Laboratory # 2

Signal Correlation and Detection II

2.1 Introduction

In Lab #1, we designed an energy-based signal/no-signal detector for determining when a signal is present. This type of detector has a wide variety of applications, from speech analysis to communication, but it has two weaknesses. First, an energy-based detector is very susceptible to noise, especially when the signal's energy is small compared to the energy of the noise. Second, such a detector cannot distinguish between different types of signals that are mixed together.

In this laboratory, we will examine an alternative detection method that addresses these concerns. It uses a computation called *correlation* to detect the presence of a signal *with a known form*. In general, correlation measures the similarity between two signals. Using correlation for detection has significant applications. For instance, it allows several signals to be sent over a single communications channel simultaneously. It also allows the use of radar and sonar in noisy environments. Later in this course, we will see that correlation forms the basis for one of the most important tools in signals and systems engineering, the *spectrum*.

2.1.1 "The Questions"

- How can we transmit many different signals on the same channel?

- How can we develop a radar detection scheme that is robust to noise, and how do we characterize its performance?

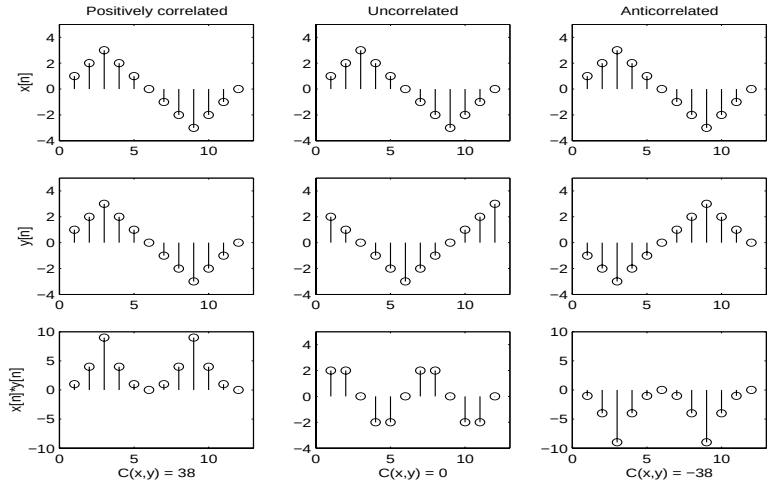

Figure 2.1: Examples of positively correlated, uncorrelated, and anticorrelated signals.

2.2 Background

2.2.1 Correlation

Suppose that we have two discrete-time signals, $x[n]$ and $y[n]$. We compute the *correlation*[1] between these two signals, $C(x, y)$, using the formula

$$C(x, y) = \sum_{n=n_1}^{n_2} x[n]y[n] \tag{2.1}$$

where n_1 and n_2 define the interval over which we are calculating the correlation. In words, we compute a correlation by multiplying two signals together and then summing the product. The result is a single number that indicates the similarity between the signals $x[n]$ and $y[n]$.

What values can $C(x, y)$ take on, and what does this tell us about the signals $x[n]$ and $y[n]$? Let us consider the examples in Figure 2.1. In the first column, $C(x, y) > 0$. When two signals have $C(x, y) > 0$, the signals are said to be *positively correlated*. Basically, this means that the signals are more similar than they are dissimilar. In the second column, we can see an example where $C(x, y)$ is zero. In this case, the two signals are *uncorrelated*. One might say that uncorrelated signals are "equally" similar and dissimilar. Notice, for instance, that the signal $x[n] \times y[n]$ is positive as often as it is negative. Finally, in the third column we see an example where $C(x, y) < 0$, which means that $x[n]$ and $y[n]$ are *anticorrelated*. This means the signals are mostly dissimilar.

Note that the positively correlated signals given in Figure 2.1 are actually identical. This is a special case; from equation (2.1), we can see that $C(x, x)$ is simply the energy of $x[n]$. When $C(x, y) = \sqrt{E(x)E(y)}$ (as in this case), the two signals are *perfectly correlated*. For two signals, this is the maximum possible correlation value. Similarly, the uncorrelated signals here are negated versions of one another, so $C(x, y) = -\sqrt{E(x)E(y)}$ (the most negative correlation value possible). When this happens, the two signals are called *perfectly anticorrelated*.

[1] We will occasionally refer to this operation as "in-place" correlation to distinguish it from "running" correlation.

Sometimes, it is useful to define a normalized correlation, $C_N(x, y)$, like this:

$$C_N(x,y) = \frac{C(x,y)}{\sqrt{E(x)E(y)}} = \frac{1}{\sqrt{E(x)E(y)}} \sum_{n=n_1}^{n_2} x[n]y[n]. \tag{2.2}$$

Normalized correlation is somewhat easier to interpret. $C_N(x, y)$ always lies in the range between -1 and 1. When two signals have a normalized correlation of 1, they are perfectly correlated; if they have a normalized correlation of -1, then they are perfectly anticorrelated.

2.2.2 Running correlation

Suppose that we want to know the distance to a certain object. We transmit a radar pulse, $x[n]$, and receive a signal, $y[n]$, that contains the reflection of our pulse off of the object. For simplicity, let's assume that we know $y[n]$ is simply a delayed version of $x[n]$, that is[2],

$$y[n] = x[n - n_0], \tag{2.3}$$

However, we do not know what the delay factor, n_0, is. Since n_0 is proportional to the distance to our object, this is the quantity that we wish to estimate. We can use correlation to help us determine this delay, but we need to extend the concept of in-place correlation.

Suppose that we first guess that n_0 is equal to zero. We correlate $x[n]$ with $y[n]$ and record the resulting correlation value as one sample of a new signal, say $r[0]$. Then, we guess that n_0 is equal to one, shift $x[n]$ over by one sample, and correlate $x[n-1]$ with $y[n]$. We record this correlation value as $r[1]$. We can continue this shift-and-correlate procedure, building up the new signal[3] $r[k]$ according to the formula

$$r[k] = C(x[n-k], y[n]) = \sum_{n=-\infty}^{\infty} x[n-k]y[n]. \tag{2.4}$$

Once we find a value of $r[k]$ that equals $E(x)$, we have found the value of n_0. This procedure of building up a signal like $r[k]$ is known as *running correlation* or *sliding correlation*. We will refer to the resulting signal ($r[k]$ above) as the *correlation signal*.

There are a couple of important features of the correlation signal. Note that the limits of summation in equation (2.4) are infinite. In general, though, we know that the support of $x[n]$ and $y[n]$ will be finite, so we do not actually need to perform an infinite summation. The duration of the correlation signal will be equal to the sum of the durations of $x[n]$ and $y[n]$ minus one[4]. There will also be *transient effects* (or *edge effects*) at the beginning and end of the correlation signal. These transient effects result from cases where $x[n-k]$ only partially overlaps $y[n]$.

An algorithm for running correlation

Here, we provide an algorithm for running correlation. One of its primary benefits is that it is easy to understand. In this algorithm, we imagine the filter as a box into which we drop one new sample of the "incoming" signal and a corresponding new sample of the correlation signal comes out. This allows the algorithm to be used in real-time: as samples of our signal

[2]Recall that a signal $x[n - n_0]$ is equal to the signal $x[n]$ shifted n_0 samples to the right.

[3]Note that we can index a signal by any variable; there is no difference between $r[k]$, $r[n]$, or $r[l]$. Here, we are simply introducing a new index variable to prevent confusion in equation (2.4).

[4]Suppose that support interval of $x[n]$ is $n_{x_1} \le n \le n_{x_2}$, while the support interval of $y[n]$ is $n_{y_1} \le n \le n_{y_2}$. For this general case, we can see that the first nonzero sample of $r[k]$ will occur at $k = n_{y_1} - n_{x_2}$. Similarly, the last nonzero sample will fall at $k = n_{y_2} - n_{x_1}$. Thus, the duration of $r[k]$ is $(n_{y_2} - n_{y_1}) + (n_{x_2} - n_{x_1}) + 1 = (n_{y_2} - n_{y_1} + 1) + (n_{x_2} - n_{x_1} + 1) - 1 = \text{duration}(x) + \text{duration}(y) - 1$.

arrive (from a radar detector, for instance), we can process the resulting signal with almost no delay.

In this algorithm we refer to the signal we are looking for (i.e., the transmitted radar signal) as $x[n]$, following (2.4). The algorithm goes like this:

1. Initialize an *input buffer*, which is simply an array with length equal to the duration of $x[n]$, to all zeros.
2. For each sample that comes in:
   (a) Update the buffer by doing the following:
      i. Discard the sample at the beginning of the buffer.
      ii. Shift the rest of the samples one place towards the beginning of the buffer.
      iii. Insert the incoming sample at the end of the buffer.
   (b) Initialize a running sum variable to zero.
   (c) For each position, n, in the buffer:
      i. Multiply the n^{th} position in the input buffer by the n^{th} sample of $x[n]$.
      ii. Add the resulting product to the running sum.
   (d) Output the running sum as the next sample of the correlation signal.

In the laboratory assignment, you will be asked to complete an implementation of this algorithm. Note that significant portions of this algorithm can be implemented very simply in MATLAB. For instance, all of (a) can be accomplished using a single line of code. Similarly, parts (b) through (d) can all be accomplished in a single line using one of MATLAB's built-in functions and its vector arithmetic capabilities.

2.2.3 Using running correlation for signal detection

Whenever we wish to use correlation for signal detection, we use a two-part system. The first part of the system performs the correlation and produces the correlation signal. The second part of the system examines the correlation signal and makes a decision. See the block diagram given in Figure 2.2.

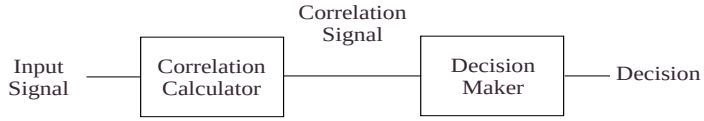

Figure 2.2: A generalized block diagram for a correlation-based detection system.

In the radar example used to motivate running correlation in Section 2.2.2, we simply checked to see if the correlation signal at a given point equals the energy of the transmitted signal. While this will work for the idealized system presented, real systems are usually much less ideal. We may have multiple reflections, distorted reflections, reductions in reflection amplitude, and other environmental noise. In order to address these and other, similar problems in a wide variety of systems, we commonly use a simple threshold comparison as our decision maker. For instance, if we compute a running correlation signal $r[n]$, we might make a decision for each sample of based on the following formula:

$$r[n] \underset{0}{\overset{1}{\gtrless}} c \tag{2.5}$$

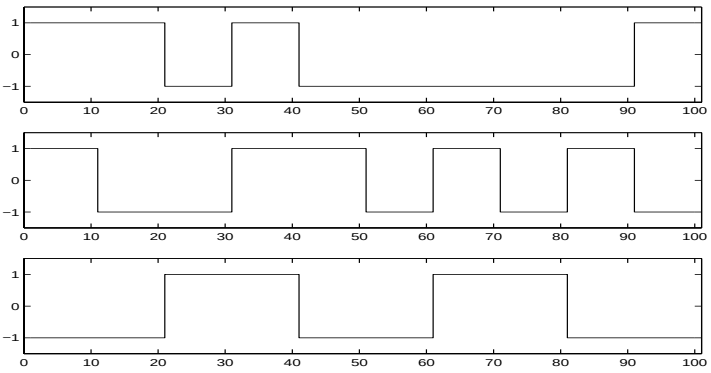

Figure 2.3: Example code signals for simultaneous communications

If the correlation signal's value for a given sample is greater than the threshold, c, we *decide* 1, or "signal present." If the value is less than the threshold, we decide 0, or "signal absent." In our radar example, for instance, we might select the threshold to be $c = E(x)/2$.

2.2.4 Using correlation for simultaneous communications

Suppose that we wish to implement a multi-user wireless communication system[5]. That is, many people will want to send different signals simultaneously; however, we only have a single communication channel (in this case, a small portion of the electromagnetic spectrum). The users of this system are completely uncoordinated, so no user has any idea who else might be using the system at any given time. How can we develop a system so that each user can use the system without experiencing interference from the other users?

It turns out that we can use a correlation-based detector to address this problem. Suppose that each user is trying to send a binary message to a friend. Each user has a different *code signal*, like those shown in Figure 2.3. Each code signal is made up of some number of *chips*, which are regions of constant signal value; the signals shown here each consist of ten chips. To send a binary "one," the user transmits his or her code signal. To send a binary "zero," a user instead transmits a negated version of his or her code signal (i.e., one that is perfectly anticorrelated with their code signal). In order to send a desired *sequence of bits*, the user concatenates these positive and negative versions of the code signal into a *sequence of code signals* and then transmits this sequence to his or her friend. When a user's friend receives the sequence of code signals, the friend correlates each code signal (in-place) with the user's code signal. When the resulting correlation is greater than zero (the threshold for this system), the friend records a "one;" when it is less than zero, the friend records a "zero".

This method has benefits when multiple users are using the channel simultaneously, or when there is other noise on the channel. In this case, when a user's friend receives a signal, it will be the sum of that user's sequence of code signals plus some other users' sequences and/or noise. However, the system is designed so that the correlation between two different code signals (or between a code signal and noise) is reasonably small. Consider the examples in Figure 2.3. The first two code signals are completely uncorrelated, as are the second two. The first and third signals are slightly anticorrelated. The normalized correlation between these signals is only -0.2. This should be compared to the correlation values of 1 and -1 that

[5]This is the problem faced by 600 Mhz cordless telephones, and the solution we will present is similar to the actual solution used.

result when a "proper" code signal is detected. Note that longer code signals have greater energy and are more easily distinguished from other code signals or from noise.

Above, we've indicated that our system uses in-place correlation. This means that this system is *synchronous*; that is, the receiver knows when bits are sent. We can actually save ourselves some work by using running correlation rather than in-place correlation, and then sampling the resulting correlation signal at the appropriate times. This is how we will implement this communication system in the laboratory assignment. Using the running correlation algorithm presented in this lab, the "appropriate times" occur in the correlation signal at the end of each code signal. That is, if our code signals are N samples long, we want to pick off the $(k \times N)^{th}$ sample to decode the k^{th} transmitted bit.

It is worth noting that the threshold used to decode bits in this communication system, which we set to zero, is actually a design parameter of the system. It might happen, for instance, that the system's noise is biased so that we tend to get slightly positive correlations when no signal is sent. In this case, we can improve performance of the system by adjusting the threshold. For another example, we might want to decide that no bit has been sent if the magnitude of the correlation is below some threshold. In this case, we actually have two thresholds. One separates "no signal" from a binary "one;" the other separates "no signal" from a binary "zero."

2.2.5 Noise, detector errors, and setting the threshold

Let's develop the radar example from Section 2.2.2 into a somewhat more realistic case. We transmit the signal $x[n]$ and receive a signal $y[n]$. $y[n]$, however, is no longer simply a single delayed replica of the transmitted signal. Instead, $y[n]$ may contain multiple reflections from several objects. We may not know how many objects are present, but we would like to determine the distance to each one. Also, we expect that $y[n]$ contains some amount of environmental noise that is unrelated to the transmitted signal.

In Section 2.2.3, we argued that a threshold-based decision maker was useful for such systems. Such a decision maker, though, will generally not perform perfectly. What happens if no reflected pulse is received, but the correlation signal falls above the threshold? Similarly, what happens if a reflected pulse is received without pushing the correlation signal above the threshold. In either case, the system has made an *error*. The likelihood of an error is dependent upon a number of factors, including the threshold itself and the amount of noise in the signal. In any real-world detection system, some possibility of error is generally unavoidable. Thus, we generally attempt to minimize the likelihood of errors, rather than expecting to eliminate them entirely.

There are two types of error that our detector can make. First, our detector could detect a reflection of the transmitted signal where no actual reflection exists. This is called a *false alarm*. A false alarm occurs when the correlation of the noise with transmitted signal exceeds the threshold. The other type of error occurs when the detector fails to detect a real reflection because the noise causes the correlation to drop below the threshold when a signal is present. This type of error is called a *miss*.

Depending on the detection system being developed, these two types of error could be equally undesirable or one could be more undesirable than another. For instance, in a defensive radar system, false alarms are probably preferable to misses, since the former are decidedly less dangerous. We can trade off the likelihood of these two types of error by adjusting the threshold. Raising the threshold decreases the likelihood of a false alarm, while lowering it decreases the likelihood of a miss.

It is often useful to know the likelihood of each type of error. There is a simple way of empirically determining these relative likelihoods. First, we obtain a signal, $n[k]$ that contains just the environmental noise that our system will see when in use. This is easy to

do by simply recording signal from the detector without transmitting a radar pulse. Then, we perform running correlation between $n[k]$ and the pulse we intend to transmit, $x[k]$. The resulting correlation signal, $n_c[k]$, is a sample of what the decision subsystem (i.e., the threshold) will see when no signal is present. We can count the number of times that $n_c[k]$ exceeds the threshold, c:

$$n_c[k] > c. \tag{2.6}$$

Then, if we divide by the total number of samples (i.e., the maximum possible number of false alarms), we have estimated the *false alarm rate*, which is the chance that any given sample will be a false alarm. Assuming that the noise characteristics do not change significantly, we can also use this technique to estimate the *miss rate*. When a signal is present, the signal values of the correlation signal will be approximately equal to the signal values without the signal *plus the energy of the transmitted pulse*. Thus, we can estimate the number of misses by counting the number of times that the following condition is met:

$$n_c[k] + E(x) < c \tag{2.7}$$

or

$$n_c[k] < c - E(x) \tag{2.8}$$

Dividing the resulting count by the total number of samples gives us an estimate of the miss rate.

The signal value distribution is also useful here. If we plot the histogram of values in $n_c[k]$, we can use this plot determine the error rates . The false alarm rate is equal to the area of the histogram that exceeds c divided by the total area of the histogram[6]. As we noted above, the signal value distribution of the correlation signal with a reflected pulse present is equal to the distribution of $n_c[k]$ shifted to the right by $E(x)$. Alternatively, we can shift the threshold to the left by $E(x)$. Then, the miss rate is equal to the area of the histogram that falls below $c - E(x)$ divided by the total area of the histogram. This allows us to set a threshold quickly simply by inspecting the histogram of $n_c[k]$.

Assuming that the distribution of $n_c[k]$ is symmetric, we can minimize the *total error rate* (which is simply the sum of the false alarm and miss rates) by setting a threshold that yields the same number of false alarms as misses. It turns out that this threshold value is equal to $E(x)/2$. We can argue for this intuitively. Since we've assumed a symmetric distribution for $n_c[k]$, the each type of error rate is dependent entirely on the distance of the thresholds c and $c - E(x)$ from the origin. To equalize the two types of error, then, the magnitude of each threshold should be the same. To achieve this, we must to set c equal to $E(x)/2$, which results in thresholds of $E(x)/2$ and $-E(x)/2$.

2.3 Some MATLAB commands for this lab

- **Calculating in-place correlation:** If you have two signals, `x` and `y`, that you wish to correlate, simply use the command

  ```
  >> c_xy = sum(x.*y);
  ```

  Note that `x` and `y` must be the same size; otherwise MATLAB will return an error.

- **The `subplot` command:** In order to put several plots on the same figure in MATLAB, we use the `subplot` command. `subplot` creates a rectangular array of axes in a figure.

[6] That is, we sum the values in this region of the histogram and divide by the sum of all values in the histogram

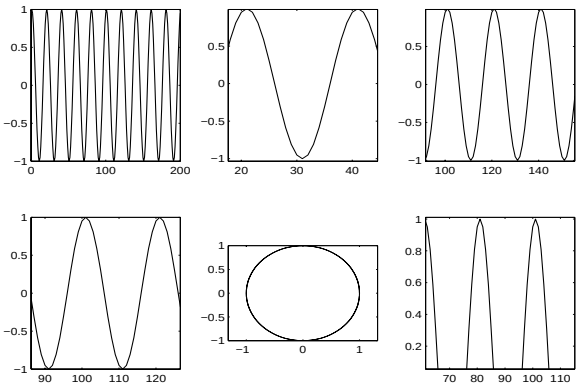

Figure 2.4: Example of a MATLAB figure with subplots.

Figure 2.4 has an example figure with such an array. Each time you call `subplot`, you activate one of the axes. `subplot` takes three input parameters. The first and second indicate the number of axes per row and the number of axes per column, respectively. The third parameter indicates which of the axes to activate by counting along the rows[7]. Thus the command:

```
>> subplot(2,3,5)
```

activates the plot with the circle in Figure 2.4.

- **The `axis` command:** The command `axis` allows us to set the axis range for a particular plot. Its single input argument is a vector of the form `[ x_min, x_max, y_min, y_max ]`. For instance, if you wish to change the display range of the currently active plot (or subplot) so that the x-axis ranges from 5 and 10 and the y-axis ranges from -100 to 100, simply execute the command

  ```
  >> axis([ 5, 10, -100, 100]);
  ```

  Other useful forms of the `axis` command include `axis tight`, which fits the axis range closely around the data in a plot, and `axis equal`, which assures that the x- and y-axes have the same scale.

- **Buffer operations in MATLAB:** It is often useful to use MATLAB's vectors as *buffers*, with which we can shift values in the buffer towards the beginning or end of the buffer by one position. Such an operation has two parts. First, we discard the number at the beginning or end of the buffer. If our buffer is a vector `b`, we can do this using either `b = b(2:end)` or `b = b(1:end-1)`. Then, we append a new number to the opposite end of the buffer using a standard array concatenation operation. Note that we can easily combine these two steps into a single command. For instance, if `b` is a row vector and we wish to shift towards the end of the buffer, we use the command

  ```
  >> b = [ new_sample, b(1:end-1) ];
  ```

- **Counting elements that meet some condition:** Occasionally we may want to determine how many elements in a vector meet some condition. This is simple in

[7]Note in particular that this is the opposite of MATLAB's usual convention!

MATLAB because of how the conditional operators are handled. Recall that for a vector, `v`, `(v == 3)` will return a vector with the same size as `v`, the elements of which are either 1 or 0 depending upon the truth of the conditional statement. Thus, to count the number of elements in `v` that equal 3, we can simply use the command

```
>> count = sum(v == 3);
```

2.4 Demonstrations in the Lab Section

- Detector error types
- Why use correlation? Or, when energy detectors break down.
- "In-place" correlation as a similarity measure
- Running correlation
- Multi-user communication

2.5 Laboratory Assignment

1. Download the file `code_signal.m` and use it to create the following signals:

```
>> code1 = code_signal(75,10);
>> code2 = code_signal(50,10);
>> code3 = code_signal(204,10);
```

   (a) Use `subplot` and `stairs` to plot the three code signals on three separate axes in the same figure. After plotting each signal, call `axis([1, 100, -1.5, 1.5])` to make sure that the signal is visible.
   - [4] Include your figure, with axis labels on *each subplot*, a figure number and caption, and the generating code in your report.

   (b) For each of the three signals generated above, calculate:
   - [3] Their mean values.
   - [3] Their energies.

   (c) Calculate the "in-place" correlation for the following pairs of signals.
   - [2] `code1` and `code2`
   - [2] `code1` and `code3`
   - [2] `code2` and `code3`

   (d) Which of the above pairs are:
   - [2] Positively correlated?
   - [2] Uncorrelated?
   - [2] Anticorrelated?

   (e) Each signal generated by `code_signal` can be thought of as a representation of a 10-bit number. This 10-bit number is the first input parameter to `code_signal`. Note that for any 10-bit number, there are ten other 10-bit numbers that are different from it by only one bit.

- [3] What is the *normalized correlation* between two 10 chip code signals that are different only by one chip?

2. Download the file `run_corr.m` from the course web page. `run_corr.m` is a "skeleton" file for an implementation of the "real-time" running correlation algorithm described in Section 2.2.2. It accepts two input signals, performs running correlation on them, and produces the correlation signal with a length equal to the sum of the lengths of the input signal minus one.

   (a) Complete the function, following the algorithm given in Section 2.2.2. You can use the completed demo version of the function, `run_corr_demo.dll` to check your function's output[8].

   - [10] Include your code in the MATLAB appendix of your report.

   (b) Use `run_corr.m` to compute the running correlation between the following pairs of signals, and plot the resulting correlation signals on the same figure using `subplot`.

   - [2] `code1` and `code2`.
   - [2] `code3` and itself.

   (c) When performing running correlation with a signal and itself, the resulting correlation signal has some special properties. Look at the correlation signal that you computed between `code3` and itself.

   - [1] Is the correlation signal symmetric? It can be shown that it should be.
   - [2] What is the maximum value of the correlation signal? How does this relate to the energy of `code3`?

3. Download the file `lab2_data.mat` and load it into your workspace. The file contains the variable `dsss`, which we will use in this problem. `dsss`[9] is a signal that is the sum of four sequences of different code signals corresponding to bit sequences from four different users. One of the code signals is a ten chip signal corresponding to the integer 170, while another is a six chip signal corresponding to the integer 25. The other two code signals are unknown to us. In this problem, we will try to extract the bit sequences for the known code signals from `dsss`. Start by generating the following code signals:

```
>> cs1 = code_signal(170,10);
>> cs2 = code_signal(25,6);
```

   (a) Start by using `run_corr` to correlate `cs1` with `dsss`. Call the resulting signal `cor1`.

   Now, since this communication technique is synchronous (as discussed in Section 2.2.4) we need to extract the appropriate samples from `cor1`. We can do this in MATLAB using the following command:

```
>> sub_cor1 = cor1(length(cs1):length(cs1):length(cor1));
```

   `sub_cor1` the contains the correlation values only at the appropriate samples for synchronous reception. We compare these values to a threshold to identify the bits.

[8] If you cannot get your function working properly, you may use `run_corr_demo.dll` to complete the rest of the assignment.

[9] This name comes from the name of the communication technique, *Direct Sequence/Spread Spectrum*.

- [2] Use `stem` to plot `sub_cor1`.
- [4] Decode the sequence of bits using zero as your threshold. (Hint: The sequence is 10 bits long, and the first 3 bits are "011".)

(b) Repeat the procedure in a and b above, this time using the code signal `cs2`. Call your correlation signal `cor2`, and the vector of extracted values `sub_cor2`.

- [2] Use `stem` to plot the signal `sub_cor2`.
- [4] Decode the sequence of bits. (Hint: there are 17 bits in this sequence.)
- [2] Since the code signal `cs2` has fewer chips (i.e., it is shorter), there is a greater chance of error. Are there any decoded bits that you suspect might be incorrect? Which ones? Why?

4. `lab2_data.mat` also contains three other signals: `radar_pulse`, `radar_received`, and `radar_noise`. The received signal contains several reflections of the transmitted radar pulse and noise. The signal `radar_noise` contains noise with similar characteristics to the noise in the received signal without the reflected pulses.

(a) First, let's take a look at the first two signals.

- [2] Calculate the energy of `radar_pulse`.
- [3] Use `subplot` to plot `radar_pulse` and `radar_received` in separate axes of the same figure.
- [1] Can you identify the reflected pulses in the received signal by visual inspection alone?

(b) Use `run_corr` to correlate `radar_pulse` with `radar_received`.

- [3] Plot the resulting correlation signal.
- [2] Where are the received pulses? Visually identify sample locations f each pulse in the correlation signal.
- [4] Given that the speed of light is 3×10^8 m/s and the sampling frequency of the detector is 10^7 samples per second, what is the approximate distance to each object[10]?

(c) In a real radar detector, the correlation signal would be compared to a threshold to perform the detection. To begin with, let's consider a threshold that is equal to one-half the energy of the transmitted pulse. Perform running correlation between `radar_pulse` and `radar_noise` call the resulting correlation signal `noise_c`.

- [2] Plot `noise_c`.
- [3] For how many samples is `noise_c` *greater* than this threshold? Use this value to estimate the false alarm rate.
- [3] For how many samples is `noise_c` *less* than this threshold minus the energy of the transmitted pulse? Use this value to estimate the miss rate.
- [2] What is the total error rate for this threshold?

(d) As discussed in Section 2.2.5, we can use a histogram to judge the number of errors as well.

- [3] Plot the histogram of `noise_c` using 100 bins.
- [3] Describe how you could derive the error numbers in problem 4c from the histogram.

[10] Remember that the radar pulse must travel to the object and then back again.

(e) Suppose that detector false alarms are considered to be more serious than detector misses. Thus, we have determined that we want to raise the threshold so that we achieve a false alarm rate of approximately 0.004. Find a threshold that satisfies this requirement.

- [4] What is your threshold?
- [3] What is the false alarm rate on this noise signal with your threshold?
- [3] What is the miss rate on this noise signal with your threshold?
- [3] What is the total error rate for the new threshold? Compare this to the total error rate of the threshold used in problem 4c.

