Laboratory # 1

Signals Statistics and Detection I

1.1 Introduction

In everyday language, a *signal* is anything that conveys information. We might talk about traffic signals or smoke signals, but the underlying purpose is the same. In the study of *signals and systems engineering*, however, we adopt a somewhat more specific notion of a signal. In this field, a signal is a numerical quantity that varies with respect to one or more independent variables. One can think of a signal as a functional relationship, where the independent variable might be time or position.

As an example, one signal might be the voltage on a wire that varies with time. Another signal might be light intensity that varies with position along a sensor array. The important aspect of these signals, though, is the mathematical representation, not the underlying medium. That is, the voltage and light signals might be exactly the same, despite the fact that the signals come from two widely different physical sources. In signals and systems engineering, we recognize that the most important aspects of signals are mathematical. Thus, we don't need to know anything about the physical behavior of voltage or light in order to usefully deal with these signals.

Throughout this course we will develop tools for analyzing, modifying, and extracting information from signals mathematically. One of the most basic (and sometimes most useful) methods involves the calculation of *signal statistics.* Calculating signals statistics provides us a substantial amount of useful information about a signal. These statistics allow us to determine "how much" signal is present (i.e., the *signal strength*), how long a signal lasts, what values the signal takes on, and so on. We will use signal statistics to develop measures of *signal quality* (with respect to a reference signal) and also to perform *signal detection* (by determining when a signal contains useful information rather than just background noise).

1.1.1 "The Questions"

- How do I quantitatively determine how "good" a signal is?
- How do I detect when a signal is present?

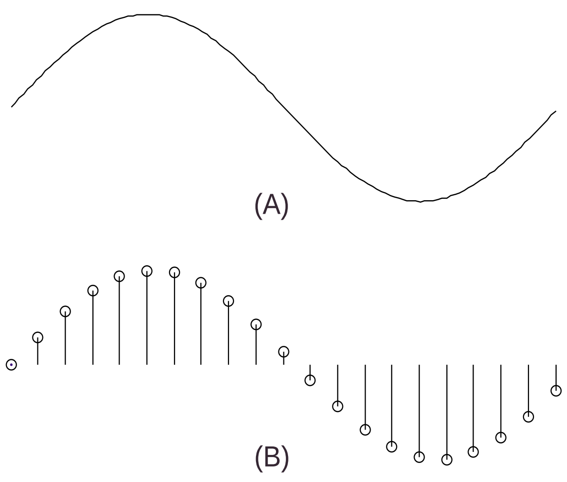

Figure 1.1: (A) A continuous-time signal. (B) A discrete-time signal.

1.2 Background

1.2.1 Continuous-time and discrete-time signals

Above, we suggested that signals often arise from measurable quantities. Let us define a signal, $s(t)$, which gives the voltage on a wire at a particular instant in time, t. We expect that the wire has a voltage at any (real-valued) time that we can think of, whether it be $t = 1$, $t = 2.874$ or $t = \pi$. A signal that has a value for any real value of the independent variable is called a *continuous-time* signal[1]

How can we represent a continuous-time signal? One way is to find a formula or rule that lets us calculate the signal value given the value of the independent variable. For instance, we might say that $s(t) = sin(t)$; this provides a handy method of representing and working with the signal. What happens, though, if we *cannot* find such a formula? Most signals have no simple, closed form representation. In this case, we need to list the signal values for every possible value of the independent variable. Unfortunately, there are infinitely many possible values for the independent variable between any other two values[2]! This means that we need an infinite amount of time and storage space to list out the signal values of a continuous-time signal; in particular, this makes representing continuous-time signals with a computer a challenge.

An alternative to continuous-time signals are *discrete-time* signals[3]. Discrete-time signals take values only when their independent variable is an integer. We denote discrete-time signals using square brackets to contain the time parameter; e.g. $s[n]$. When dealing with discrete-time signals, there are only a finite number of values that we need to store to represent a signal in a finite interval. This is very important because it allows us to represent and manipulate signals using computers. Throughout these laboratories, we will use MATLAB to help us work with discrete-time signals. Figure 1.1 shows examples of continuous-time and discrete-time signals.

Note that in some cases it is possible to convert a continuous-time signal into a discrete-

[1]In some situations, the independent parameter is not "time" but something else, like "distance". In such cases, the signal can be said to be *continuous-space* or, more generally, *continuous-parameter.*

[2]This is called an *uncountably infinite* number of values.

[3]Again, this is could be called a *discrete-space* or *discrete-parameter* in situations where the independent variable is not time

time signal by *sampling*. Basically, sampling exploits the fact that most signals vary relatively slowly. If we record enough values per unit of time, we can usually get a good discrete-time representation of the signal. We will study sampling in detail in Chapter 4 of the textbook, but we will use sampling to work with continuous-time signals before then.

1.2.2 Signals in Matlab

In Matlab, signals are simply arrays of numbers. Let us consider an example. Suppose we want to represent the following discrete-time signal in Matlab:

$$s[n] = \begin{cases} n^2 & 5 \le n \le 15 \\ 0 & else \end{cases} \tag{1.1}$$

In Matlab, we do this by creating two vectors: a *support vector* and a *signal vector*. The support vector represents the *support interval* of the signal, i.e. the set of integers from the first time at which the signal is nonzero to the last. For this example, the support vector can be created with the command

```
>> n = 5:15
```

Then, the signal vector can be created with the command

```
>> s = n.^2
```

Note that we (usually) only define the signal within the range that it is nonzero. That is, we assume that the signal is zero outside of the range for which it is defined. We will also occasionally assume that the support vector for a signal `s` is given by `n = 1:length(s)`.

To plot this discrete-time signal, use the stem command:

```
>> stem(n,s)
```

You can also use the `plot` command; however, `plot` draws straight lines between plotted points, which may not be desirable.

It is also possible to represent continuous-time signals in Matlab, by converting them to discrete-time with sampling. To do so, we first need to define a *sampling interval*, T_s. The sampling interval is the amount of time that separates two samples of the signal, and is generally given in seconds (or "seconds per sample"). We can also define the *sampling frequency*, f_s, to be one over the sampling interval. That is, $f_s = 1/T_s$. The sampling frequency is given in inverse seconds[4] (or "samples per second"). Matlab generally uses sampling frequency rather than sampling interval, but the sampling interval is a somewhat more intuitive idea.

We first construct a *time axis*[5] using this sampling frequency. Let's define the sinusoid $x(t) = sin(2\pi t)$ over the range from 0 to 5 seconds by sampling it at a frequency of 20 Hz.

```
>> fs = 20;
>> t = 0:1/fs:5;
>> x = sin(2*pi*t);
```

Here, `fs` is the sampling frequency, `t` is the time axis, and `x` is the signal. To plot the sampled signal, use the `plot` command like this:

```
>> plot(t,x);
```

Note the important difference between `plot(t,x)` and `plot(x)`. In the first case, the x-axis is given in *time*. In the second case, however, the x-axis is given in *samples* with the support vector assumed to be `1:length(x)`. It is important that you always know what the axes of your plots represent and that you label them with the `xlabel` and `ylabel` commands.

[4]Inverse seconds is occasionally given units of "Hertz" or Hz.
[5]The time axis is effectively a support vector for sampled signals.

1.2.3 Signal Statistics

We can calculate signal statistics in both continuous-time and in discrete-time. For sampled signals, we can also sometimes calculate an approximation to the continuous-time statistics. The following lists a number of signal statistics, their discrete- and continuous-time versions, and MATLAB code for calculating the discrete-time version. In some cases, the statistics for the continuous-time signal can be approximated using a sampled version; in most cases, the approximation becomes better as the sampling frequency, f_s, increases.

1. **Support Interval**. A signal's *support interval* (also occasionally known as just the signal's *support* or its *interval*) is the smallest interval that includes all non-zero values of the signal.

   $$\text{Continuous-time:} \quad t_1 \le t \le t_2 \tag{1.2}$$
   $$\text{Discrete-time:} \quad n_1 \le n \le n_2 \tag{1.3}$$
   $$\text{MATLAB:} \quad \texttt{n = n1:n2} \text{ for a signal } \texttt{s}. \tag{1.4}$$

2. **Duration**. The *duration* of a signal is simply the length of the support interval.

   $$\text{Continuous-time:} \quad t_2 - t_1 \tag{1.5}$$
   $$\text{Discrete-time:} \quad n_2 - n_1 + 1 \tag{1.6}$$
   $$\text{MATLAB:} \quad \text{Assumed } \texttt{length(s)} \text{ for a signal } \texttt{s}. \tag{1.7}$$
   $$\text{Sampled:} \quad (n_2 - n_1 + 1) \approx f_s(t_2 - t_1) \tag{1.8}$$

3. **Fundamental Period**. Certain signals (called *periodic signals*) repeat themselves in time. The *fundamental period*, T_0, of a periodic signal is the smallest time between repetitions in the signal. A signal may not be periodic, in which case it will have no fundamental period. Note that truly periodic signals necessarily have infinite duration. We will often talk about the "period" of signals that are nearly periodic over some finite interval, even though they are not truly periodic.

   $$\text{Continuous-time:} \quad s(t) = s(t + T_0) \tag{1.9}$$
   $$\text{Discrete-time:} \quad s[n] = s[n + T_0] \tag{1.10}$$

4. **Maximum and Minimum Value**. These values are the largest and smallest values that a signal takes on over some interval defined by n_1 and n_2. In MATLAB these values are found using the `min` and `max` commands.

   $$\text{MATLAB:} \quad \mathit{Maximum}(\texttt{s}) = \texttt{max(s((n1:n2)-n(1)+1))}^{6} \tag{1.11}$$
   $$\text{MATLAB:} \quad \mathit{Minimum}(\texttt{s}) = \texttt{min(s((n1:n2)-n(1)+1))} \tag{1.12}$$

5. **Average Value**. The *average value*, M, is the value around which the signal is "centered" over some interval.

   $$\text{Continuous-time:} \quad M(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s(t)dt \tag{1.13}$$
   $$\text{Discrete-time:} \quad M(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n_1}^{n_2} s[n] \tag{1.14}$$
   $$\text{MATLAB:} \quad M(\texttt{s}) = \texttt{mean(s((n1:n2)-n(1)+1))} \tag{1.15}$$
   $$\text{Sampled:} \quad M(s[n]) \approx M(s(t)) \tag{1.16}$$

[6] For all of these statistics, you can use `s(n1:n2)` if the support vector, `n`, is `1:length(s)`. Further, if you wish to calculate the statistic over the entire signal, you can simply use `s`.

6. **Mean-squared value**. The *mean-squared value* (or *MSV*) of a signal, MS, is defined as the average squared valued of the signal over an interval. The MSV is also called the **average power**, because the squared value of a signal is considered to be the instantaneous power of the signal.

$$\text{Continuous-time:} \quad MS(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt \tag{1.17}$$

$$\text{Discrete-time:} \quad MS(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n_1}^{n_2} s^2[n] \tag{1.18}$$

$$\text{MATLAB:} \quad MS(\texttt{s}) = \texttt{mean(s((n1:n2)-n(1)+1).\^{}2)} \tag{1.19}$$

$$\text{Sampled:} \quad MS(s[n]) \approx MS(s(t)) \tag{1.20}$$

7. **Root mean squared value**. The *root mean squared value* (or *RMS value*) of a signal over some interval is simply the square root of mean squared value.

$$\text{Continuous-time:} \quad RMS(s(t)) = \sqrt{\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt} \tag{1.21}$$

$$\text{Discrete-time:} \quad RMS(s[n]) = \sqrt{\frac{1}{n_2 - n_1 + 1} \sum_{n_1}^{n_2} s^2[n]} \tag{1.22}$$

$$\text{MATLAB:} \quad RMS(\texttt{s}) = \texttt{sqrt(mean(s((n1:n2)-n(1)+1).\^{}2))} \tag{1.23}$$

$$\text{Sampled:} \quad RMS(s[n]) \approx RMS(s(t)) \tag{1.24}$$

8. **Signal Energy**. The *energy* of a signal, E, indicates the strength of a signal is present over some interval. Note that energy equals the average power times the length of the interval.

$$\text{Continuous-time:} \quad E(s(t)) = \int_{t_1}^{t_2} s^2(t)dt \tag{1.25}$$

$$\text{Discrete-time:} \quad E(s[n]) = \sum_{n_1}^{n_2} s^2[n] \tag{1.26}$$

$$\text{MATLAB:} \quad E(\texttt{s}) = \texttt{sum(s((n1:n2)-n(1)+1).\^{}2)} \tag{1.27}$$

$$\text{Sampled:} \quad E(s[n]) \approx f_s E(s(t)) \tag{1.28}$$

9. **Signal Value Distribution**. The *signal value distribution* is a plot indicating the relative occurrence of values in a signal. There is no formula defining the signal value distribution, but it can be approximated using a *histogram.* A histogram counts the number of samples that fall within particular ranges, or *bins.* Note that the y-axis is effectively arbitrary, and that the coarseness of the approximation is determined by the number of histogram bins that are used. Figure 1.2 shows an example of a signal value distribution and the histogram approximation to that distribution.

$$\text{MATLAB:} \quad \texttt{hist(s((n1:n2)-n(1)+1),num\_bins);} \tag{1.29}$$

1.2.4 Measuring signal distortion and error

Suppose that we wish to transmit a signal from one location to another. This is a common task in *communication systems.* A common problem with this task is that the signal is often

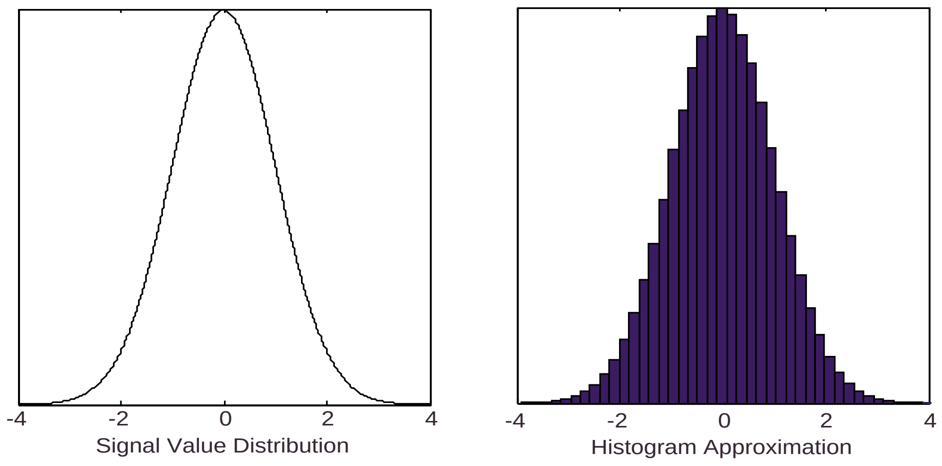

Figure 1.2: Signal value distribution and a discrete histogram approximation

modified or *distorted*. Thus, the received signal is not the same as the transmitted signal. Typically, we want to reduce the amount of distortion as much as possible. However, this requires that we have a method of measuring the amount of distortion in a signal. In order to develop such a measure, we'll look at a *signal plus noise* model of signal distortion.

Suppose we are transmitting a signal $s[n]$ over FM radio. Someone tunes in to our radio station and receives a modified version of our signal, $s'[n]$. We can represent this modification mathematically as the addition of an *error signal*, $v[n]$, like this:

$$s'[n] = s[n] + v[n]. \tag{1.30}$$

Assuming that we have both $s[n]$ and $s'[n]$, we can easily calculate $v[n]$ as

$$v[n] = s'[n] - s[n]. \tag{1.31}$$

Note that if $s[n]$ and $s'[n]$ are identical, $v[n]$ will be zero for all n. This suggests that we can simply measure the signal strength of $v[n]$ by using one of the energy or power statistics.

Mean squared value is a natural choice because it normalizes the error with respect to the length of the signal. Sometimes, though, the RMS value is more desirable because it produces error values that are directly comparable to the values in $v[n]$[7]. When we measure the MSV of an error signal, we sometimes call it the *mean squared error* or *MSE*. Similarly, the RMS value of an error signal is often called the *root mean squared error* or *RMSE*.

In MATLAB, we will usually want to calculate the MSE or RMSE over the entire length of the signals that we have. Supposing that we are given a signal `s` and a modified version `s_mod` (with the same size), we can calculate the MSE and RMSE like this:

```
>> mse = mean((s - s_mod).^2);
>> rmse = sqrt(mean((s - s_mod).^2));
```

Notice that we could also subtract `s_mod` from `s`; the order doesn't matter because of the square operation. Also note that you *must* include the period before the exponentiation operator in order to correctly square each sample.

[7]Mean square values are comparable to the square of the values in $v[n]$

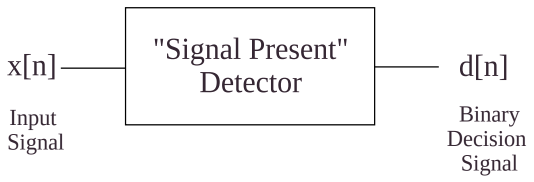

Figure 1.3: An “overview” block diagram for the signal/no signal detector.

1.2.5 Signal detection

Suppose that we are designing a continuous speech recognition and transcription system for a personal computer. The computer has a microphone attached to it, and it “listens” to the user’s speech and tries to produce the text that was spoken. However, the user is not speaking continuously; there are periods of silence between utterances. We don’t want to try to recognize speech where there is silence, so we need some means of determining when there is a speech signal present.

This is an example of *signal detection.* There many different types of signal detection. Sometimes signal detection involves finding a signal that is obscured by noise, such as radar detection. In other applications, we need to determine if a particular signal exists in a signal that is the sum of many signals. The “signal present” detector for our speech recognition system is a simpler form of signal detection, but it still important in many applications. One particular communication system, known as *on/off keying*, generally makes use of such a “signal present” detector. Here, we’ll consider the design of such a detector, shown in Figure 1.3.

Specifying the detector’s behavior

The first step is to specify specifically what our system needs to do. From the description above, we know that we will receive a signal as an input. For simplicity, we’ll assume that we are given an entire discrete-time signal. What must our system do? We need some sort of indication as to when a signal is present. However, the signal that we are given will contain both periods of silence and periods with valid signal. Suppose we break the signal into *blocks* of N samples, such that the first N samples make up the first block, the second N samples make up the second block, and so on. Then, we can make a “signal present” decision for each block separately. The output of our system should consist of two arrays. The first is an array containing present/not present decisions (as ones and zeros) for each block. The second is an “x-axis” array that indicates the sample number that starts each blocks referred to in the first array. This second array will allow us to plot the decisions on the same figure as the signal itself.

How will we make the decision for each block, though? Since we can assume a signal that is relatively free of noise and has zero mean, we can simply calculate an energy-based statistic for each block and compare the result to a threshold. If the statistic exceeds the threshold, we say that a signal is present. Otherwise, we assume that no signal is present. Using signal energy, though, is not ideal; the necessary threshold will depend on the block size. We may want to change the block size, and we should be able to keep the threshold constant. Using average power is a better option, but we would like our threshold to have a value comparable to the values of the signal. Thus, the RMS value seems to be an ideal choice.

Note that our detector will have two parameters. One is the block size, and the other is the threshold. We will need to find reasonable values for these parameters when we make

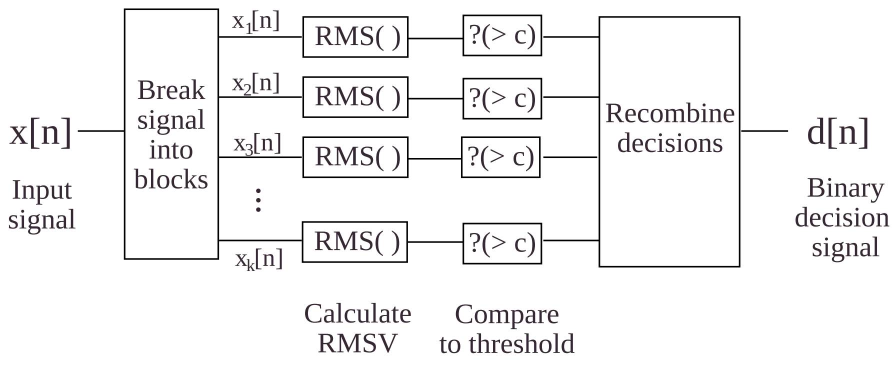

Figure 1.4: A detailed block diagram for the signal/no signal detector.

the detector itself. A more detailed block diagram of the detector can be found in Figure 1.4.

Detector algorithm

Now that we've specified the behavior of the detector, let's come up with an algorithm for performing the detection. Of course, this is not the only way to implement this detector.

- Define a `block_size`.
- Define a `threshold`.
- Calculate the `number_of_blocks`.
- Define your `x_axis`.
- For each block of the signal:
  - Calculate the RMS value of the current block
  - Compare the RMS value to the threshold
  - Store the result in the `output` array
- Return the `output` array

What follows are some details about the algorithm:

1. First, note that we want `number_of_blocks` to be an integer. For this calculation, recall that the function `length` returns the number of samples in a vector. Also, note that the `floor` command rounds down to the nearest integer

2. Suppose that the block size is 512. The vector `x_axis` should contain the numbers `[1, 513, 1025, ...]` and so on. You can generate `x_axis` with a single line of code by using the `:` operator[8].

3. There are actually two separate ways to implement the "for each block" part of this algorithm in MATLAB. One involves using a `for` loop, while the other makes use of the `reshape` command and MATLAB's vector and matrix arithmetic capabilities.

[8]Type `help colon` if you need assistance with this operator.

Both take roughly the same amount of code, but the second way is somewhat faster. You can implement whichever version of the algorithm that you choose in the lab assignment.

(a) If you implement the algorithm using a `for` loop, you should first initialize the `output` array to "empty" using the command "`output =[];`". Then, loop over the values in `x_axis`. Within the loop, you need to determine what values of n_1 and n_2 to use in the RMS value calculation for a given value of the loop counter. Then, compare the RMS value that you calculate to the threshold and append the result to the end of `output`[9].

(b) An alternative to the `for` loop is to use the `reshape` command to make a matrix out of our signal with one block of the signal per column. If you choose to use `reshape`, you first need to discard all samples beyond the first `block_size` × `number_of_blocks` samples of the input signal. `reshape` this shorter signal into a matrix with `block_size` rows and `number_of_blocks` columns[10]. Then, use `.^` to square each element in the matrix, use `mean` to take the mean value of each column, and take the `sqrt` of the resulting vector to produce a vector of RMS values. Finally, compare this vector to `threshold` to yield your output vector.

1.3 Some MATLAB commands for this lab

- **Zooming on Figures:** In MATLAB, you can interactively zoom in and out on figure windows. To do so, you can either find a "+ magnifying glass" icon on the figure window, or you can type `zoom on` at the command lines. Then, you can click and drag a zoom box on the figure window to get a closer look at that portion of the figure. Also very useful is the `zoom xon` command, which only enables zooming in the x-direction; this is usually how we will want to zoom in on our signals.

- **Using line styles and `legend`:** Whenever you plot two or more signals on the same set of axes, you must make sure that the signals are distinguishable and labeled. Generally, we do this using line styles and the `legend command`. The `plot` command gives you a wide range of options for changing line styles and colors. For instance, the commands

```
>> hold on
>> plot(1:10,1:10,'-')
>> plot(1:10,2:11,':')
>> plot(1:10,3:12,'--')
```

plot lines using solid, dotted, and dashed lines. Type `help plot` for more details about using different line styles and colors. The `legend` command adds a figure legend for labeling the different signals. For instance, the command

```
>> legend('Solid (lower)', 'Dotted (middle)', 'Dashed (higher)')
```

adds a legend with labels for each of the three signals on the figure. Note that signal labels are given in the order that the signals were added to the figure.

[9]Use either `output(end+1) = result;` or `output = [output, result];`

[10]Remember to assign the output of `reshape` to something! No MATLAB function ever modifies its input parameters unless you explicitly reassign the output to the input.

- **Labeling Figures:** Any time that you create a figure for this laboratory, you need to include axis labels, a figure number, and a caption:

  ```
  >> xlabel('This is the x-axis label');
  >> ylabel('This is the y-axis label');
  >> title('Figure 1: This is a caption describing the figure');
  ```

  Note that it is recommended that you use your word processor to produce figure numbers and captions, rather than using the `title` command. You also need to include the code that you used to produce the figures, including label commands. Note that each `subplot` of a figure must include its own axis labels.

- **Function Headers:** At the top of the file containing a function declaration, you must have a line like this:

  ```
  function [out1, out2, out3] = function_name(in1, in2, in3)
  ```

  where `in1`, `in2`, and `in3` are input parameters and `out1`, `out2`, and `out3`. Note that you can name the parameters anything you like, and there can be any number of them. The word `function` is a MATLAB keyword. Also, you do not need to explicitly return the output parameters. Instead, MATLAB will take their values at the end of the function's execution and return them to the calling function.

- **`sum`, `mean`, `min`, and `max`:** Given a vector (i.e., a one-dimensional array), these MATLAB functions calculate the sum, mean, minimum, and maximum (respectively) of the numbers in the array and returns a single number. If these functions are given a matrix (i.e., a two-dimensional array), they calculate the appropriate statistic on each column of the matrix and return a row-vector containing one result for each column.

- **`for` loops:** `for` loops in MATLAB have the following form

  ```
  >> for index = row_vector
  >>   % Code in the loop goes here
  >> end
  ```

  `index` is the loop counter (which, of course, can be called anything you like). `row_vector` is a row vector that contains all the values that will be assigned to `index` on each iteration of the loop. Thus, the loop will execute `size(row_vector,2)` times. If `row_vector` is actually a matrix (or a column vector), each column of `row_vector` will be assigned to `index` in turn.

  It is common to simply define the range of iteration using the `:` operator, like this:

  ```
  >> for index = 1:20
  >>   % This loop is executed twenty times
  >> end
  ```

- **Reshaping arrays:** The `reshape` command is used to change the shape of an array:

  ```
  >> new_array = reshape(array,[new_rows, new_columns]);
  ```

  `array` must have `new_rows*new_columns` elements[11]. `new_array` will have dimensions of `new_rows` $\times$ `new_columns`.

[11]The number of elements in `array` can be checked using the command `prod(size(array))`.

- **Logical operators:** The logical operators (`>`, `<`, `>=`, `<=`, `==`, `~=`) perform a test for equality or various forms of inequality. They all operate in the same manner by evaluating to 1 ("true") or 0 ("false") depending upon the truth value of the operator. After executing the following statement, for instance,

  ```
  >> result = (x > 0);
  ```

  `result` will contain a one or a zero if `x` is a single number. If `x` is an array, `result` will be an array of ones and zeros with a size equal to `x`, where each element indicates whether the corresponding element in `x` is in fact greater than 0.

1.4 "Demonstrations" in the Lab Section

- Laboratory policies
- Signals in Matlab – sampling
- Signal statistics
- Approximating continuous-time signal statistics with sampled signals
- Model of discrete-time signal as signal plus noise
- The signal/no-signal detector

1.5 Laboratory Assignment

Note that in this and all following laboratory assignments, the bullets • indicate items that you must include in your laboratory. The number in brackets indicates how many points each item is worth.

1. Use the following Matlab commands to create a signal:

   ```
   >> n = 1:50;
   >> s = sin(2*pi*n/50);
   ```

   (a) Use `stem` to plot the signal. Make sure that you include[12]:
   Also, remember to tell them to put the code for their figure in the Matlab appendix.
   - [2] The figure itself[13].
   - [1] An x-axis label and a y-axis label.
   - [1] A figure number and a caption that describes the figure.
   - [1] The code you used to produce the signal and the figure. This should be included in an appendix at the end of your report[14]. Make sure you clearly indicate which problem the code belongs to.

[12]Note that *every* figure that you produce in a laboratory for this class must include these things!

[13]On Windows systems, you can select "Copy Figure" from Edit menu on the figure window to copy the figure to the clipboard and then paste it into your report.

[14]You should include *all* Matlab code that you use in the appendix. However, you do not need to include code that is built into Matlab or code that we provide to you.

(b) Calculate the following statistics over the length of the signal (i.e., let $n_1 = 1$ and $n_2 =$ `length(s)`), and include your results in your report[15].
   - [2] Maximum value
   - [2] Minimum value
   - [2] Mean value
   - [2] Mean squared value
   - [2] RMS value
   - [2] Energy

(c) Suppose that `s` is the result of sampling a continuous-time signal with a sampling frequency of $f_s = 100$ Hz. Use the discrete-time statistics to estimate the following statistics for the continuous-time signal:
   - [2] Signal duration
   - [2] Energy
   - [2] Average power
   - [2] RMS Value

2. Download the file `lab1_data.mat` from the course web page. Place it in the present working directory or in a directory on the path, and type

```
>> load lab1_data
```

This file contains two signals which will be loaded into your workspace. You will use the signal `clarinet`[16] in this problem and in Problem 3. The other signal, `mboc`, will be used in Problem 4.

(a) Define the support vector for `clarinet` as `1:length(clarinet)`. Then, use `plot` to plot the signal `clarinet`.
   - [4] Include the figure (with axis labels, figure number, caption, and MATLAB code) in your report.

(b) Zoom in on the signal so that you can see four or five "periods[17] ."
   - [4] Include the zoomed-in figure (with axis labels, figure number, caption, and MATLAB code) in your report.

(c) Estimate the "fundamental period" of `clarinet`. Include:
   - [3] Your estimate for the discrete-time signal in samples.
   - [3] Your estimate for the original continuous-time signal in seconds.

(d) Use the `hist` command to estimate the signal value distribution of `clarinet`. Use 50 bins.
   - [4] Include the figure (with axis labels, code, etc.) in your report.
   - [1] From the histogram, make an educated guess of the MSV and RMSV. Explain how you arrived at these guesses.

(e) Calculate the following (discrete-time) statistics over the length of the signal:
   - [2] Mean value

[15]Remember to include the code you used to calculate these in your MATLAB appendix.

[16]This is a one-second recording of a clarinet, recorded at a sampling frequency of 22,050 Hz. To listen to the sound, use the command `soundsc(clarinet,22050)`.

[17]This signal, like all real-world signals, is not exactly periodic; however, it is approximately periodic.

- [2] Energy
- [2] Mean squared value
- [2] RMS value

3. In this problem, we'll measure the amount of distortion introduced to a signal by two "systems." Download the two files `lab1_sys1.m` and `lab1_sys2.m`. Apply each system to the variable `clarinet` using the following commands:

```
>> sys1_out = lab1_sys1(clarinet);
>> sys2_out = lab1_sys2(clarinet);
```

   (a) Use `plot`[18] and MATLAB's zoom capabilities to display roughly one "period" of:
      - [3] The input and output of `lab1_sys1` on the same figure.
      - [3] The input and output of `lab1_sys2` on the same figure.

   (b) What happens to the signal when it is passed through these two systems? Look at your plots from the previous section and describe the effect of:
      - [3] `lab1_sys1.m` on `clarinet`.
      - [3] `lab1_sys2.m` on `clarinet`.

   (c) Calculate the RMS error introduced by each system.
      - [3] RMS error introduced by `lab1_sys1`.
      - [3] RMS error introduced by `lab1_sys2`.
      - [1] Which system introduces the least error? Is this what you would have expected from your plots?

4. Download the files `sig_nosig.m` and `lab1_data.mat` (if you haven't already) from the course web page. The first is a "skeleton" m-file for the signal/no signal detector function. The second contains a speech signal, `mboc`[19], that we will use to test the detector.

   (a) Following the detector description given in Section 1.2.5, complete the function in `sig_nosig.m`. Use a threshold of 0.2 and a block size of 512. Verify the operation of your completed function on the `mboc` signal by comparing its output to that of `sig_nosig_demo.dll`[20].
      - [15] Include the code for your completed version of `sig_nosig.m` in the appendix of your lab report.

   (b) Call `sig_nosig`[21] like this:

```
>> [detection,n] = sig_nosig(mboc);
```

   Then, plot the signal `mboc` (with `plot`) and the output of `sig_nosig` (using `stairs(n,detection,'k:');`) on the same figure.
      - [4] Include this plot in your report.

[18]Make sure the two signals are easily distinguishable by using different line styles. Also, any time that you plot multiple signals on a single set of axes, you *must* use `legend` or some other means to label the signals!

[19]This signal has also been recorded with a sampling frequency of 22,050 Hz.

[20]`sig_nosig_demo.dll` is a completed version of the function in `sig_nosig.m`. This demo function has been *compiled* for use on Matlab version 6 on Windows-based systems ONLY.

[21]If you did not successfully complete this function, you may use the compiled demo function for this part of the assignment.

(c) The threshold given above isn't very good; it causes the detector to miss significant portions of the signal. Change the threshold until all significantly visible portions of the signal are properly marked as "signal" by the detector, but the regions between these portions are marked as "no signal."

- [4] What threshold did you find?
- [6] Include the figure (like the one you generated in Problem 4b) that displays the output of your detector with this new threshold.