized conditionals.

3. Create a function called `clip` that takes three arguments, `x`, `x_min`, & `x_max`, and ensures that all elements in x are between the minimum and maximum values. For example,

   ```
   clip([-1 3 9; 0 1 -3], 0, 1) == [0 1 1; 0 1 0];
   ```

4. Put it all together. Create a function called `update` that takes in the current game state G, convolves G with K to get the neighbor counts U, adds growth(U) to G, `clip` s the result between 0 and 1, and returns it as the new game state.

5. Finally, let's watch it play! Create a random board G to start with, of a size you decide works well and use `board = imshow(G);` to show the board, capturing the graphics handle in `board`.

   In a loop (yes, really), update G using `update`, and display the new board by assigning to `board.CData` and calling `drawnow` to redraw it. Also, to slow things down, call `pause .2` after displaying the new board.

Bonus Extend the game of life to *Primordia:*

- **(a)** the board should start with random integers between 0 and 9;
- **(b)** the growth function should now return 1 for u_{ij} between 20 and 24, -1 for u_{ij} strictly less than 19 or more than 31, and 0 otherwise;
- **(c)** state values should be clipped between 0 and 12, inclusive; and
- **(d)** the board should receive normalized graphics data (0 to 1) so that it shows a color gradient properly.

That's it. See what happens!

Bonus Well…I do love creativity.

A On Gram-Schmidt Orthogonalization

Since I received a few requests specifically for this last year, I'll briefly go over the Gram-Schmidt process. Hopefully this helps clarify a few points for you all.

A.1 Inner Products

Any linear algebra text will give you the definition of an inner product on a vector space. All we need is the notion that an inner product between two vectors (read: signals!) tells us how much those vectors "align" with each other. For real-valued vectors in $\mathbb{R}^n$, the standard inner product is the well-known *dot product*.

For discrete-time signal vectors, the usual inner product is

$$\langle x, y \rangle = \sum_{n=-\infty}^{\infty} x[n]\, \bar{y}[n],$$

$\bar{\cdot}$ denoting complex conjugate. In MATLAB, we never deal with anything else, so I won't discuss the usual continuous-time inner product

$$\langle x, y \rangle = \int_{-\infty}^{\infty} x(t)\, \bar{y}(t)\, dt.$$

Two vectors are *orthogonal* if the inner product between them is zero – that is, they do not align (or anti-align) at all.

A.2 Norms

Once you have an inner product, the usual L^2-norm is

$$\|x\| = \sqrt{\langle x, x \rangle}.$$

This is, notably, always real, always nonnegative, and always well-defined.

Given a vector x, one can normalize it to a unit vector u_x which points in the same direction by dividing x by its norm:

$$u_x = \frac{1}{\|x\|} x.$$

Then u_x has norm 1, which may be verified using the above expressions. Not that you care to verify it. Trust me, $\|u\|_x = 1$.

A.3 Bases

A basis for a vector space is a set of linearly independent vectors which span the space. I characterize this as "the smallest set of vectors out of which you can construct the entire space." For example, the entire $\mathbb{R}^2$ plane requires just two vectors to construct – given a unit vector in the x direction and a unit vector in the y direction, for example, any other vector in the plane is representable as a linear combination of those two:

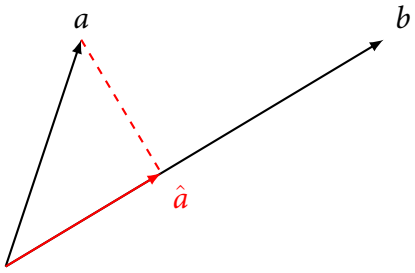
$$(5, 6.3) = 5(1, 0) + 6.3(0, 1).$$

A basis is orthogonal if the vectors in that basis are orthogonal to each other, and orthonormal if those vectors additionally have norm 1. The basis above is orthonormal.

I should note again at this point that I'm deliberately *not* being formal. Don't press any of this too hard.

A.4 Projection

Projection (read: orthogonal projection) deals with finding out how much of one vector points in the direction of another vector:



To project a onto b to get $\hat{a}$, take the inner product of a with b, then use that to scale a unit vector u_b in the direction of b appropriately:

$$\hat{a} = \left(\frac{\langle a, b\rangle}{\|b\|}\right)\left(\frac{1}{\|b\|}b\right) = \frac{\langle a, b\rangle}{\|b\|^2}b.$$

Essentially, this finds out how much of a points in the direction of b, then scales u_b to that length.

Note that the dashed line in the above is equal to $a - \hat{a}$, and is orthogonal to $\hat{a}$! This is Fred's famous Orthogonality Principle™at work – if we project a vector onto a space, what remains is orthogonal to that space.

A.5 The Gram-Schmidt Algorithm

This algorithm allows one to take a motley collection of vectors and create an orthogonal basis from them. The idea is to repeatedly do the above: given a (linearly independent) set of vectors $\{v_1, v_2, \ldots, v_n\}$, take one at a time, project it onto the space of previous vectors, and subtract that projection. This will yield a set of vectors which are orthogonal to each other and span the same space as the original set. Those basis vectors $\{w_1, w_2, \ldots, w_n\}$ are defined as follows:

$$\begin{aligned} w_1 &= v_1, \\ w_2 &= v_2 - \frac{\langle v_2, w_1 \rangle}{\|w_1\|^2} w_1, \\ w_3 &= v_3 - \frac{\langle v_3, w_1 \rangle}{\|w_1\|^2} w_1 - \frac{\langle v_3, w_2 \rangle}{\|w_2\|^2} w_2, \end{aligned}$$

and so on. In general,

$$w_k = v_k - \sum_{m=1}^{k-1} \frac{\langle v_k, w_m \rangle}{\|w_m\|^2} w_m.$$

Normalizing each w_k will yield an orthonormal basis $\{u_{w_1}, u_{w_2}, \ldots, u_{w_k}\}$, which is the goal!

A.6 This Stuff in MATLAB

I can't give everything away, but the following *should* be enough to get you started:

```
a = [0 2 7j 8];
b = [9 1 -4 -6j];
c = [6j 0 2 -1];

% take the inner product of a and b
% note: <a, b> = dot(b, a), because for some INSANE reason MATLAB conjugates
% the first argument instead of the second.
inner_ab = dot(b, a);
inner_ab = sum(a.*conj(b)); % mostly equivalent but slower

```

```
% normalize b
u_b = b/norm(b);
u_b = b/sqrt(dot(b, b)); % mostly equivalent but less explicit

% project a onto b
a_hat = dot(b, a)/dot(b, b)*b;
a_hat = dot(u_b, a)*u_b; % unit vectors are great, right?

% find the component of a orthogonal to b
a_orth = a - a_hat;

% project c onto {u_b, u_a_orth}
u_a_orth = a_orth/norm(a_orth);
c_hat = dot(u_b, c)*u_b + dot(u_a_orth, c)*u_a_orth;

% and so on...
```

hi! you made it to the last page.

there's nothing of note here. just wanted to say: thanks for sticking with it. if you've made it this far, you've read through altogether too much of my writing, ostensibly in the interest of learning something useful. i appreciate that. really. sticking with cooper long enough to get out the other side has taken a bit of a toll, honestly – seeing others willing to do that, to fight their way through and find something of value in it, is heartening.

good luck with the rest of your tenure here – i'll be long gone. wave if we meet again, though.

– cat :3