Laboratory 1

Signals, Signal Statistics, and Signal 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 the wires from a microphone as it varies with time. Another signal might be the light intensity as it 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 mathematically the same, despite the fact that the signals come from two very different physical sources. In signals and systems engineering, we recognize that the most important aspects of signals are mathematical. Thus, we don't necessarily need to know anything about the physical behavior of voltage or light to deal with these signals.

What purposes do signals serve? Let us highlight a few of the many important ones. First, a signal can embody a sensory experience, as in a sound that we would like to hear or a picture that we would like to see. Second, a signal can convey symbolic information, as in the text of a newspaper. Third, a signal can serve to control some system. For example, in a typical modern automobile, an electronic control signal determines how much gasoline is emitted by the fuel injectors. Last, we mention that a signal can embody an important measurement, for example, the speed of a vehicle or the EKG of some patient.

What is the advantage of having a sound or a picture or text or control information or a measurement embodied in a signal? For one thing, it enables us to transmit it to a remote location or to record it. In many, but not all, cases, these are done electronically, either with analog or digital hardware. For another, the signals we encounter frequently need to be *processed*, which can also be done electronically with analog or digital hardware. For example, a signal may contain unwanted noise that needs to be removed; this is an example of what is called *noise reduction* or *signal recovery*. Alternatively, the desired information or sensory experience may need to be extracted from the signal, as in the case of AM and FM radio signals, which need to be *demodulated* before we can listen to them, or in the case of CT scan signals, which need extensive processing before an X-ray like image can be viewed.

Finally, in many situations, the purpose of signals is to permit decisions to be made. This kind of signal processing is variously called *signal classification*, *signal recognition*, or *signal detection*. As examples, a radar system processes the signal received from its antenna to determine whether or not it contains a reflected pulse, which would indicate the presence of an airplane in the direction to which the antenna is pointed. The bill changer in a vending machine processes the signal produced by its optical sensor to determine if the inserted piece of paper is a valid dollar bill. A speech recognition system processes the signal produced by a microphone to determine the words that are spoken. A speaker recognition system processes the same signal to determine the identity of the person speaking. A heart monitoring system processes an electrical EKG signal to determine if arythmia is occuring. A digital modem processes the received signal to determine what bits are being transmitted. These are a just a few of the situations in which signals must be processed to make decisions.

As one of part of this lab assignment, we will implement and tune a simple signal processing system for detecting whether or not a recorded signal contains a spoken sound. In later lab assignments, we will develop more sophisticated decision making systems – for detecting the presence or absence of radar pulses, for decoding a sequence of key presses from the signal produced by a touchtone telephone, and for deciding which of several vowel sounds has been spoken into a microphone.

Throughout this course we will develop tools for analyzing, modifying, processing 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 can I quantitatively determine a signal's "quality"?
- How can I detect the presence of "speech" within a segment of a speech signal?

1.2 Background

1.2.1 Continuous-time and discrete-time signals

In its most elementary form, a *signal* is a time-varying numerical quantity, for example, the time-varying voltage produced by a microphone. Equivalently, a signal is a numerically valued function of time. That is, it is an assignment of a numerical value to each instance of time. As such, it is customary to use ordinary mathematical function notation. For example, if we use s to denote the signal, i.e. the function, then $s(t)$ denotes the *value* of the signal at time instance t. In common usage, the notation $s(t)$ also has an additional interpretation — it may also refer to the entire signal. Usually, the context will make clear which interpretation is intended.

We will deal with many different signals and to keep them separate we will use a variety of symbols to denote them, such as such as r, x, y, z, x'. Occasionally, we will use other symbols to denote time, such as t', s, u. In some situations, the independent parameter t represents something other than "time", such as "distance". This happens, for example, when pictures are considered to be signals.

As illustrated in Figure 1.1, there are two basic kinds of signals. When the time variable t ranges over all real numbers, the signal $s(t)$ is said to be a *continuous-time* signal. When the time variable t ranges only over the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the signal $s(t)$ is said to be a *discrete-time* signal. To distinguish these, from now on we will use a somewhat different notation for discrete-time signals. Specifically, we will use one of the letters i, j, k, l, m, n to denote the time variable, and we will enclose the time-variable in square brackets, rather than parentheses, as in $s[n]$. Thus, for example, $s[17]$ denotes the value of the discrete-time signal $s[n]$ at time $n = 17$. Note that for discrete-time signals, the time argument has no "units". For example, $s[17]$ simply indicates the 17th sample of the signal. When the independent parameter t or n represents something other than time, for example distance, then

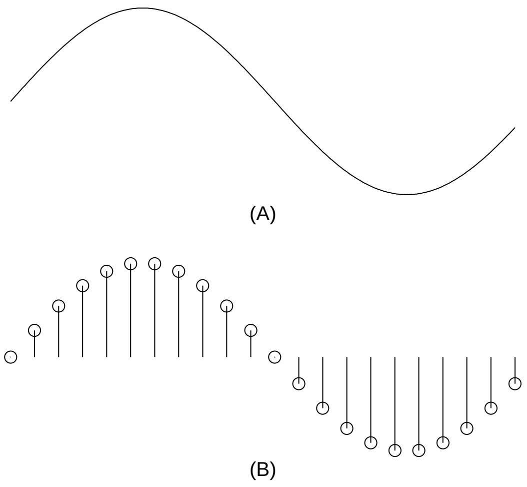

Figure 1.1: (A) A continuous-time signal. (B) A discrete-time signal.

the signal can be said to be *continuous-space* or *discrete-space*, respectively, or more generally, *continuous-parameter* or *discrete-parameter*.

It is important to reemphasize the inherent ambiguity in the notation $s(t)$ and $s[n]$. Sometimes $s(t)$ refers to the value of the signal at the specific time t. At other times, $s(t)$ refers to the entire signal. Usually, the intended meaning will be clear from context. The same two potential interpretations apply to $s[n]$.

1.2.2 Describing Signals

Some continuous-time signals can be described with formulas, such as $s(t) = \sin(t)$ or

$$s(t) = \begin{cases} 0 & t < 0 \\ \cos(t) & t \geq 0. \end{cases} \tag{1.1}$$

For other signals, there are no such formulas. Rather they might simply be measured and recorded, as with an analog tape recorder. Similarly, some discrete-time signals can be described with formulas, such as $s[n] = \sin(n)$ or

$$s[n] = \begin{cases} 0 & n < 0 \\ \cos[n] & n \geq 0 \end{cases} , \tag{1.2}$$

and some are described simply by recording their values for all values of n.

Often, a discrete-time signal is obtained by *sampling* a continuous-time signal. That is, if T_s is a small time increment, then the discrete-time signal $s[n]$ obtained by sampling $s(t)$ with *sampling interval* or *sampling period* T_s is defined by

$$s[n] = s(nT_s) \text{ , } -\infty < n < \infty \tag{1.3}$$

For example, if $s(t) = \sin(t)$ and $T_s = 3$, then the discrete-time signal obtained by sampling with sampling interval T_s is $s[n] = \sin(3n)$. The reciprocal of T_s is called the *sampling rate* or *sampling frequency* and denoted $f_s = 1/T_s$. Its units are samples per second. The discrete-time signal in Figure 1.1 was obtained by sampling the continuous-time signal shown above it.

In the above example, we have allowed the time parameter to be negative as well as positive, which begs the question of how to interpret negative time. Time 0 is generally taken to be some convenient reference time, and negative times simply refer to times prior to this reference time.

Nowadays, signals are increasingly processed by digital machines such as computers and DSP chips. These are inherently discrete-time machines. They can record and work with a signal in just two ways: as a formula or as a sequence of samples. The former applies to continuous-time and discrete-time signals. For example, a computer can easily compute the value of the continuous-time signal $s(t) = \sin(t)$ at time $t = \sqrt{2}$ or the value of the discrete-time signal $s[n] = \cos(n)$ at time $n = 17$. However, the latter works only with discrete-time signals. Thus, whenever a digital machine works with a continuous-time signal, it must either use a formula or it must work with its samples. That is, it must work with the discrete-time signal that results from sampling the continuous-time signal. This admonition applies to us, because in this and future lab assignments, many of the signals in which we are interested are continuous-time, yet we will process them exclusively with digital machines, i.e. ordinary computers.

Except in certain ideal cases, which never apply perfectly in real-world situations, sampling a continuous-time signal entails a "loss". That is, the samples only partially "capture" the signal. Alternatively, they constitute an approximate representation of the original continuous-time signal. However, as the sampling interval decreases (equivalently, the sampling rate increases), the loss inherent in the sampled signal decreases. Thus in practical situations, when the sampling interval is chosen suitably small, one can reliably work with a continuous-time signal by working with its samples, i.e. with the discrete-time signal obtained by sampling at a sufficiently high rate. This will be the approach we will take in this and future lab assignments, when working with continuous-time signals that cannot be described with formulas.

When digital machines are used to process signals, in addition to sampling, one must also *quantize*, or *round*, the sampled signal values to the limited precision with which numbers are represented in the machine, e.g. to 32-bit floating point. This engenders another "loss" in the signal representation. Fortunately, for the computers we will use in performing our lab experiments, this loss is so small as to be negligible. For example, MATLAB uses 64-bit double-precision floating point representation of numbers. (Lab 5 is an exception; in that lab, we will consider systems that are designed to produce digital signal representations with as few bits as possible.)

1.2.3 Signal support and duration

The *support* of a signal is the smallest time interval that includes all non-zero values of the signal. For example, the continuous-time signal $s(t) = \cos(t), 0 \leq t \leq 3, s(t) = 0$, else has support interval $[0, 3]$. The discrete-time signal $s[n] = \cos(n), 0 \leq n \leq 3$, has support interval $[0, 3] = \{0, 1, 2, 3\}$. The *duration* of a signal is simply the length of its support interval. In the previous examples, the duration of $s(t)$ is 3, and the duration of $s[n]$ is 4. Note that the support and duration of a signal can be either finite or infinite.

1.2.4 Periodicity

Periodicity is a property of many naturally occurring or man-made signals. A continuous-time signal $s(t)$ is said to be periodic with period T, where T is some positive real number, if

$$s(t + T) = s(t) \text{ , for all } t \tag{1.4}$$

If $s(t)$ is *periodic with period* T, then it is also periodic with period $2T$, $3T$, ... The *fundamental period* T_o of $s(t)$ is the smallest T such that $s(t)$ is periodic with period T.

Similarly, a discrete-time signal $s[n]$ is said to be periodic with period N, where N is some positive integer, if

$$s[n + N] = s[n] \text{ , for all } n \tag{1.5}$$

If $s[n]$ is periodic with period N, then it is also periodic with period $2N$, $3N$, ... The fundamental period N_o is the smallest N such that $s[n]$ is periodic with period N.

1.2.5 Signals in MATLAB

While signals can be represented by formulas or by recorded signal values, when working in MATLAB, we generally use the latter. That is, we represent a signal as a *vector* (i.e., a one-dimensional *array*) of numbers.

Discrete-time signals

We begin with an example. Suppose we want to represent the following discrete-time signal as an array in MATLAB:

$$s[n] = \begin{cases} n^2 & 5 \le n \le 15 \\ 0 & \text{else} \end{cases} \tag{1.6}$$

In MATLAB, we do this by creating two vector: a *support vector* and a *value* or *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
```

This causes `n` to be the array of 11 numbers $5, 6, \ldots, 15$. Next, the signal vector can be created with the command

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

which causes `s` to be the array of 11 numbers $25, 36, \ldots, 225$.

Note that as in the above example, we usually only specify the signal within the range of times that it is nonzero. That is, we usually do not include zero values outside the support interval in the signal vector.

It is often quite instructive to plot signals. To plot the discrete-time signal `s[n]`, 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 important to note that in MATLAB, when `i` is an integer then `s(i)` is not necessarily the signal value at time `i`. Rather it is the signal at time `n(i)`. Thus, `stem(n,s)` and `stem(s)` result in similar plots with different labelings of the time axis. Occasionally, it will happen that `n(i) = i`, in which case `s(i) = s(n(i))` and `stem(n,s)` and `stem(s)` result in identical plots with identical time axis labels.

Continuous-time signals

We begin with an example. Suppose we wish to represent the following continuous-time signal as an array in MATLAB:

$$s(t) = \begin{cases} t^2 & 5 \le t \le 15 \\ 0 & \text{else} \end{cases} \tag{1.7}$$

We first choose a sampling interval `Ts` with a command such as

```
>> Ts = 1/20
```

We then create a support vector with the command

```
>> t = 5:Ts:15
```

Finally, we create a signal vector with the command

```
>> s = t.^2
```

What have we done? To represent $s(t)$, we have created a support vector `t` that contains the sample times $5, 5 + 1/20, 5 + 2/20, 5 + 3/20, \ldots, 15$, and we have created a signal vector `s` that contains the samples $s(5), s(5 + 1/20), s(5 + 2/20), s(5 + 3/20), \ldots, s(15)$. That is, for $n = 1, \ldots, 301$, `s(n)` contains the signal value at time `t(n) = 5 + (n-1)/20`.

Note that when representing a continuous-time signal as an array, it is usually important to choose the sampling interval `Ts` small enough that the signal changes little over any time interval of `Ts` seconds.

As with discrete-time signals, it is frequently instructive to plot a continuous-time signal. This is done with the command

```
 >> plot(t,s)
```

which plots the points (`t(1),s(1)`), (`t(2),s(2)`), . . ., and connects them with straight lines. Connecting these points in this manner produces a plot that approximates the original continuous-time signal $s(t)$, which takes values at all times (not just integers) and which usually does not change significantly between samples (assuming `Ts` is chosen to be small enough). Note that `plot(s)` produces a similar plot, but the horizontal axis is labeled with sample "indices" (i.e., the number of the corresponding sample) rather than sample times. When working with continuous-time signals, it is important that you always use `plot(t,s)` rather than `plot(s)`. It also important that your plot indicates what the axes represent, which can be done using the `xlabel` and `ylabel` commands.

1.2.6 Signal Statistics

When dealing with a signal, it is often useful to obtain a rough sense of the range of values it takes and of the average size of its values. We do this by computing one or more *signal statistics*.

The following lists a number of common signal statistics. It gives the defining formula for each for both continuous-time and discrete-time signals. Also included is MATLAB code for calculating the statistic[1] for a discrete-time signal. If we wish to compute a statistic for a continuous-time signal when we only have a sampled representation, we can use the discrete-time statistic to approximate the continuous statistic. The formulas needed for this approximation are included here with the label "sampled." (In most cases, this approximation becomes better as the sampling interval T_s decreases.) For completeness, signal support and duration are also defined below.

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.

Continuous-time:	$t_1 \le t \le t_2$	(1.8)
Discrete-time:	$n_1 \le n \le n_2$	(1.9)
MATLAB:	`n = n1:n2` for a signal `s`.	(1.10)

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

Continuous-time:	$t_2 - t_1$	(1.11)
Discrete-time:	$n_2 - n_1 + 1$	(1.12)
MATLAB:	Assumed `length(s)` for a signal `s`.	(1.13)
Sampled:	$(t_2 - t_1) = (n_2 - n_1 + 1)T_s$	(1.14)

[1]This code assumes that the signal vector `s` is defined only over the range of times for which we wish to compute the statistic. More generally, if `n` is the support vector and `n1` and `n2` define a subset of the support vector over which we wish to calculate our statistic, we can compute the statistic over only this range, `n1:n2`, by replacing the signal `s` with the shorter signal `s((n1:n2)-n(1)+1)`.

3. **Periodicity**. Periodicity was described in section 1.2.4. The key formulas are included here.

$$\text{Continuous-time:} \quad s(t) = s(t+T) \tag{1.15}$$

$$\text{Discrete-time:} \quad s[n] = s[n+N] \tag{1.16}$$

$$\text{Sampled:} \quad T \approx NT_s \tag{1.17}$$

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.

$$\text{MATLAB:} \quad Maximum(\mathtt{s}) = \mathtt{max(s)} \tag{1.18}$$

$$\text{MATLAB:} \quad Minimum(\mathtt{s}) = \mathtt{min(s)} \tag{1.19}$$

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

$$\text{Continuous-time:} \quad M(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s(t)dt \tag{1.20}$$

$$\text{Discrete-time:} \quad M(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s[n] \tag{1.21}$$

$$\text{MATLAB:} \quad M(\mathtt{s}) = \mathtt{mean(s)} \tag{1.22}$$

$$\text{Sampled:} \quad M(s(t)) \approx M(s[n]) \tag{1.23}$$

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.

$$\text{Continuous-time:} \quad MS(s(t)) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt \tag{1.24}$$

$$\text{Discrete-time:} \quad MS(s[n]) = \frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s^2[n] \tag{1.25}$$

$$\text{MATLAB:} \quad MS(\mathtt{s}) = \mathtt{mean(s.\hat{}2)} \tag{1.26}$$

$$\text{Sampled:} \quad MS(s(t)) \approx MS(s[n]) \tag{1.27}$$

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.

$$\text{Continuous-time:} \quad RMS(s(t)) = \sqrt{\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s^2(t)dt} \tag{1.28}$$

$$\text{Discrete-time:} \quad RMS(s[n]) = \sqrt{\frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} s^2[n]} \tag{1.29}$$

$$\text{MATLAB:} \quad RMS(\mathtt{s}) = \mathtt{sqrt(mean(s.\hat{}2))} \tag{1.30}$$

$$\text{Sampled:} \quad RMS(s(t)) \approx RMS(s[n]) \tag{1.31}$$

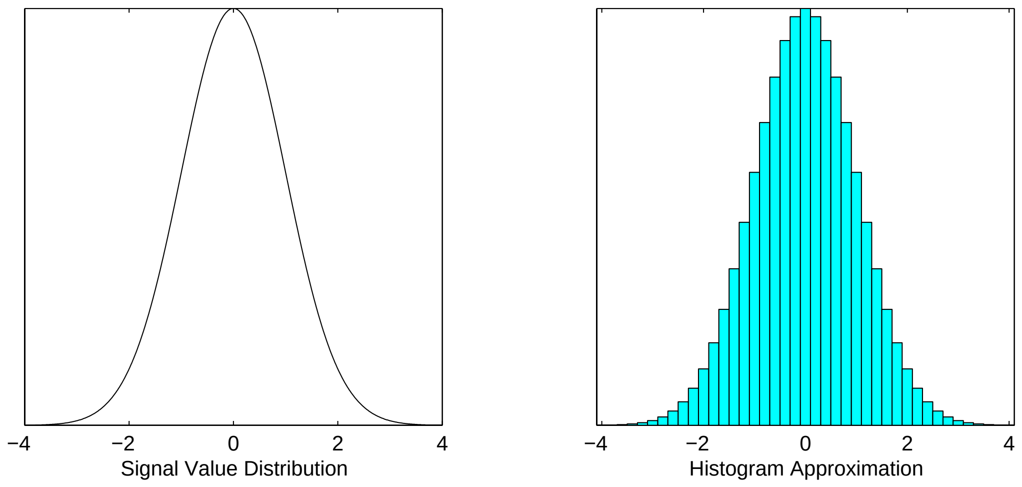

Figure 1.2: Signal value distribution and a discrete histogram approximation

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.

$$\text{Continuous-time:} \quad E(s(t)) = \int_{t_1}^{t_2} s^2(t)dt \tag{1.32}$$

$$\text{Discrete-time:} \quad E(s[n]) = \sum_{n=n_1}^{n_2} s^2[n] \tag{1.33}$$

$$\text{MATLAB:} \quad E(\texttt{s}) = \texttt{sum(s.\^{}2)} \tag{1.34}$$

$$\text{Sampled:} \quad E(s(t)) \approx E(s[n])T_s \tag{1.35}$$

9. **Signal Value Distribution**. The *signal value distribution* is a plot indicating the relative frequency of occurrence of values in a signal. There is no closed-form definition of 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.

MATLAB: `hist(s,num_bins);` (1.36)

1.2.7 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 is that the signal is often modified or *distorted* in the communication process. 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, $r[n]$. We can represent this modification mathematically as the addition of an *error* *signal*, $v[n]$, like this:

$$r[n] = s[n] + v[n]. \tag{1.37}$$

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

$$v[n] = r[n] - s[n]. \tag{1.38}$$

Note that if $s[n]$ and $r[n]$ 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 $v[n]^2$. 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` from `s_mod`; 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.

1.2.8 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 are 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. As another example, some digital transmission systems send bits using what is known as *on/off keying*. They send an electrical pulse to represent "1" and send nothing at all to represent "0". The receiver for such a system uses a "pulse present" detector.

In this laboratory, we will consider the design of a "signal present" detector to identify spoken segments of a speech signal. Note, however, that the detector we will design can be used for many other applications as well. Figure 1.3 shows a block diagram of such a detector. The detector consists of two blocks. The first block computes one or more statistics that we will use to perform the detection. The second block uses the computed statistic(s) to make the final decision.

Specifying the detector's operation

The first step is to specify 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 speech is present in a signal. However, the signal that we are given

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

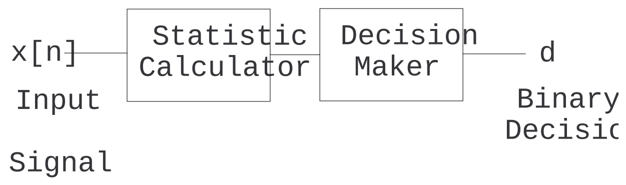

Figure 1.3: An “overview” block diagram for a “signal present” detector.

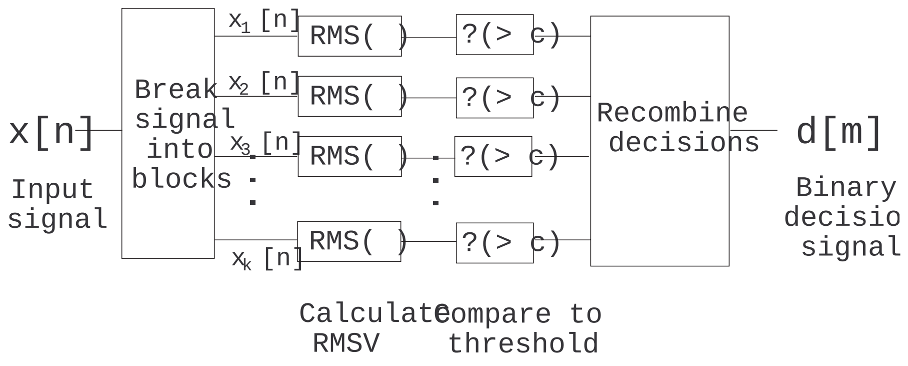

Figure 1.4: A detailed block diagram for the “signal present” detector.

will contain both periods of silence and periods with speech. For some integer N, let us break the signal into *blocks* of N samples, such that the first N samples make up the first block, the next N samples make up the second block, and so on. Then, we will make a “speech present” or “speech not present” decision separately for each block. The output of our system will consist of a signal with a 0 or 1 for each block, where a 1 denotes “speech present” and a 0 denotes “speech not present”. To describe this signal in MATLAB, our system will produce a signal support vector containing containing the index of the first sample of each block and a signal vector containing the 0 or 1 for each block. Choosing the support vector in this way will allow us to plot the decisions on the same figure as the signal itself.

How will we make the decision for each block? Since we can assume a signal that is relatively free of noise, we can simply calculate the energy for each block and compare the result to a threshold. If the statistic exceeds the threshold, we decide that speech is present. Otherwise, we decide that speech is not 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, and this is what we will use. In summary, for each block the detector computes the RMS value R and compares it to a threshold c. The decision is

$$\begin{array}{rl} \text{signal present,} & \text{if } R \geq c \\ \text{signal not present,} & \text{if } R < c \end{array}$$

Note that our detector system will has two design parameters. One is the block size N, and the other is the threshold c. To tune the system, we will need to find reasonable values for these parameters when we make 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.

It is assumed that the input signal is contained in a signal vector `x` whose support vector is simply `[1, 2, ..., length(x)]`.

- Define a variable called `block_size`, representing N.
- Define a variable called `threshold`, representing c.
- Calculate the `number_of_blocks`.
- Determine the support vector `support_output` for the output signal.
- 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. Then the vector `support_output` should contain the numbers `[1, 513, 1025, ...]` and so on. You can generate `support_output` with a single line of code by using the `:` operator[3].
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. 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 `support_output`. 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`[4].
   (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` $\times$ `number_of_blocks` samples of the input signal. `reshape` this shorter signal into a matrix with `block_size` rows and `number_of_blocks` columns[5]. 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.

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

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

[5]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.

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 enables zooming *only* 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.

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

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

  In this case, the loop counter, `index`, is set to 1 on the first execution loop, 2 on the second, and so on. The loop will execute a total of 20 times. We can, of course, use any variable for the loop counter.

  A more general form of the `for` loop is given by

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

  where `index` is the loop counter and `row_vector` is a row vector that contains all the values that will be assigned to `index` on successive iterations of the loop. Thus, the loop will execute `size(row_vector,2)` times[6].

- **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[7]. `new_array` will have dimensions of `new_rows` $\times$ `new_columns`.

- **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 Session

- 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 present" detector

[6]If `row_vector` is actually a matrix (or a column vector), each column of `row_vector` will be assigned to `index` in turn.
[7]The number of elements in `array` can be checked using the command `prod(size(array))`.

1.5 Laboratory Assignment

Note that in this and all following laboratory assignments, the bullets (i.e, •) indicate items that you must include in your laboratory.

1. (A simple signal and its statistics) Use the following MATLAB commands to create a signal:

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

   (a) (Plotting a signal with labels) Use `stem` to plot the signal. Make sure that you include[8]:

   - [2] The figure itself[9].
   - [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[10]. Make sure you clearly indicate which problem the code belongs to.

   (b) (Calculating signal statistics) 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[11].

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

   (c) (Approximating continuous-time statistics in discrete-time) Suppose that `s` is the result of sampling a continuous-time signal with a sampling interval $T_s = 1/100$. Use the discrete-time statistics to estimate the following statistics for the continuous-time signal $s(t) = \sin(4\pi t)$:

   - [2] Signal duration
   - [2] Energy
   - [2] Average power
   - [2] RMS Value

2. (Statistics of real-world signals) 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
```

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

[9] 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. Also, to make your report compact, you should make all figures as small as possible, while being just large enough that the important features are clearly discernable.There are two ways to shrink plots, you can shrink them in your lab report document, or you can shrink the MATLAB window before copying and pasting. Shrinking the MATLAB window is generally preferable because it does not shrink the axis labels. Note, you may need to specify the appropriate copy option, so that what is in fact copied is the shrunk rather than original version of the plot

[10] 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.

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

This file contains two signals which will be loaded into your workspace. You will use the signal `clarinet`[12] in this problem and in Problem 3. The other signal, `mboc`, will be used in Problem 4.

(a) (Plot the real-world signal) 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) Zoom in on the signal so that you can see four or five "periods[13] ."

- [4] Include the zoomed-in figure (with axis labels, figure number, caption, and MATLAB code) in your report.

(c) (Find the signal's period) 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) (Approximate the SVD) 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 statistics) Calculate the following (discrete-time) statistics over the length of the signal:

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

3. (Looking at and measuring signal distortion) 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) (Examine the effects of the systems) Use `plot`[14] 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) (Describe the effects of the systems) 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) (Measure the distortion) Calculate the RMS error introduced by each system.

[12]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)`.

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

[14]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!

- [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. (Developing an energy detector) In this problem, we will develop a detector that identifies segments of a speech signal in which speech is actually present. 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`[15], that we will use to test the detector.

   (a) (Write the detector function) Following the detector description given in Section 1.2.8, 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`[16].

   - [15] Include the code for your completed version of `sig_nosig.m` in the appendix of your lab report.

   (b) (Plot the results of your function) Call `sig_nosig`[17] like this:

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

   Then, plot the output of `sig_nosig` (using "`stairs(n,detection,'k:');`") and the signal `mboc` (with `plot`) on the same figure.

   - [4] Include this plot in your report.

   (c) (Adjusting the threshold) 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 "speech" by the detector, but the regions between these portions are marked as "no speech." (Note that you *cannot* use the compiled demo function for this part of the assignment.)

   - [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.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

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

[16]`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.

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