EECS 16A Designing Information Devices and Systems I
Spring 2018 Lecture Notes Note 22

Module Goals

In this module, we introduce a family of ideas that are connected to **optimization**, which is one of the modern pillars of EECS and the topic of EECS 127. Here, we will also be building on our understanding of the inner-product structure of vector spaces as well as the general principle of **relaxation** wherein we try to solve a perhaps harder problem in order to get an approach that is actually easier.

This module uses the practical example of localization — figuring out where you are — as a source of examples and a connection to the lab. The ideas we will learn, and the skills that we develop, just as in previous modules, are far more generally applicable.

Key Concepts In this Note

By the end of this note, you should be able to do the following:

1. Understand why and when we would want to use Cross-Correlation
2. Be able to mechanically identify signals using Cross-Correlation
3. Be able to triangulate position if there is no noise in your distance measurements.

Introduction

Suppose we have a receiver R_x that's receiving signals from various beacons B_1, B_2..., like in the following diagram.

Our goal is to figure out where the receiver is. In order to do that, we need to be able to estimate the distance between the receiver and the beacon transmitters. As we will see, once we have those distances, we can figure out where we are.

To get the distance, we can rely on the fact that the speed of propagation of the signal through the environment, v, is constant (the speed of light for radio waves, the speed of sound for audio waves). We then figure out how long it took that signal to get from the beacon to our receiver, t, and then convert to distance, $d = vt$.

Assume first that the beacon transmitters and the receiver all have a perfectly calibrated clock. Further assume that each transmitter is transmitting a repeating signal that has some period N (this is essentially what GPS satellites and many other systems do). This period should be longer than any uncertainty that exists in the distance to the beacon transmitter.

Our goal is to figure out which beacons we can hear and at what delay. To do this we use a procedure called cross correlation, but before we talk about that, we need to first model the problem more carefully and relax it a bit.

Modeling and relaxing the problem

We are going to work in the digital realm where time ticks discretely in steps. Our receiver takes N consecutive measurements (after this, they will repeat by the periodicity of all transmissions) and arranges them into a column vector $\vec{y}$. Assume that the ith beacon is transmitting a signal $\vec{s}_i$ also of length N. However, because of the finite distance, each of these signals are delayed when they get to the receiver.

We will use $y[k]$ to refer to the kth position within the vector $\vec{y}$ and similarly for the $\vec{s}_i$. To avoid notational clutter, we will not worry if k becomes larger than $(N-1)$ because it is to be understood that k is to be viewed "mod N." i.e. All that matters is the remainder[1] of k after we divide by N. In other words, it wraps around the way that the hands of an old-fashioned clock wrap around because of periodicity.

So our model is that $y[k] = \sum_i \alpha_i s_i[k-\tau_i]$ where τ_i is the signal delay experienced from the ith transmitter to the receiver. The delays τ_i do not change with k — the receiver is assumed to be stationary as are the transmitters. The α_i represent the signal attenuation experienced by the transmitted beacon before it gets to the receiver.

Now, the challenge is that the unknowns that we are interested in, the τ_i, enter into our observations in a seemingly nonlinear[2] way. They shift the signals. The α_i on the other hand are nicely behaved and enter our observations linearly. How can we deal with this?

To understand, we first consider the problem with a single transmitter. So, we have $y[k] = \alpha s[k-\tau]$ where τ is unknown and could be any one of $0, 1, 2, 3, \ldots, N-1$. We have N equations (for the N different k values corresponding to our N distinct observations through time) and two unknowns. The problem is that one of the unknowns, the τ, is annoyingly nonlinear in how it enters the observations.

So, what can we do? Just as you did in the image-stitching homework problem with corresponding points,

[1] "mod N" means going 0, 1, 2, ..., N-2, N-1, 0, 1, 2, ... We count in a circle so the number that follows N-1 is 0 instead of N.

[2] Recall the definition of linearity. A set of variables enter linearly into an equation if (a) setting them all to zero would result in their contribution to that equation being zero, (b) multiplying them all by γ would multiply their contribution by γ, and (c) setting them to the sum of two possible values would result in their contributions corresponding to those values adding up to give their new contribution. Notice that the τ_i here satisfy none of these conditions.

we can *relax* the problem. After all, we have plenty of equations. So, why not add more unknowns? Let's solve the "harder" problem of assuming that all of the possible τ could be simultaneously present. $y[k] = \sum_{j=0}^{N} \beta_j s[k-j]$. Here we assume that our signal is composed on N different signals, each shifted by a fixed amount, j. β_j are the unknown weights of each of the shifted signals. We've replaced the τ unknown with N of these β_j unknowns, giving us N linear equations and N unknowns.

Let's write this in matrix form. We define a special matrix $C_{\vec{x}}$ to be

$$C_{\vec{x}} = \begin{bmatrix} x[0] & x[N-1] & x[N-2] & x[N-3] & \dots & x[1] \\ x[1] & x[0] & x[N-1] & x[N-2] & \dots & x[2] \\ x[2] & x[1] & x[0] & x[N-1] & \dots & x[3] \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x[N-1] & x[N-2] & x[N-3] & x[N-4] & \dots & x[0] \end{bmatrix}$$

and then our equation is $C_{\vec{s}}\vec{\beta} = \vec{y}$. Each column of the matrix $C_{\vec{x}}$ is the signal $s[k]$ delayed by a different amount.

This matrix $C_{\vec{s}}$ consists of all the circular shifts of the vector $\vec{s}$. The term for such matrices is "circulant" matrices and the command `linalg.circulant` in IPython (import scipy) will automatically return the circulant matrix corresponding to a vector.

To be able to solve this system of equations uniquely, we need the circular shifts of $\vec{s}$ to be linearly independent. Then the columns of $C_{\vec{s}}$ are linearly independent and the system of equations will have one unique solution. If indeed the $\vec{y}$ came from exactly one of those shifts, the solution to the equation would be a $\vec{\beta}$ that has a single nonzero β_j in exactly the position corresponding to τ and zeros everywhere else. We could then set $\tau = j$ and $\alpha = \beta_j$.

So, by relaxing the problem, we were able to solve it easily using our standard bag of tricks: solving systems of linear equations.

There is one challenge with this particular trick. It only works in the special case when there is exactly one transmitter. This is because if we try to apply it to the case when we have more than one beacon signal being transmitted simultaneously (as is the case in lab), we would get a multiple of N unknowns while still having only N equations. How should we solve this problem?

To understand this, we are going to step back and see if there is another way forward.

Cross-Correlation

The **cross-correlation** is a measurement of the similarity between two vectors $\vec{y}$ and $\vec{u}$ - basically a sliding inner product. We compute the inner product $\langle \vec{y}, \vec{u} \rangle$, and then the same with $\vec{u}$ shifted by 1, then shifted by 2, and so on. IPython has the function `numpy.correlate` which basicaly does this[3].

[3]There are a few small differences. First, the cross-correlation we define is related to what IPython gives you when you use the "same" mode of `numpy.correlate`. Second, there is a complex conjugation in IPython that we are ignoring here because we are interested in vectors that are just real. In general, the vector that is being shifted needs to be complex-conjugated for reasons that will become clear when we talk about the inner product for complex vectors. Third, there is the matter of ordering. The order matters because one of the vectors gets shifted while doing the cross-correlation and the other does not. Our notation is explicit and we call out the vector that is being shifted as the one that creates the cross-correlation operator $S_{\vec{u}}$. This acts (from the left)

For us, cross-correlation will often be circular and normalized. Circular in that we consider circular shifts (that wrap the signal around) as is natural for periodic beacons; and normalized in that perfect correlation gives a value of 1.

To compute the normalized cross-correction of a vector $\vec{y}$ with $\vec{u}$, we compute the cross-correlation operator (this is a matrix) $S_{\vec{u}} = \frac{1}{\|\vec{u}\|^2} C_{\vec{u}}^T$. Here $C_{\vec{u}}$ is the circulant matrix consisting of circular shifts of $\vec{u}$ in the columns. The cross-correlation vector between $\vec{u}$ and $\vec{y}$ will be given by $S_{\vec{u}}\vec{y}$. Another way of writing this is

$$S_{\vec{u}}\vec{y}[j] = \frac{1}{\|\vec{u}\|^2} \sum_{n=0}^{N-1} \vec{y}[n]\vec{u}[(n-j)_N]$$

where $S_{\vec{u}}\vec{y}[j]$ represents the jth entry of the cross-correlation. (Unnormalized cross-correlation has the same form as above but without dividing $\|\vec{u}\|$. We will specify if you are to use the normalized version.)

Cross-correlation is not commutative (ie, the cross-correlation of $\vec{y}$ with $\vec{u}$ is not the same as the cross-correlation of $\vec{u}$ with $\vec{y}$). In this class, we will use the convention that the second signal is the one that shifts, and it shifts to the right. There are other conventions that you may see in other places (for example, Wikipedia uses a slightly different convention). In future classes, you'll learn about *convolution* which is an operation similar to cross-correlation but commutative.

If we want to find a signal $\vec{u}$ in an observation $\vec{y}$, we can compute the cross-correlation, and the jth entry tells us how much $\vec{y}$ contains the jth shift of $\vec{u}$. In other words, it is $\frac{1}{\|\vec{u}\|^2} \left\langle \vec{y}, \text{Shift}_j \vec{u} \right\rangle$. The largest cross-correlations are deemed to correspond to time-shifts of the signal $\vec{u}$ of interest — they are deemed to be actually present in $\vec{y}$.

This taking the "largest" is how we are using optimization to help us do information extraction. This sense will become even clearer when we talk about Least-Squares in the next lecture.

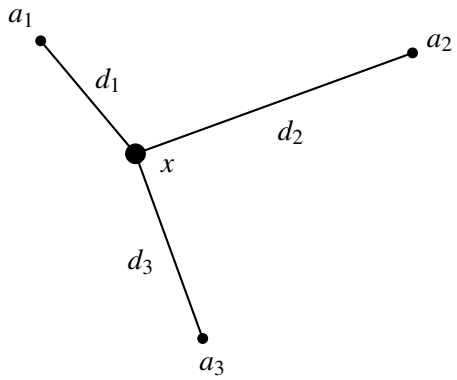
We can also look at the **auto-correlation**, which is defined to be the cross-correlation of a signal with itself. This is 1 in the first position by construction. The signal is a good candidate for being used as a beacon signal if the auto-correlation is small everywhere else. There are lots of good signals out there once N is large enough.

Trilateration

Now that we can extract the delays and hence the distances to beacon transmitters, we can actually figure out where we are.

Let's imagine we have a situation like the one below. We know the locations of the beacons $\vec{a}_1, \vec{a}_2, \vec{a}_3$, but don't know the location of the point at $\vec{x}$ (we'll be trying to find out what $\vec{x}$ is). We do know the distances d_1, d_2, d_3. We're trying to find the coordinates of $\vec{x}$ in this diagram:

on the vector that is not being shifted. In IPython, the one being shifted is the second argument to the `numpy.correlate` function. Fourth, there is the accounting for where we place the 0 shift. For convenience in writing, we place the zero shift at the beginning. For convenience in plotting, IPython places the zero shift in the middle. Finally, in the discussion below, we normalize the cross-correlation for ease of interpretation. (i.e. divide by $\frac{1}{\|\vec{u}\|^2}$.) IPython does not normalize.



We know that:

1. $\|\vec{x}-\vec{a}_1\|^2 = d_1^2$
2. $\|\vec{x}-\vec{a}_2\|^2 = d_2^2$
3. $\|\vec{x}-\vec{a}_3\|^2 = d_3^2$

Rewriting these using transpose notation we get:

$$\vec{x}^T\vec{x} - 2\vec{a}_1^T\vec{x} + \|\vec{a}_1\|^2 = d_1^2 \tag{1}$$
$$\vec{x}^T\vec{x} - 2\vec{a}_2^T\vec{x} + \|\vec{a}_2\|^2 = d_2^2 \tag{2}$$
$$\vec{x}^T\vec{x} - 2\vec{a}_3^T\vec{x} + \|\vec{a}_3\|^2 = d_3^2 \tag{3}$$

But we have squared terms here involving unknowns. That's not really conducive to our style of computation - we prefer only linear terms in unknowns, so we can use linear algebra.

For this, we use a trick[4]. Let's subtract equation 1 from equation 2, and separately again from equation 3. Then we get:

$$2(\vec{a}_1-\vec{a}_2)^T\vec{x} = \|\vec{a}_1\|^2 - \|\vec{a}_2\|^2 - d_1^2 + d_2^2$$

and

$$2(\vec{a}_1-\vec{a}_3)^T\vec{x} = \|\vec{a}_1\|^2 - \|\vec{a}_3\|^2 - d_1^2 + d_3^2$$

We can then stick these into a matrix, which will only have linear terms:

[4]This is not the best way to solve this problem when we allow noise into the system as in the next note, but it has the advantage of being simpler to understand. So, we will stick with this approach in the lab and in the next lecture note, even though for better accuracy, it is useful to deploy slightly more complicated approaches that instead linearize the equations around a potential solution and then iterate to converge to the best answer.

$$\begin{bmatrix} 2(\vec{a}_1 - \vec{a}_2)^T \\ 2(\vec{a}_1 - \vec{a}_3)^T \end{bmatrix} \vec{x} = \begin{bmatrix} \|\vec{a}_1\|^2 - \|\vec{a}_2\|^2 - d_1^2 + d_2^2 \\ \|\vec{a}_1\|^2 - \|\vec{a}_3\|^2 - d_1^2 + d_3^2 \end{bmatrix}$$

.

This can be solved for a location. Three circles uniquely define a point in 2D. The argument extends in 3D to four spheres. And in 4D to five hyper-spheres. (In the real-world, time is the fourth dimension we need to solve for.)