EECS 16A Designing Information Devices and Systems I
Fall 2016 Official Lecture Notes Note 15

Introduction to Correllation and Localization

Recap

The story so far in 16a has been largely building from the following very simple doctrine:

1. When faced with an situation where we would like to extract some information from measurements, we first choose variables to represent all the unknowns that we are interested in, along with others that we may not be interested in.
2. For every observation that we have or fact that we know, we write down an equation in terms of those unknown variables. This captures all of our knowledge into a format that we (or a computer) can manipulate systematically.
3. If we are lucky, these equations are all linear in terms of the variables we have and we can collect the variables together into a giant unknown vector $\vec{x}$ and write all the equations we have into matrix form $A\vec{x} = \vec{b}$.
4. Somehow, we solve this system of equations and extract the information that we want. If we can't solve it exactly, we get a solution that has some uncertainty in it, but we know exactly how to express this uncertainty in terms of some completely free variables.

Gaussian Elimination, as a mechanical technique for solving systems of linear equations, can in this setting be viewed as a technique for logical deduction. Each equation plays an informational role analogous to a logical proposition and the Gaussian steps can be viewed as ways of generating new collections of propositions from old ones.

Of course, we are not restricted to using Gaussian Elimination. As we saw during the circuits module, using our physical intuition and symmetries in the problem can help us speed up deduction by a lot. But at the end of the day, we still end up essentially solving a system of equations. However, we trade the application of quintessentially human thinking skills (context awareness, recognizing symmetries, seeing patterns, grouping things intelligently, applying equivalences, decomposing problems into simpler subproblems, etc.) for reducing our mechanical work.

All of the major circuit theory results for linear circuit networks simultaneously have a purely linear-algebraic proof/justification[1] as well as an explanation rooted in physical intuition. You need to understand both to proceed intelligently in general. You have certainly seen how the mathematical skill of being able

[1]For example, superposition is a natural consequence of how solving a system of equations can be viewed as applying an appropriate inverse-type-matrix to $\vec{b}$. The $\vec{b}$ in a circuits problem is just a vector that has independent current and voltage sources in it. The relevant columns of the inverse-matrix are just the solutions to the circuit with that independent source active at level 1 and all others off.

to systematically work through equations helps in circuits. What might be surprising to many of you is that the same mental muscles that are responsible for physical intuition also come into play in more abstract settings. It turns out that you can build a "physical intuition" about mathematical concepts too. This is vital for attacking more advanced problems.

Module Goals

The goal in this module is to push beyond simple logical deduction into the counterpart of what is often called "Occam's Razor[2] in logic: namely that the simplest explanation is the one that we should choose if we have a choice.

To do this, we introduce a family of ideas that are connected to **optimization** — which is really 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.

Similarly, Norton/Thevenin theorems are just a consequence of the operation of Gaussian Elimination for a system that has two different groups of nodes and branches that only interact through their common interconnection. This can be seen by partitioning the unknowns into three groups: (a) those local to the first circuit; (b) those common to both circuits; (c) those local to the second circuit. The current into the port and the voltage across the the port are the only common variables. So, if we partition the variables so that (a) are on top, (b) are in the middle, and (c) are on the bottom, Gaussian elimination proceeds and doesn't even touch the second circuit's equations (c) until it has already got exactly one equation that only involves the common variables in (b). (All the others above have been eliminated.) That is to say that group (b) gets boiled down to a single equation that represents the combined effect of all of the first circuit. This single equation then proceeds to hit the second circuit's equations in group (c). This single equation is exactly what Norton/Thevenin abstract. It is the interface through which the second circuit interacts with the first.

[2]The Latin phrase: "*entia non sunt multiplicanda praeter necessitatem*" means "entities must not be multiplied beyond necessity" and is often used as a classical statement of the principle. We shall find that this incipient quantification (the plurality of entities) is closer in spirit to what we do than the more vague term "simple" used in modern times.

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 is constant (the speed of light for radio waves, the speed of sound for audio waves) and just figure out how long it took that signal to get from the beacon to our receiver.

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 the GPS satellites do and many other systems do as well.) This period is 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 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[3] 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

[3]Don't worry if this is the first time you are seeing the language "mod N." It is a very simple concept related to how beats are counted in music, the days of the week, hours in the day, etc. Counting mod N just 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. A fancy way of saying a very simple thing.

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[4] 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 just 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 just 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 HW's image-stitching problem with corresponding points, 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. $y[k] = \sum_{j=0}^{N} \beta_j s[k-j]$. Now, the τ unknown is gone and in its place we have N of these β_j unknowns instead. We have N equations and N unknowns.

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

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

and then our equation is $C_{\vec{s}}\vec{\beta} = \vec{y}$.

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.

We know 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.

[4]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.

Orthogonality and Orthogonal Matrices

At the heart of the previous approach was to invert a $C_{\vec{s}}$ matrix. Before we talk about our special case, it is worth asking what kind of matrices are very easy to invert?

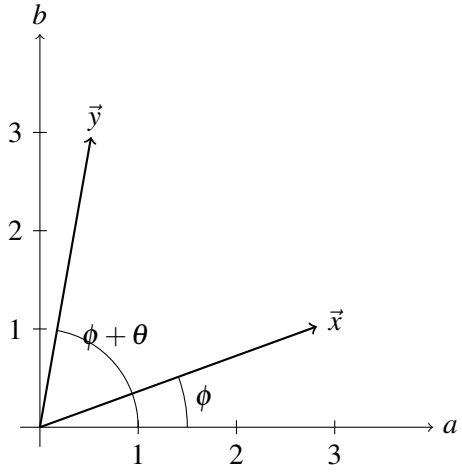
The easiest kind of matrix to invert is of course a diagonal matrix. We just take the reciprocal along the diagonal. But in our case of circulant matrices, there is essentially only one kind of signal $s[k]$ that would give rise to a diagonal matrix. This would be a pulse transmitted at time 0 so $s[0] = \beta$ and zeros at all other times $s[k] = 0$ for $k \neq 0$. Neither interesting nor practical.

It turns out that another easy-to-invert category of matrices are those whose columns are all orthogonal to each other.

Review Orthogonality

Let's say that $\vec{a} \in \mathbb{R}^N$ and $\vec{b} \in \mathbb{R}^N$. We know from previous encounters with orthogonality that the Euclidean inner product is $\left\langle \vec{a}, \vec{b} \right\rangle = \|\vec{a}\| \left\|\vec{b}\right\| \cos\theta$ where θ is the angle between the two vectors. Also we know that if this quantity is zero ($\left\langle \vec{a}, \vec{b} \right\rangle = \|\vec{a}\| \left\|\vec{b}\right\| \cos\theta = 0$), then $\vec{a}$ and $\vec{b}$ are orthogonal to each other.

Let's try to show this for a case in $\mathbb{R}^2$. Say $\vec{x} = [x_1 \;\; x_2]^T$ and $\vec{y} = [y_1 \;\; y_2]^T$. Here's a drawing of them.



Now by definition of the Euclidean inner product, we know that $\langle \vec{x}, \vec{y} \rangle = x_1 y_1 + x_2 y_2$.

At this point, we see the following, using trigonometry:

$$x_1 = \|\vec{x}\| \cos\phi$$
$$y_1 = \|\vec{y}\| \cos(\phi + \theta)$$
$$x_2 = \|\vec{x}\| \sin\phi$$
$$y_2 = \|\vec{y}\| \sin(\phi + \theta)$$

Now if we take the equalities that we just found and substitute them into the definition of the Euclidean product, then by applying the classic difference angle formula ($\cos(A-B) = \cos A \cos B + \sin A \sin B$) we see that it simplifies down to:

$$\langle \vec{x}, \vec{y} \rangle = \|\vec{x}\| \, \|\vec{y}\| \cos\theta$$

Recall that cos is zero exactly when the angle θ is a right angle (either 90 or 270 degrees). So a zero inner product indeed corresponds to classical perpendicularity.

Examples of Orthogonal Vectors

Two orthogonal vectors are shown below. It can be easily verified that their inner product is zero.

$$\vec{v}_1 = [1 \quad -1 \quad 1 \quad -1]$$
$$\vec{v}_2 = [1 \quad 1 \quad 1 \quad 1]$$

These kind of vectors with just $+1$ and -1 in them are very closely related to the kinds of signals transmitted by the location beacons in the lab, the signals emitted by GPS satellite, and the way that CDMA phones work in 3G wireless systems and 802.11b worked in early generations of WiFi.

Orthogonal Matrices

Suppose that we have a matrix whose columns are all orthogonal to each other. i.e. Suppose the columns $\vec{c}_i$ of the matrix C satisfy $\langle \vec{c}_j, \vec{c}_\ell \rangle = 0$ if $j \neq \ell$. We already know that $\langle \vec{c}_j, \vec{c}_j \rangle = \|\vec{c}_j\|^2$ by the definition of the Euclidean norm. Such a matrix C is termed orthogonal and we can see that the inverse of C is very easy to calculate since we can just look at the transpose of C and see what hapens when we multiply C by it.

$C^T C$ can be viewed as a matrix whose ℓth column of the jth row consists of the inner product $\langle \vec{c}_j, \vec{c}_\ell \rangle$. This means that all of the off-diagonal entries are zero by assumption. To get the inverse of an orthogonal matrix C, all we need to do is to divide the jth row of C^T by $\|\vec{c}_j\|^2$.

Approximately Orthogonal Matrices

It would be really easy to work with a beacon transmission that was such that it was orthogonal to all of its own shifts. But this seems very hard to do. Actually, only the same single pulse case works for this. So, why don't we relax our requirement a bit. What if the matrix of shifted signals were approximately orthogonal?

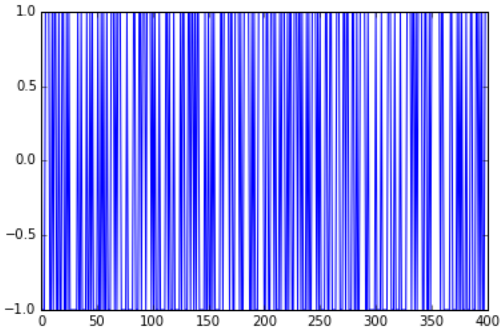
What should "approximately orthogonal" mean? Our context is the taking of the inverse. It would be great if we could just replace taking the inverse by doing inner products. This is because taking inner products just asks the question: how similar is the received vector to this particular vector? It asks this single question over and over again. It doesn't think about the global effect of the combination of vectors the way that an actual inverse does.

So, we are going to consider a matrix A approximately orthogonal if $A^T A$ is approximately diagonal — by which we mean that the diagonal terms are much bigger than the off-diagonal terms.

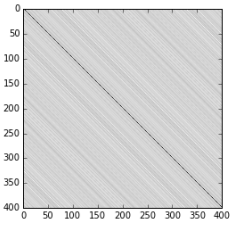
It's fine to idly speculate about how convenient such approximately orthogonal matrices would be, but do they exist?

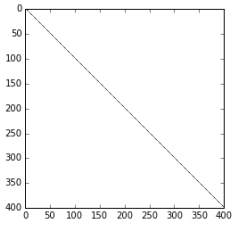
This is one of the (many) areas where large matrices and high-dimensional vector spaces actually behave in nicer ways than the more intuitive two and three dimensional spaces. Whereas approximately orthogonal matrices in two or three dimensions are somewhat special (take a few orthogonal vectors and then perturb them a bit), they are literally all over the place in higher dimensional spaces.

To see this, we fire up IPython and just try a 400 dimensional vector filled with $+1$ and -1 literally generated by virtually tossing 400 coins at random:

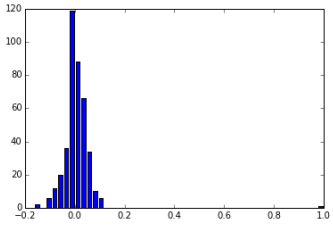


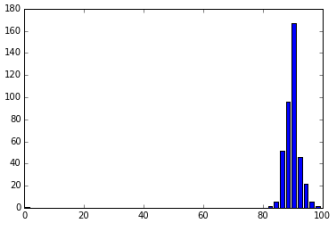
By construction, this vector $\vec{u}$ has Euclidean norm 20. Now, let's consider $C_{\vec{u}}$, the circulant matrix obtained from shifts of $\vec{u}$. Let's try $\frac{1}{400}C_{\vec{u}}^T C_{\vec{u}}$ and see what this looks like as compared to $C_{\vec{u}}^{-1}C_{\vec{u}}$. It is hard to write out a 400x400 matrix but we can plot them by intensity:





Notice how the circular structure to the matrix $C_{\vec{u}}$ creates this band-structure to $\frac{1}{400}C_{\vec{u}}^T C_{\vec{u}}$. The diagonal is solid and filled with 1s. All the other terms are much smaller, and none is bigger than 0.2 in magnitude. Here is a plot of the frequency of entries, first expressed in terms of the normalized inner product and then in terms of the angles (in degrees) between the vectors.





Notice how basically, all the time-shifts are approximately orthogonal to each other, within 10 degrees of perpendicular.

The deeper reason for this phenomenon will be explored in the third course in this trilogy, EECS 70, but it has to do with the laws of large numbers in probability. The important thing for us here is that this phenomenon justifies our use of inner products as a similarity measure between vectors in the context of searching for a time-shift of a beacon.

Cross-Correlation

The **Cross-correlation** is a measurement of the similarity between two vectors $\vec{u}$ and $\vec{y}$ - basically a sliding inner product. We compute the inner product $\langle \vec{u}, \vec{y} \rangle$, 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[5].

For us, the natural sense of cross-correlation will 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 we will be interested in getting correlations around 1.

If we want to find a signal $\vec{u}$ in an observation $\vec{y}$, 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 $\vec{u}$. The cross-correlation vector between $\vec{u}$ and $\vec{y}$ will be given by $S_{\vec{u}}\vec{y}$. The jth entry in the resulting vector is just how much the $\vec{y}$ contains the jth shift of $\vec{u}$. In other words, it is $\frac{1}{\|\vec{u}\|^2} \left\langle \text{Shift}_j \vec{u}, \vec{y} \right\rangle$. The largest cross-correlations, if significantly large (say larger than $\frac{\ln N}{\sqrt{N}}$ for a period N signal consisting of ± 1), 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.

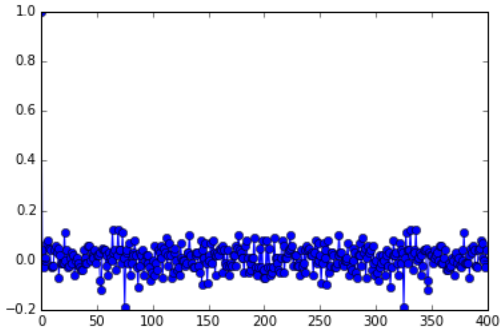
To see why this approach is reasonable, it is good to look at the **auto-correlation** which is just defined to be the cross-correlation of a signal with itself. This is clearly 1 in the first position by construction. The signal is a good candidate for being used as a beacon signal if the auto-correlation is small everywhere else. As we have seen from our random example, there are lots of good signals out there once N is large enough.

WARNING: Just because correlation works for this signal doesn't mean that it will work for any signal. There are signals out there that are horrible candidates for beacon transmission. (Can you think of any examples?)

[5] 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) 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 just 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.

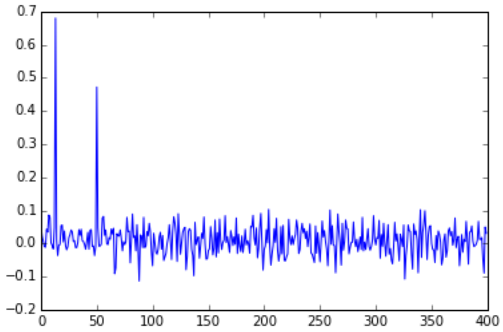
An extended example with plots

It is illustrative to look at the auto-correlation function for the random coin-toss signal we looked at earlier:

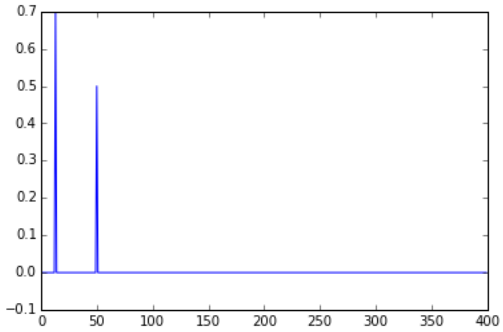


Notice again how it is mostly small as the histogram earlier had indicated.

Now, consider a $\vec{y}$ that was obtained by taking 0.7 times the shift of this signal by 13 steps and adding together 0.5 times a shift of this signal by 50 steps. Let's see what the cross-correlation of this with $\vec{u}$ looks like:



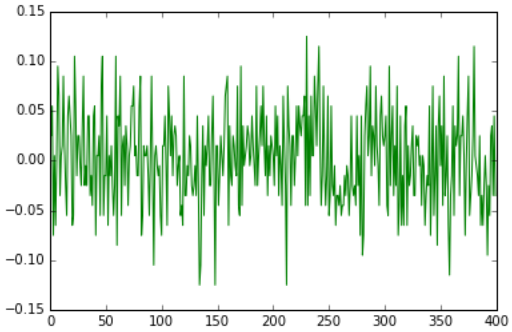
Notice how we can essentially read off not just the positions 13 and 50 but also the weights 0.7 and 0.5. Although it is interesting to notice that the weights aren't exactly right. For comparison, here's what we would've gotten had we used the inverse of $C_{\vec{u}}$ directly:



Here there is absolutely no ambiguity and recovery is perfect not just for the positions but also the weights.

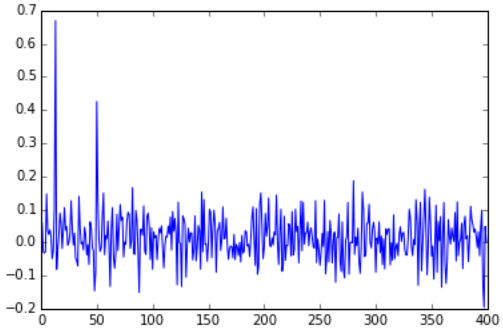
The great part about this approach is that it can be generalized to more than one beacon.

Here is the cross-correlation of $\vec{u}$ with another signal $\vec{v}$ also obtained by tossing coins at random:

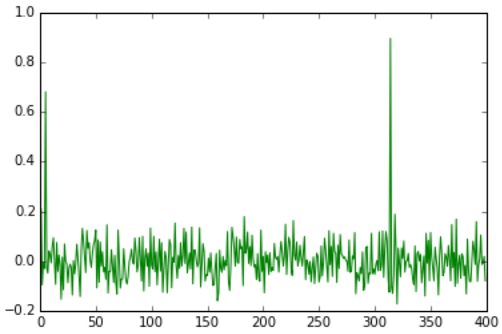


The important thing here is that all the entries are small. So this vector is approximately orthogonal to all the shifts of $\vec{u}$.

Now we consider a new $\vec{y}$ that was obtained by taking 0.7 times the shift $\vec{u}$ by 13 steps, adding together 0.5 times a shift of $\vec{u}$ by 50 steps, adding together 0.9 times a shift of $\vec{v}$ by 314 steps, and adding together 0.6 times a shift of $\vec{v}$ by 5 steps. By just cross-correlating with $\vec{u}$ we can easily read off the positions of the signal components by looking at the peaks:



And doing the same for $\vec{v}$ as well:



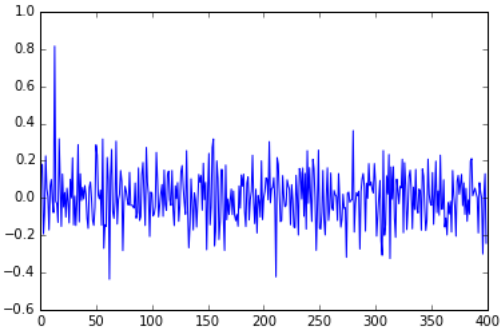
The amplitudes are a little off, but the positions are what are critical and we get them.

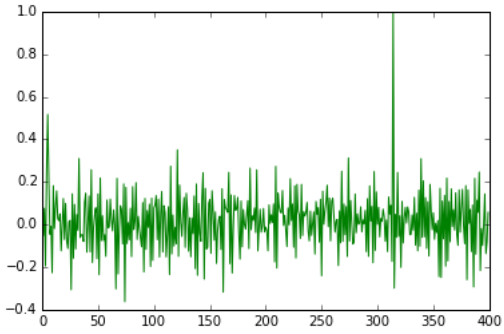
The cool thing here is that there are actually 800 possible vectors that we are allowing linear combinations of and only 400 measurements. But we are still somehow able to recover the 4 specific vectors from this that we wanted. The general structure that we are implicitly exploiting here is sometimes called "sparsity" and this is studied more in EECS 16B as well as exploited greatly in both EECS 127 (optimization) and EECS 189 (machine learning), to say nothing of its implicit exploitation in courses like EECS 120 and 123.

Would the inverse have worked as well? (Optional — for your interest only)

It is natural to wonder why we have spent so much effort in introducing the cross-correlation operation that leverages the inner product structure of the vector space. Is it just computational? Could we just use the inverse as well for this purpose?

The answer turns out to be no. Just using the inverse in place of the cross-correlation is a bad idea if we have shifts of more than one signal present in the observation. The following example should show you graphically that you should be wary. Here is a plot of the inverse of $C_{\vec{u}}$ being applied to $\vec{y}$ in the example above to try to find the $\vec{u}$ components. Followed by the same with the inverse of $C_{\vec{v}}$.





Notice that the second spike at 50 in the first is now completely gone. It is burried in the noise. The dominant spikes in the second are visible but the weaker one is definitely harder to distinguish and there are more spurious spikes that are tempting.

The deeper reasons for this are beyond the scope of this lecture note, but at a very coarse level, the intuition is that the inverse is perfectly matched to the exact problem that it was made to solve. But in getting that perfect, it is more vulnerable to interference from other signals. The cross-correlation approach is more robust to the presence of other signals. Of course, we cannot have too many signals or certainly, everything will break down.

In the lab, you will use the cross-correlation approach to locate audio beacons in the received signal. GPS uses cross-correlation to find the satellite beacons in the received signal.

Non-circular correlations

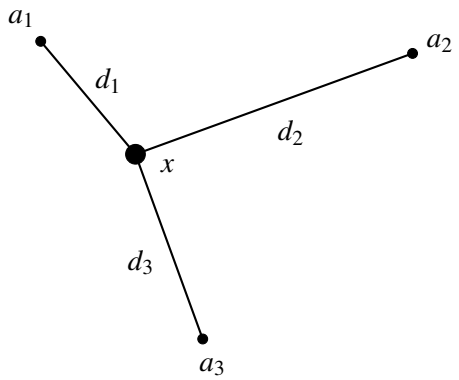
All of our discussion was centered on periodic signals like GPS and the ones in lab. If you want to define cross-correlation for non-periodic signals, this is also possible. Practically speaking, however, an easy trick is just to shoehorn those into the circular case by extending the signal with enough zeros (this is called zero-padding) and then pretending that a finite signal is periodic. Then, the appropriate slice of the cross-correlation will correspond to what you want without any of these wrap-around effects causing trouble.

In IPython, this corresponds to using the "full" mode of `numpy.correlate`.

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.

Now, 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:

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

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[6]. 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:

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

.

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.)

[6]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.