EECS 16A Designing Information Devices and Systems I
Fall 2020 Optional Bonus Problems

These are some optional problems that you might find interesting. They are not required. No submission is required. Note that OMP is not in scope for the final and absolutely nothing related to an OMP questions will be on the final; these are only shared for fun.

1. Audio File Matching

(Contributors: Aviral Pandey, Michael Kellman, Moses Won, Spencer Kent, Titan Yuan, Urmita Sikder, Vijay Govindarajan)

Learning Goal: *This problem motivates the application of correlation for pattern matching applications such as Shazam.*

Many audio processing applications rely on representing audio files as vectors, referred to as audio *signals*. Every component of the vector determines the sound we hear at a given time. We can use inner products to determine if a particular audio clip is part of a longer song, similar to an application like *Shazam*.

Let us consider a very simplified model for an audio signal, $\vec{x}$. At each timestep k, the audio signal can be either $x[k] = -1$ or $x[k] = 1$.

(a) Say we want to compare two audio files of the same length N to decide how similar they are. First, consider two vectors that are exactly identical, namely $\vec{x}_1 = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^T$ and $\vec{x}_2 = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^T$. What is the inner product of these two vectors? What if $\vec{x}_1 = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^T$ but $\vec{x}_2$ oscillates between 1 and -1? Assume that N, the length of the two vectors, is an even number.

Use this to suggest a method for comparing the similarity between a generic pair of length-N vectors.

(b) Next, suppose we want to find a short audio clip in a longer one. We might want to do this for an application like *Shazam*, which is able to identify a song from a short clip. Consider the vector of length 8, $\vec{x} = \begin{bmatrix}-1 & 1 & 1 & -1 & 1 & 1 & -1 & 1\end{bmatrix}^T$.

We want to find the short segment $\vec{y} := \begin{bmatrix}y[0] & y[1] & y[2]\end{bmatrix}^T = \begin{bmatrix}1 & 1 & -1\end{bmatrix}^T$ in the longer vector. To do this, perform the linear cross correlation between these two finite length sequences and identify at what shift(s) the linear cross correlation is maximized. Apply the same technique to identify what shift(s) gives the best match for $\vec{y} = \begin{bmatrix}1 & 1 & 1\end{bmatrix}^T$.

(If you wish, you may use iPython to do this part of the question, but you do not have to.)

(c) Now suppose our audio vector is represented using integers beyond simply just 1 and -1. Find the short audio clip $\vec{y} = \begin{bmatrix}1 & 2 & 3\end{bmatrix}^T$ in the song given by $\vec{x} = \begin{bmatrix}1 & 2 & 3 & 1 & 2 & 2 & 3 & 10\end{bmatrix}^T$. Where do you expect to see the peak in the correlation of the two signals? Is the peak where you want it to be, i.e. does it pull out the clip of the song that you intended? Why?

(If you wish, you may use iPython to do this part of the question, but you do not have to.)

(d) Let us think about how to get around the issue in the previous part. We applied cross-correlation to compare segments of $\vec{x}$ of length 3 (which is the length of $\vec{y}$) with $\vec{y}$. Instead of directly taking the cross correlation, we want to normalize each inner product computed at each shift by the magnitudes of both segments, i.e. we want to consider $\frac{<\vec{x}_k, \vec{y}>}{\|\vec{x}_k\|\|\vec{y}\|}$, where $\vec{x}_k$ is the length 3 segment starting from the k-th index of $\vec{x}$. This is referred to as normalized cross correlation. Using this procedure, now which segment matches the short audio clip best?

(e) We can use this on a more 'realistic' audio signal – refer to the IPython notebook, where we use normalized cross-correlation on a real song. Run the cells to listen to the song we are searching through, and add a simple comparison function `vector_compare` to find where in the song the clip comes from. Running this may take a couple minutes on your machine, but note that this computation can be highly optimized and run super fast in the real world! Also note that this is not exactly how Shazam works, but it draws heavily on some of these basic ideas.

2. Trilateration With Noise!

(Contributors: Craig Schindler, Michael Kellman, Rahul Arya, Sashank Krishnamurthy, Vijay Govindarajan)

Learning Goal: *This problem will help to understand how noise affects the accuracy of trilateration and consistency of the corresponding system of equations.*

In this question, we will explore how various types of noise affect the quality of triangulating a point on the 2D plane to see when trilateration works well and when it does not.

First, we will remind ourselves of the fundamental equations underlying trilateration.

(a) There are four beacons at the known coordinates $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$. You are located at some unknown coordinate (x, y) that you want to determine. The distance between your location and each of the four beacons are d_1 through d_4, respectively. Write down one equation for each beacon that relates the coordinates to the distances using the Pythagorean Theorem.

(b) Unfortunately, the above system of equations is nonlinear, so we can't use least squares or Gaussian Elimination to solve it. We will use the technique discussed in lecture to obtain a system of linear equations. In particular, we can subtract the first of the above equations from the other three to obtain three linear equations. Write down these three linear equations.

(c) Combine the three equations in the above system into a single matrix equation of the form

$$\mathbf{A} \begin{bmatrix} x \\ y \end{bmatrix} = \vec{b}.$$

(d) Now, go to the IPython notebook. In the notebook we are given three possible sets of measurements for the distances of each beacon from the receiver:

i. `ideal_distances`: the ideal set of measurements, the true distances of our receiver to the beacons. $d_1 = d_2 = d_3 = d_4 = 5$.

ii. `imperfect_distances`: imperfect measurements. $d_1 = 5.5, d_2 = 4.5, d_3 = 5, d_4 = 5$.

iii. `one_bad_distances`: mostly perfect measurements, but d_1 is a very bad measurement. $d_1 = 6.5$ and $d_2 = d_3 = d_4 = 5$.

Plot the graph illustrating the case when the receiver has received `ideal_distances` and visually solve for the position of the observer (x, y). What is the coordinate?

(e) You will now set up the above linear system using IPython. Fill in each element of the matrix $\mathbf{A}$ that you found in part (c).

(f) Similarly, fill in the entries of $\vec{b}$ from part (c) in the `make_b` function.

(g) Now, you should be able to plot the estimated position of (x, y) using the supplied code for the `ideal_distances` observations. Modify the code to estimate (x, y) for `imperfect_distances` and `one_bad_distances`, and comment on the results.

In particular, for `one_bad_distances` would you intuitively have chosen the same point that our trilateration solution did knowing that only one measurement was bad?

3. OMP Exercise

(Contributors: Aviral Pandey, Edward Doyle, Titan Yuan, Urmita Sikder)

Learning Goal: *The objective of this problem is to practice the steps of Orthogonal Matching Pursuit (OMP).*

Suppose we have a vector $\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^T \in \mathbb{R}^4$. The vector $\vec{x}$ is related to the vector $\vec{b}$ in the following way:

$$\mathbf{M}\vec{x} \approx \vec{b}$$

$$\begin{bmatrix} 1 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \approx \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix}$$

For this undetermined and possibly noisy problem of finding $\vec{x}$, assume that $\vec{x}$ is sparse: it has 2 non-zero entries and 2 zero entries. Use Orthogonal Matching Pursuit to estimate x_1 to x_4.

(Note: Unlike previous examples you may have seen, we will not cross correlate $\vec{b}$ with the columns of $\mathbf{M}$. Instead, we will just compute the inner product of $\vec{b}$ with every column of $\mathbf{M}$.)

4. Sparse Imaging

(Contributors: Aviral Pandey, Gireeja Ranade, Jiazhen Chen, Nirmaan Shankar, Titan Yuan, Urmita Sikder)

Learning Goal: *This problem aims to apply Orthogonal Matching Pursuit (OMP) to reconstruct a sparse image. This is a faster recovery method for sparse images if the sparsity is known.*

Recall the imaging lab where we projected masks onto an object to scan it into our computer using a single pixel measurement device, that is, a photoresistor. In that lab, we were scanning a 30×40 image having 1200 pixels. In order to recover the image, we took exactly 1200 measurements because we wanted our 'mask matrix' $\mathbf{A}$ to be invertible. In this problem, we have a 2D image $\mathbf{I}$ of size 91×120 pixels for a total of 10920 pixels. We will explore how to reconstruct the image using only 6500 measurements.

The image is made up of mostly black pixels (represented by a zero) except for 476 white ones (represented by a one). In cases where there are a small number of non-black pixels, we can reduce the overall number of samples necessary using the orthogonal matching pursuit algorithm. This reduces the time required for scanning an image, a real-world concern for lengthy processes like MRI where people have to stay still while being imaged.

Although the imaging illumination masks we used in the lab consisted of zeros and ones, in this question, we are going to have masks with real values. Say that we have an imaging mask $\mathbf{M_0}$ of size 91×120. The measurements using this imaging mask can be represented as follows:

First, let us put every element in the matrix $\mathbf{I}$ into a column vector $\vec{i}$ of length 10920. This operation is referred to as vectorization. Likewise, let us vectorize the mask $\mathbf{M_0}$ to $\vec{m}_0$ which is a column vector of length 10920. Then the measurement using the mask $\mathbf{M_0}$ can be represented as

$$b_0 = \vec{m}_0^T \vec{i}.$$

Say we have a total of K measurements, each taken with a different illumination mask. Then, these measurements can collectively be represented as

$$\vec{b} = \mathbf{A}\vec{i},$$

where $\mathbf{A}$ is an $K \times 10920$ size matrix whose rows contain the vectorized forms of the illumination masks, that is

$$\mathbf{A} = \begin{bmatrix} — & \vec{m}_1^T & — \\ — & \vec{m}_2^T & — \\ & \vdots & \\ — & \vec{m}_K^T & — \end{bmatrix}.$$

To show that we can reduce the number of samples necessary to recover the sparse image $\mathbf{I}$, we are going to only generate 6500 masks. We will generate $\mathbf{A}$ so that the columns of $\mathbf{A}$ are approximately orthogonal with each other. The following question refers to the part of IPython notebook file accompanying this homework related to this question.

(a) In the IPython notebook, we call a function `simulate` that generates masks and the measurements. You can see the masks and the measurements in the IPython notebook file. Complete the function `OMP` that does the OMP algorithm described in lecture.

Remark: When you look at the vector `measurements` you will see that it has zero average value. Likewise, the columns of the matrix containing the masks $\mathbf{A}$ also have zero average value. To satisfy these conditions, they need to have negative values. However, in an imaging system, we cannot project negative light. One way to get around this problem is to find the smallest value of the matrix $\mathbf{A}$ and subtract it from all entries of $\mathbf{A}$ to get the actual illumination masks. This will yield masks with positive values, hence we can project them using our real-world projector. After obtaining the readings using these masks, we can remove their average value from the readings to get measurements as if we had multiplied the image using the matrix $\mathbf{A}$.

(b) Run the code `rec = OMP((height, width), sparsity, measurements, A)` and see the image being correctly reconstructed from a number of samples smaller than the number of pixels of your figure. What is the image?

(c) We have supplied code that reads a PNG file containing a sparse image, takes measurements, and performs OMP to reconstruct the original image. An example input image file is also supplied together with the code. Recover `smiley.png` using OMP with 6500 measurements.

You can answer the following parts of this question in very general terms. Try reducing the number of measurements. Does the algorithm start to fail in recovering your sparse image? Why do you think it fails?

