

INTRODUCTION

TO

DIGITAL SIGNAL

PROCESSING

INTRODUCTION

Purpose/objective of the course: To provide sufficient background on Digital Signal Processing (DSP) concepts so students can understand and use commercial software for DSP and use DSP for research projects. Detailed manual design procedures will be covered only to the extent necessary to use the available software.

1.0 DIGITAL SIGNAL PROCESSING

What do we mean by “Digital Signal Processing”?

DIGITAL - means *discrete* in nature – i.e. the signal levels are chosen from a finite set of levels, as opposed to *continuous or analog* signals, which can have an infinite number of levels. In practice, digital nearly always means *binary*, that is, two-level signals – our standard TTL or CMOS levels as used in computers and other digital systems. Note that signals used in DSP systems may be developed from analog signals by sampling and analog-to-digital conversion (discussed at some length in a later section) or may be available as digital signals initially, as from another digital system.

DSP signals are also discrete in time, i.e. they represent samples taken at specific instants in time. Thus, we use notation like $x[n]$ or $y[n]$ to represent these signals, where n is an integer that represents, effectively, the sample number.

SIGNAL – means some *physical quantity* whose variations convey information. A signal can be mechanical, hydraulic, pneumatic, optical (visible, UV or infra-red light), temperature, etc. However, we generally deal with electrical signals, either because they were initially developed as electrical, or because they have been converted to electrical.

PROCESSING – refers to the applications we want to implement or operations we want to perform on the digital signal. The two major, end-result applications for digital signal processing are *digital filters* and the *fast Fourier transform (FFT)*. However, there are innumerable other applications or types of processing, carried out because they are important in themselves or because they are steps in implementing filters or FFTs.

SYSTEM PROPERTIES: To simplify the work and to allow the use of the many standard design techniques, we will make certain assumptions about the properties of the systems we will deal with.

Linearity Linearity dictates that for a single input, the output is proportional to the input, and for two or more inputs, the output must be the sum of the individual responses of the two inputs. Mathematically, this is expressed as:

if $x_1[n]$ produces $y_1[n]$, and $x_2[n]$ produces $y_2[n]$,

then $ax_1[n] + bx_2[n]$ produces $ay_1[n] + by_2[n]$

Linearity is the basis for the concept of superposition and for convolution which we will see shortly.

Time-invariant. We assume the system properties do not vary with time (or at least over the time period we are concerned about.)

Causality. Causality implies that output changes do not occur before input changes. Although this cannot happen in a real situation, non-causal considerations sometimes arise in theoretical derivations. If all the samples for a certain signal have been collected and stored, causality is not an issue, since all samples both before and after a selected value of n are available. In addition, any non-causal signal can be made causal just by delaying all the samples by an appropriate amount.

Most real signals and systems we deal with are causal, but just as an example we will look at one non-causal system. To that end, consider the moving-average system, frequently used to smooth a set of data points. The moving average system computes the average of a certain number of points around a specific point, then replaces the value at that point with the average. Thus, a typical moving average computation might be to take the average of the two points prior to a certain point, plus the point itself, and the two points past the point in question.

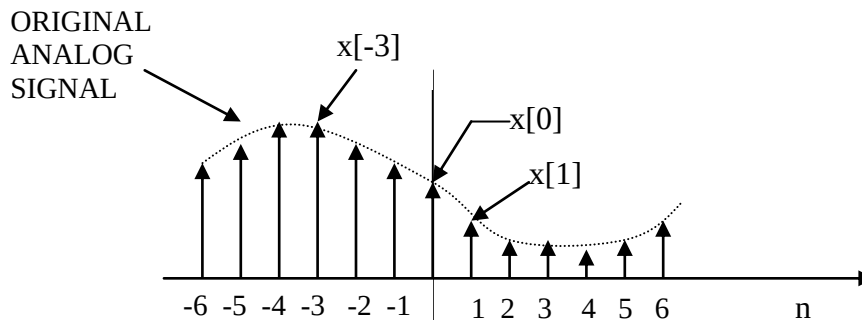
Thus, the computation for point n requires data from the points numbered $n+1$ and $n+2$, which obviously aren't available at time n . However, if the data consists of stored points, the average can easily be computed.

There are other important properties of DSP systems, such as stability (every bounded input produces a bounded output). We will assume these, but not discuss any further.

2.0 SAMPLED SIGNALS AND SEQUENCES

Let us now consider the **nature** of the signals used in digital signal processing. These signals are frequently, but not always, developed by sampling a continuous-time signal. Sometimes they are called discrete-time signals, to reflect the fact that they have meaning only at discrete points in time. Sampled signals are also discrete in amplitude as well as time, since the normal analog to digital conversion process is finite in its resolution, and thus forces the amplitude to be chosen from a finite set of values.

If we call the sampled signal $x[n]$, to indicate that the values of the signal are a function of the sample number or **index** “ n ”, then the individual values of the signal are represented by $x[0]$, $x[1]$, $x[2]$, etc., as shown by the following:



Note that the index n can be negative as well as positive, to indicate that the signal sample was taken before the point designated $n=0$. This is not really a complication, since the definition of $n=0$ is frequently arbitrary.

Since the digital signals represent samples of continuous-time signals, taken at discrete points in time, they are actually a **set of numbers** representing the **values** of the continuous-time signal at the instants in time at which it was sampled. For example, typical sequences might be:

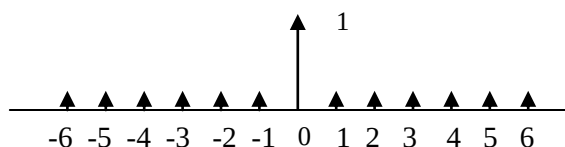
1,2,3,4,5 or 2, -4, -6, 8 or 0.330, -1.25, 5.69, 3.45, -5.77 etc.

Thus we have to get accustomed to the idea that the digital signals we deal with are just a string or **sequence** of numbers. In fact, they are usually referred to as sequences instead of signals. This string or sequence of numbers is handled just like any other set of numeric data: it can be treated as a vector, stored in an array format, manipulated by a spread sheet, etc. A sequence is frequently called a vector, although it has no physical meaning like a force vector or electromagnetic field vector does.

It is also quite possible that the sequence of numbers was **not** derived by sampling a continuous-time signal. They may represent data collected from measurements made in some physical or natural system, such as traffic flow, growth rate of trees, ocean water temperature, etc., which actually **are** sampled quantities, but may be the result of intermediate calculations.

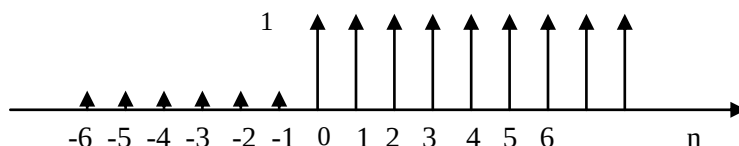
There are several special sequences used in DSP that are not derived from the sampling of a real-world signal. Rather, they are **defined** instead of sampled, and thus have somewhat different notation.

The first is the **unit impulse**. It is designated as $\delta[n]$, and defined as 1 for $n=0$, and 0 elsewhere. Graphically, it is



The unit impulse is particularly important in DSP because DSP signals, being just sequences of numbers, can be represented by a series of impulses, each weighted to represent the actual value of the sequence for each value of n . Thus, DSP system analysis frequently amounts to analysis of the system response to a sequence of impulses individually, and subsequent superposition or addition of the individual responses to find the complete response to the input sequence.

Another useful defined sequence is the **unit step**. It is denoted by $u[n]$, and consists of a sequence of impulses with value of one for $n \geq 0$, and 0 otherwise.



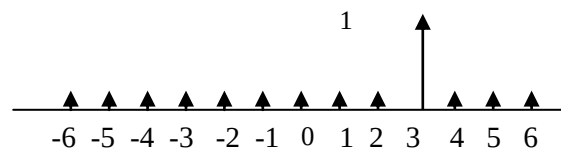
The unit step $u[n]$

There are several other functions that can be defined, including the exponential sequence and the sinusoidal sequence. The names are more or less self-explanatory.

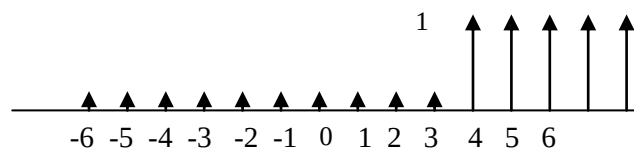
Note that there is one major difference in the notation for **signals**, such as $\mathbf{x[n]}$ or $\mathbf{y[n]}$, and **defined functions** such as the unit impulse $\delta[\mathbf{n}]$ or unit step $\mathbf{u[n]}$. For a signal such as $\mathbf{x[n]}$, there is, in general, a numerical value associated with every value of $\mathbf{n}$. Thus, $x[0]$ has a distinct value, as does $x[5]$, $x[10]$ and $x[-4]$, etc.

However, for a defined function such as $\mathbf{u[n]}$, its values are determined by its definition, i.e. $\delta[\mathbf{n}] = 1$ for $n=0$, and 0 elsewhere; $\mathbf{u[n]} = 1$ for $n \geq 0$, and 0 elsewhere. It would not be correct to speak of $\delta[5]$ or $u[-4]$ or any other value of n . The values are determined by the definition, not by defining individual samples.

We **can**, however, talk about shifted functions. For example, $\delta[n-3] = 1$ for $n = 3$, and 0 elsewhere, and $u[n-4]$ is a unit step that starts at $n = 4$. Some illustrations are:

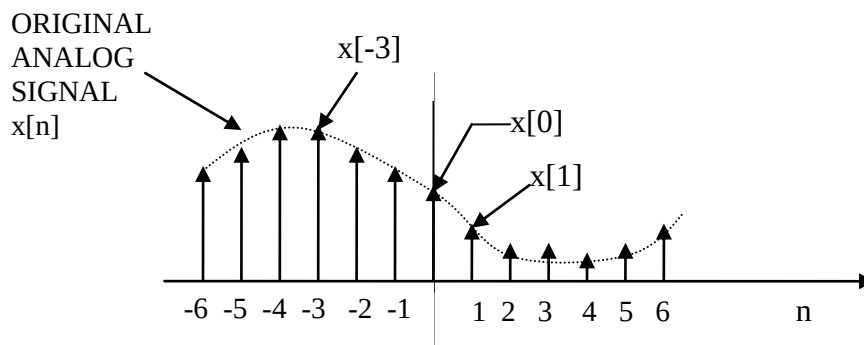


The delayed unit impulse, $\delta[n-3]$

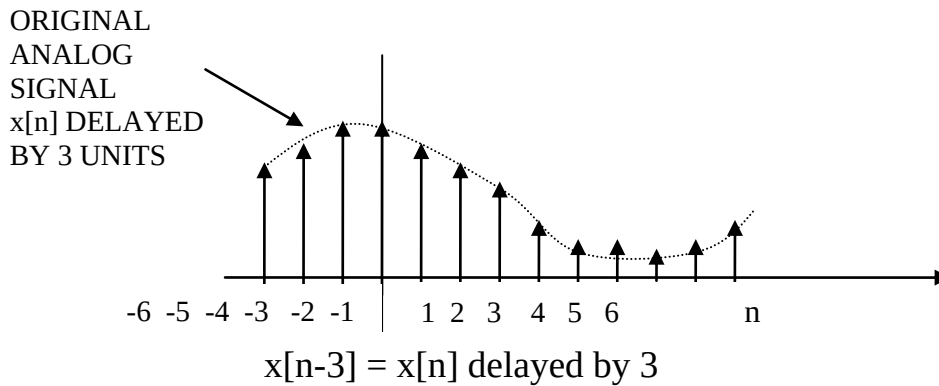


The delayed unit step $u[n-4]$

A similar shifting concept applies to signals. The notation $x[n-3]$ represents signal $x[n]$ delayed by 3 units, and $y[n+4]$ represents the signal $y[n]$ advanced by 4 units. Consider, for example, the signal we used at the beginning of this section:



If this signal is just shifted (delayed) by 3 units, we have $x[n-3]$:



Relationships between the unit step and unit impulse. The unit step can be written in terms of the unit impulse. Since the step is really represented by an infinite sequence of impulses starting at 0, the step can be written as:

$$u[n] = \sum_{k=0}^{\infty} \delta[n-k] \text{ i.e. a sequence of impulses from 0 to infinity}$$

Similarly, the unit impulse can be expressed as

$$\delta[n] = u[n] - u[n-1]$$

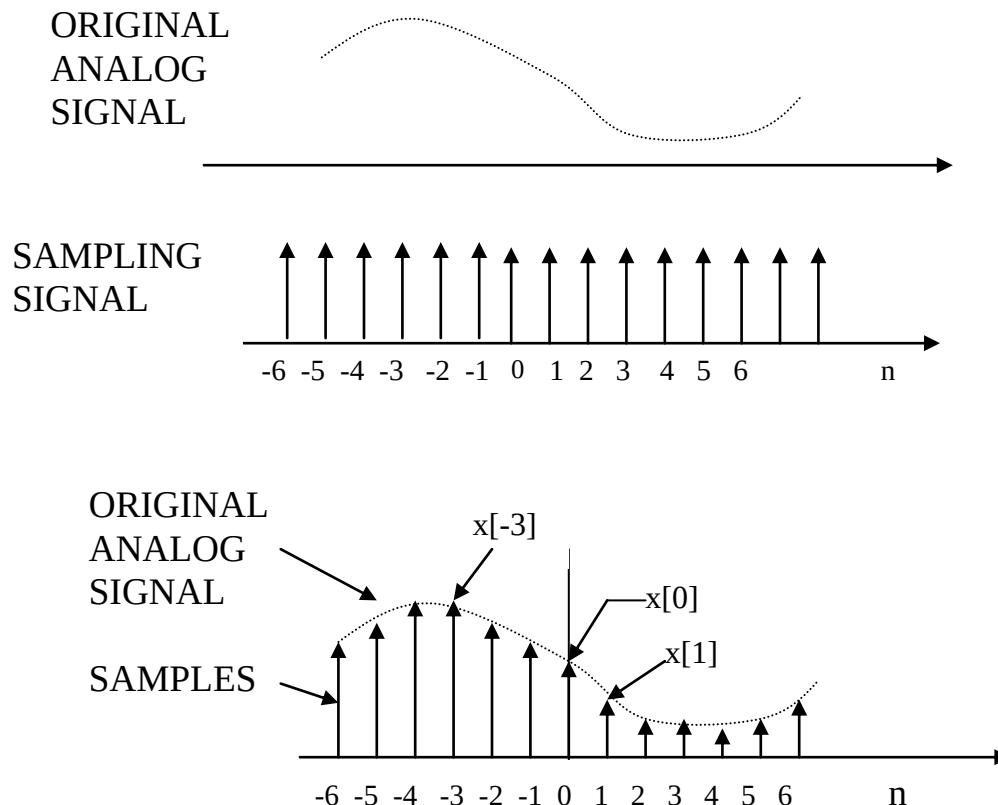
i.e. a unit step less a unit step delayed by one. This form of equation is known as a first-order difference equation. It is analogous to the differential equation used in the analysis of continuous-time systems. As we will see shortly, the difference equation is key in the modeling and analysis of DSP systems.

3.0 SAMPLING AND ANALOG-TO-DIGITAL CONVERSION

3.1 SAMPLING

To carry out any digital signal processing, most real-world signals, which are predominately analog, must be converted to digital form. Although the process is conceptually straightforward – just feed the analog signal to an analog-to-digital converter – there are a number of issues connected with that process that must be considered.

First, we note that for analysis purposes it is convenient to model the conversion process as the multiplication of the analog signal by an impulse train. Since each impulse has a value of unity, the individual samples generated by this process will have the value of the analog signal at the instant that the impulse occurred. The diagram we used initially to illustrate the derivation of a digital signal from an analog signal is also useful here:



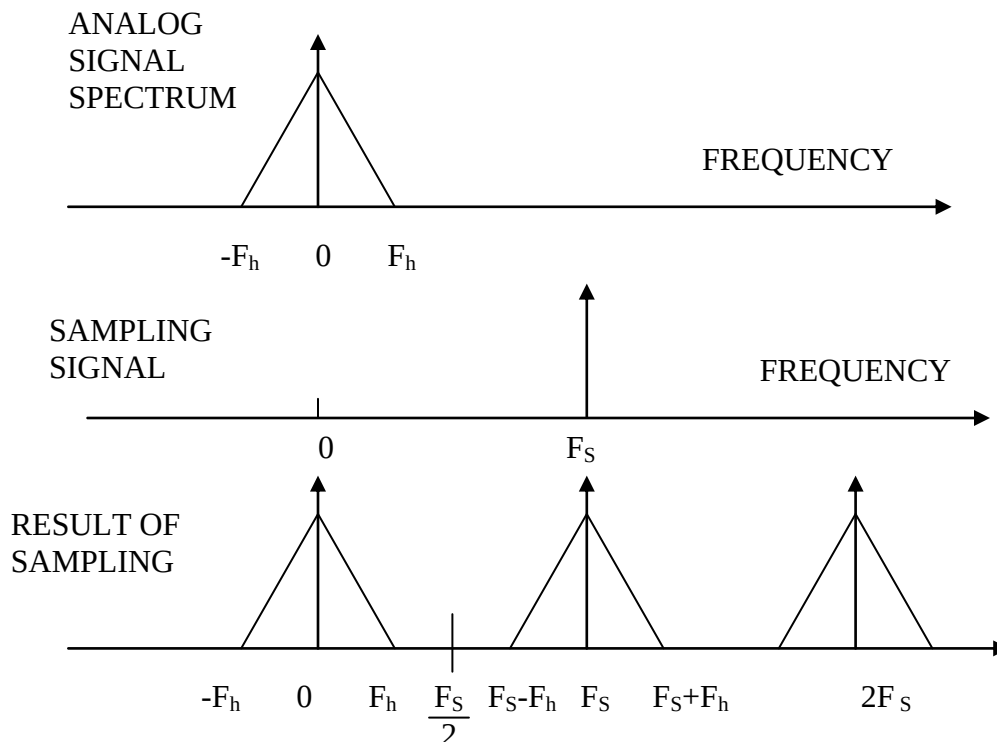
We can see from the diagram that the multiplication of the analog signal by a train of unit-valued impulses will produce the sequence of weighted impulses shown.

The fact that the two signals are effectively multiplied allows us to use some traditional signal processing concepts to describe the results. Recall from previous theory that multiplying two single-frequency signals together produces a result containing the sums and differences of the two original frequencies. This is confirmed by a trig identity which states that:

$$2\sin(a)\sin(b) = \sin(a+b) + \sin(a-b)$$

Since the impulse function contains its fundamental frequency, as well as all of its harmonics, the sampling operation will produce the spectrum of the analog signal centered about all the harmonics of the frequency of the impulse. The frequency of the impulse function is called the **sampling frequency**.

The result of sampling an analog signal having a given spectrum, in which the highest frequency is f_h , is shown below:

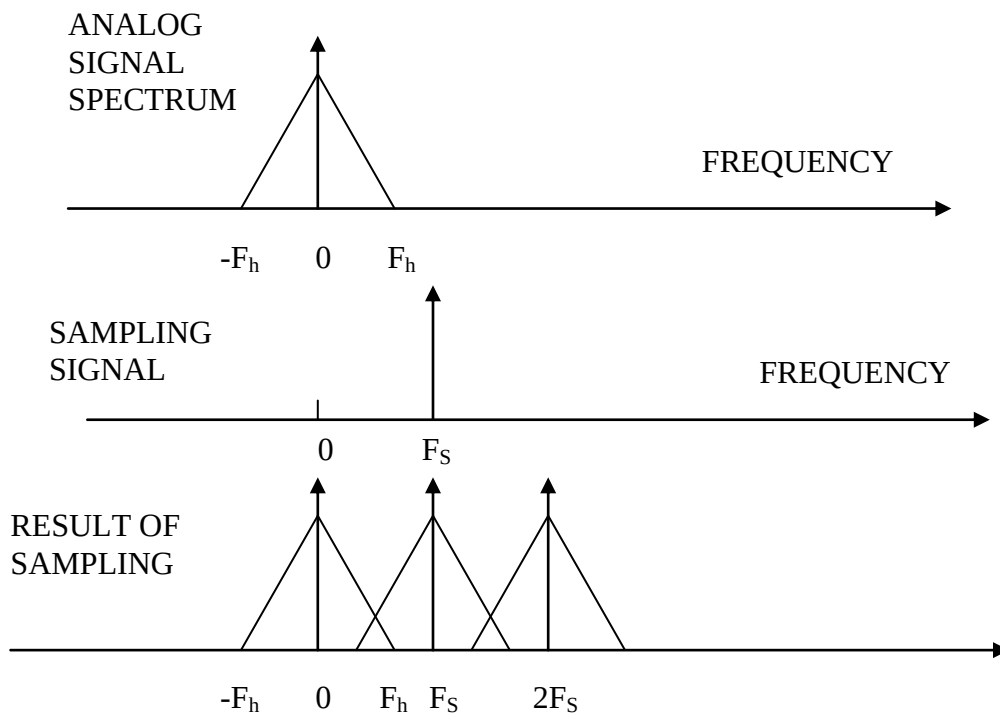


In effect, copies of the original spectrum appear at all multiples of the sampling frequency. With a sharp filter, it would be possible to recover all the original information just by separating out the basic spectrum from all the copies.

Note that in this case, the sampling frequency F_s is more than twice the highest frequency in the signal, F_h . This is obvious from the fact that $F_s - F_h > F_h$ so $F_s > 2F_h$. When this is the case, the original signal can be filtered out from the sampled spectrum. So, although the samples effectively contain all the copies of the original spectrum, that spectrum can be recovered without distortion.

We use the criterion that the original spectrum can be recovered by a sharp filter as a justification for a certain sampling rate. This does not imply that we really plan to recover the original spectrum. If that's all we did, we haven't really accomplished anything. The real reason for using such a criterion is that if it is satisfied, we know that the signal is correctly represented by the sampled signal, and that it isn't distorted by the *alias*, which we discuss next.

Let us now consider a slightly different situation – in which the sampling frequency is less than twice the highest frequency in the signal. The spectrum would then look like:



It is obvious from the figure that some energy from the first copy of the spectrum is injected into the original spectrum, so that even sharp filtering cannot remove that energy from the original spectrum while keeping all the information in the original

signal. This phenomenon is called **aliasing**, and the unwanted portion of the first copy of the spectrum that is aliased into the spectrum of the original signal is referred to as the **alias** signal.

The presence of the aliasing problem means that the sampling frequency must be at least twice that of the highest frequency in the signal being sampled. This can be accomplished in two ways:

1. The original signal can be low-pass filtered to reduce the bandwidth to less than half of the sampling frequency. There are obviously limits on how far the bandwidth can be reduced and still retain the desired signal characteristics. Sometimes the filter can be a very simple RC filter, as for instance, when the signal is greatly oversampled – say at a rate 5 to 10 times F_h . In other situations, where oversampling is either impossible or not convenient, more complex filters will be required. In such cases, attention must be paid to the phase characteristic as well as the amplitude characteristics of the filter. Non-uniform phase shift, which is characteristic of many active filters, can be especially troublesome for data signals.
2. The sampling frequency can be increased to more than twice the highest signal frequency. There are obvious limits on this approach also, as the circuitry involved must be able to handle the sampling frequency involved.

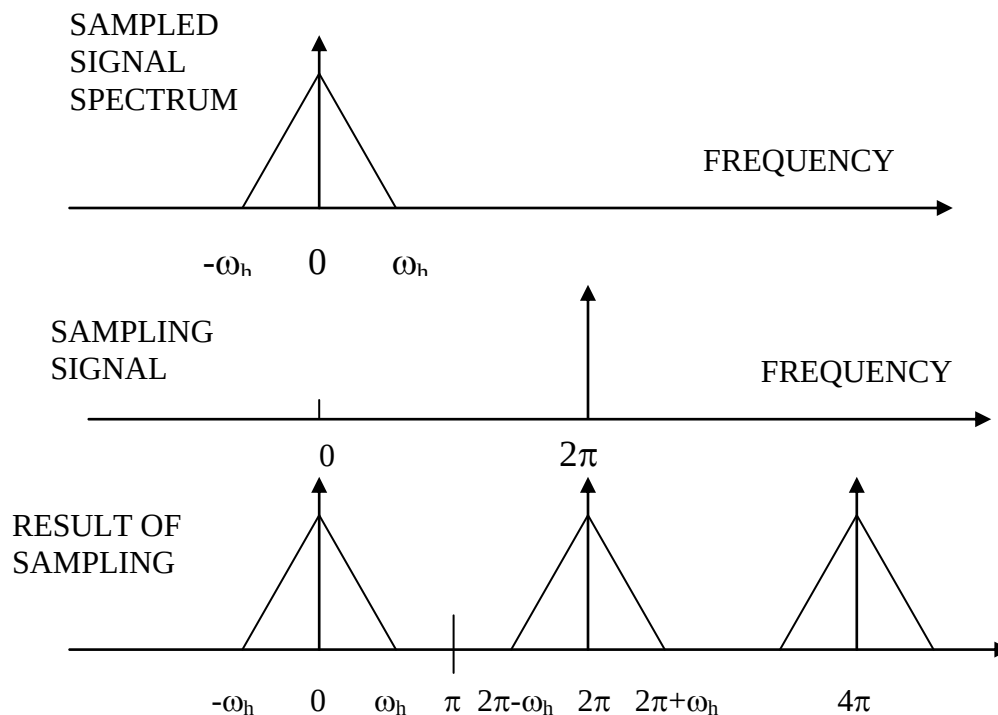
In practice, a sampling rate of 2.5 to 3 times the highest frequency in the signal is actually used, because any filters used to recover the original signal are not ideal, and allowance must be made for that characteristic.

Nyquist Theorem The requirement that the sampling rate exceed twice F_h , the highest frequency in the signal, is known as the **Nyquist Theorem** or the **Nyquist criterion**. The frequency F_h is called the **Nyquist frequency** by some authors, and the frequency $2F_h$ that must be exceeded by the sampling rate is called the **Nyquist rate**. Others, including Matlab, define the Nyquist Frequency as half the sampling rate.

A note of caution: When considering the bandwidth of the signal, we must also consider the frequency characteristics of any noise in the spectrum. Although the signal may have already been bandwidth limited by some other mechanism, e.g. by a phone system, there may be noise in the signal outside the signal bandwidth, and low-pass filtering may be required to keep the noise from aliasing.

3.2 FREQUENCY NORMALIZATION OF SAMPLED SIGNALS

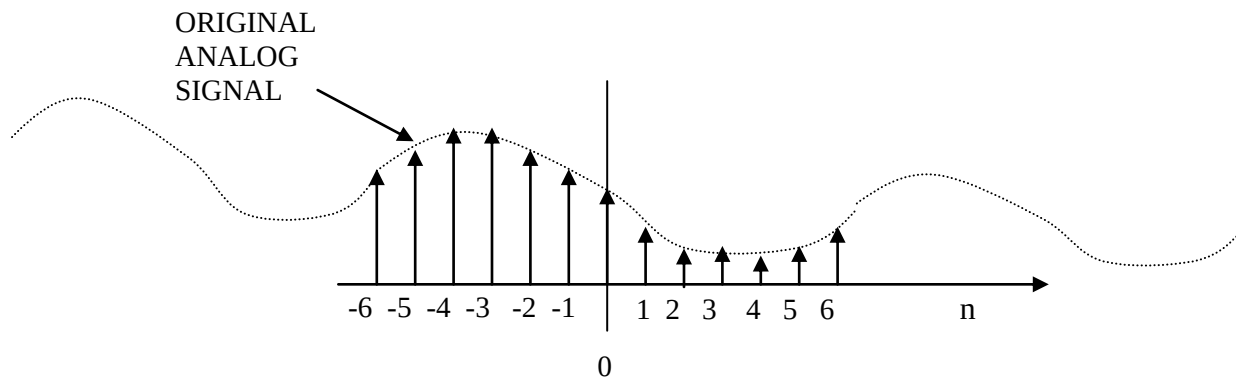
When discussing sampled signals, it is customary to first normalize the frequencies involved. One approach is to normalize to the sampling frequency F_s and use radian frequency instead of conventional cycles/second frequency. With these conventions, the sampling frequency becomes 1 in cycles/sec, or 2π in radian frequency. The symbol ω is used for frequency of the sampled variable, so that $\omega = 2\pi$ is the sampling frequency, and other frequency points are expressed in terms of that. Thus, the Nyquist frequency is π , and the spectrum of the signal will also be normalized to the sampling frequency. The spectrum of the sampled signal shown earlier would then be:



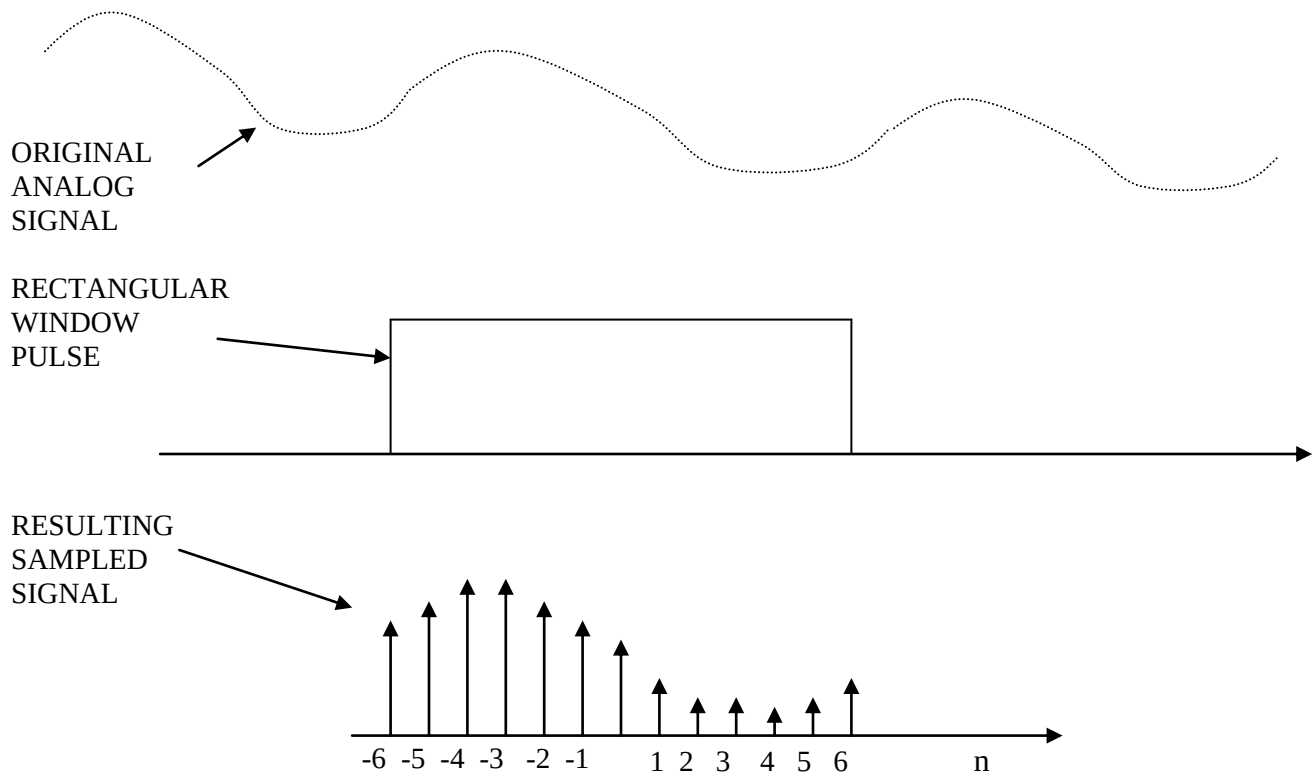
This type of normalization is used in the theoretical derivation of many of the DSP concepts. Commercial DSP software, however, frequently uses different frequency notation. One point to note is that the actual sampling frequency is involved in some of the DSP operations, so that value will have to be known. Some packages use actual frequency, so no conversion is necessary. Others, such as Matlab which we will use, normalize to half the sampling frequency, so that the Nyquist frequency becomes 1, and the sampling frequency becomes 2.

3.3 THE USE OF WINDOWS IN SAMPLING

When a signal is sampled, the sampling operation generally starts abruptly - that is at some arbitrary point on the sampled waveform. Consider again the example we have been using:



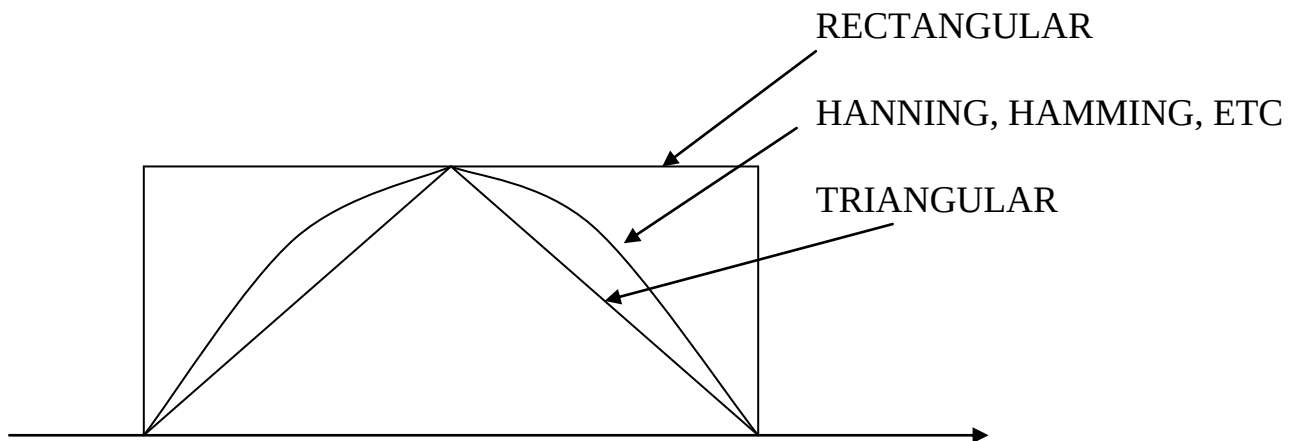
The portion of the signal that is sampled is generally a part of a much longer signal. The sampling starts abruptly at some point and stops at some other point. The abrupt start and stop introduces frequency components into the sampled signal that did not exist in the original. The abrupt start and stop of sampling has the effect of multiplying the original analog signal with a rectangular pulse or “window” before sampling:



To reduce the effects of the abrupt transitions, it is customary to modify the shape of the original analog signal so that it approaches zero smoothly at the ends of the sample period, instead of abruptly. This is done by multiplying by a **window function** which is

selected to reduce the end effects while minimizing the resultant effect on the spectrum of the signal being sampled. The rectangular pulse is effectively the window used when no specific window is deliberately applied.

There are several standard windows whose characteristics in both the time domain and the frequency domain are well known. The general response of the popular ones is shown by:



It is not our purpose to investigate these in any detail, but just to recognize why and how they are used. As we will see when we consider some actual DSP operations, provisions for incorporating windows are frequently included. The design of digital filters frequently uses windows, since one popular technique is to duplicate the impulse response of an analog IIR filter, and application of a window is necessary to limit the number of terms. In fact, one design technique is sometimes called the “windowing” approach.

3.4 ANALOG-TO-DIGITAL CONVERTERS (ADCS)

There are many different kinds of ADCs. Some are more suitable than others for DSP applications. We will briefly review the characteristics of the most popular ones, since choice of an ADC is an important part of the DSP design process when the signals are initially in analog form. Some of them require a sample and hold circuit to keep the analog signal at a constant level during the conversion process. Others “track” the input, and their output reflects the value of the analog signal during the last step of the conversion process.

The delta/sigma ADC. This type uses a simple one-bit converter to sample the input signal at a very high rate. The one-bit signal is then digitally filtered to produce a signal at a much lower rate, but with higher resolution. For example, the initial one-bit

conversion could be at a rate of several MHz, which is then converted by a digital filter to a rate of tens of kHz, but with resolution typically of 8 to 12 bits, but sometimes as high as 18 to 20 bits. The very high sampling rate means that sampling noise is well above the frequency range of the signal, so can be effectively filtered out by the digital filter.

The dual-slope integrating ADC counts the number of clock pulses required to discharge a capacitor at a rate proportional to the input signal, when the capacitor was initially charged to a level proportional to a reference voltage. Thus, the number of counts represents the value of the input signal. Because the counting operation can be lengthy, and its time is not predictable, this type of ADC is more suitable for hand-held meters and other human interfaces than for DSP.

The charge-balance ADC operates in a similar fashion, except that it accumulates charge on a capacitor as it develops a counter value proportional to the value of the signal. It is also slow and not well suited for DSP. Note that both the dual-slope and the charge-balance ADCs require a conversion time proportional to the actual value (number of counts in the counter) of the input.

The successive-approximation ADC uses a digital-to-analog converter (DAC) in a feedback loop to perform its conversion. It starts its operation by setting the output of the DAC to half scale, i.e. only the most significant bit is a ONE. It then compares the input to the DAC output. It resets the MSB if the input is lower, and keeps it if the input is higher. Then the input is compared with the MSB resulting from the previous step, added to the next most significant bit. Another decision is made, based on the size of the input relative to the DAC output.

A total of N operations are thus necessary for N bits. The conversion time is proportional to the number of bits involved rather than the number of counts. A 12-bit successive approximation converter, for example, gives the same resolution as a 4096 count integrating or charge balance type.

The flash converter is the fastest of all types, as it does its entire conversion in one step, regardless of the number of bits. For N bits, it uses 2^N resistors in a voltage divider and 2^N analog comparators to determine exactly which of the 2^N possible voltage levels are represented by the input. Digital encoding circuits then convert the comparator outputs to binary words. Flash ADCs can perform a complete conversion in 10 ns. or less.

This speed comes at a price, however. The device is substantially more complex, and generally requires much more power because of the many comparators and the extreme

speed required. It is used for the most demanding requirements, such as sampling oscilloscopes and digitization of radio-frequency signals. Because the complexity increases as 2^N , flash converters are generally limited to about 8 bits.

3.5 FINITE PRECISION PROBLEMS

In DSP work, there is always a concern because the precision of the data is limited. ADCs are based on a specific number of bits – anywhere from 7 or 8 for flash to as many as 20 for the delta-sigma. In addition, in many calculations involving two numbers, the results require more bits than the original numbers did. Thus, there is a loss of precision from the rounding that must be done to fit the results into the word size of the system.

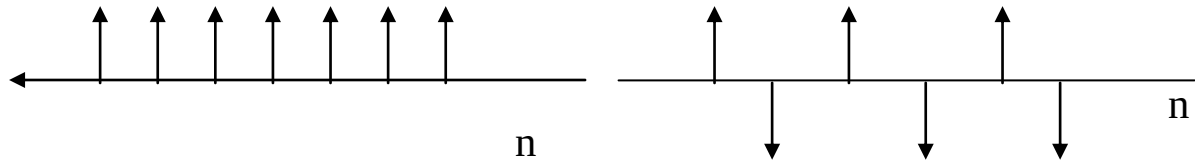
QUANTIZATION NOISE. All of these effects mean that there is nearly always some difference between the number assigned to a sequence value by the limited-precision system, and what its real value would be with an infinite-precision system. The difference is generally uncorrelated from one sequence value to the next, so it is completely random. Thus, it can be treated as **random noise**, and its effect predicted on a statistical basis. In fact, the finite-precision effect can be analyzed in terms of signal-to-noise ratio, just like random noise in any system.

Research and experimentation have shown that the effects of this “noise” depend on the word size (number of bits), the rounding scheme used in multi-stage calculations, and the binary code used (fixed vs. floating point). In general, 16 bits is adequate for most purposes, and decent results can be obtained down to 12 bits or fewer, depending on the applications. There is also the possibility for making some tradeoffs with other parameters. For example, in filter design, sometimes adding stages will allow lower precision in the numbers themselves.

LIMIT CYCLES Another result of the finite precision available in DSP calculations is the production of **limit cycles**. These are constant amplitude output signals which appear at the output of a DSP process when the output is expected to approach zero. They occur because forcing the result of a computation to match one of the available output levels of a limited-resolution system - e.g. one of 256 for an 8-bit system or one of 65536 for a 16-bit system – is inherently a rounding or truncation process.

If the result of an operation is closer to one non-zero level, (say 0000 0001), than it is to zero (0000 0000), it will be rounded to 0000 0001 instead of just 0, and subsequent values may produce the same result, so that the output never reaches zero.

Limit cycles may also alternate in polarity, depending on the calculation being made.
Examples of limit cycles are:



4.0 LINEAR DIFFERENCE EQUATIONS

Most problems that are treated by digital signal processing techniques can be described by **linear difference equations**. These are analogous to the differential equations used in modeling continuous-time systems. These equations simply express the relationship of the current output with the past outputs and the current and past inputs. They are written in terms of the current inputs and outputs, usually called $\mathbf{x[n]}$ and $\mathbf{y[n]}$, and the previous inputs and outputs expressed as delayed inputs and outputs.

We have already seen one example of a linear difference equation: the expression for a unit impulse in terms of two unit step functions: $\delta[n] = u[n] - u[n-1]$. Let us consider another one involving compound interest.

Let us consider a savings account to be a system, with input $\mathbf{x[n]}$ and output $\mathbf{y[n]}$.



Further, let us assume we want to invest a certain sum of money at selected intervals, and that it will draw interest at a rate of $\mathbf{i}$. We let $\mathbf{x[n]}$ be the **input** (amount invested) and $\mathbf{y[n]}$ equal the **output** of the system or the value of the account at any interval $\mathbf{n}$ periods after we start the deposits. We can see that output $\mathbf{y[n]}$ is equal to the current deposit plus the previous value with appropriate interest added. Noting that $\mathbf{y[n-1]}$ is the value of $\mathbf{y[n]}$ at the previous value of $\mathbf{n}$, we can say that

$$\mathbf{y[n] = x[n] + y[n-1] + iy[n-1] = x[n] + (1+i)y[n-1]}$$

That is, $\mathbf{y[n]}$ equals the current deposit plus the previous deposit with interest added

For example, when $\mathbf{n = 0}$, $\mathbf{y[0] = x[0]}$; (just the current deposit)

$$\text{when } \mathbf{n=1}, \quad \mathbf{y[1] = x[1] + (1+i)y[0]}$$

$$\text{when } \mathbf{n=2}, \quad \mathbf{y[2] = x[2] + (1+i)y[1]}$$

As we will see shortly, it is possible to solve the difference equations for the output $\mathbf{y[n]}$ as a function of $\mathbf{x[n]}$ and $\mathbf{i}$ by using the z-transform. If $\mathbf{x[n]}$ is a single deposit, it is

represented as a unit impulse, $\delta[n]$, having a value of 1 at $n=0$ and 0 otherwise. In that case, the use of the z-transform yields

$$y[n] = (1+i)^n x[n]u[n].$$

The $u[n]$ accounts for the fact that there is no output before the input starts and can be dropped for many physical systems. As an example using this formula, if we deposited \$1000 when $n=0$, with no additional deposits, at 5% interest, after 10 periods

$$y[n] = 1000(1+.05)^{10} = \$1628.89.$$

The above formula is not too difficult to work out just using intuition or other techniques. However, if we assumed that an equal deposit is made at $n=0$ and each interval thereafter, then we treat $x[n]$ as a unit step, and the application of z-transforms yields the solution for $y[n]$ as:

$$y[n] = \frac{(1+i)^{n+1} - 1}{i}$$

This result is not nearly as intuitive as the simple formula for a single deposit. (Mathematical tables usually show an exponent of just n , not $n+1$, because they start at $n=1$ instead of $n=0$.) To consider an example of this formula, if we made a deposit of \$1000 for 10 consecutive periods, and received 5% interest, the formula shows that we would accumulate a total of \$12,577.89.

The equation for compound interest is a simple example of a linear difference equation. The more general equation is given by

$$\sum_{k=0}^N a_k y(n-k) = \sum_{r=0}^M b_r x(n-r)$$

As we will see later, this general form is very widely used to represent the behavior of DSP systems.

5.0 z-TRANSFORMS

5.1 DEFINITIONS

We mentioned earlier that the linear difference equations that represent sampled-data systems can be solved by using z transforms. In manipulating equations that represent discrete-time signals, the z-transform plays a role similar to that of the Laplace transform in processing differential equations for continuous-time signals. In both cases, the procedure is to write the equation describing the system, transform the equation to the Laplace or z-transform domain, solve for the desired variable, then transform back to the original domain.

The desired (and usual) result is a closed-form solution for the variable of interest. In the case of sampled-data systems, that would be an equation for $y[n]$ in terms of inputs and the various “components” or processing steps in the system. These would include the multiply, add, and delay functions discussed in the section on signal flow diagrams.

In terms of a formal definition, the z-transform $\mathbf{X(z)}$ of a sequence $\mathbf{x[n]}$ is defined as:

$$\mathbf{X(z)} = \sum_{\mathbf{n} = -\infty}^{\infty} \mathbf{x[n]z^{-n}}$$

As with many other such matters in engineering, however, we do not usually need to employ the formal definition, either to find the z-transform or its inverse. The transforms for many functions have already been worked out and tabulated, just as they have been for integrals, Laplace transforms, Fourier transforms, etc. Thus finding the z-transform or its inverse is largely a matter of putting the expressions into a form that corresponds with an entry in the table.

Since the z-transform, by definition, is an infinite sum, we have to be concerned about whether it **converges**, i.e. produces a non-infinite result. The convergence properties of a given transform depend on the sequence being summed. Furthermore, the transform usually converges only for certain values of z, and the region defined by those values of z is called the **region of convergence** (ROC). Some sequences do not converge at all, and thus have no ROC.

Some of the more popular and useful z-transform pairs are listed in the table below. Note that the region of convergence is included for each entry, as the definition of a transform for a given sequence **must** include the region of convergence.

Sequence	Transform	Region of Convergence
$x[n]$	$X(z)$	R_x
1. $\delta[n]$	1	All z
2. $u[n]$	$\frac{1}{1 - z^{-1}}$	$ z > 1$
3. $-u[-n-1]$	$\frac{1}{1 - z^{-1}}$	$ z < 1$
4. $\delta[n-m]$	z^{-m}	All z except 0 if $m > 0$ or ∞ if $m < 0$
5. $a^n u[n]$	$\frac{1}{1 - az^{-1}}$	$ z > a $
6. $-a^n u[-n-1]$	$\frac{1}{1 - az^{-1}}$	$ z < a $

There are also some characteristics which are known as **properties** of z-transforms, as opposed to transform pairs. Some of the important ones of these are:

7. $x[n-m]$	$z^{-m}X(z)$	R_x except for possible deletion or addition of origin or ∞
8. $m^n x[n]$	$X(z/m)$	$ m R_x$

There are many other tabulated z-transforms, but those in the table above will be sufficient for our purposes.

In the table, note particularly property 7. This is the time shifting property, which is very important in DSP operations, as we will see shortly.

5.2 FINDING THE z-TRANSFORM AND INVERSE z-TRANSFORM

To find the z-transform of a sequence or the inverse z-transform, we must manipulate the form we have into a form that allows us to use the table.

The first and simplest approach, which we employ when possible, is simply to manipulate the z-transform algebraically into a form that has a direct equivalent in the table. Let us use the compound interest example to illustrate this approach.

The difference equation we found earlier for the single-deposit compound interest was

$$y[n] = x[n] + (1+i)y[n-1]$$

To find the z transform of the desired output, namely $Y(z)$, we find the z-transform term by term. Thus we have

$$Y(z) = X(z) + (1+i)z^{-1}Y(z)$$

Factoring out $Y(z)$, we have

$$Y(z)[1-(1+i)z^{-1}] = X(z)$$

$$\text{from which } Y(z) = \frac{X(z)}{1-(1+i)z^{-1}}$$

If $x[n]$ is just a **single** deposit at $n=0$, we treat it as a unit impulse, and from the table its z-transform $X(z)$ is just 1. Thus we can write

$$Y(z) = \frac{1}{1-(1+i)z^{-1}}$$

This is **exactly** in the format of transform pair #5, with $a = 1 + i$.

Thus, taking the inverse transform,

$$y[n] = a^n u[n] = (1 + i)^n$$

If deposits are made **periodically** instead of making just a single deposit, a little more work is required to find $y[n]$ in a closed form. In this case, we treat $x[n]$ as a unit step, and from the table, its z -transform is $1/(1-z^{-1})$. Then

$$Y(z) = \frac{X(z)}{1-(1+i)z^{-1}} = \frac{1}{1-z^{-1}} \cdot \frac{1}{1-(1+i)z^{-1}}$$

To put this equation into a format compatible with the table, we use the technique known as **partial fractions**. This approach requires us to factor the $Y(z)$ into several fractions, each of which corresponds to a table entry. We break down the $Y(z)$ into two fractions, as follows:

$$Y(z) = \frac{1}{1-z^{-1}} \cdot \frac{1}{1-(1+i)z^{-1}} = \frac{A}{1-z^{-1}} + \frac{B}{1-(1+i)z^{-1}}$$

The task, then, is to solve for the two constants A and B . We can accomplish this by multiplying through by each denominator separately. If we multiply both sides of the equation by $1-z^{-1}$, we have

$$= \frac{(1-z^{-1})}{1-z^{-1}} \cdot \frac{1}{1-(1+i)z^{-1}} = \frac{(1-z^{-1})A}{1-z^{-1}} + \frac{(1-z^{-1})B}{1-(1+i)z^{-1}}$$

and canceling terms in the numerator and denominator, we have:

$$\frac{1}{1-(1+i)z^{-1}} = A + \frac{(1-z^{-1})B}{1-(1+i)z^{-1}}$$

and setting $z^{-1} = 1$,

$$\frac{1}{1-(1+i)} = A + \frac{(1-1)B}{1-(1+i)}$$

and $A = -1/i$

Similarly, multiplying through by the denominator of the B term, namely

$1-(1+i)z^{-1}$, we find that $B = (1+i)/i$

Then the z -transform becomes

$$Y(z) = \frac{-1/i}{1 - z^{-1}} + \frac{(1+i)/i}{1-(1+i)z^{-1}}$$

Again using #5 in the table, we find the inverse transform to be

$$y[n] = \frac{-1}{i} u[n] + \frac{(1+i)(1+i)^n}{i} u[n] = \frac{(1+i)^{n+1} - 1}{i}$$

where we have dropped the $u[n]$ because we know that nothing happens before $n = 0$.

One other technique we want to discuss is the ***power series expansion***. This technique requires that the z-transform be written in the form of a power series, such as

$$Y(z) = az^{-1} + bz^{-2} + cz^{-3} + \dots$$

This form frequently occurs when the difference equation is of the form

$$y[n] = ay[n-1] + by[n-2] + cy[n-3] + \dots$$

which arises frequently in applications such as finite-impulse response digital filters.

In this case, the inverse transform can be found term by term, as can be seen just by comparing the two forms above.

5.3 SYSTEM TRANSFER FUNCTIONS

We have been discussing system behavior in terms of an input $x[n]$ or $X(z)$ and an output $y[n]$ or $Y(z)$. Usually, we want to know just the overall behavior of the system, in terms of an output vs. an input. When dealing with sequences, i.e. functions involving $x[n]$ and $y[n]$, it is not convenient to express system behavior in a simple format, because of the shifted terms such as $x[n+3]$ or $y[n-5]$. However, when the system behavior is expressed in terms of z -transforms, we can define a straightforward relationship between input and output, as the ratio

$\frac{Y(z)}{X(z)}$ known as the *system function* $H(z)$.

For example, the expression we had for compound interest was:

$$Y(z) = \frac{X(z)}{1-(1+i)z^{-1}}$$

From this, $H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1-(1+i)z^{-1}}$

A very important form of this relationship results when we consider the system's response to an impulse. Recall that for the compound interest system, the difference equation was:

$$y[n] = x[n] + (1+i)y[n-1]$$

For the single-deposit situation, we let $x[n]$ be a unit impulse. However, the z -transform for the unit impulse is just 1 . Thus, the output is given by

$$Y(z) = \frac{1}{1-(1+i)z^{-1}}$$

Which is exactly the same as the general form of the system function

$$H(z) = \frac{Y(z)}{X(z)} \text{ from above.}$$

This result holds generally, so we can say that the system function of a system is the z -transform of its impulse response. The impulse response is generally called $h[n]$.

A similar relationship shows that the frequency response of a linear time-invariant system is the Fourier transform of the impulse response. The converse is also true, meaning that the impulse response can be obtained by finding the inverse Fourier transform of the frequency response. We will use this concept later in the design of digital filters.

5.4 INFINITE AND FINITE IMPULSE RESPONSE SYSTEMS (IIR and FIR SYSTEMS)

The *impulse response* is a very important characteristic of a system. There are two general categories of impulse response. In some systems, the response to an impulse never completely dies out. For example, in the single-deposit compound interest example, we came up with the result that

$$y[n] = a^n u[n] = (1 + i)^n$$

Obviously, in this case $y[n]$ does not go to zero – it actually grows. However, it is entirely possible that the constant a could be less than 1 , in which case $y[n]$ would decrease with n , but would never go to zero. This is an example of *an infinite-impulse response* system, commonly known as an **IIR** system.

It is also possible that the impulse response of a system *does* go to zero after a certain time. Consider a difference equation of the form below, which arises in certain digital filter designs:

$$y[n] = ax[n] + bx[n-1] + cx[n-2]$$

We can find the impulse response just by letting the inputs be impulses, so that

$$y[n] = ax[n] + bx[n-1] + cx[n-2] = a\delta[n] + b\delta[n-1] + c\delta[n-2]$$

which is just the sum of three impulses, delayed by one, two or three units respectively, and weighted by the constants a , b , and c . Thus the output goes to zero after $n=2$, and this system is said to have *finite impulse response*, called **FIR**.

IIR and FIR systems each have their own unique properties. These are especially important in the design of digital filters, and will be considered in more detail when we get to the section on filters.

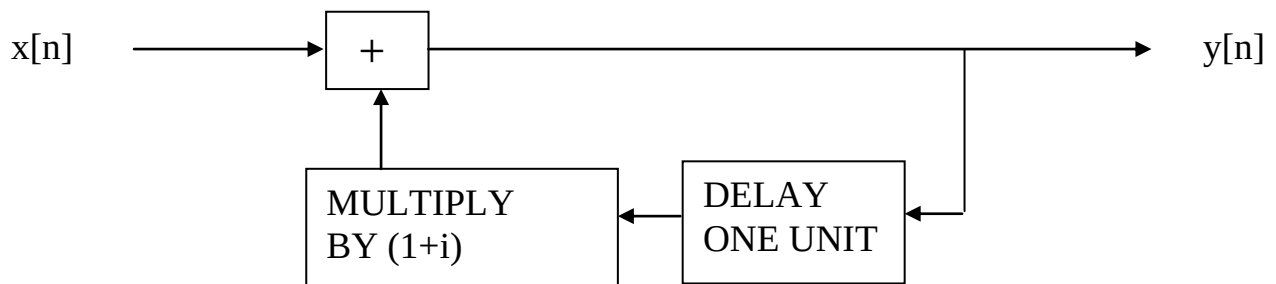
5.5 SIGNAL FLOW GRAPH NOTATION

It should be apparent from the discussion on difference equations that there are only three operations involved in the typical DSP process: delay, addition and multiplication. **Signal flow graphs** provide a convenient way to represent these operations graphically, and so are very useful for depicting the operation of difference equations. Signal flow graphs have branches which represent each of the three operations.

Consider now the compound interest difference equation which we developed earlier

$$y[n] = x[n] + (1+i)y[n-1]$$

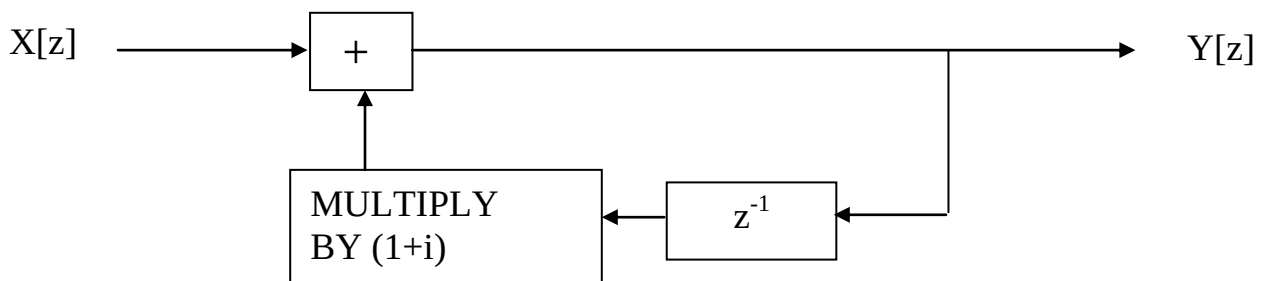
To implement this equation, the block diagram would be:



The equivalent z-transform equation was

$$Y(z) = X(z) + (1+i)z^{-1}Y(z)$$

Noting the similarity of this equation to the difference equation, it is easy to see that the z-transform equation is represented by the block diagram:

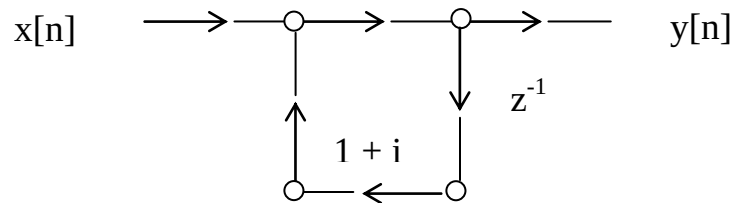


The block diagram form can be simplified by representing each block as just a branch, with the function of the branch denoted by notation on the branch. The direction of signal flow is shown by an arrow on the branch. The gain is shown by a number next to the arrow, and is assumed to be unity if there is no number. A delay of one unit is denoted by z^{-1} . The notation is frequently mixed, so that the z^{-1} notation from z-transforms is used along with the $y[n]$ and $x[n]$ notation from difference equations. Thus, the equations for compound interest would be represented by the **flow diagram**:

Equations:

$$y[n] = x[n] + (1+i)y[n-1]$$

$$Y(z) = X(z) + (1+i)z^{-1}Y(z)$$



We can see that the notation, although hybrid in nature, is not confusing, and the diagram provides a simple way to show the operation of the system, and consequently a straightforward way to mechanize such a requirement.

This form used here is called the direct form, because it is drawn directly from the system function. As we will see shortly, other forms can be drawn by manipulating the more complex system functions into different forms.

More complex expressions can also be represented and mechanized by flow diagrams. The simple expression for $y[n]$ or $Y(z)$ that we mechanized above are fairly easy to diagram. We can see a way to draw the flow diagram for more complex expressions by noting a relationship between the z-transform expression and the flow diagram. For the flow diagram above, the z-transform expression was

$$Y(z) = \frac{X(z)}{1 - (1+i)z^{-1}}$$

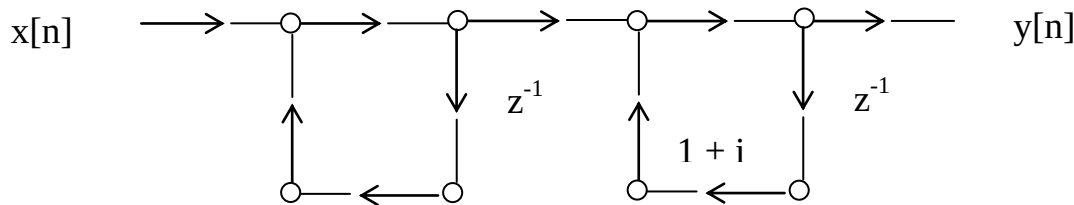
from which, the **system function**, $H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - (1+i)z^{-1}}$

In this expression, it can be seen that the numerator represents the forward path (just 1 in this example), while the denominator represents $1 -$ (the feedback path) in the flow

diagram. Furthermore, z-transforms are linear and can be cascaded to implement multiplication or paralleled to implement addition. Thus, for the expression

$$\frac{Y(z)}{X(z)} = \frac{1}{1 - z^{-1}} \cdot \frac{1}{1 - (1+i)z^{-1}}$$

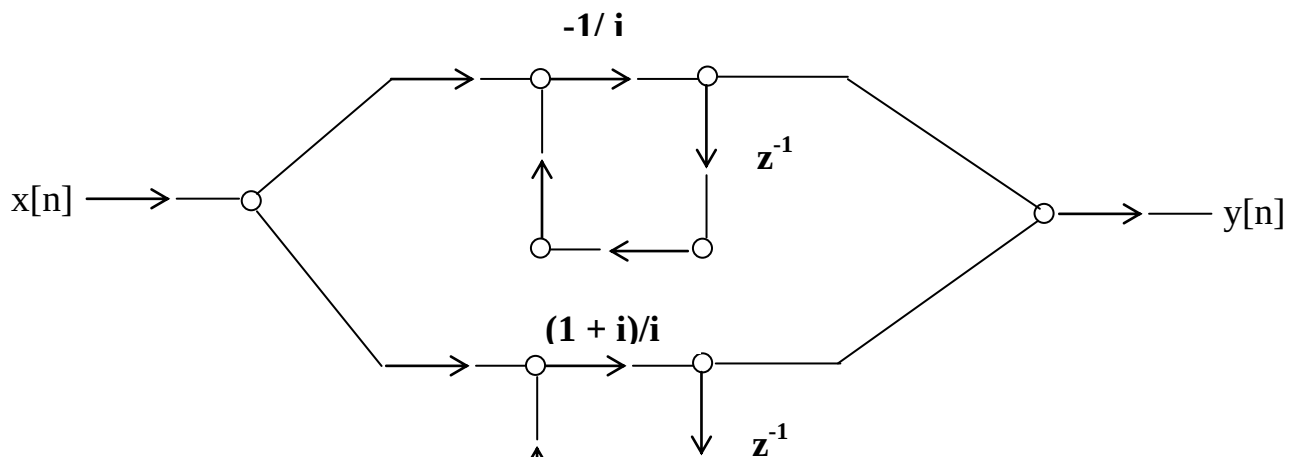
the flow diagram for the **cascade form** for the current system function would be



On the other hand, if we factored the expression into a sum of terms, say

$$\frac{Y(z)}{X(z)} = \frac{-1/i}{1 - z^{-1}} + \frac{(1+i)/i}{1 - (1+i)z^{-1}}$$

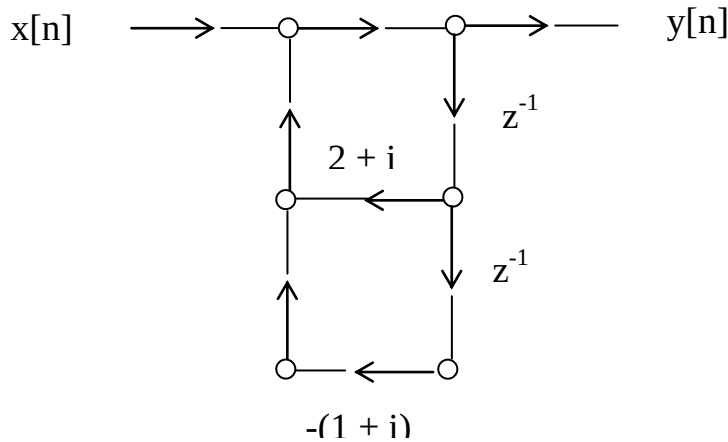
we could represent the system by two networks in parallel, each implementing one of the terms on the right hand side of the equation above. The flow diagram can be constructed by recalling the rule for each term: the numerator represents the forward path, and the denominator represents 1-(feedback path). Following this procedure, the **parallel form** representing the system function above would be:



The **direct form** for this example can be obtained directly from the system function. Multiplying the two factors of the system function to get a single second order term we have

$$\frac{Y(z)}{X(z)} = \frac{1}{1 - z^{-1}} \frac{1}{1 - (1+i)z^{-1}} = \frac{1}{1 - (2+i)z^{-1} + (1+i)z^{-2}}$$

Keeping in mind that the numerator still represents the forward path and the denominator represents $1 - (\text{feedback})$, this system function can be drawn as:



For systems with more than just a simple 1 in the numerator, the flow graphs are more complex, as there will be more than just a 1 in the forward path. In such cases, the cascade form lends itself to further simplification because the delay paths in a forward path and a reverse path can sometimes be combined, giving rise to **direct form I** and **direct form II** subsections.

As we will see later, signal flow graphs are used extensively to analyze and portray the behavior of a variety of DSP systems.

5.6 SIGNIFICANCE OF THE SYSTEM FUNCTION

The system function $H(z)$, the ratio of the system output to the input, plays a key role in DSP design. To reiterate, it is the z -transform of the system impulse response. If we have the system function, we can easily find the system output. If we have

$$H(Z) = \frac{Y(z)}{X(z)}$$

We can multiply $\mathbf{H(z)}$ by the input $\mathbf{X(z)}$ to get $\mathbf{Y(z)}$, the system output. We can then take the inverse z transform of $\mathbf{Y(z)}$ to get the output $\mathbf{y[n]}$.

From the system function, we can also tell if the system is a FIR (Finite Impulse Response) or IIR (Infinite Impulse Response) system.

As an example of an IIR system, consider the system for the compound interest problem. For that system we found that
$$\frac{\mathbf{Y(z)}}{\mathbf{1-(1+i)z^{-1}}} = \mathbf{X(z)}$$

and we found that for a single deposit $x[n] = \delta[n]$, $\mathbf{X(z)} = 1$ and $\mathbf{y[n]} = (1+i)^n$

We then used this $\mathbf{y[n]}$ as an example of an IIR system – one whose output never drops completely to zero. Note that the corresponding system function that we found,

$$\mathbf{H(z)} = \frac{1}{\mathbf{1-(1+i)z^{-1}}}$$

has an expression in z for the denominator. In this simple example, the numerator is 1, but it could be another polynomial expression in z.

As an example of an FIR system, one that has only a finite impulse response, we used

$$\mathbf{y[n]} = \mathbf{ax[n]} + \mathbf{bx[n-1]} + \mathbf{cx[n-2]}.$$

Since the $[n-i]$ is just a delay operator, we can find the z-transform of the expression on a term by term basis, getting

$$\mathbf{Y(z)} = \mathbf{aX(z)} + \mathbf{bz^{-1} X(z)} + \mathbf{cz^{-2} X(z)} \quad \text{and}$$

$$\mathbf{H(z)} = \frac{\mathbf{Y(z)}}{\mathbf{X(z)}} = \mathbf{a + bz^{-1} + cz^{-2}}$$

Now if $\mathbf{X(z)}$ is a single impulse function, its z-transform is just 1, so

$$\mathbf{H(z)} = \mathbf{a + bz^{-1} + cz^{-2}}$$

This term has a numerator only (we can say the denominator is 1). This factor – numerator only – is a characteristic of FIR systems.

In general, DSP systems will have the form,

$$\frac{Y(z)}{X(z)} = \frac{a_0 z^0 + a_1 z^{-1} + a_2 z^{-2} + \dots}{b_0 z^0 + b_1 z^{-1} + b_2 z^{-2} + \dots}$$

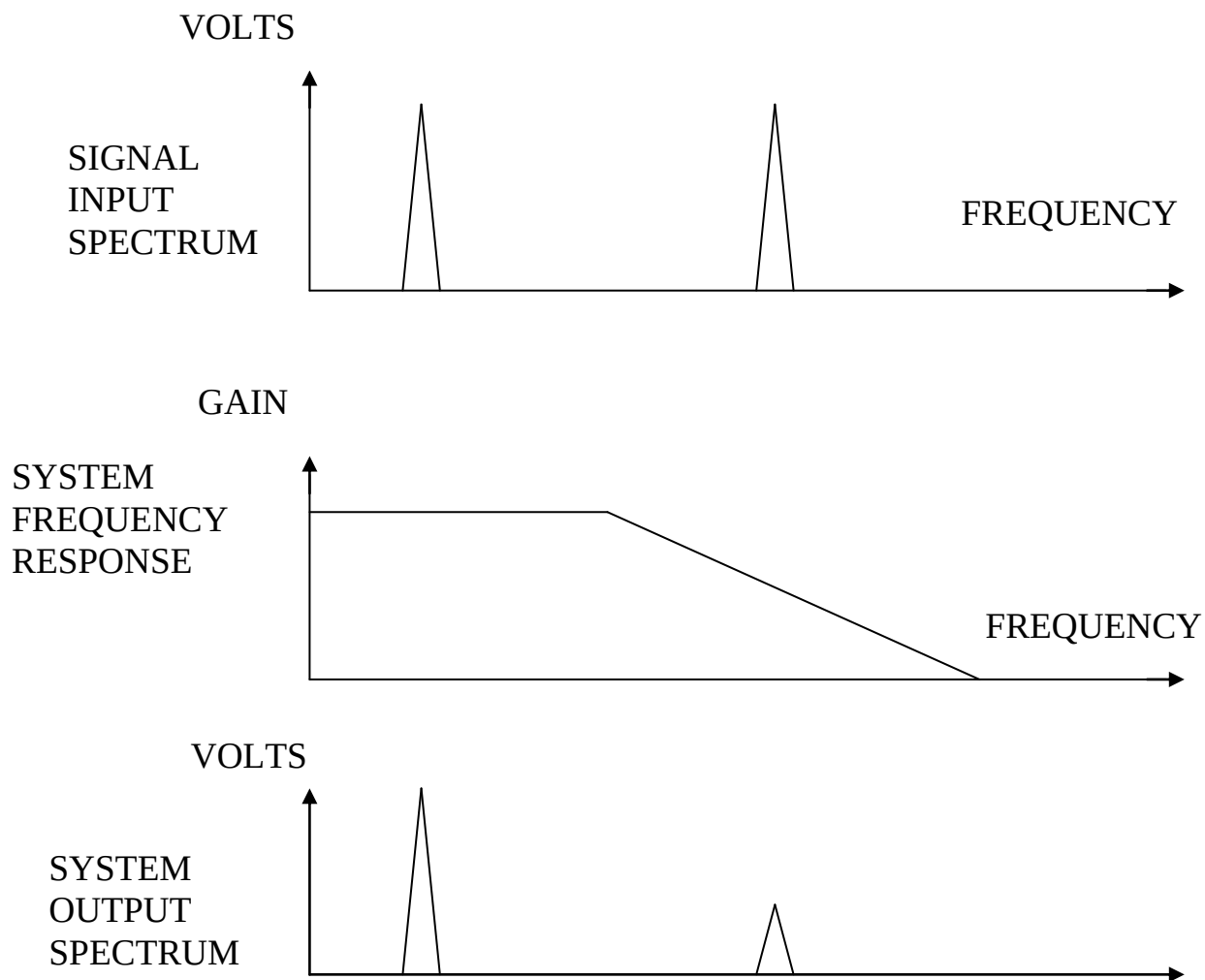
i.e., a ratio of polynomials in z^{-1} . If the denominator is 1, the system is an FIR system. The numerator can be either 1 or some polynomial. When we study digital filters we will see that manual design procedures, as well as programs like Matlab, just generate two sets of numbers – one for the a coefficients and one for the b coefficients in the numerator and denominator polynomials. In Matlab, they are called vectors ***num*** and ***den*** for numerator and denominator.

6.0 RESPONSE OF A SYSTEM TO AN INPUT SIGNAL

Much of what we do in engineering is to compute the response of a system to a particular input. In general, we can discuss either a response in the time domain or the frequency domain. For example, we may have a mechanical system, and we want to know what is the deflection vs time that results from the application of a steady force, or what is the frequency-domain response as a result of a sharp blow or an applied vibration. Similarly, in an electrical system or circuit, we may want to know what the output is when a certain input signal is applied.

6.1 FREQUENCY DOMAIN

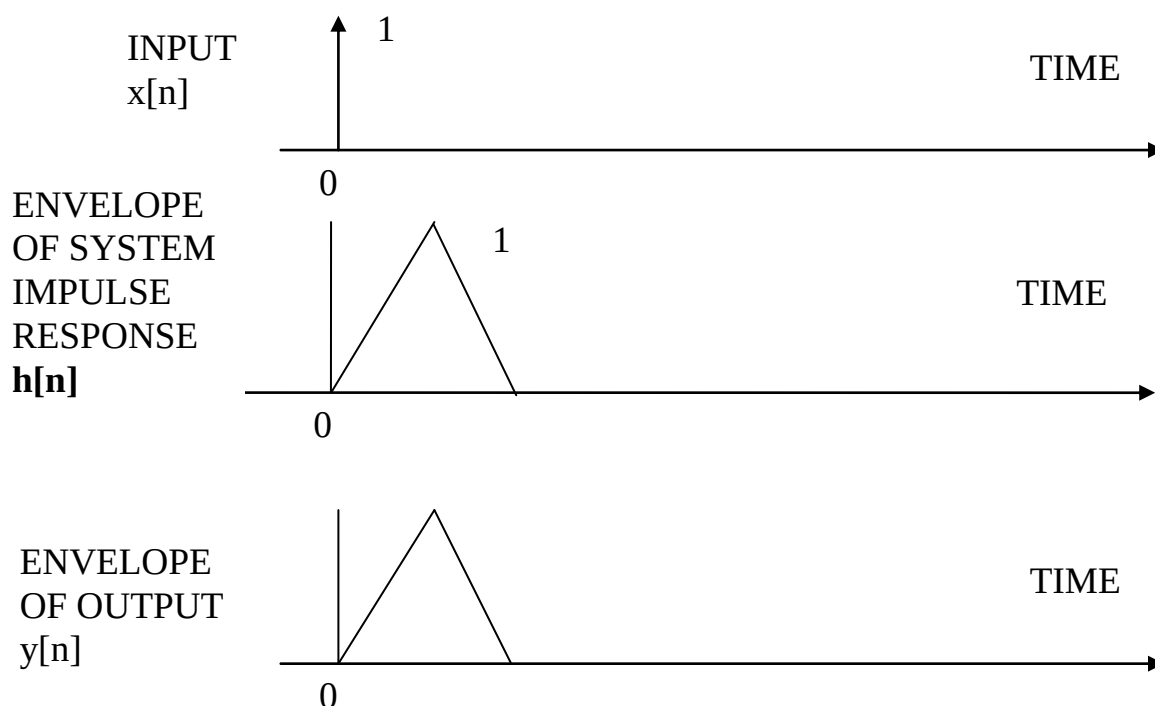
First, let us look at the situation when we know the signal and circuit characteristics in the frequency domain. Let us assume we have a signal and system with characteristics as follows:



6.2 TIME DOMAIN

If we have time-domain information, however, the determination of the output is not so simple. The signals used in DSP systems are usually time-domain signals, because the process of acquiring such signals, i.e. sampling, is inherently a time-domain process – the samples are captured at multiple time increments. Thus, we need a way to predict the output of a system, when the signal is in the time domain format. With DSP, this computation is performed with an operation known as **convolution**, specifically the output of a system is given by the convolution of the input with the impulse response of the system.

To see why this is true, let us look at an example. Suppose we have an input signal that consists of a single impulse, with value assumed to be 1 for convenience, and a system whose response to an impulse is as shown below:



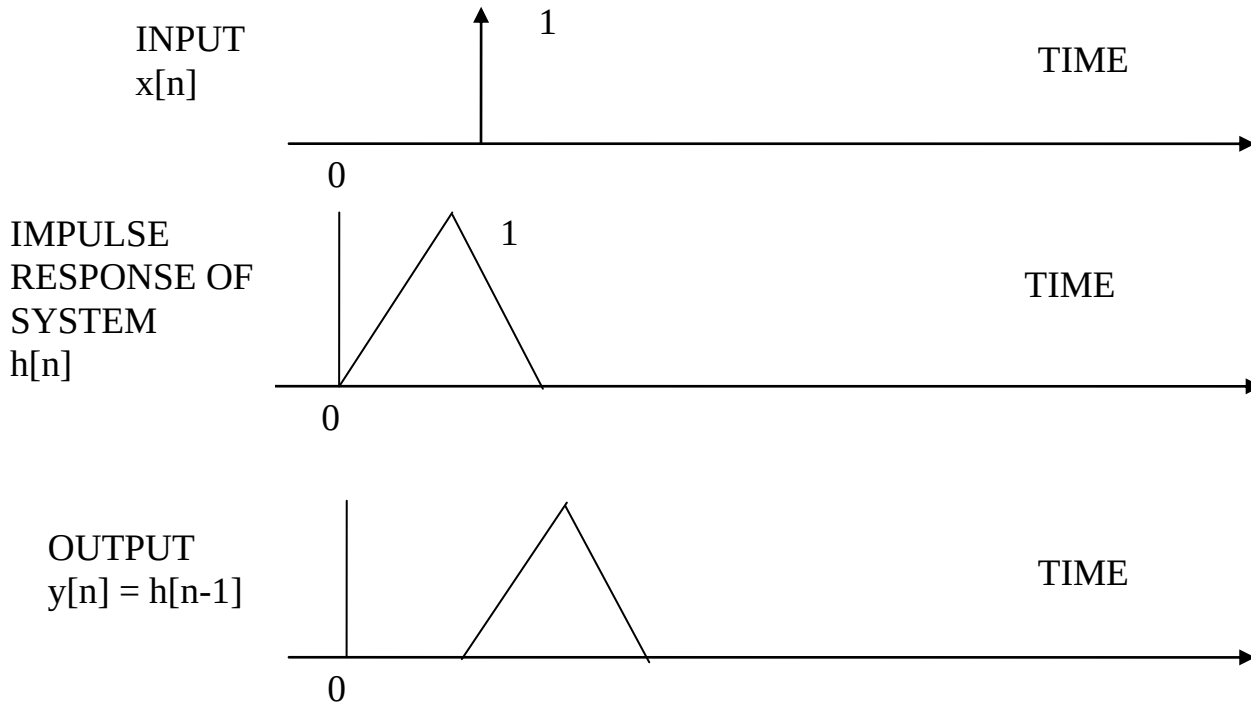
We can see that the output $y[n]$ is given by $y[n] = x[n] h[n]$ where $x[n]$ is just a unit impulse with the value of 1. (remember we are dealing with the values at sample times – the width of the impulse has no meaning)

To reflect the fact that $x[n]$ is the (possible) first term of a longer sequence, and that $h[n]$ is not delayed, (actually delayed by 0 units) we could write that

$$y[0] = x[0] h[n-0]$$

where we are now using $x[0]$ for the value of the first term instead of assuming that it is one.

Suppose now that the input consisted of an impulse, delayed by one unit. We could write it as $x[1]$, the second term of the sequence $\mathbf{x[n]}$. The output of the system would be the same impulse response as before, but delayed by one unit because the input was delayed by one unit. Thus it would be the impulse response $h[n]$ delayed by one unit, which we would write as $h[n-1]$. Then the relationships would be:



Thus, we could write the second component of the output as

$$\mathbf{y[1] = x[1] h[n-1]}$$

to indicate that it is the product of the second term of $x[n]$ and $h[n]$ delayed by one unit. Then the third term of $y[n]$ would be

$$\mathbf{y[2] = x[2] h[n-2]}$$

and so on for all the terms of $x[n]$. Noting the format, we could then write that for N terms total, the entire expression for $y[n]$ could be written as

$$\mathbf{y[n] = x[0] h[n-0] + x[1] h[n-1] + x[2] h[n-2] + x[3] h[n-3] + + x[N-1]h[n-(N-1)]}$$

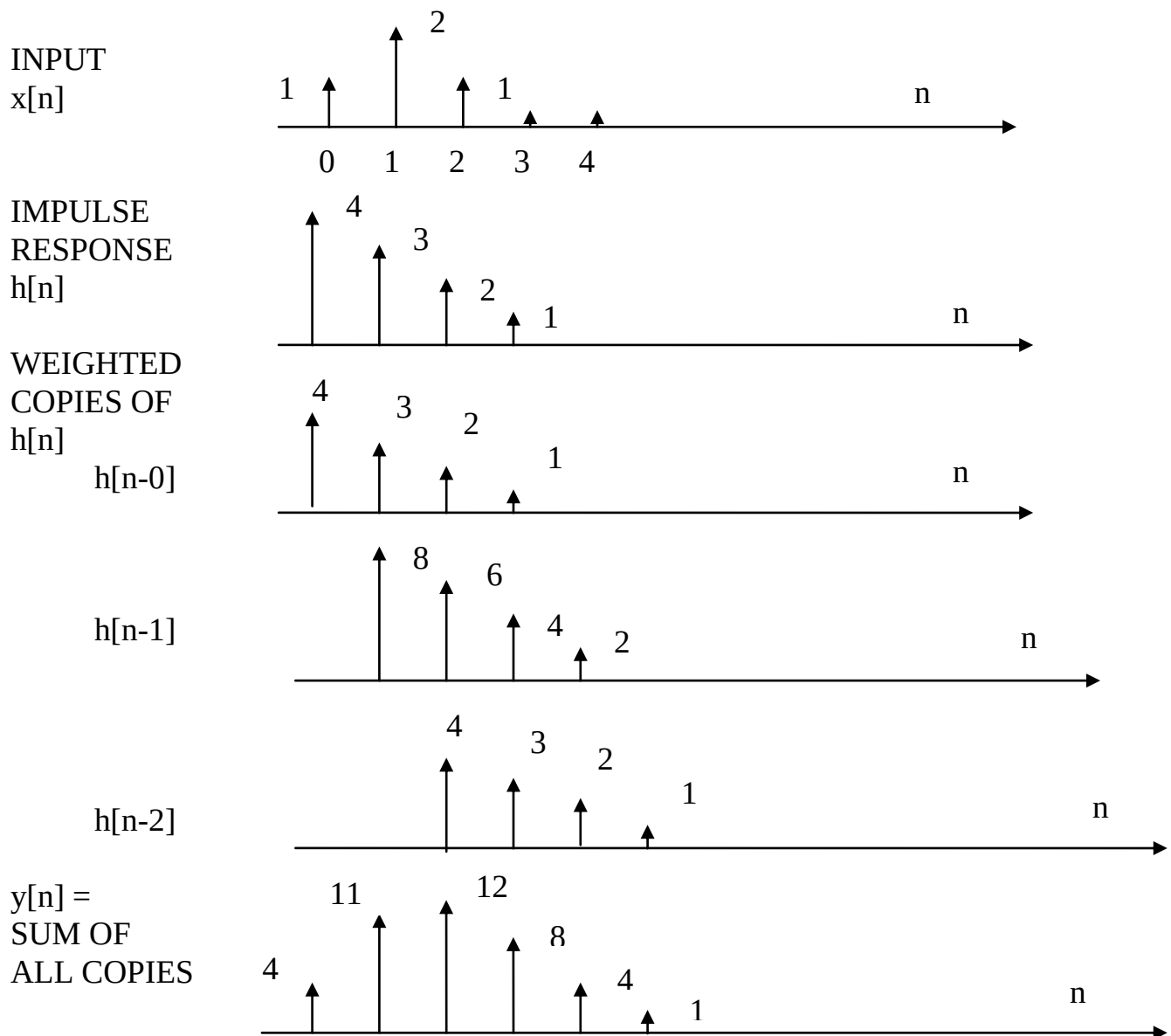
$$= \sum_{k=0}^{N-1} x[k]h[n-k].$$

$k=0$

This is the classical formula for discrete convolution, and is the basis for computing the response of a linear discrete-time system when driven by a sampled-data sequence.

6.3 CONVOLUTION

From the discussion on convolution, we can see that if we are given an impulse response $\mathbf{h[n]}$ and an input signal $\mathbf{x[n]}$, it is quite easy to apply convolution to compute the output $\mathbf{y[n]}$. The approach is: for an input signal $\mathbf{x[n]}$ having N terms, we add up N copies of the impulse response $\mathbf{h[n]}$, each with the appropriate delay and weight. Let us consider an example. Assume we have the following input $\mathbf{x[n]}$ and impulse response $\mathbf{h[n]}$.



Note that the convolution sum for discrete-time systems can be generated simply by applying the definition. This is not true for continuous-time systems, where the convolution sum is an integral instead of a sum. In continuous-time operations, the sum is usually evaluated by reversing the axis of one of the factors, say $h[n]$, and carrying out the integration. While this approach can also be used in discrete-time operations, and is sometimes preferred from a computational viewpoint, the method using the definition is easier to understand.

We will now look at computer programs, specifically Matlab, used for Digital Signal Processing. We will work through some examples to get familiar with the program, then move to the computer lab to do some actual DSP design.

10.0 THE DISCRETE AND FAST FOURIER TRANSFORMS

The second major application of Digital Signal Processing is in computing the spectrum or frequency content of a signal. This is done by the process of the Discrete Fourier Transform, with several modifications that greatly increase the speed of the computation, resulting in a set of algorithms collectively known as the ***Fast Fourier Transform***.

10.1 BACKGROUND

The DFT and FFT algorithms are based on the well-known Fourier Series concept. This principle says that any periodic waveform can be represented by a sequence of harmonically-related sine waves (or cosines or exponentials). For example, a square wave is made up of a sine wave of the same fundamental frequency as the square wave, plus sine waves at all odd harmonics of that fundamental (with various weights applied).

In the continuous-time world, the Fourier series is defined by two equations. The function being considered is represented by the series:

$$f(\omega_0) = a_0/2 + \sum_{k=1}^{\infty} (a_k \cos k\omega_0 t + b_k \sin k\omega_0 t)$$

where the coefficients a_k and b_k are determined by the equations

$$a_k = (2/T) \int_{-T/2}^{T/2} f(\tau) \cos k\omega_0 \tau d\tau \quad \text{and} \quad b_k = (2/T) \int_{-T/2}^{T/2} f(\tau) \sin k\omega_0 \tau d\tau$$

Since the sum of a sine and cosine is an exponential, the Fourier series equations are frequently written in exponential form, where

$$f(\omega_0) = \sum_{k=-\infty}^{\infty} c_k e^{jk\omega_0 t}$$

$$\text{where the coefficients } c_k = (1/T) \int_{-T/2}^{T/2} f(\tau) e^{-jk\omega_0 \tau} d\tau$$

$$-T/2$$

When the signal is non-periodic, or what we might call a transient, the Fourier series becomes the **Fourier transform**, in which there are an infinite number of sinusoids or exponentials, and their magnitudes are given by an integral instead of a summation:

$$F(\omega_n) = (1/2\pi) \int_{-\infty}^{\infty} f(t) e^{-j\omega_n t} dt$$

This is a more general case, and applies to any signal.

10.2 THE DISCRETE FOURIER TRANSFORM

With the application of a fair amount of mathematical manipulation, the continuous-time Fourier transform can be extended to the discrete-time case. Fortunately, we do not need to go through the manipulations to understand and apply the results.

In the discrete-time world, the Fourier transform equations become:

$$x[n] = (1/N) \sum_{k=0}^{N-1} X[k] e^{-j(2\pi/N)kn} \quad \text{frequently called the **synthesis** equation}$$

where

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{j(2\pi/N)kn} \quad \text{frequently called the **analysis** equation}$$

Note that we have switched to the notation usually used in discrete-time signal processing, with $x[n]$ representing the signal in the form of a sequence, and $X[k]$ representing the frequency components or **spectrum** of $x[n]$. Also, the range of the summations from 0 to $N-1$ indicates that these formulas apply to finite-length sequences only. This restriction to finite length is not a serious limitation, as the sequences can be made long enough to extract quite accurate information (sequence values or spectral components). There are also methods involving overlapping segments that allow us to handle very long signals.

These two formulas are usually referred to as the discrete Fourier transform equations. The first is the inverse discrete Fourier transform (IDFT) or the synthesis formula,

which tells us how to represent the signal as a sequence if we know the spectrum $X[k]$. The second is the discrete Fourier transform (DFT) or analysis formula, which tells us how to find the frequency spectrum if we know the sequence coefficients.

Note the similarity of the two equations. Each involves a known quantity, either the signal sequence $x[n]$ or the spectrum $X[k]$, multiplied by an exponential. The exponential is the same in both cases, except for the sign. This similarity allows essentially the same algorithm to be used in computing the discrete Fourier transform and the inverse discrete Fourier transform.

As a point of interest, the exponential in the transform equations is frequently replaced by the complex quantity W_N for ease of manipulation. Thus we let

$$W_N = e^{-j(2\pi/N)}$$

so that the two equations become

$$x[n] = (1/N) \sum_{k=0}^{N-1} X[k] W_N^{kn} \quad \text{the *synthesis* equation}$$

and

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \quad \text{the *analysis* equation}$$

Introduction of the W_N factor not only makes the notation simpler, but aids in utilizing some of the symmetry and other properties that simplify the DFT. This form is used in many theoretical derivations for the FFT.

10.3 THE FAST FOURIER TRANSFORM

The spectrum of a sequence can be computed by using the discrete Fourier transform, as just discussed. However, there is still a substantial amount of computation involved, as we can see when we consider the fact that the computations may involve a large number of points. Computation of the DFT for 1024 or 4096 or even 64K points is not unusual. There are several ways to reduce the amount of computation, which take advantage of the symmetry and other properties of the DFT.

However, the major reduction in computational effort stems from breaking up the very large summations into multiple smaller ones, a process called **decimation**. Let us consider again the formula for the DFT, applied to a sequence of finite length N. Then the equation becomes:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j(2\pi/N)kn} \quad \text{for } k = 0, 1, 2, \dots, N-1$$

In general, both $x[n]$ and the exponential are complex numbers. Thus, the straightforward computation of the transform requires N complex multiplications for each value of k, or at total of N^2 complex multiplications for the entire transform. For example, if $N = 16$, a total of $16^2 = 256$ complex multiplications are required. Now, if the computations are broken down into two equal parts, each requires $8^2 = 64$ complex multiplications, or a total of $2 \times 64 = 128$. Thus, the complexity is reduced by a factor of two. This process can be continued, and the result is that the total number of computations is reduced to $N \log_2 N$, rather than N^2 .

To get an idea of the reduction in computation, let us consider the difference for several values of N, and the reduction in computation achieved:

N	$N \log_2 N$	N^2	Reduction factor	Improvement
256	2048	65536	.03125	32
1024	10240	1,048,576	.009766	102.4
4096	49152	16,772,216	.00293	341.2

The advantage continues to increase with the size of the transform. Obviously, this technique dramatically reduces the amount of computation required.

The process of decimation, along with elimination of computations because of symmetry and other properties, leads to the technique known as the fast Fourier transform, or FFT. As noted earlier, this is one of the main applications of DSP.

Because of the efficiency of the FFT, it is relatively easy to program a computer to calculate the spectrum of a signal. There are many programs available in standard programming languages, and Matlab has built-in routines to perform the FFT. Thus, we

can calculate the spectrum of a signal without knowing very much about the theory behind the FFT. We simply input the signal, and call a Matlab function.

10.4 EXAMPLES OF FFT CALCULATIONS

Let us calculate and plot the spectra of the two signals we used in the filter design project. These were the input signal *s*, and the filter output *sf*. We will just do an FFT analysis and plot for each of them to see the effect of the filter on the spectrum.

The Matlab statement for the FFT is `FFT(s,N)` where *s* is the signal vector and *N* is the (optional) length or number of points. As usual, we can get a complete description of the syntax and usage of the statement from the Matlab help facility:

» help fft

FFT Discrete Fourier transform.

FFT(X) is the discrete Fourier transform of vector X. If the length of X is a power of two, a fast radix-2 fast-Fourier transform algorithm is used. If the length of X is not a power of two, a slower non-power-of-two algorithm is employed. FFT(X,N) is the N-point FFT, padded with zeros if X has less than N points and truncated if it has more. If X is a matrix, the FFT operation is applied to each column.

See also IFFT, FFT2, IFFT2, FFTSHIFT.

We proceed to produce the spectrum of the signals as follows:

```
% Take the FFT of the original signal and its absolute value
% since s has a length of 128, a power of two, we can use the fast algorithm
```

```
ffts = fft(s)
absffts=abs(ffts)
```

Matlab prints out two vectors, for the fft results in complex form and in magnitude form.

```
% Take the FFT of the filtered signal and its absolute value
```

```
fftsf=fft(sf)
absfftsf=abs(fftsf)
```

Matlab prints out two more vectors for the filtered signal.

% to plot, we need a frequency vector, for the scale in plotting

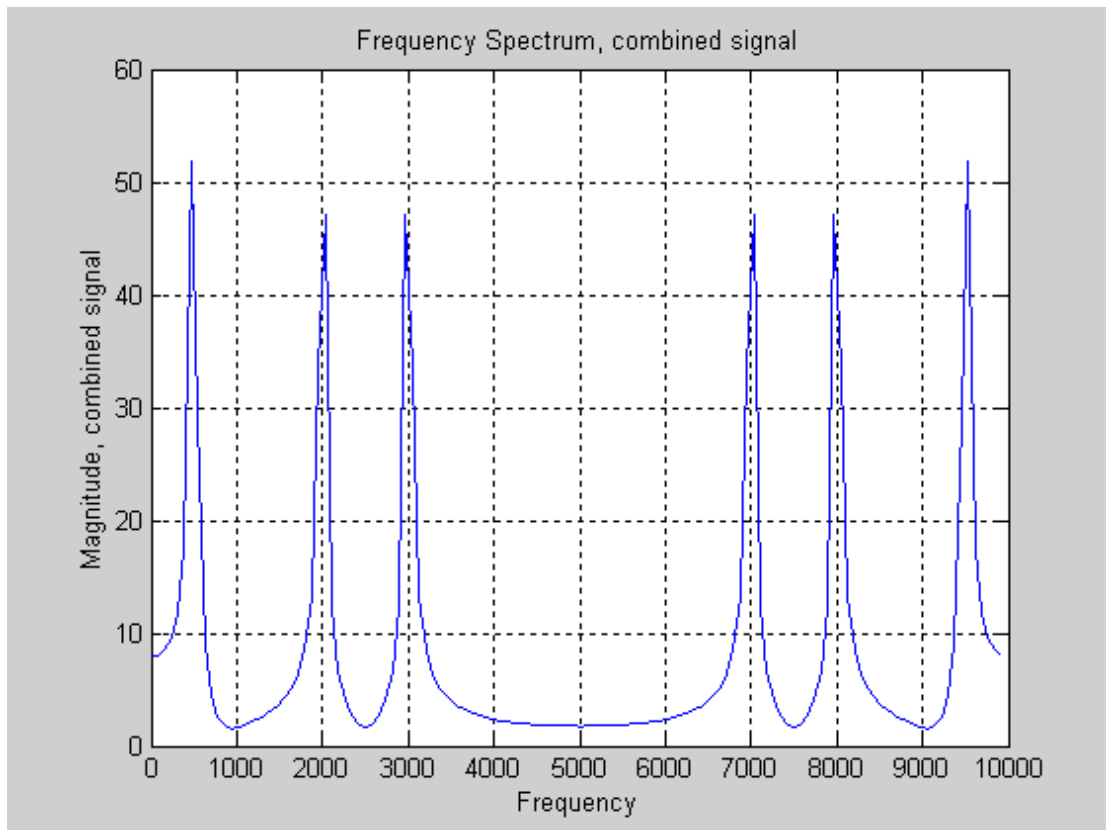
% so let's generate a 128-point vector, ranging from 0 to half of fs

```
f=((0:127)/128)*fs/2
```

% Plot the original signal and the filtered signal on the same plot:

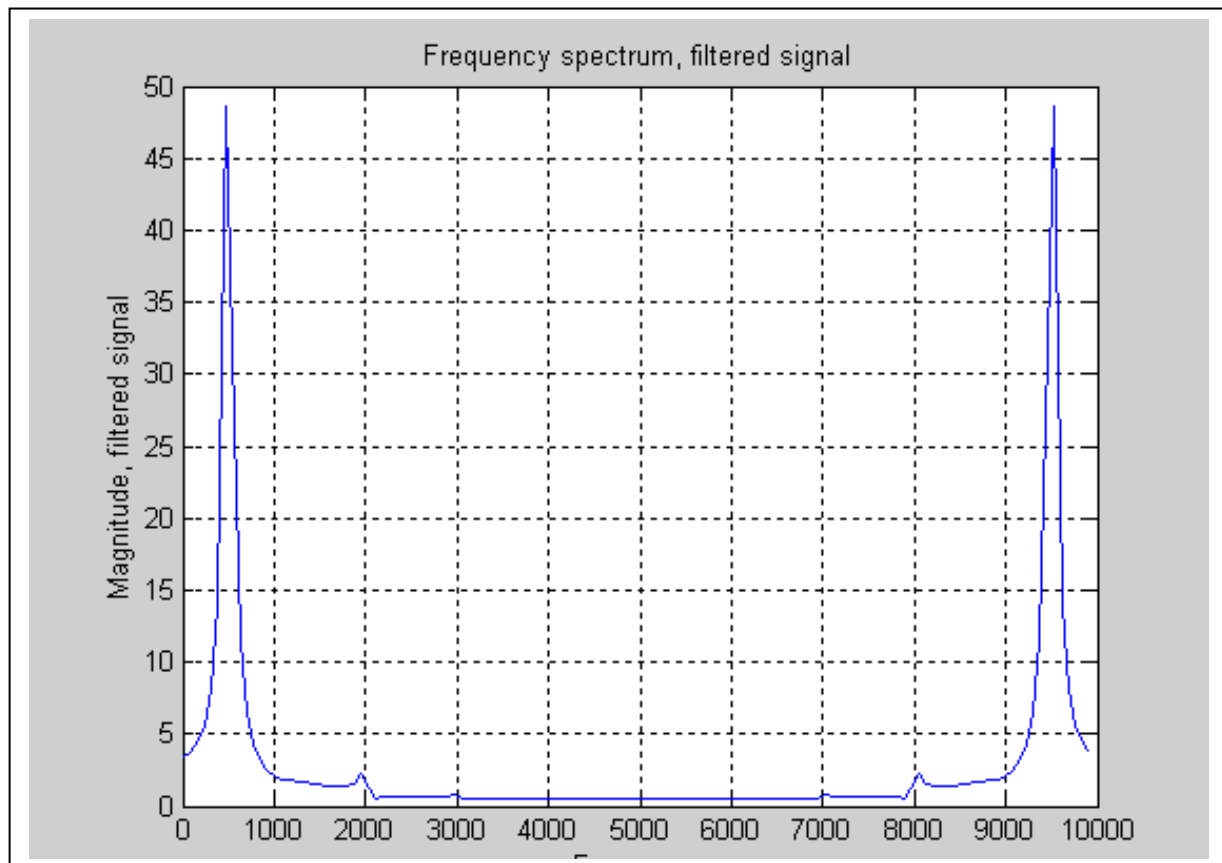
```
plot(f,absffts,f,absfftsf)
```

The spectrum of the original signal shows all three components at equal values:



The filtered signal shows only the portion of the signal within the passband, with only a little indication of the other two frequency components.

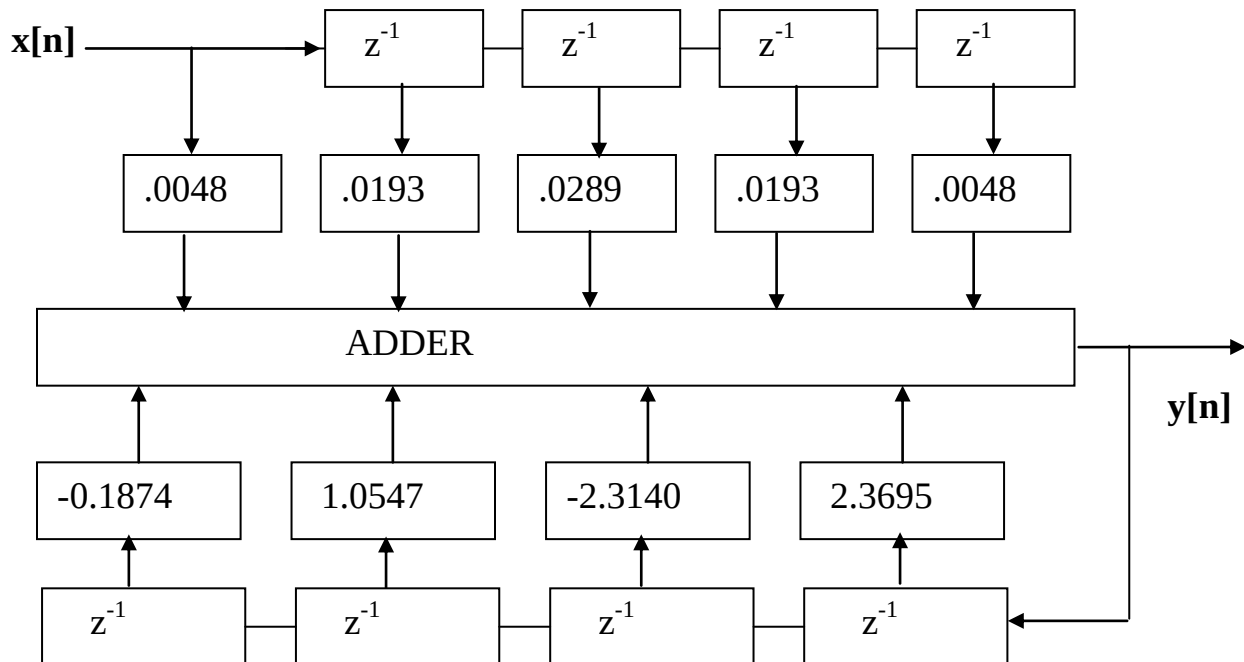
Or, if we just look at the filtered signal with
`plot(f, absfftsf)`, we have



Additional examples to be worked out independently.

11.0 SPECIAL PURPOSE DSP PROCESSORS

It should be apparent that many DSP operations consist of the same two operations - multiplication and addition - repeated over and over again. For example, let us look again at the block diagram for the Butterworth low pass filter we designed earlier.



We can see that the output, $y[n]$, is computed just by multiplying various samples of the input $x[n]$ and past outputs $y[n]$ by constants and adding the results. Because of this very regular and straightforward requirement, it should be no surprise that processors have been designed specifically for this purpose. Since they perform the same operation over and over again, they can be designed to do these specific operations very efficiently, allowing the DSP computations to be done very rapidly.

One very important result of this characteristic is that modern DSP chips can process data in real time in many applications. Thus, digital filtering, extraction of speech parameters, etc. can be done as fast as the signal comes in, and outputs are provided at the same rate, with only a processing delay. The basic requirement for processing to be “real time” is that all processing that takes place after one sample is received, be completed before the next sample is received.

We will not study the architecture of these processors in any detail, but will review the major characteristics involved to get a feel for what we can expect from dedicated DSP chips. Some of the features of modern DSP chips include:

Single-cycle instruction execution. Provides very fast execution of the most common instructions. Also gives predictable execution times for a routine, which is very important in predicting whether or not a specific processor can handle a certain application. This feature also implies that branches, subroutine calls, etc., do not require extra time.

Large on-chip memory. Provides on-chip storage for important data such as filter coefficients. Eliminates the need for the delay involved in accessing off-chip memory.

Optimized architecture. For example, many DMA channels allow for moving data to and from memory without interrupting processing, and a special hardware unit for multiply/accumulate (MAC).

On-chip hardware management of looping. Allows zero-overhead loops and conditional execution.

Separate program and data buses, on-chip. Speeds operation and allows the single-cycle execution.

Hardware circular buffers and barrel shifters. Allow for efficient calling of data used repeatedly, in sequence, such as filter coefficients.

Floating point math. The use of floating-point operations greatly improves the dynamic range of a system, so that the range can be tailored to the signal involved. The result is much lower quantizing noise. Floating point processors are more expensive than fixed-point, but many of them are available.

In addition, many chips have additional features that enhance their usefulness. These include serial ports, on-chip RAM, ROM or PROM, multiprocessing support and a host interface. Because of the popularity of the dedicated DSP chips, substantial development assistance is available, in the form of development software, development boards, in-circuit emulators and various types of training. It is interesting to note that because of the speed and the high level of development of DSP chips, many of them are used as general purpose processors.

Two of the major manufacturers of DSP chips are Texas Instruments and Analog Devices. Brief literature on their products are contained in the Appendix.

12. EXAMPLE: Digital Speech Production.

One good example of the use of digital signal processing is the production of speech. There are many ways to do that, and most involve some DSP.

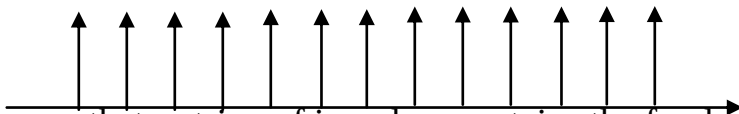
Digitize and store, usually using Pulse Code Modulation, or PCM. The most obvious way, and the simplest conceptually is simply to convert the speech to digital form and store the samples for later D/A conversion and output as analog speech. Data rates of 64 k bits/sec and more are usually used to ensure good fidelity in uncompressed audio. “Telephone quality” speech starts out at 64k bits/sec but is compressed somewhat. Sometimes the weighting of bits is changed as the magnitude of the audio changes, leading to Adaptive Pulse Code Modulation (APCM). More complex techniques used to reduce the data rate include Adaptive Differential Pulse Code Modulation, (ADPCM). CDs use 44.1 k samples/sec, which translates to 352.8k bits/sec for 8-bit encoding and more for larger sample sizes. Obviously these approaches take a lot of storage and require a lot of data to be moved around, so are not very efficient.

Phoneme synthesis. On the other end of the efficiency scale are techniques that store tiny snippets of speech, called phonemes, which can produce intelligible speech at rates of a few k bits/sec. Such speech, however is very mechanical-sounding, and represents what most people would call “computer speech”.

Linear Predictive Coding. Another approach which is quite efficient but takes a lot of computation is Linear Predictive Coding (LPC). This approach is modeled after the human speech process, which we now discuss.

Humans produce only two distinct types of sound when speaking. One is the so-called “unvoiced” sounds, such as the “sssss...” or “th” sound. These are easily produced electronically by random noise. In the computer implementation, a random-number generator serves the purpose.

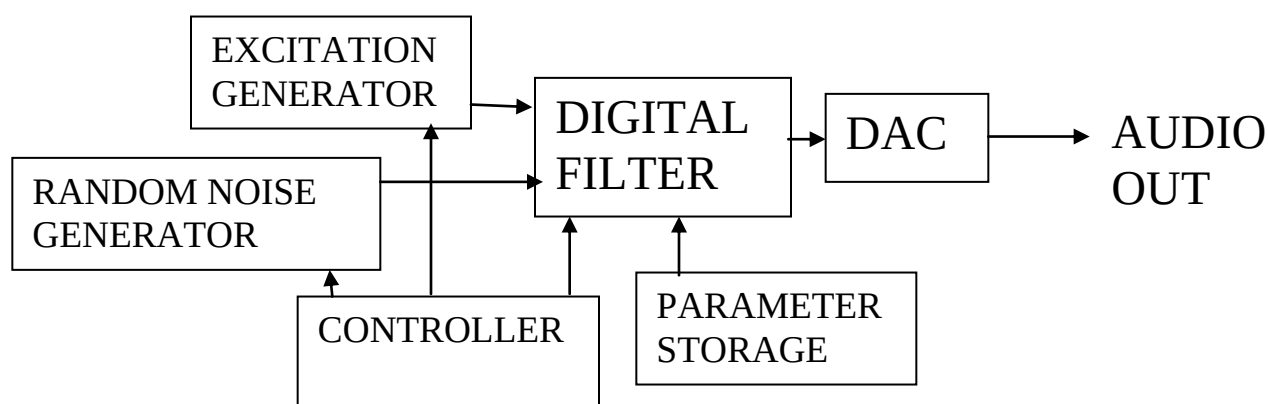
The other type of sound we produce is the “voiced” sounds, which start out as simple train of impulses. When the doc says “stick out your tongue and say ‘ahhh’”, the sound we make is just a string of impulses, produced at a rate of a few hundred Hz. All the rest of the process of speaking or singing is accomplished by modifications of the basic train of impulses.



It is well known that a string of impulses contains the fundamental frequency of the

string, plus all harmonics – theoretically all harmonics at constant amplitude for ideal impulses. The actual sound that comes out of our mouth is then the fundamental and its harmonics, as modified by the various cavities in our throat, mouth, nose, etc., acting as acoustical filters.

It is easy to see how this process can be replicated in electronic hardware. The string of impulses is just multiple copies of the basic impulse function $\delta[n]$. The filtering effect of our mouth, nose, etc, can be accomplished by conventional digital filters. The filter parameters are fixed for short periods of time, requiring changes only every 50 times per second or so – the rate at which human speech changes. So the only data stored are the filter parameters and the basic frequency of the excitation for a given passage of speech. With careful design, a data rate of a few thousand bits per second can be achieved. A very much simplified diagram for a system to produce LPC speech is:



The Texas Instruments “Speak and Spell” speaking toy, sold in the 1980s, used this technique. It used filters on the order of 10 to 13 stages. It also used a different filter configuration, namely the lattice configuration, rather than the direct forms we have covered.

While the mechanism for reconstructing speech using LPC is rather easy to understand, it should be realized that deriving the filter parameters and the excitation is a rather complex process. It involves extracting the parameters from spoken speech, and uses some fairly sophisticated digital signal processing. In any case, the whole field of speech reproduction is a good example of an application of DSP.

Matlab allows us to write a file, called an m-file, that executes Matlab commands. The m-file that goes through all the steps in this lecture, as well as the application of the Fast Fourier Transform, is given below

```
%M-file to design and plot results of 4th order Butterworth filter,  
%1000 Hz cutoff, 10000 Hz sampling rate  
% set sampling frequency to 10kHz  
fs = 10000  
pause  
% normalize the 3dB frequency to half of fs  
wn = 1000/(fs/2)  
pause  
%design Butterworth filter  
[b,a]= butter (4,wn)  
pause  
%Matlab responds with b and a vectors  
% compute frequency response for 128 points  
[h, w] = freqz (b, a, 128);  
pause  
%Matlab prints out complex response vector and frequency vector  
% to plot we will need magnitude of response  
mag = abs (h);  
pause  
%Matlab prints out magnitude vector  
%Plot the magnitude vs frequency  
plot(w,mag),grid  
xlabel ('Frequency normalized to Fs = 2pi')  
ylabel ('Magnitude')  
title('Linear plot of 4th Order Butterworth Filter Response')  
pause  
%We can also plot semilog - with the x axis using log scale  
semilogx(w,mag),grid
```

```

xlabel ('Frequency normalized to Fs = 2pi')
ylabel (' Magnitude')
title('Semilog Plot of 4th Order Butterworth Filter Response')
pause

%To plot phase we need to extract phase from response vector
phase = angle(h);
pause

%Plot the phase
plot(w,phase),grid
xlabel('Frequency normalized to Fs = 2pi')
ylabel ('Phase' )
title('Phase Response of 4th Order Butterworth Filter')
pause

%Now test the effect of the filter by setting up a time vector of 128 points
%and an input signal consisting of three frequencies
t = (1:128)/fs;
pause

s = sin(2*pi*t*500) + sin(2*pi*t*2000) + sin(2*pi*t*3000);
pause

%Plot the signal, with a grid
plot(t,s),grid

xlabel('Time')

ylabel('Signal magnitude')

title ('Combined 500, 2000 and 3000 Hz signals')

pause

%Now filter the signal through our filter

sf=filter(b,a,s);

pause

%and plot the results

```

```

plot(t,sf),grid

xlabel('Time')

ylabel('Signal Magnitude')

title('Filtered signal')

pause

%now take the FFT of the original signal and find its magnitude
ffts = fft(s);

pause

absffts = abs(ffts);

pause

%take the FFT of the filtered signal and its magnitude
fftsf = fft(sf);

pause

absfftsf = abs(fftsf);

pause

%to plot, let's-generate a 128-point vector, from 0 to fs
f=((0:127)/128)*fs;

pause

%Now plot the original signal, showing all three components
plot(f,absffts),grid

xlabel('Frequency')

ylabel('Magnitude, combined signal')

title ('Frequency Spectrum, combined signal')

pause

```

```
%now plot the filtered signal  
  
plot(f,absfftsf),grid  
  
xlabel('Frequency')  
  
ylabel('Magnitude, filtered signal')  
  
title('Frequency spectrum, filtered signal')
```