EECS 16A Designing Information Devices and Systems I
Spring 2016 Elad Alon, Babak Ayazifar Homework 11

This homework is due April 19, 2016, at Noon.

1. **Homework process and study group**

   Who else did you work with on this homework? List names and student ID's. (In case of hw party, you can also just describe the group.) How did you work on this homework?
   Working in groups of 3-5 will earn credit for your participation grade.

   Solution: I worked on this homework with...

   I first worked by myself for 2 hours, but got stuck on Problem 5 so I went to office hours on...

   Then I went to homework party for a few hours, where I finished the homework.

2. **Mechanical Gram-Schmidt**

   (a) Use Gram-Schmidt to find a matrix U whose columns form an orthonormal basis for the column space of V.

$$V = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

   and show that you get the same resulting vector when you project $w = [\,1,\ -1,\ 0,\ -1,\ 0\,]^T$ onto V and onto U, i.e. show that

$$V(V^TV)^{-1}V^Tw = U(U^TU)^{-1}U^Tw \tag{2}$$

   Solution: We start with the columns of V as our basis for the column space of V and we want to find an orthonormal basis for this same space using Gram-Schmidt. For notational convenience define

$$V = \begin{bmatrix} | & | & | \\ v_1 & v_2 & v_3 \\ | & | & | \end{bmatrix} \tag{3}$$

   We summarize the first few steps of the Gram-Schmidt algorithm as follows

   i. $u_1' = v_1; \qquad u_1 = u_1'/||u_1'||.$
   ii. $u_2' = v_2 - \langle v_2, u_1\rangle u_1; \qquad u_2 = u_2'/||u_2'||.$
   iii. $u_3' = v_3 - \langle v_3, u_1\rangle u_1 - \langle v_3, u_2\rangle u_2; \qquad u_3 = u_3'/||u_3'||.$

   For the column space of V, this is

   i. $u_1' = v_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$. Since u_1' is already normalized, we simply set $u_1 = u_1'$.

ii.

$$u_2' = v_2 - \langle v_2, u_1 \rangle u_1 = \begin{bmatrix}1\\1\\1\\0\\0\end{bmatrix} - \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\\1\\0\\0\end{bmatrix} \qquad u_2 = \begin{bmatrix}0\\1/\sqrt{2}\\1/\sqrt{2}\\0\\0\end{bmatrix} \tag{4}$$

iii.

$$u_3' = v_3 - \langle v_3, u_1 \rangle u_1 - \langle v_3, u_2 \rangle u_2 = \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} - \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} - \frac{2}{\sqrt{2}}\begin{bmatrix}0\\1/\sqrt{2}\\1/\sqrt{2}\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\\1\end{bmatrix} \qquad u_3 = \begin{bmatrix}0\\0\\0\\1/\sqrt{2}\\1/\sqrt{2}\end{bmatrix} \tag{5}$$

Thus the matrix U is given by

$$U = \begin{bmatrix}1 & 0 & 0\\0 & 1/\sqrt{2} & 0\\0 & 1/\sqrt{2} & 0\\0 & 0 & 1/\sqrt{2}\\0 & 0 & 1/\sqrt{2}\end{bmatrix} \tag{6}$$

Note that whatever basis we use for a subspace, when we project a vector onto that subspace, we get the same vector. For example, when we project the vector $w = [\,1,\ -1,\ 0,\ -1,\ 0\,]^T$ onto the subspace using the V basis we get

$$V(V^TV)^{-1}V^Tw = \begin{bmatrix}1&1&1\\0&1&1\\0&1&1\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&3&3\\1&3&5\end{bmatrix}^{-1}\begin{bmatrix}1&0&0&0&0\\1&1&1&0&0\\1&1&1&1&1\end{bmatrix}\begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} \tag{7}$$

$$= \begin{bmatrix}1&1&1\\0&1&1\\0&1&1\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}3/2&-1/2&0\\-1/2&1&-1/2\\0&-1/2&1/2\end{bmatrix}\begin{bmatrix}1\\0\\-1\end{bmatrix} \tag{8}$$

$$= \begin{bmatrix}1&1&1\\0&1&1\\0&1&1\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}3/2\\0\\-1/2\end{bmatrix} = \begin{bmatrix}1\\-1/2\\-1/2\\-1/2\\-1/2\end{bmatrix} \tag{9}$$

and when we project w onto the subspace using the U basis we get the same resulting vector

$$U(U^TU)^{-1}V^Tw = \begin{bmatrix}1 & 0 & 0\\ 0 & 1/\sqrt{2} & 0\\ 0 & 1/\sqrt{2} & 0\\ 0 & 0 & 1/\sqrt{2}\\ 0 & 0 & 1/\sqrt{2}\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 & 0\\ 0 & 0 & 0 & 1/\sqrt{2} & 1/\sqrt{2}\end{bmatrix}\begin{bmatrix}1\\ -1\\ 0\\ -1\\ 0\end{bmatrix} \tag{10}$$

$$= \begin{bmatrix}1 & 0 & 0\\ 0 & 1/\sqrt{2} & 0\\ 0 & 1/\sqrt{2} & 0\\ 0 & 0 & 1/\sqrt{2}\\ 0 & 0 & 1/\sqrt{2}\end{bmatrix}\begin{bmatrix}1\\ -1/\sqrt{2}\\ -1/\sqrt{2}\end{bmatrix} = \begin{bmatrix}1\\ -1/2\\ -1/2\\ -1/2\\ -1/2\end{bmatrix} \tag{11}$$

Note however that the projection using the U basis was much simpler. Since U^TU is the identity, we didn't need to do a matrix inversion.

(b) Use Gram-Schmidt to find a matrix U whose columns form an orthonormal basis for the column space of V.

$$V = \begin{bmatrix}1 & 1 & -1\\ 0 & 1 & -1\\ 1 & 0 & 0\\ 1 & -1 & 1\\ 0 & -1 & 1\end{bmatrix} \tag{12}$$

and show that you get the same resulting vector when you project $w = [\,1,\ -1,\ 0,\ -1,\ 0\,]^T$ onto V and onto U.

Solution: We repeat the process from in part (a).

i. $u_1' = \begin{bmatrix}1 & 0 & 1 & 1 & 0\end{bmatrix}^T$; $\quad u_1 = \begin{bmatrix}1/\sqrt{3} & 0 & 1/\sqrt{3} & 1/\sqrt{3} & 0\end{bmatrix}^T$

ii.

$$u_2' = \begin{bmatrix}1\\ 1\\ 0\\ -1\\ -1\end{bmatrix} - \left(\begin{bmatrix}1\\ 1\\ 0\\ -1\\ -1\end{bmatrix}^T\begin{bmatrix}1/\sqrt{3}\\ 0\\ 1/\sqrt{3}\\ 1/\sqrt{3}\\ 0\end{bmatrix}\right)\begin{bmatrix}1/\sqrt{3}\\ 0\\ 1/\sqrt{3}\\ 1/\sqrt{3}\\ 0\end{bmatrix} = \begin{bmatrix}1\\ 1\\ 0\\ -1\\ -1\end{bmatrix} \qquad u_2 = \begin{bmatrix}1/2\\ 1/2\\ 0\\ -1/2\\ -1/2\end{bmatrix} \tag{13}$$

iii.

$$u_3' = \begin{bmatrix}-1\\ -1\\ 0\\ 1\\ 1\end{bmatrix} - \left(\begin{bmatrix}-1\\ -1\\ 0\\ 1\\ 1\end{bmatrix}^T\begin{bmatrix}1/\sqrt{3}\\ 0\\ 1/\sqrt{3}\\ 1/\sqrt{3}\\ 0\end{bmatrix}\right)\begin{bmatrix}1/\sqrt{3}\\ 0\\ 1/\sqrt{3}\\ 1/\sqrt{3}\\ 0\end{bmatrix} - \left(\begin{bmatrix}-1\\ -1\\ 0\\ 1\\ 1\end{bmatrix}^T\begin{bmatrix}1/2\\ 1/2\\ 0\\ -1/2\\ -1/2\end{bmatrix}\right)\begin{bmatrix}1/2\\ 1/2\\ 0\\ -1/2\\ -1/2\end{bmatrix} = \begin{bmatrix}0\\ 0\\ 0\\ 0\\ 0\end{bmatrix} \tag{14}$$

Notice that the third vector came out as all zeros. This happened because v_3 is linearly dependent on v_1 and v_2. Thus once we have found an orthonormal basis for the span of the first two columns of V, we have found a basis for the span of the entire column space. Since v_3 is in the span of u_1

and u_2 when we subtract the projection of v_3 onto u_1 and u_2 from v_3 we get zero. This is one of the nice properties of Gram-Schmidt. If we are performing Gram-Schmidt on a set of vectors that are linearly dependent, we will simply get the zero vector when we come to a linearly dependent vector. When this happens we can simply throw out the zero vector and keep going with the algorithm.

Note also that v_1 and v_2 were already orthogonal to each other so all we really did in the Gram-Schmidt algorithm was normalize them.

$$U = \begin{bmatrix} 1/\sqrt{3} & 1/2 \\ 0 & 1/2 \\ 1/\sqrt{3} & 0 \\ 1/\sqrt{3} & -1/2 \\ 0 & -1/2 \end{bmatrix} \tag{15}$$

We now want to test projecting the vector $w = [\ 1,\ -1,\ 0,\ -1,\ 0\ ]^T$ onto the subspace using the V basis and see if we get the same as when we project onto the subspace using the U basis. Since V has linearly dependent columns we don't need all of them to span the column space of V. The first two will do. Thus the projection of w onto the column space of V using the V basis can be calculated as

$$\begin{bmatrix} | & | \\ v_1 & v_2 \\ | & | \end{bmatrix} \left(\begin{bmatrix} | & | \\ v_1 & v_2 \\ | & | \end{bmatrix}^T \begin{bmatrix} | & | \\ v_1 & v_2 \\ | & | \end{bmatrix} \right)^{-1} \begin{bmatrix} | & | \\ v_1 & v_2 \\ | & | \end{bmatrix}^T w = \tag{16}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & -1 \\ 0 & -1 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & -1 \\ 0 & -1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \tag{17}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & -1 \\ 0 & -1 \end{bmatrix} \left(\begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \tag{18}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & -1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1/3 & 0 \\ 0 & 1/4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/4 \\ 1/4 \\ 0 \\ -1/4 \\ -1/4 \end{bmatrix} \tag{19}$$

Projecting using the columns of U, we get

$$\begin{bmatrix}1/\sqrt{3} & 1/2\\ 0 & 1/2\\ 1/\sqrt{3} & 0\\ 1/\sqrt{3} & -1/2\\ 0 & -1/2\end{bmatrix}\left(\begin{bmatrix}1/\sqrt{3} & 1/2\\ 0 & 1/2\\ 1/\sqrt{3} & 0\\ 1/\sqrt{3} & -1/2\\ 0 & -1/2\end{bmatrix}^T\begin{bmatrix}1/\sqrt{3} & 1/2\\ 0 & 1/2\\ 1/\sqrt{3} & 0\\ 1/\sqrt{3} & -1/2\\ 0 & -1/2\end{bmatrix}\right)^{-1}\begin{bmatrix}1/\sqrt{3} & 1/2\\ 0 & 1/2\\ 1/\sqrt{3} & 0\\ 1/\sqrt{3} & -1/2\\ 0 & -1/2\end{bmatrix}^T\begin{bmatrix}1\\-1\\0\\-1\\0\end{bmatrix} = \tag{20}$$

$$\begin{bmatrix}1/\sqrt{3} & 1/2\\ 0 & 1/2\\ 1/\sqrt{3} & 0\\ 1/\sqrt{3} & -1/2\\ 0 & -1/2\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}0\\1/2\end{bmatrix} = \begin{bmatrix}1/4\\1/4\\0\\-1/4\\-1/4\end{bmatrix} \tag{21}$$

3. The Framingham Risk Score

For most of the parts of this problem, your work will be done in the appropriate section of the ipython notebook.

In Homework 1, we did a problem where we calculated the parameters of the Framingham risk score for predicting cardiovascular disease (CVD). In this problem, we will revisit the parameters of the Framingham risk score in a more realistic setting using the more sophisticated optimization tool of linear least squares. In the problem in Homework 1, we determined four parameters of Framingham risk score from the data from four patients – this amounts to solving four equations with four unknowns. This makes sense if we knew the correct parameters originally but then forgot them. Suppose, however, that we were trying to come up with the correct parameters for the Framingham risk score in the first place. How would we do it?

As a review, the Framingham risk score estimates the 10-year cardiovascular disease (CVD) risk of an individual. There are multiple factors (predictors) that weigh in the calculation of the score. In Homework 1, we simplified the score to only use four factors. Here we will look at more complex version of the score that takes into account six factors including age, total cholesterol, level of high-density lipoprotein (HDL) cholesterol, systolic blood pressure (SBP), whether or not the individual smokes, and whether or not the individual is diabetic.

Scores like this are determined empirically after tracking the characteristics of many medical patients. Once we have data from hundreds or thousands of test subjects, we want to find the parameters that best model the data we are seeing so that we can use our score to predict the probability of heart disease for a new patient. Of course there will be some variability in the probability of heart to disease for each individual but we want to design the parameters of our score so that it predicts their risk as closely as possible.

Linear least squares is a powerful tool for fitting these kind of models to minimize the error between the observed risk of heart disease for each individual and the predicted risk from the model. Linear least squares can even be a powerful tool in many cases when we expect our model to be nonlinear in the data. As long as we can transform the problem so that the model is **linear in the parameters** then we can use linear least squares. For example in the Framingham case, we have reason to believe (from medical modeling considerations) that the probability of the individual suffering from CVD in the next 10 years has the form

$$p = 1 - 0.95^{e^{(R-26.1931)}} \tag{22}$$

where the score R is calculated based on the values of age, total cholesterol (TC), HDL cholesterol, systolic blood pressure (SBP), whether or not the patient is diabetic (DIA), and whether or not the patient smokes (SMK) as follows

$$\begin{aligned} R =& x_1 \cdot \ln\Big(\text{AGE (years)}\Big) + x_2 \cdot \ln\Big(\text{TC (mg/dL)}\Big) + \\ & x_3 \cdot \ln\Big(\text{HDL (mg/dL)}\Big) + x_4 \cdot \ln\Big(\text{SBP (mm Hg)}\Big) + \\ & x_5 \cdot \Big(\text{DIA (binary)}\Big) + x_6 \cdot \Big(\text{SMK (binary)}\Big) \end{aligned} \tag{23}$$

DIA and SMK are binary variables that indicate whether or not the subject has diabetes and smokes respectively whereas AGE, TC, HDL, and SBP are all numeric values. Note also that AGE, TC, HDL, and SBP are passed through the natural log function $\ln(\cdot)$ where as DIA and SMK are not. For patient k, we will represent the probability as p^k and the score as R^k.

(a) We want to transform the probabilities and the input data (AGE, TC, HDL, SBP, DIA, SMK) for patient k into the form

$$b^k = x_1 A_1^k + x_2 A_2^k + x_3 A_3^k + x_4 A_4^k + x_5 A_5^k + x_6 A_6^k \tag{24}$$

in order to solve for the parameters $\vec{x} = [x_1,\, x_2,\, x_3,\, x_4,\, x_5,\, x_6]^T$. How can we transform the probabilities and the input data to express Equation (22) in the form of Equation (24), i.e. express b^k, A_1^k, A_2^k, A_3^k, A_4^k, A_5^k and A_6^k be in terms of p^k, AGE^k, TC^k, HDL^k, SBP^k, DIA^k, SMK^k. In the ipython notebook, load in the data file `CVDdata.mat` and apply these transformations to the appropriate variables.

Credit: The data was obtained from the Center for Disease Control and Prevention's (CDC) National Health and Nutrition Examination Survey (NHANES) dataset (October 2015). http://www.cdc.gov/nchs/nhanes.htm

Solution: By manipulating Equation (22), we get

$$1 - p = 0.95^{e^{R-26.1931}} \tag{25}$$

$$\log_{0.95}(1-p) = e^{R-26.1931} \tag{26}$$

$$\ln\Big(\log_{0.95}(1-p)\Big) + 26.1931 = R \tag{27}$$

Plugging in the expression for R in Equation (23), we get

$$\ln\Big(\log_{0.95}(1-p)\Big) + 26.1931 = x_1 \cdot \ln\Big(\text{AGE (years)}\Big) + x_2 \cdot \ln\Big(\text{TC (mg/dL)}\Big) + \tag{28}$$

$$x_3 \cdot \ln\Big(\text{HDL (mg/dL)}\Big) + x_4 \cdot \ln\Big(\text{SBP (mm Hg)}\Big) + \tag{29}$$

$$x_5 \cdot \Big(\text{DIA (binary)}\Big) + x_6 \cdot \Big(\text{SMK (binary)}\Big) \tag{30}$$

Thus the appropriate transformations are

$$b^k = \ln\Big(\log_{0.95}(1-p^k)\Big) + 26.1931 \tag{31}$$

$$A_1^k = \ln(\text{AGE}^k) \tag{32}$$

$$A_2^k = \ln(\text{TC}^k) \tag{33}$$

$$A_3^k = \ln(\text{HDL}^k) \tag{34}$$

$$A_4^k = \ln(\text{SBP}^k) \tag{35}$$

$$A_5^k = \text{DIA}^k \tag{36}$$

$$A_6^k = \text{SMK}^k \tag{37}$$

(b) Now that we have transformed the problem into a linear problem, we want to use linear least squares to estimate the parameters $\vec{x}$. In order to do this we set up a system of equations in matrix form

$$\vec{b} = A\vec{x}$$

where A is a tall matrix and $\vec{b}$ is a tall vector. What form should $\vec{b}$ and A have in terms of b^k, A_1^k, A_2^k, A_3^k, A_4^k, A_5^k, A_6^k? The data we loaded in python is for 91 patients. Construct $\vec{b}$ and A using the loaded data. What are the dimensions of $\vec{b}$ and A?

Solution:

$$\vec{b} = [\, b^1, \dots b^k \dots, b^{91} \,]^T \tag{38}$$

$$A = \begin{bmatrix} A_1^1 & A_2^1 & A_3^1 & A_4^1 & A_5^1 & A_6^1 \\ \vdots & & \vdots & \vdots & & \vdots \\ A_1^k & A_2^k & A_3^k & A_4^k & A_5^k & A_6^k \\ \vdots & & \vdots & \vdots & & \vdots \\ A_1^{91} & A_2^{91} & A_3^{91} & A_4^{91} & A_5^{91} & A_6^{91} \end{bmatrix} \tag{39}$$

The dimension of $\vec{b}$ are 91×1. The dimensions of A are 91×6.

See also `sol11.ipynb`.

(c) We want to choose our estimate $\hat{\vec{x}}$ to minimize $||\vec{b} - A\vec{x}||^2$. Use the linear least squares formula to find the best fit parameters $\hat{\vec{x}}$. What is the $\hat{\vec{x}}$ that you found?

Solution: See `sol11.ipynb`.

(d) Now that we've found the best fit parameters $\hat{\vec{x}}$, write an expression for $\hat{\vec{b}}$, our model's prediction of the values of $\vec{b}$ given the data A and our estimate of the parameters $\hat{\vec{x}}$ and compute the squared error $||\vec{b} - \hat{\vec{b}}||^2$. What is the squared error that you computed?

Solution: See `sol11.ipynb`.

(e) Since this problem has many parameters it is difficult to visualize what is going on. One thing we can do to get a feel for the data and check that our fit is good is to plot it in a lower dimensional subspace. For example, we could plot b by A_1 or A_2 or A_3 individually. (A_1 is the vector of A_1^k's for all patients. It is the first column of A. Similarly for A_2 and A_3.) In your ipython notebook, plot b by A_1, b by A_2, and b by A_3 individually using the plotting option `'ob'` to plot the data as blue dots. For each plot you should see a blue point cloud. What is actually happening here is that we're projecting the data onto the A_1-b plane, the A_2-b plane, and the A_3-b plane respectively. Now plot your models prediction $\hat{b}$ by A_1, A_2, and A_3 on the appropriate plots using the plotting option `'or'` to plot the predictions as red dots (refer to the ipython notebook for reference code). Does it look like your model's fit is good?

Solution: See `sol11.ipynb`. From the plots, it looks like the fits are fairly good. Keep in mind, however, that plotting only tells us qualitatively if our fits are good. The squared error gives a quantitative value for the error in our model.

(f) To better visualize the linearity of the model, we will calculate the risk as a function of A_2 alone and plot it. The rest of the predictors will be fixed in this part. We will use the following values for the other parameters. Age=55 years, HDL=25 mg/dL, SBP=220 mm Hg, DIA=1 and SMK=1. In the IPython notebook, we have generated a block of code that you need to complete to make the calculation of predicted b values from the above parameters. Fill the code and plot the estimated b values vs A_2 values. Is the plot linear?

Hint: don't forget to apply the appropriate transformation to the different parameters.

Solution: See `sol11.ipynb`.

Yes, the plot is linear as expected.

(g) Try changing the parameters $\vec{x}$ slightly and re-plotting. Does it look like the fit is getting better or worse? Is the squared error increasing or decreasing?

Solution: See `sol11.ipynb`.

We can see clearly from the plots that the fit is getting worse – our model's predictions are moving away from the actual measured values. The squared error is increasing significantly.

(h) (BONUS) Transform b and $\hat{b}$ back into the form of p and transform A_1, A_2, A_3 back into the form of AGE, TC, and HDL and re-plot. What do you see?

Solution: See `sol11.ipynb`.

In these plots, we similar point clouds, but since we are plotting the data in it's original nonlinear form, the data no longer lies near a linear subspace.

(i) (BONUS) Use the values for b from part (f) to calculate p as a function of TC. Plot the curve p vs TC. Is the plot linear? What does this plot portray?

Solution: See `sol11.ipynb`.

This plot is no longer linear since we have transformed the equation back into it's original nonlinear form. Here we are looking at the model's prediction of the relationship between p and TC as opposed to b and A_2 which are the transformed versions of p and TC.

Note: Some of the values in the algorithm were modified from the original study values.

4. Finding Signals in Noise

Disclaimer: This problem looks long. But almost all of the parts only involve running provided IPython code, and commenting on the output. The last part is open-ended, but can be done by re-using code from the previous parts. It is important to understand the previous parts in order to tackle the last part.

In this problem, we will explore how to use correlation and least-squares to find signals, even in the presence of noise and other interfering signals.

(a) Suppose there is a transmitter sending a known signal s_1, which is periodic of length $N = 1000$. Say $\vec{s}_1$ is chosen to be a random $\{+1, -1\}$ vector (for example, by tossing a coin N times and replacing every heads with $+1$ and every tails with -1). For convenience, let us also normalize s_1 to norm 1. That is, the vector $\vec{s}_1$ looks like:

$$\vec{s}_1 = \frac{1}{\sqrt{n}}\begin{bmatrix}+1 & -1 & -1 & +1 & -1 & \dots\end{bmatrix}$$

(Where the ± 1 entries are chosen by random coin toss).

We claim that such a vector $\vec{s}_1$ is "approximately orthogonal" to circular-shifts of itself. That is, if $\vec{s}_1^{(j)}$ denotes circularly-shifting s_1 by j, then for all $j \neq 0$:

$$\left|\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle\right| \lessapprox \varepsilon$$

for some small epsilon.

Run the provided IPython code to generate a random $\vec{s}_1$, and plot its autocorrelation (all inner-products with shifted versions of itself). Run this a few times, for different random vectors $\vec{s}_1$. Around how small are the largest inner-products $\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle$, $j \neq 0$?

Recall that we normalized such that $\langle \vec{s}_1, \vec{s}_1 \rangle = 1$.

Solution: Running the provided code a few times, you should have observed that individual inner-products $\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle, j \neq 0$ are usually less than 0.06, and the maximum magnitude of inner-product $|\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle|$ for all $j \neq 0$ is usually around 0.1. (For example, you could have done this by looking at the maximum of the autocorrelation plot at indices $j \neq 0$.) So we can let

$$\varepsilon = 0.1$$

. Let us also define

$$\varepsilon_2 = 0.06$$

as a bound on the deviation of an individual inner-product. (It is find to just use ε as a bound instead of ε_2, but we will carry ε_2 through to be slightly more precise).

Notice that $\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle$ for $j \neq 0$ is much smaller than $\langle \vec{s}_1, \vec{s}_1 \rangle = 1$.

(b) Suppose we received a signal y, which is s_1 delayed by an unknown amount. That is,

$$\vec{y} = \vec{s}_1^{(j)}$$

for some unknown shift j. To find the delay, we can choose the shift j with the highest inner-product $\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle$.

Run the provided IPython code to plot the cross-correlation between $\vec{y}$ and $\vec{s}_1$ (ie, all the shifted inner-products). Can you identify the delay? Briefly comment on why this works, using the findings from the previous part. *(Hint: What does $\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle$ look like, for $k = j$? For $k \neq j$?)*

Solution: Running the provided code, we see that the autocorrelation peaks at index $k = 10$. That is, the inner-product $\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle$ is maximum for the shift $k = 10$. Thus we identify the delay as $j = 10$.

This works because the inner-product $\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle = \left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} \right\rangle$ will usually be small (at most ε) if $k \neq j$ (from the previous part), but will be exactly 1 if $k = j$. Thus the maximum index k will usually identify the correct shift.

(Note: We used the word "usually" here as a technicality, since it is possible, though unlikely, that our random choice of $\vec{s}_1$ happens to have high correlation with its shifted versions. For example, it is possible our "random" choice resulted in $\vec{s}_1 = 1/\sqrt{N} \begin{bmatrix} 1 & 1 & 1\ldots & 1 \end{bmatrix}$, ie the normalized all-ones vector. However, this is unlikely to occur – random vectors are usually almost-orthogonal to their shifted selfs, as you saw in the previous part. We will omit the "usually" for the rest of the solutions, for readability sake.)

(c) Now suppose we received a slightly noisy signal:

$$\vec{y} = \vec{s}_1^{(j)} + 0.1\vec{n}$$

where the "noise" source $\vec{n}$ is chosen to be a random normalized vector, just like s_1.

Run the provided IPython code to compute $\langle \vec{s}_1, \vec{n} \rangle$. Run this a few times, for different random choices of s_1, n. Around how small is $|\langle \vec{s}_1, \vec{n} \rangle|$? How does this compare to $\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle$, from your answer in part (a)?

Solution: Running the provided code a few times, you should have observed that $|\langle \vec{s}_1, \vec{n} \rangle|$ is usually around 0.03, and almost always less than $0.06 = \varepsilon_2$.

This is similar to how the inner-products $\left\langle \vec{s}_1, \vec{s}_1^{(j)} \right\rangle, j \neq 0$ looked in the previous parts. That is, the inner-product of a random signal with its shifted self is roughly the same as the inner-product of a signal with random noise.

(d) Can we identify the delay from this noisy reception? In this case, we do not know the noise $\vec{n}$, and we do not know the delay j. (But, as before, we know the signal $\vec{s}_1$ being transmitted).

Run the provided IPython code to plot the cross-correlation between $\vec{y}$ and $\vec{s}_1$. Briefly comment on why this works to find the delay, using the findings from the previous part.

Solution: Yes, we can identify the delay in this case, as demonstrated by the provided code.

This works roughly because, for a fixed $k \neq j$, the inner-product:

$$\begin{aligned}\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle &= \left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} + 0.1\vec{n} \right\rangle \\ &= \left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} \right\rangle + (0.1)\left\langle \vec{s}_1^{(k)}, \vec{n} \right\rangle \\ &\approx \varepsilon_2 \pm (0.1)\varepsilon_2\end{aligned}$$

is small. And in fact, *all* the inner-products $k \neq j$ are small; the maximum over all $k \neq j$ is roughly bounded by:

$$\begin{aligned}\max_{k\neq j}\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle &= \max_{k\neq j}\left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} + 0.1\vec{n} \right\rangle \\ &= \max_{k\neq j}\left(\left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} \right\rangle + (0.1)\left\langle \vec{s}_1^{(k)}, \vec{n} \right\rangle\right) \\ &\leq \max_{k\neq j}\left\langle \vec{s}_1^{(k)}, \vec{s}_1^{(j)} \right\rangle + (0.1)\max_{k\neq j}\left\langle \vec{s}_1^{(k)}, \vec{n} \right\rangle \\ &\lessapprox \varepsilon + (0.1)\varepsilon\end{aligned}$$

(Where we used our findings from the previous parts in the last step).

On the other hand, for $k = j$, the inner-product:

$$\begin{aligned}\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle &= \left\langle \vec{s}_1^{(j)}, \vec{s}_1^{(j)} + 0.1\vec{n} \right\rangle \\ &= \left\langle \vec{s}_1^{(j)}, \vec{s}_1^{(j)} \right\rangle + (0.1)\left\langle \vec{s}_1^{(j)}, \vec{n} \right\rangle \\ &\approx 1 \pm (0.1)\varepsilon_2\end{aligned}$$

is large.

(e) What if the noise was higher? For example:

$$\vec{y} = \vec{s}_1^{(j)} + \vec{n}$$

Does cross-correlation still work to find the delay? (Use the provided IPython notebook).

What about very high noise?

$$\vec{y} = \vec{s}_1^{(j)} + 10\vec{n}$$

Does cross-correlation still work to find the delay? If not, can you explain why? (use findings from previous parts).

Solution: Noise with the same amplitude as the signal ($\vec{y} = \vec{s}_1^{(j)} + \vec{n}$) still works, as seen in the IPython notebook.

We may expect this, since (similar to the rough calculations from the previous part), the maximum correlation for all $k \neq j$:

$$\max_{k\neq j}\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle \lessapprox \varepsilon + (1)\varepsilon \approx 0.2$$

is still small compared to $\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle \approx 1$.

However, very high noise ($\vec{y} = \vec{s}_1^{(j)} + 10\vec{n}$) no longer works – there is no clear autocorrelation peak. Again, we may expect this, because now the maximum correlation for $k \neq j$ is roughly:

$$\max_{k \neq j} \left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle \approx 10\varepsilon \approx 1.1$$

which is larger than $\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle \approx 1$. Thus, it is likely that some inner-product $\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle, k \neq j$ is greater than the "correct" inner-product $\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle$.

In other words, the noise masks the signal in this case.

(f) Now suppose there are two transmitters, sending known signals $\vec{s}_1$ and $\vec{s}_2$ at two unknown delays. That is, we receive

$$\vec{y} = \vec{s}_1^{(j)} + \vec{s}_2^{(k)}$$

for unknown shifts j, k. Both signals $\vec{s}_1$ and $\vec{s}_2$ are chosen as random normalized vectors, as before.

We can try to find the first signal delay by cross-correlating $\vec{y}$ with $\vec{s}_1$ (as in the previous parts). Similarly, we can try to find the second signal delay by cross-correlating $\vec{y}$ with $\vec{s}_2$. Run the provided IPython code to estimate the delays j, k. Does this method work to find both delays? Briefly comment on why or why not.

Solution: Yes, this method works in this case, as demonstrated by the provided code.

We may expect this, because when we try to find the delay of $\vec{s}_1$ by computing inner-products $\left\langle \vec{s}_1^{(k)}, \vec{y} \right\rangle$, the signal $\vec{s}_2$ looks like "noise". In fact, $\vec{s}_2$ is chosen as an independent random normalized vector, just like $\vec{n}$ in the previous part. Thus, this works for exactly the same reason that the "medium noise" case ($\vec{y} = \vec{s}_1^{(j)} + \vec{n}$) worked in the previous part.

(g) Now, suppose the second transmitter is very weak, so we receive:

$$\vec{y} = \vec{s}_1^{(j)} + 0.1\vec{s}_2^{(k)}$$

Does the method of the previous part work reliably to find signal s_1? What about s_2? (Run the provided code a few times, to test for different choices of random signals). Briefly justify why or why not.

(Hint: $\vec{s}_1$ looks like "noise" when we are trying to find $\vec{s}_2$. Based on the previous parts, would you expect to be able to find $\vec{s}_2$ under such high noise?)

Solution: This still works to find $\vec{s}_1$, since the signal $\vec{s}_2$ looks like random noise of very low amplitude. However, this no longer works reliably to find $\vec{s}_2$, for the same reason that the "very high noise" case in part (e) failed. That is, when we are trying to find $\vec{s}_2$, the signal $\vec{s}_1$ appears as random noise, at 10 times the amplitude of $\vec{s}_2$. Thus, as in part (e), we may expect that the "noise" masks the signal in this case.

(h) To address the problem of the previous part, suppose we use the following strategy: First, cross-correlate to find the delay j of the strongest signal (say, $\vec{s}_1$). Then, subtract this out from the received $\vec{y}$, to get a new signal $\vec{y}' = \vec{y} - \vec{s}_1^{(j)}$. Then cross-correlate to find the second signal in $\vec{y}'$.

Run the provided IPython code to test this strategy for the setup of the previous part (with a strong and weak transmitter). Does it work? Briefly comment on why or why not.

Solution: Yes, this new strategy works to find both signals. The received signal is:

$$\vec{y} = \vec{s}_1^{(j)} + 0.1\vec{s}_2^{(k)}$$

If we know the coefficient of $\vec{s}_1$ exactly (1 in this case), and we find the shift of $\vec{s}_1$ correctly, then when we subtract out the contribution of $\vec{s}_1$, we find

$$\vec{y}' = \vec{y} - \vec{s}_1^{(j)} = 0.1\vec{s}_2^{(k)}$$

Which only contains signal $\vec{s}_2$. Thus we can cross-correlate to find its shift.

(i) Finally, suppose the amplitudes of the sent signals are also unknown. That is:

$$\vec{y} = \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)}$$

for unknown amplitudes $\alpha_1 > 0$, $\alpha_2 > 0$, and unknown shifts j, k.

Can we use inner-products (cross-correlation) to find the amplitudes as well? For example, suppose we find the correct shift j via cross-correlation. Then, briefly comment on why the following holds:

$$\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle \approx \alpha_1$$

Run the provided IPython notebook to try this method of estimating the coefficients α_1, α_2. Roughly how close are the estimates to the actual amplitudes? (Run the code a few times, to test different choices of random signals).

Solution: Yes, taking inner-products works reasonably to find the coefficients. This is because, if we find the shift j correctly, then:

$$\begin{aligned}
\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle &= \left\langle \vec{s}_1^{(j)}, \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)} \right\rangle \\
&= \alpha_1 \left\langle \vec{s}_1^{(j)}, \vec{s}_1^{(j)} \right\rangle + \alpha_2 \left\langle \vec{s}_1^{(j)}, \vec{s}_2^{(k)} \right\rangle \\
&\approx \alpha_1(1) \pm \alpha_2 \varepsilon_2 \\
&\approx \alpha_1
\end{aligned}$$

Where the last line is because ε_2 is small.

(Note: In our case, $\alpha_1 = 0.7, \alpha_2 = 0.5$. This strategy would not have worked if α_2 was much larger than α_1, because then $\alpha_2 \varepsilon_2$ in the last line would be large compared to α_1).

Running the provided code, we see the estimated coefficients are usually within ± 0.03 of the true coefficients, and almost always within $\pm 0.06 = \varepsilon_2$ of the true coefficients.

(j) Repeat the above for when there is some additional noise as well:

$$\vec{y} = \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)} + 0.1\vec{n}$$

Roughly how close are the estimates to the actual amplitudes? (Run the code a few times, to test different choices of random signals).

Solution: The estimates are slightly worse, but not by much. Repeating the estimates of the previous part, we see:

$$\begin{aligned}
\left\langle \vec{s}_1^{(j)}, \vec{y} \right\rangle &= \left\langle \vec{s}_1^{(j)}, \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)} + 0.1\vec{n} \right\rangle \\
&\approx \alpha_1(1) \pm \alpha_2 \varepsilon_2 \pm 0.1\varepsilon_2 \\
&\approx \alpha_1
\end{aligned}$$

Again, running the provided code, we see the estimated coefficients are usually within ± 0.03 of the true coefficients, and almost always within $\pm 0.06 = \varepsilon_2$ of the true coefficients.

(k) Let us try to improve the coefficient estimates. Once we have identified the correct shifts j,k, we can setup a Least-Squares problem to find the "best" amplitudes α_1, α_2 such that

$$\vec{y} \approx \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)}$$

Set up a Least-Squares problem in the form

$$\min ||A\vec{x} - \vec{b}||$$

to estimate α_1, α_2. What is the matrix A and vector b?

(Hint: A will have 1000 *rows and* 2 *columns)*

Use the provided IPython notebook to try this method of estimating the coefficients α_1, α_2. Roughly how close are the estimates to the actual amplitudes? Did this improve on the strategy from the previous part?

Solution: Let $\vec{b} = \vec{y}$, and let

$$A = \begin{bmatrix} \vec{s}_1^{(j)} & \vec{s}_2^{(k)} \end{bmatrix}$$

(ie, the columns of A are the two shifted signals).

Then, our linear least-squares objective is

$$\min_{\vec{x}} ||A\vec{x} - \vec{y}|| = \min_{x_1,x_2} || \begin{bmatrix} \vec{s}_1^{(j)} & \vec{s}_2^{(k)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \vec{y}|| = \min_{x_1,x_2} ||x_1 \vec{s}_1^{(j)} + x_2 \vec{s}_2^{(k)} - \vec{y}||$$

That is, we are trying to find coefficients x_1, x_2 of the known signals which best approximate the received signal (under the assumption that the noise is "small"). We estimate α_1 as $\hat{x}_1$ and α_2 as $\hat{x}_2$.

Running the code, we see that this yields a much better estimate for the coefficients, typically within $\pm 0.1\varepsilon_2 = 0.006$ of the true coefficients. (In fact, if there is no noise, this will find the coefficients exactly provided $\vec{s}_1$ and $\vec{s}_2$ are linearly independent – can you see why?)

(l) Now try to use these tools on your own. Suppose there are 3 transmitters, transmitting known signals $\vec{s}_1, \vec{s}_2, \vec{s}_3$. These signals were chosen as random normalized vectors (as above). You received a combination of these signals at unknown delays and unknown amplitudes, and corrupted by some unknown noise:

$$\vec{y} = \alpha_1 \vec{s}_1^{(j)} + \alpha_2 \vec{s}_2^{(k)} + \alpha_3 \vec{s}_3^{(l)} + 0.005\vec{n}$$

(Where $\vec{n}$ was also chosen as a random normalized vector as above).

You are provided with the known signals $\vec{s}_1, \vec{s}_2, \vec{s}_3$ and the received signal $\vec{y}$ in the IPython notebook. **Try to find the unknown delays** j,k,l**.** *(Hint: Some of the signals may be weak).*

There is a test function provided that will try to "decode" your candidate shifts j,k,l into a message. You should recognize the message.

Describe your procedure, and include your entire workflow (any plots you generated, etc) in your IPython solutions.

Solution: See the IPython solutions for a candidate workflow.

We notice that signals 1,2 are strong, and we can find the shifts j,k by simple cross-correlation. However, we cannot immediately find signal 3 – it is masked by the noise and the other two signals. We can try to estimate the coefficients α_1, α_2 by taking inner-products as in part (i), but our estimates won't be good enough – if we subtract out a bad estimate of the first two signals, their residual components will still mask the third signals (this is demonstrated in the IPython solutions). Thus, we try the method of part (k) to estimate α_1, α_2 using linear least-squares (the signal $\vec{s}_3$ can be considered as just more noise). This gives us a better estimate, which we can use to subtract out the first two signals, and finally cross-correlate to find the third. The entire workflow is similar to OMP.

The shifts were: $j = 140, k = 740, l = 960$, and the decoded message was 'Cal'.

5. Sparse imaging

Recall the imaging lab where we have projected masks on an object to scan it to our computer using a single pixel measurement device, that is, a solar cell! In that lab, the 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 'measurement matrix' to be invertible.

In Tuesday's lecture we saw that an iterative algorithm that does "matching and peeling" can enable reconstruction of a sparse vector while reducing the number of samples that need to be taken from it. In the case of imaging, the idea of sparsity corresponds to most parts of the image being black with only a small number of light pixels. In these cases, we can reduce the overal number of samples necessary. This would reduce the time required for scanning the image. (This is a real-world concern for things like MRI where people have to stay still while being imaged.)

In this problem we have a 2D image I of size 91×120 pixels, for a total of $10,920$ pixels. The image is made up of mostly black pixels except for 476 of them that are white.

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 — i.e. the light intensity is going to vary in a controlled way. Say that we have an imaging mask M_0 of size 91×120. The measurements using the solar cell using this imaging mask can be represented as follows.

First, let us vectorize our image to $\vec{i}$ which is a length $10,920$ column vector. Likewise let us vectorize the mask M_0 to $\vec{m}_0$ which is a length $10,920$ column vector. Then the measurement using M_0 can be represented as

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

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

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

where A is an $M \times 10,920$ size matrix whose rows contain the vectorized forms of the illumination masks, that is

$$A[j,:] = m_j^\top,$$

where we used the `numpy` array slicing notation $A[j,:]$ to denote the jth row of A.

To show that we can reduce the number of samples necessary to recover the sparse image I, we are going to only generate 6500 masks. The columns of A are going to be approximately uncorrelated with each other. The following question refers to the part of IPython notebook file accompanying this homework related to this question.

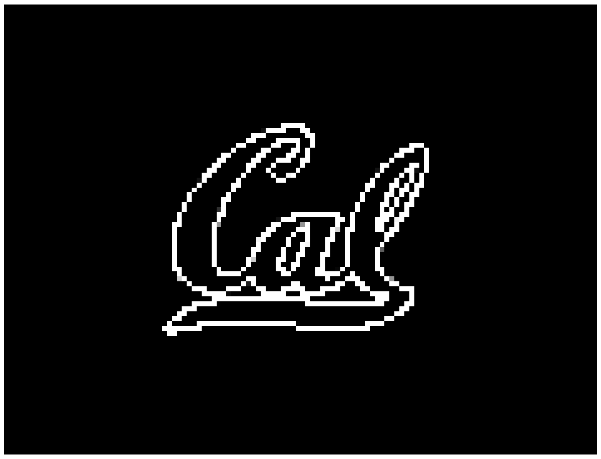
(a) In the IPython notebook, we call a function `simulate` that generates masks and the measurements. You can see the masks and the measurements in the IPython notebook file. We would like you to complete the function `OMP` that does the OMP algorithm described in lecture.

Solution: The solution is given in `sol11.ipynb`.

Remark: Note that this remark is not important for solving this problem, it is about how such measurements could be implemented in our lab setting. 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 `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 A and subtract it from all entries of 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 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?

Solution: The reconstructed image is the following.



(c) (Bonus) We have supplied code that reads a PNG file containing a sparse image, takes measurements, and performs OMP to recover it. An example input file is also supplied together with the code. Generate an image of size 91×120 pixels of sparsity less than 400 and recover it using OMP with 6500 measurements. Add your IPython notebook outputs. 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? Make an image having fewer white pixels. How much can you reduce the number of measurements that need to be taken?

Solution: The answer to this question depends on the size of your input image and the total number of non-zero pixels in it. In order to successfully recover the image, the number of measurements (masks used for imaging) needs to increase as the number of non-zero pixels increases. The reason is, as we have more nonzero pixels, they start to contribute in the measurements (in terms of the example given in the lecture, it is like having more users try to transmit their messages at the same time). In order to not be affected by these contributions, we need the signature of each pixel to be 'more orthogonal' to others. In our setting, this is achieved by taking more measurements with different masks.

6. Speeding up OMP

Consider the OMP Imaging problem.

(a) Modify the code to run faster by using a Gramm-Schmidt orthonormalization to speed it up. (Edit the code given to you in `prob11.ipynb`.)

Solution:

We have added some Gramm-Schmidt code — using a silly name of "cheapo-least-squares" to emphasize that projection onto an orthogonal basis is easy.

The solution in the IPython nodebook does Gramm-Schmidt as it goes along and finds vectors. (The code is far from efficient in its implementation since it copies vectors too often.)

Ask in Piazza if you have questions about how this works.

(b) **(Bonus)** Do any other modifications you want to further speed up the code.
(Hint: when possible, how would you safely extract multiple peaks corresponding to multiple pixels in one go and add them to the recovered list? Would this speed things up?)

Solution:

See the code in the IPython notebook.

This approach grabs many pixels at once. This saves a lot of effort in computing correlations over and over again. The threshold to decide how many pixels are safe to grab is chosen based on knowledge that derives from topics covered in 126 and touched upon in 70. Play with that number 6. For example, try to change it to 1 or 2. See what happens. Notice that it starts grabbing some false pixels.

7. **Your Own Problem** Write your own problem related to this week's material and solve it. You may still work in groups to brainstorm problems, but each student should submit a unique problem. What is the problem? How to formulate it? How to solve it? What is the solution?