INTRODUCTION

TO

DIGITAL SIGNAL

PROCESSING

by

Dr. James Hahn

Adjunct Professor

Washington University

St. Louis

1/27/11 12:02 PM

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 $\mathbf{x_1[n]}$ **produces** $\mathbf{y_1[n]}$**, and** $\mathbf{x_2[n]}$ **produces** $\mathbf{y_2[n]}$**,**

then $\mathbf{ax_1[n] + bx_2[n]}$ **produces** $\mathbf{ay_1[n] + by_2[n]}$

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.

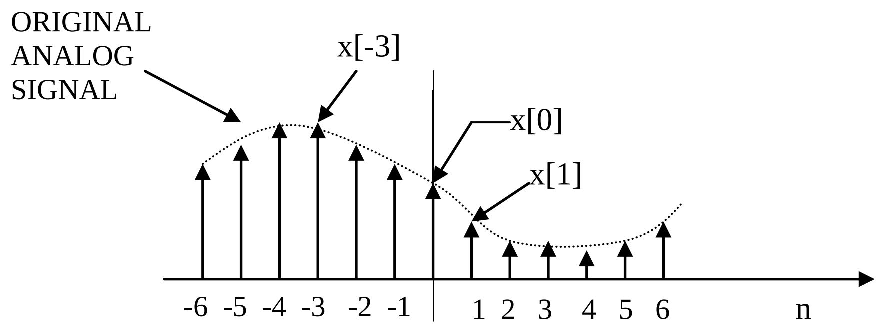
Thus, the computation for point n requires data from the points numbered n+1 and n+2, which obviously aren't available at time n. However, if the data consists of stored points, the average can easily be computed.

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:

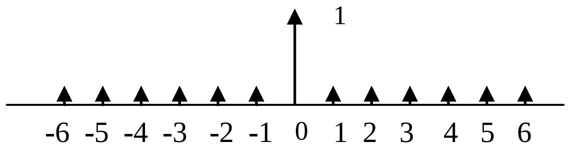
1,2,3,4,5 or 2, -4, -6, 8 or 0.330, -1.25, 5.69, 3.45, -5.77 etc.

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.

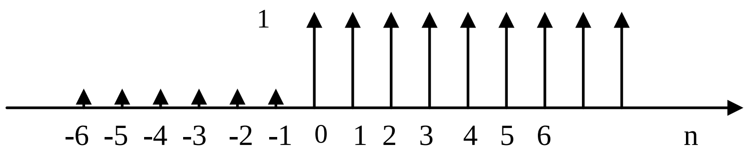
There are several special sequences used in DSP that are not derived from the sampling of a real-world signal. Rather, they are ***defined*** instead of sampled, and thus have somewhat different notation.

The first is the ***unit impulse***. It is designated as **δ[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≥0, and 0 otherwise.



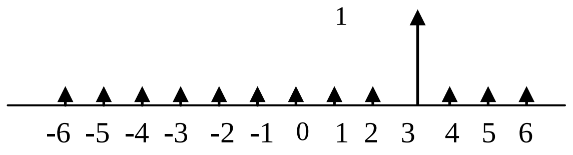
The unit step u[n]

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.

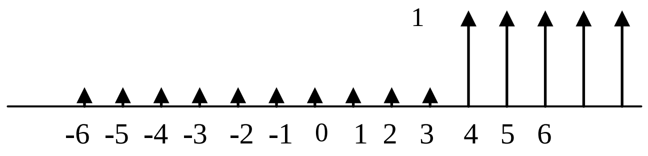
Note that there is one major difference in the notation for ***signals***, such as **x[n]** or **y[n],** and ***defined functions*** such as the unit impulse **δ[n]** or unit step **u[n].** For a signal such as **x[n],** there is, in general, a numerical value associated with every value of **n.** Thus,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

However, for a defined function such as **u[n],** its values are determined by its definition, i.e. **δ[n]** =1 for n=0, and 0 elsewhere; **u[n]** = 1 for n≥0, and 0 elsewhere. It would not be correct to speak of δ[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, δ[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:

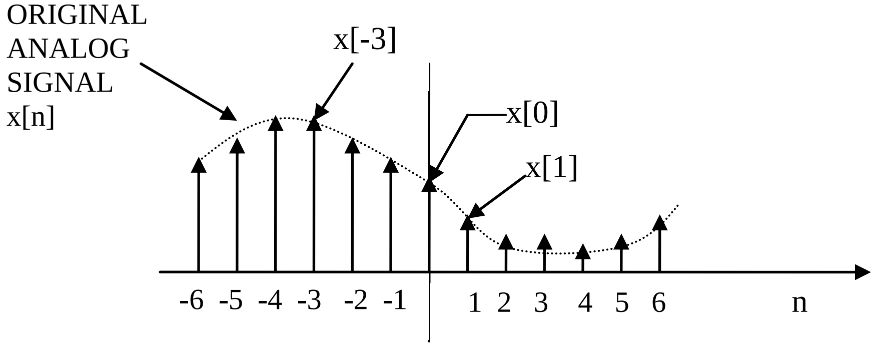


The delayed unit impulse, δ[n-3]

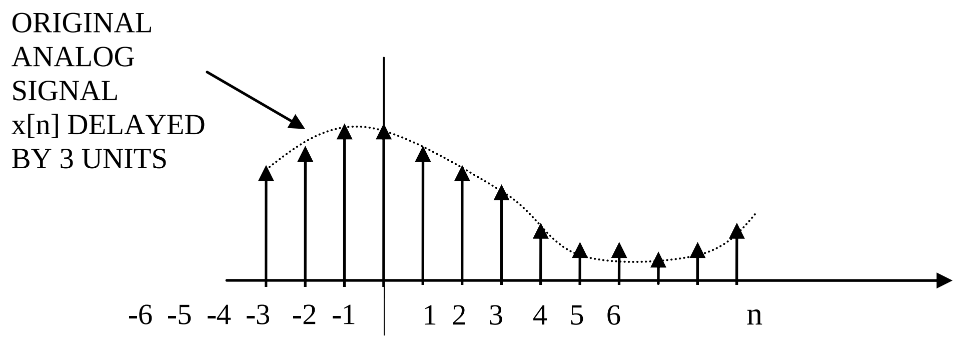


The delayed unit step u[n-4]

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]:



x[n-3] = x[n] delayed by 3

<u>Relationships between the unit step and unit impulse</u>. 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:

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

Similarly, the unit impulse can be expressed as

$$\boldsymbol{\delta}\mathbf{[n] = u[n] - u[n-1]}$$

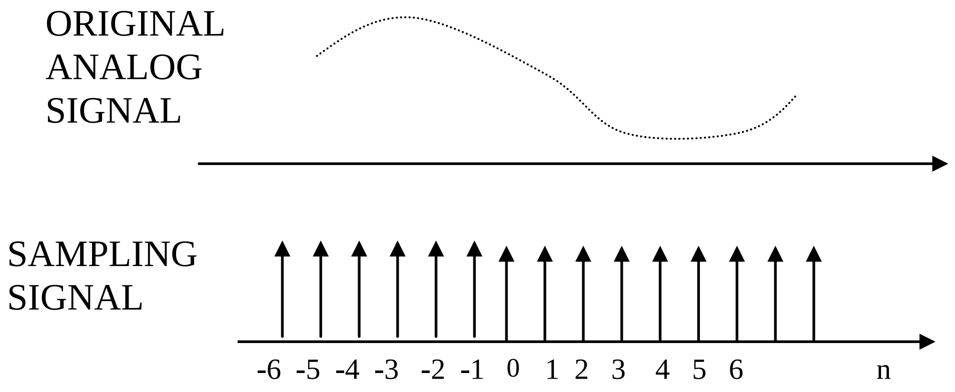
i.e. a unit step less a unit step delayed by one. This form of equation is known as a first-order difference equation. It is analogous to the differential equation used in the analysis of continuous-time systems. As we will see shortly, the difference equation is key in the modeling and analysis of DSP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

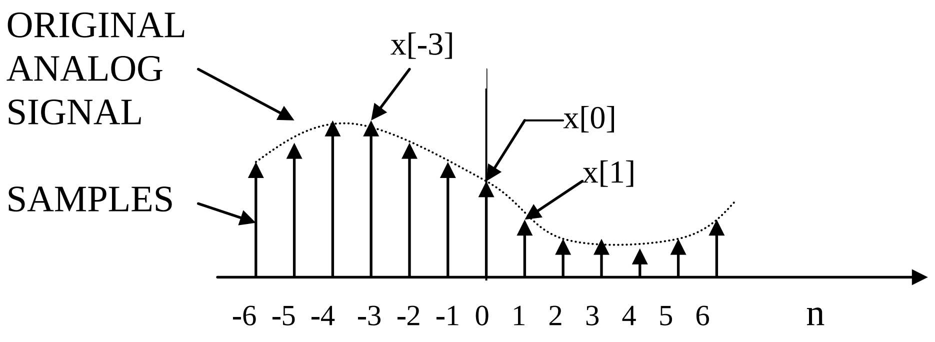
3.0 SAMPLING AND ANALOG-TO-DIGITAL CONVERSION

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:





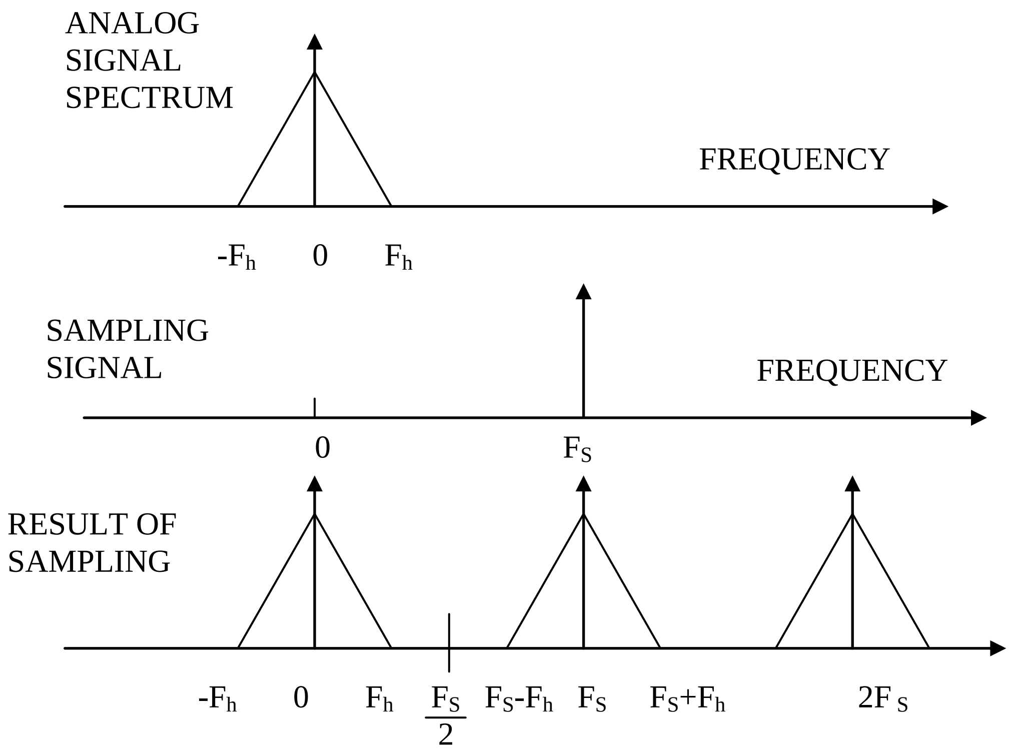
We can see from the diagram that the multiplication of the analog signal by a train of unit-valued impulses will produce the sequence of weighted impulses shown.

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:

$$\mathbf{2\sin(a)\sin(b) = \sin(a+b) + \sin(a-b)}$$

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:

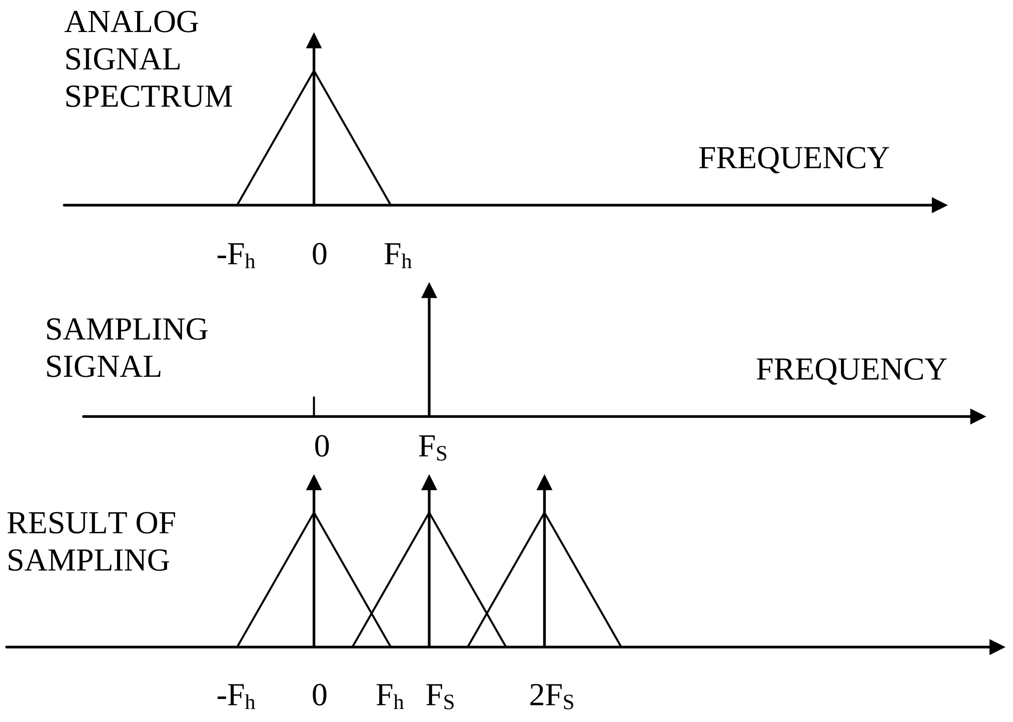


In effect, copies of the original spectrum appear at all multiples of the sampling frequency. With a sharp filter, it would be possible to recover all the original information just by separating out the basic spectrum from all the copies.

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.

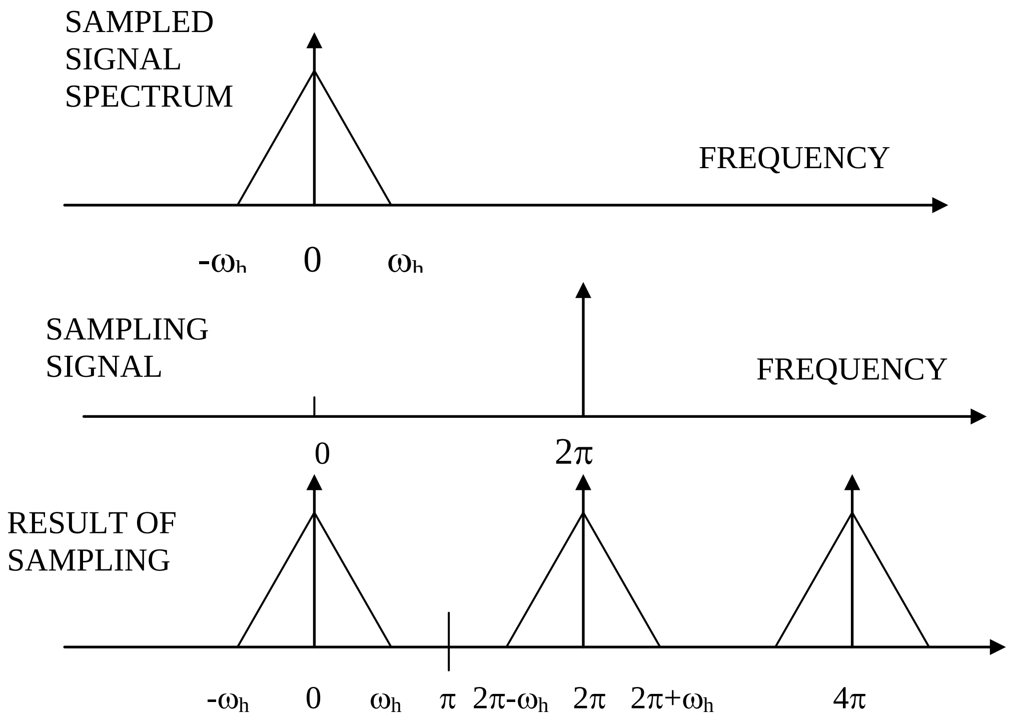
2. The sampling frequency can be increased to more than twice the highest signal frequency. There are obvious limits on this approach also, as the circuitry involved must be able to handle the sampling frequency involved.

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

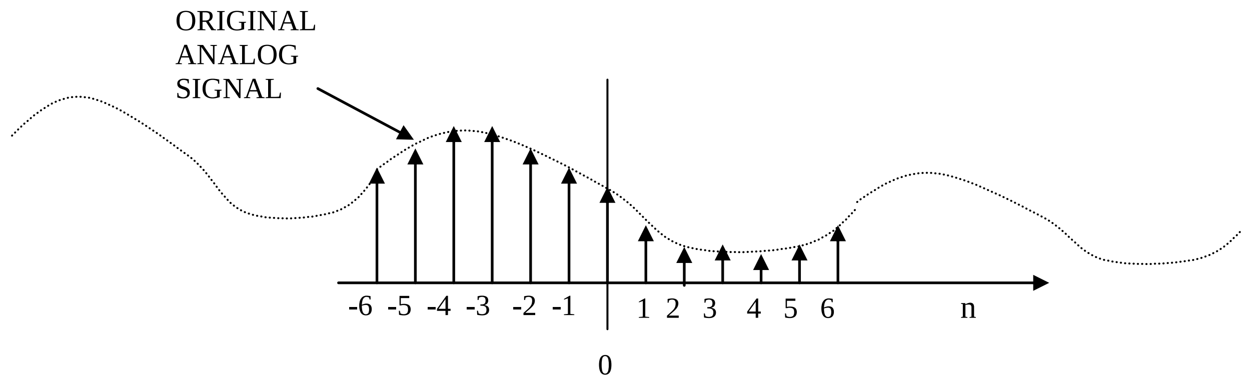
When discussing sampled signals, it is customary to first normalize the frequencies involved. One approach is to normalize to the sampling frequency F_S and use radian frequency instead of conventional cycles/second frequency. With these conventions, the sampling frequency becomes 1 in cycles/sec, or 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:



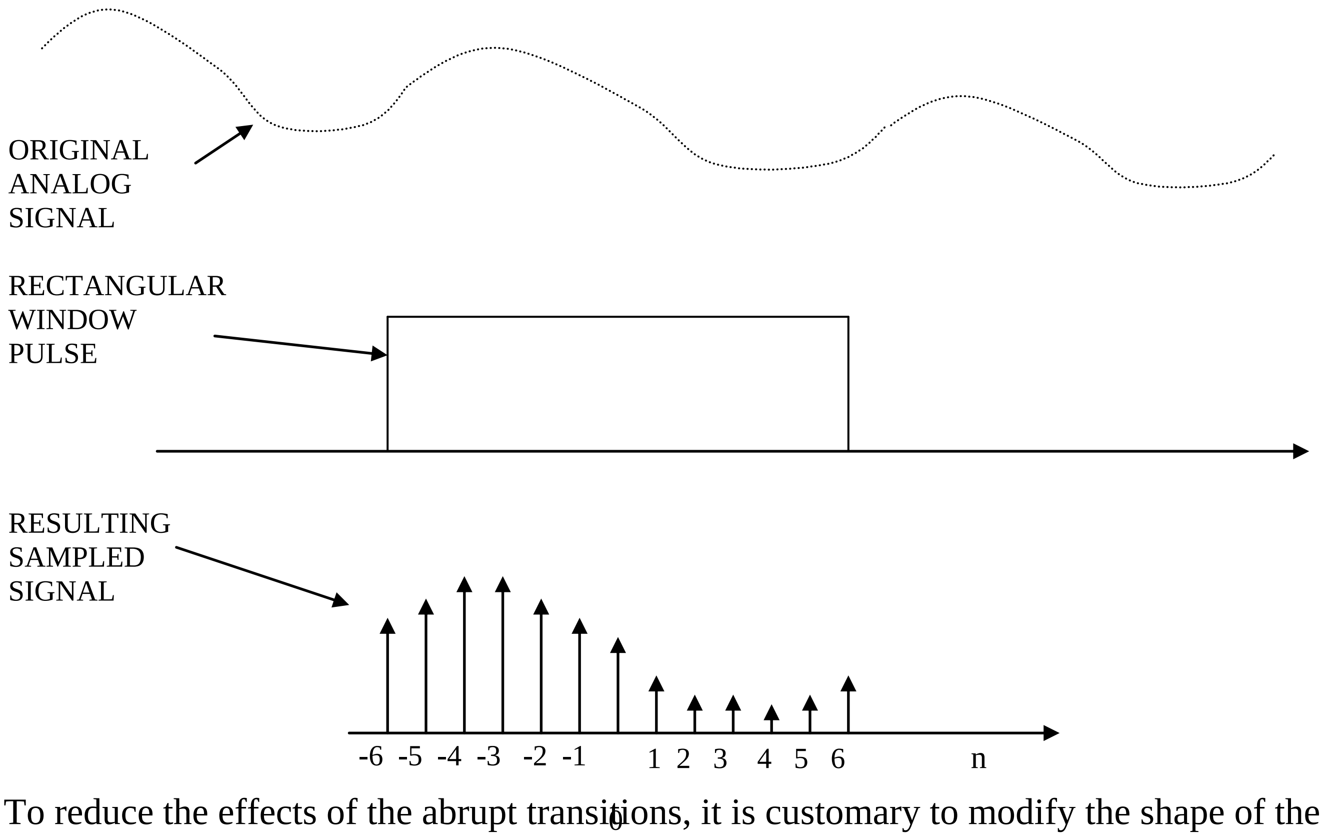
This type of normalization is used in the theoretical derivation of many of the DSP concepts. Commercial DSP software, however, frequently uses different frequency notation. One point to note is that the actual sampling frequency is involved in some of the DSP operations, so that value will have to be 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:

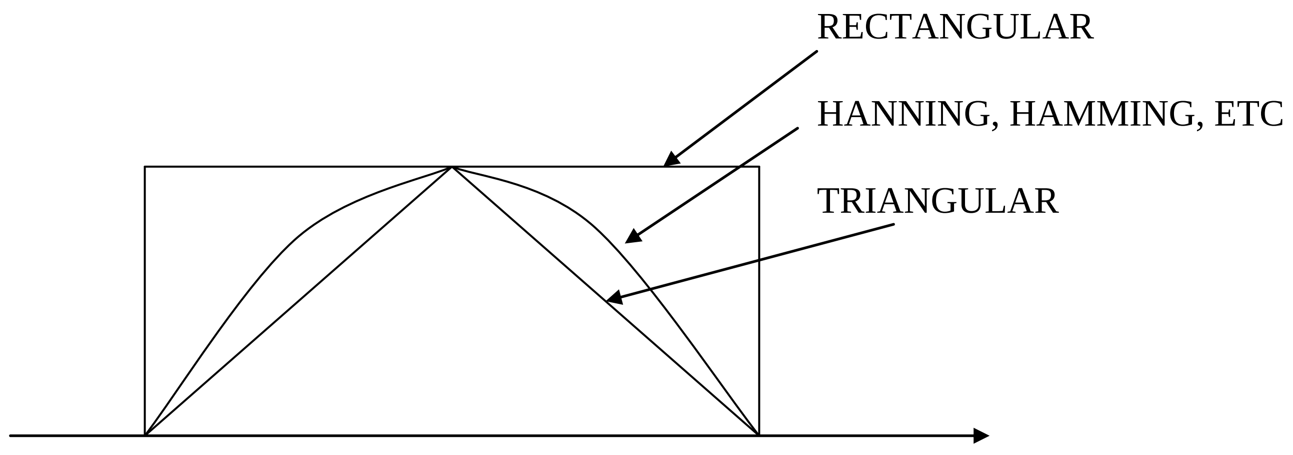


The portion of the signal that is sampled is generally a part of a much longer signal. The sampling starts abruptly at some point and stops at some other point. The abrupt start and stop introduces frequency components into the sampled signal that did not exist in the original. The abrupt start and stop 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.

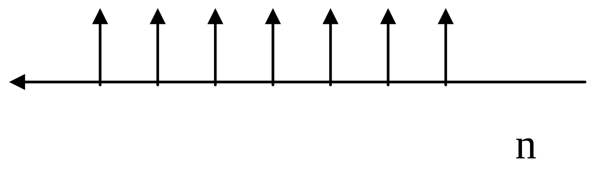
QUANTIZATION NOISE. All of these effects mean that there is nearly always some difference between the number assigned to a sequence value by the limited-precision system, and what its real value would be with an infinite-precision system. The difference is generally uncorrelated from one sequence value 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.

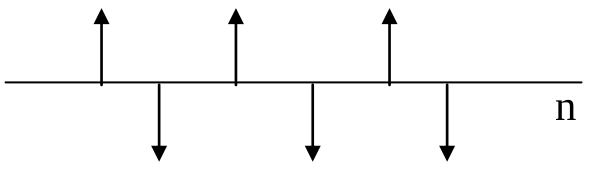
Research and experimentation have shown that the effects of this “noise” depend on the word size (number of bits), the rounding scheme used in multi-stage calculations, and the binary code used (fixed vs. floating point). In general, 16 bits is adequate for most purposes, and decent results can be obtained 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:



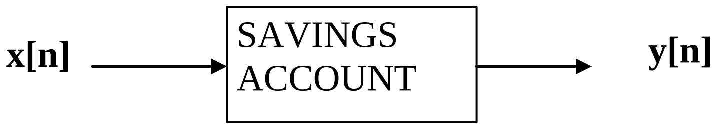


4.0 LINEAR DIFFERENCE EQUATIONS

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 **x[n]** and **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: **δ[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 x[n] and output 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 ***i.*** We let **x[n]** be the ***input*** (amount invested) and **y[n]** equal the ***output*** of the system or the value of the account at any interval **n** periods after we start the deposits. We can see that output **y[n]** is equal to the current deposit plus the previous value with appropriate interest added. Noting that **y[n-1]** is the value of **y[n]** at the previous value of **n**, we can say that

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

That is, **y[n]** equals the current deposit plus the previous deposit with interest added

For example, when n = 0**,** **y[0] = x[0];** (just the current deposit)

when n=1, **y[1] = x[1] +(1+i) y[0]**

when n=2**,** **y[2] = x[2] +(1+i) y[1]**

As we will see shortly, it is possible to solve the difference equations for the output **y[n]** as a function of **x[n]** and ***i*** by using the z-transform. If **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 **X(z)** of a sequence **x[n]** is defined as:

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

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. $u[n]$	**$\frac{1}{1 - z^{-1}}$**	**$\lvert z \rvert > 1$**
3. $-u[-n-1]$	**$\frac{1}{1 - z^{-1}}$**	**$\lvert z \rvert < 1$**
4. $\delta[n-m]$	**z^{-m}**	**All z except 0 if $m>0$ or ∞ if $m<0$**
5. $a^n u[n]$	**$\frac{1}{1 - az^{-1}}$**	**$\lvert z \rvert > \lvert a \rvert$**
6. $-a^n u[-n-1]$	**$\frac{1}{1 - az^{-1}}$**	**$\lvert z \rvert < \lvert a \rvert$**

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. $m^n x[n]$	**$X(z/m)$**	**$\lvert m \rvert R_x$**

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

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

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

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

Factoring out Y(Z), we have

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

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

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

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

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

Thus, taking the inverse transform,

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

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

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

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:

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

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 $\mathbf{1 - z^{-1}}$**,** we have

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

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

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

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

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

and $\mathbf{A = -1/i}$

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

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

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

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

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

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

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

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

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

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:

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

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

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

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.

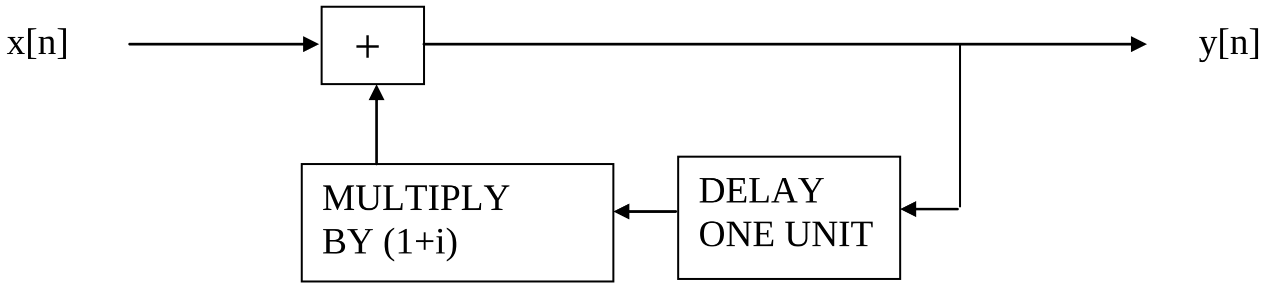
5.5 SIGNAL FLOW GRAPH NOTATION

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]

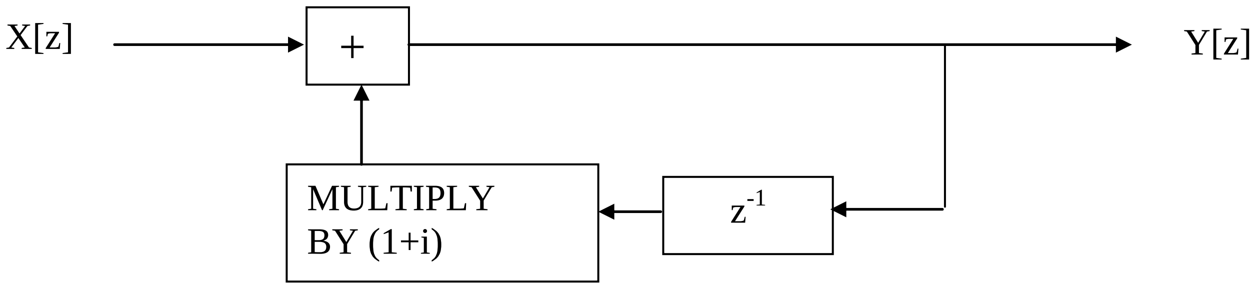
To implement this equation, the block diagram would be:



The equivalent z-transform equation was

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

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:

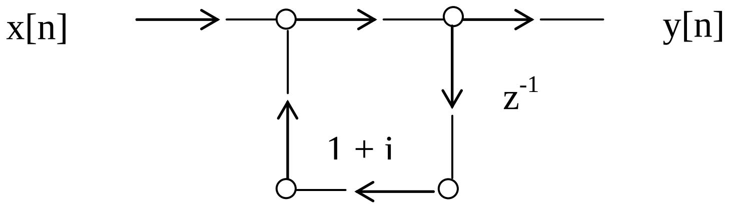


The block diagram form can be simplified by representing each block as just a branch, with the function of the branch denoted by notation on the branch. The direction of signal flow is shown by an arrow on the branch. The gain is shown by a number next to the arrow, and is assumed to be unity if there 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

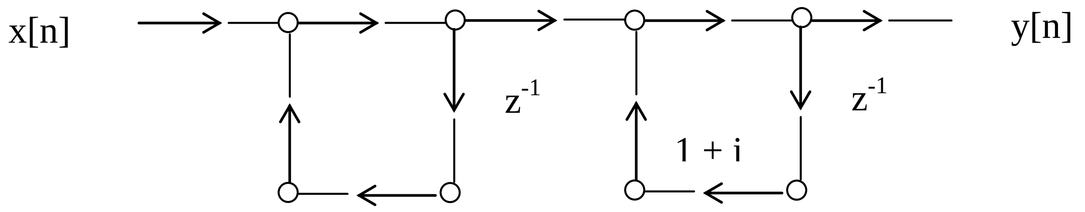
$$\mathbf{Y(z) = \frac{X(z)}{1-(1+i)z^{-1}}}$$

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

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

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

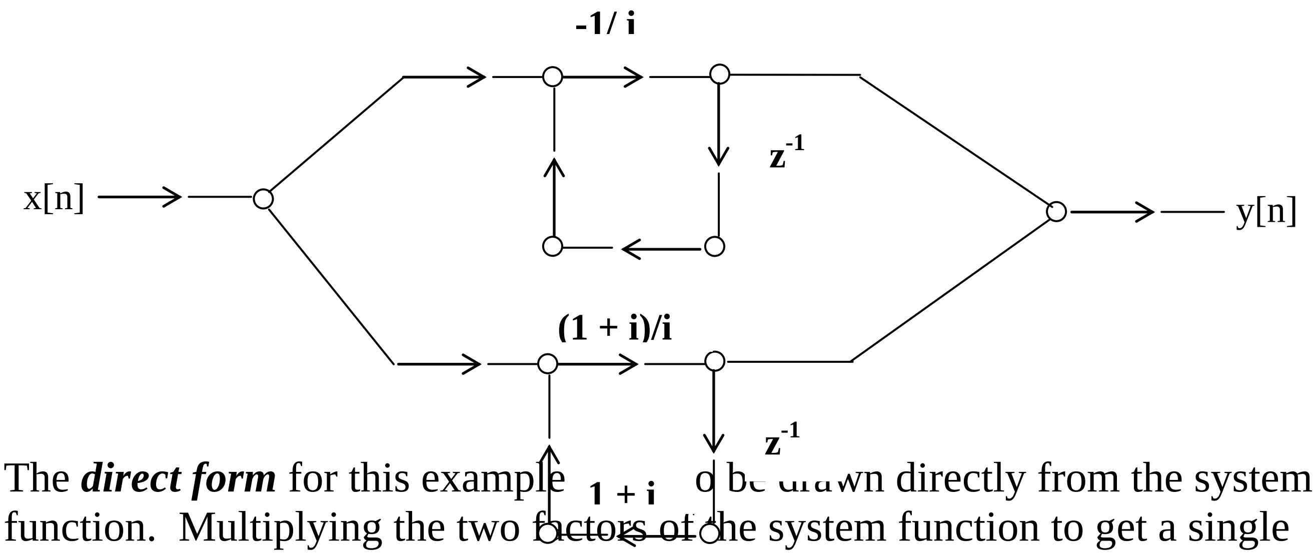
the flow diagram for the ***cascade form*** for the current system function would be



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

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

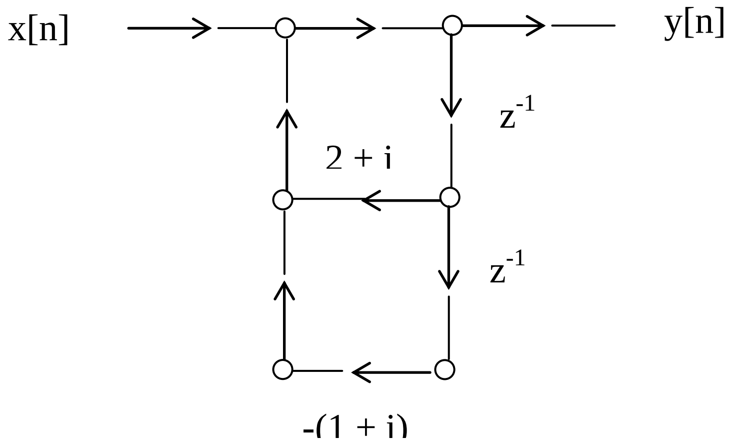
we could represent the system by two networks in parallel, each implementing one of the terms on the right hand side of the equation above. The flow diagram can be constructed by recalling the rule for each term: the numerator represents the forward path, and the denominator represents 1-(feedback path).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



The ***direct form*** for this example could be drawn 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 – (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 **H(z)** by the input **X(z)** to get **Y(z)**, the system output. We can then take the inverse z transform of **Y(z)** to get the output **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 $\mathbf{Y(z) = \dfrac{X(z)}{1-(1+i)z^{-1}}}$

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

We then used this **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,

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

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

y[n] = ax[n] + bx[n-1] + 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

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

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

Now if **X(z)** is a single impulse function, its z-transform is just 1, so

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

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,

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

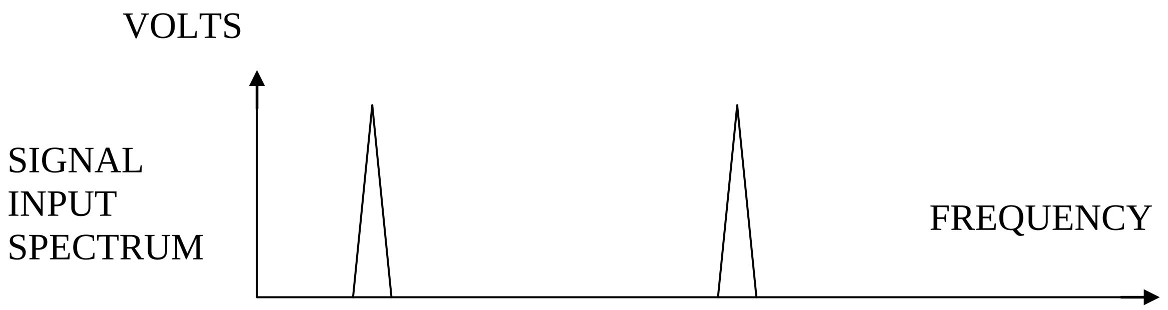
i.e., a ratio of polynomials in $\mathbf{z^{-1}}$. If the denominator is 1, the system is an FIR system. The numerator can be either 1 or some polynomial. When we study digital filters we will see that manual design procedures, as well as programs like Matlab, just generate two sets of numbers – one 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.

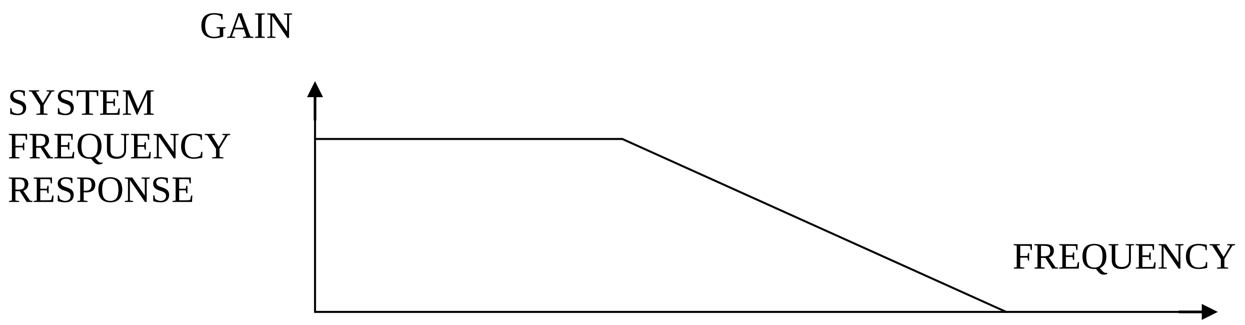
6.0 RESPONSE OF A SYSTEM TO AN INPUT SIGNAL

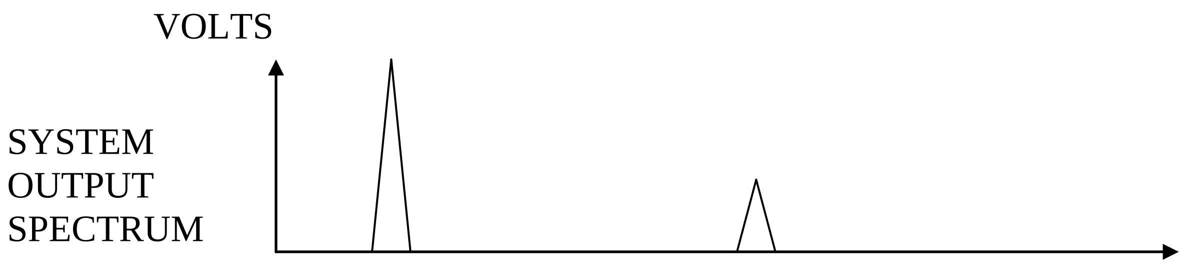
Much of what we do in engineering is to compute the response of a system to a particular input. In general, we can discuss either a response in the time domain or the frequency domain. For example, we may have a mechanical system, and we want to know what is the deflection vs time that results from the 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:



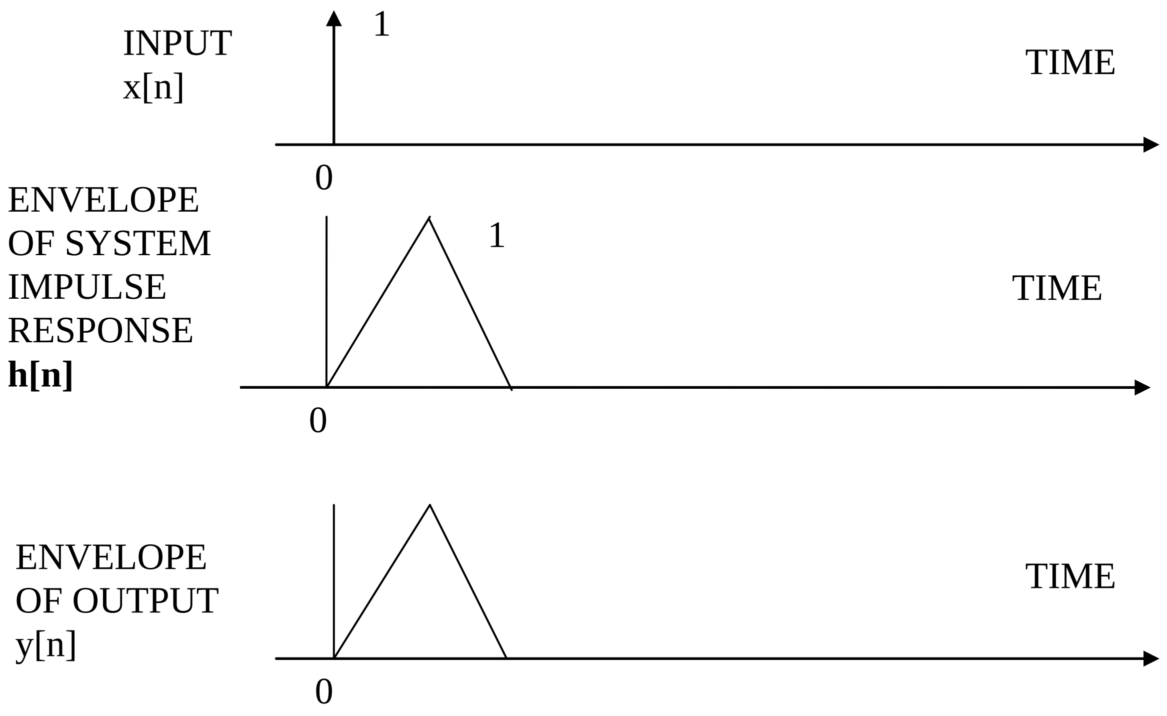




6.2 TIME DOMAIN

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:



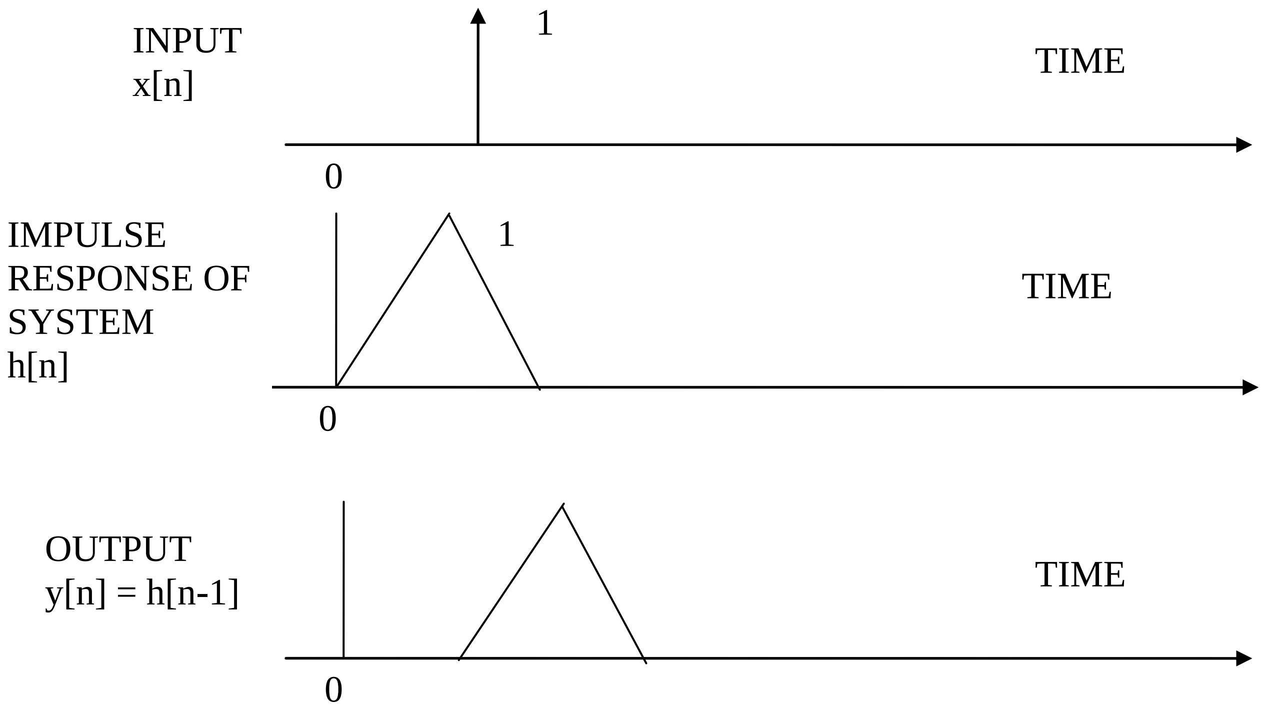
We can see that the output **y[n**] is given by **y[n] = x[n] h[n]** where **x[n]** is just a unit impulse with the value of 1. (remember we are dealing with the values at sample times – the width of the impulse has no meaning)

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 **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

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

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

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)]

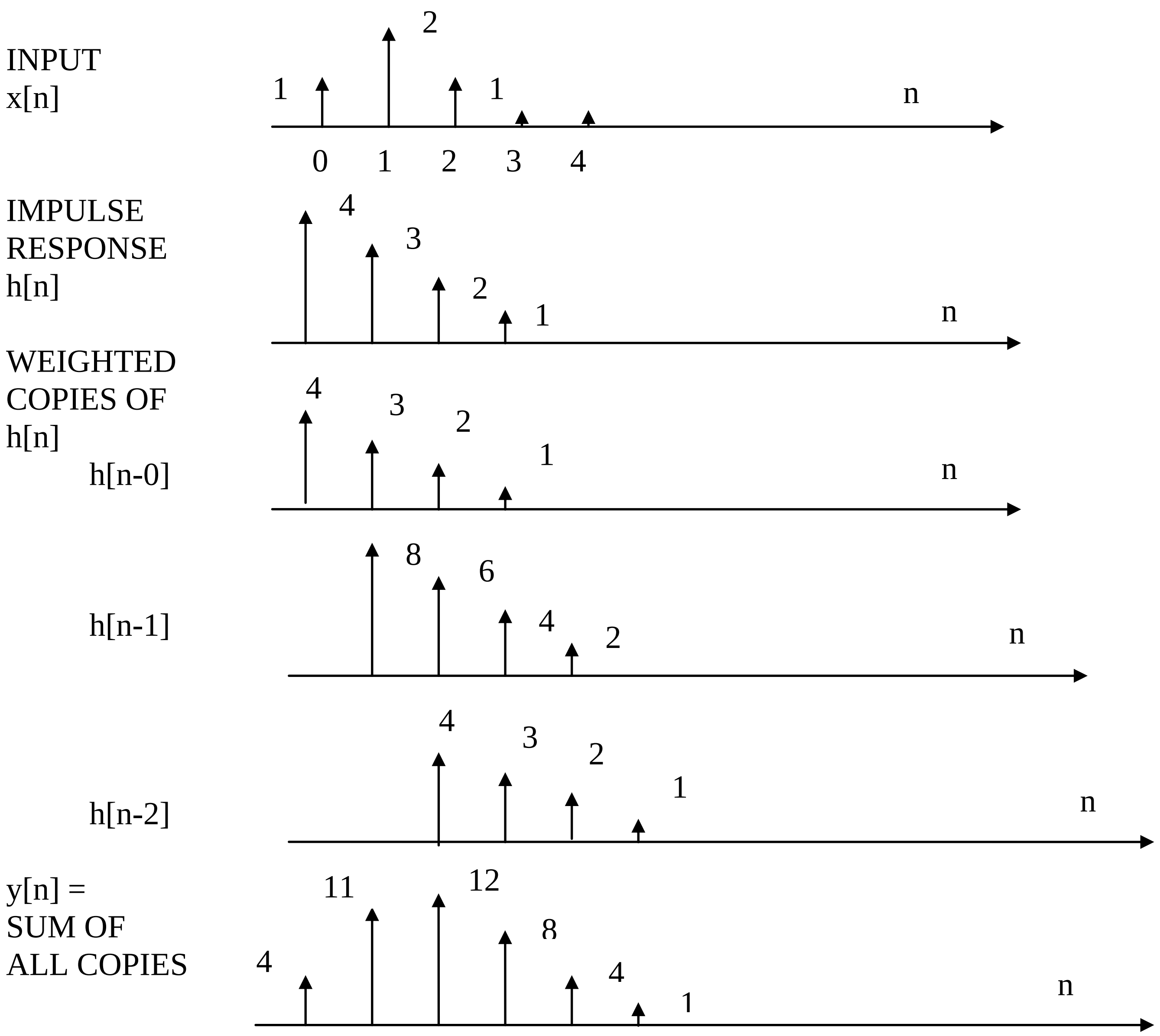
$$= \sum^{N-1} x[k]h[n-k].$$

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 **h[n]** and an input signal **x[n]** , it is quite easy to apply convolution to compute the output **y[n].** The approach is: for an input signal **x[n]** having N terms, we add up N copies of the impulse response **h[n],** each with the appropriate delay and weight. Let us consider an example. Assume we have the following input **x[n]** and impulse response **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.

7.0 COMPUTER PROGRAMS FOR DSP

It should be apparent by now that actual digital signal processing mostly involves extensive and repetitive numerical computations – after the basic concepts have been reduced to the form suitable for computations. As such, it should be obvious that a computer could be programmed to carry out the numerical computations.

In fact, it is the availability of the necessary computing power that makes DSP practical in so many applications. Many general-purpose microprocessors are fast enough to perform useful DSP computations in real time. Furthermore, there are now many processors designed specifically for digital signal processing. We will look at this concept later.

Before a design is committed to specific hardware, the algorithms have to be developed and tested. Many programs have been developed for this purpose. Some are designed just for specific DSP operations, such as digital filter design or Fast Fourier Transforms. Others are very general-purpose computational programs which include DSP as one of their capabilities.

One of the most popular of the general-purpose programs that handle DSP operations is Matlab. Its name comes from the fact that it was initially intended to be a "Matrix laboratory", that is, it was designed primarily to handle matrix manipulations. Over the years, capabilities have been added so that it now supports control system design, optimization, neural networks, system identification, signal processing and others. Each of these capabilities is handled by a ***Toolbox*** of that name. ***The Signal Processing Toolbox*** provides a large number of DSP operations, including those that we have discussed. Our laboratory work in this course will use the Matlab Signal Processing Toolbox.

7.1 INTRODUCTION TO MATLAB

7.1.1 Entering Matlab Commands. When we run Matlab, we start out with a simple command window that contains a list of a few commands to help us get started – basically some of the Matlab help commands. These are followed by the Matlab prompt: >> We are then free to start entering commands.

Matlab is designed to make the entry of commands as natural as possible. Users are encouraged to think of it, at first, as just a simple calculator. For example, if we just enter (after the command prompt)

>> **2+2** Matlab will come back with**:** **ans = 4**

We can also use algebraic expressions, such as

>> x = 5, y=6, z = x+y and Matlab will respond with **x=5 y=6 z = 11**

We can enter a more complicated expression, using traditional algebraic notation such as

>> x = sqrt(3)/2 and Matlab will return: **x = 0.866**

In this mode, Matlab is considered equivalent to a scientific calculator. It can handle most scientific and engineering functions, such as sin, cos, tan, their inverses, the hyperbolic forms, exponentials, rounding, logs, roots and complex numbers.

Consistent with its heritage as a matrix manipulation language, Matlab has extensive array capabilities. An array is listed as a sequence of numbers between two brackets. For example, a one-dimensional array containing the numbers 0 through 4 is:

>> **x = [0 1 2 3 4]**

The array elements must be separated by spaces or commas.

This form is particularly important in DSP because most DSP signals consist of a sequence of numbers, i.e. a one dimensional array. All the usual array operations are available, assuming the arrays involved are of the correct size if more than one are involved.

For example, if we input a second array,

>> y = [0 2 4 6 8] and then type

>> **z = x + y** Matlab returns: **z = 0 3 6 9 12**

also, if we type

>> **w = 2*x** Matlab returns: **w = 0 2 4 6 8**

More complex operations such as mathematical expressions like:

>>   **v = 2*x –1 + y**

would return an array with the elements that result from the application of the equation to each element independently.

We have been specifying what are effectively **row vectors**. Although this is appropriate for DSP operations, we should recognize that **column vectors** and multi-dimensional arrays can also be handled by Matlab. We will address that capability as needed.

7.1.2   Reviewing and editing Matlab commands Matlab remembers all previous variable assignments, computational results, etc. Thus it can tell us the value of any variable previously assigned or the result of a previous computation. If we type

>>y   Matlab will return   **y = 0 2 4 6 8**

and if we type

>>z   Matlab returns   **z = 0 3 6 9 12**

We can examine previous commands by using the cursor control keys. Pressing the ↑ key recalls the most recent command, and subsequent pushes of the ↑ key scrolls back one more step. Similarly, the ↓ key scrolls through the commands in a forward direction. The ← and → keys move the cursor within a command, allowing us to make changes.

We can find out the names of all the variables we have defined by typing

>> **who**

Matlab responds with:

Your variables are:

[list of variables appears here**]**

7.1.3 Matlab organization Matlab is organized as a basic Matlab program, with specialty add-ons called ***toolboxes*** that provide specialized functions. The toolbox we will be most concerned about is the ***signal processing toolbox***. It provides many capabilities for analyzing analog as well as digital signal processing operations. We will not generally need to make clear distinctions between functions that are handled by the basic Matlab program and those that are specific to the signal processing toolbox. As long as the signal processing toolbox is available, we can use its features along with the standard Matlab features without regard for the actual source of the capability.

7.2 MATLAB FUNCTIONS AND OPERATIONS

Matlab has many built-in functions and operations. The tables below summarize those of most relevance to us. We will refer to these as needed in the design process.

7.2.1 General operations

Matlab has many different general purpose commands - far too many to list here. Some of the more useful ones are:

Arithmetic operations. The operators **+ - * / ^** for addition, subtraction, multiplication, division and exponentiation all function as expected and apply to matrices as well as scalars.

The **relational operators** = == < > for assignment, equality, less than and greater than are valid, as are logical operators & | ~ and xor for logical AND, logical OR, logical NOT and exclusive OR respectively.

Basic **trigonometric functions**, such as sin, cos, tan, etc., as well as their inverses and hyperbolic forms are all available.

Control commands. There are a number of commands that control what Matlab does as a programming language, as opposed to calculating. Some of these are the for, if, else, elseif, while, end and return, which function as expected. Also, any command preceded by a % sign is considered a comment, and any command terminated by a; (semicolon) will not automatically print the results of executing that command.

Plot commands provide one of the major features of Matlab. The basic command plot(Y) plots a linear 2-D graph of the columns of Y versus their index. If Y is complex, the statement is equivalent to plotting the real and imaginary parts separately. The command plot(X,Y) plots vector X vs vector Y. If either is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up. The plot command has several other variations. Note that this command can be used to plot raw data, or the results of computations.

7.2.2 Frequency Response

Most of the standard frequency response functions can be computed and plotted by Matlab. The freqs function indicates indicates the use of the s-domain or Laplace transform transfer function, while freqz indicates the z-transform or sampled-data domain. Note that Matlab uses the letter W or w as the dummy variable for radian frequency, instead of the more common Greek ω.

bode	bode (num, den)	Bode plots
	bode (num, den, W)	
	bode (A, B, C, D, iu)	
	bode (A, B, C, D, iu, W)	
freqs	[H,W]=freqs (num, den)	continuous-time
	H=freqs (num, den, W)	frequency response
	[H,W]=freqs (num, den, n)	(Laplace transform function)
freqz	[H,W]=freqz (num, den)	discrete-time frequency
	[H,W]=freqz (num, den, n)	response
	H=freqz (num, den, W)	(z transform function)
	[H,f]=freqz (num, den, n, Fs)	
nyquist	nyquist(num, den)	Nyquist plots
nyquist	(num, den, W)	
nyquist	(A, B, C, D, iu)	
nyquist	(A, B, C, D, iu)	

7.2.3 Frequency Domain Analysis

abs	abs (x)	magnitude of a complex variable
angle	angle (x)	phase angle of a complex variable
grpdelay	[Gd, W]=grpdelay (num, den)	discrete-time group delay
	[Gd, W]=grpdelay (num, den, n)	
	Gd=freqz(num, den, W)	

[Gd, f]=freqz (num, den, n, Fs)

psd	psd (x) psd (x, nfft) psd (x, nfft, Fs) psd (x, nfft, Fs, window)	power spectral density estimation
specgram	specgram (x)	spectrogram
unwrap	unwrap (x)	

<u>7.2.4 Prototype Filter Design</u>

buttap	[Z, P, K]=buttap (n)	Butterworth analog lowpass prototype filter design
cheb1ap	[Z, P, K]=cheb1ap (n, Rp)	Chebyshev I analog lowpass prototype filter design
cheb2ap	[z, p, k] =cheb2ap (n, Rs)	Chebyshev II analog lowpass prototype filter design
ellipap	[Z, P, K]=ellipap (n, Rp, Rs)	elliptic analog lowpass prototype filter design

<u>7.2.5 Filter Design</u>

butter	[num, den]=butter (n, Wn) [num, den]=butter (n, Wn, [Wn1, Wn2]) [num, den]=butter (n, Wn, ‘high’) [num, den]=butter (n, [Wn1, Wn2], ‘stop’) [num, den]=butter (n, Wn, ‘s’) [num, den]=butter (n, Wn, [Wn1, Wn2], ‘s’) [num, den]=butter (n, Wn, ‘high’, ‘s’) [num, den]=butter (n, [Wn1, Wn2], ‘stop’, ‘s’)	Butterworth digital or analog filter design
cheby1	[num, den]=cheby1 (n, Rs, Wn) [num, den]= cheby1 (n, Rs, Wn, [Wn1 Wn2]) [num, den]= cheby1 (n, Rs, Wn, ‘high’) [num, den]= cheby1 (n, Rs, [Wn1, Wn2], ‘stop’) [num, den]= cheby1 (n, Rs, Wn, ‘s’) [num, den]= cheby1 (n, Rs, Wn, [Wn1, Wn2], ‘s’) [num, den]= cheby1 (n, Rs, Wn, ‘high’, ‘s’) [num, den]= cheby1 (n, Rs, [Wn1, Wn2], ‘stop’, ‘s’)	Chebyshev I digital or analog filter design
cheby2	[num, den]=cheby2 (n, Rs, Wn)	Chebyshev II digital or

	[num, den]= cheby2 (n, Rs, Wn, [Wn1 Wn2])	analog filter design
	[num, den]= cheby2 (n, Rs, Wn, ‘high’)	
	[num, den]= cheby2 (n, Rs, [Wn1, Wn2], ‘stop’)	
	[num, den]= cheby2 (n, Rs, Wn, ‘s’)	
	[num, den]= cheby2 (n, Rs, Wn, [Wn1, Wn2], ‘s’)	
	[num, den]= cheby2 (n, Rs, Wn, ‘high’, ‘s’)	
	[num, den]= cheby2 (n, Rs, [Wn1, Wn2], ‘stop’, ‘s’)	
ellip	[num, den]=ellip (n, Rp, Wn)	elliptic digital or analog filter design
	[num, den]= ellip (n, Rp, Rs, [Wn1 Wn2])	
	[num, den]= ellip (n, Rp, Rs, ‘high’)	
	[num, den]= ellip (n, Rp, [Wn1, Wn2], ‘stop’)	
	[num, den]= ellip (n, Rp, Rs, ‘s’)	
	[num, den]= ellip (n, Rp, Rs, [Wn1, Wn2], ‘s’)	
	[num, den]= ellip (n, Rp, Rs, ‘high’, ‘s’)	
	[num, den]= ellip (n, Rp, [Wn1, Wn2], ‘stop’, ‘s’)	
remez	num=remez (n, F, M)	Parks-McClellan optimal digital FIR filter design
	num=remez (n, F, M, W)	
	num=remez (n, F, M, ‘Hilbert’)	
	num=remez (n, F, M, ‘differentiator’)	
yulewalk	[num, den]=yulewalk (n, F, M)	Yule-Walker digital IIR filter design

7.2.6 Frequency Domain Transformations

lp2bp	[numt, dent]=lp2bp (num, den, Wo, Bw)	lowpass to bandpass transformation
	[At, Bt, Ct, Dt]=lp2bp (A, B, C, D, Wo, Bw)	
lp2bps	[numt, dent]=lp2bp (num, den, Wo, Bw)	lowpass to bandstop transformation
	[At, Bt, Ct, Dt]=lp2bp (A, B, C, D, Wo, Bw)	
lp2hp	[numt, dent]=lp2hp (num, den, Wo)	lowpass to highpass transformation
	[At, Bt, Ct, Dt]=lp2hp (A, B, C, D, Wo)	
lp2lp	[numt, dent]=lp2lp (num, den, Wo)	lowpass to lowpass
	[At, Bt, Ct, Dt]=lp2lp (A, B, C, D, Wo)	
polystab	dent=polystab (den)	discrete-time stabilization

7.2.7 Fast Fourier Transformation

fft	fft (x) fft (x, n)	fast Fourier transform
fft2	fft2 (x) fft2 (x, nr, nc)	2-D fast Fourier transform
ifft	ifft (x) ifft (x, n)	inverse fast Fourier transform
ifft2	ifft2 (x) ifft2 (x, nr, nc)	2-D inverse fast Fourier transform

7.2.8 Window Functions

boxcar	boxcar (n)	boxcar or rectangular window
hamming	hamming (n)	Hamming window
hanning	hanning(n)	Hanning window
kaiser	kaiser (n, beta)	Kaiser window
triang	triang (n)	triangular window

7.2.9 Continuous to Discrete Conversion

bilinear	[Zd, Pd, Kd]=bilinear (Z, P, K, Fs) [numd, dend]=bilinear (num, den, Fs) [Ad, Bd, Cd, Dd]=bilinear (A, B, C, D, Fs) [Zd, Pd, Kd]=bilinear (Z, P, K, $F_{S,}$ Fp)	bilinear transformation
c2d	[Ad, Bd]=c2d (A, B, Ts)	
c2dm	Ad, Bd, Cd, Dd]=c2cm (A, B, C, Ts, ‘zoh’) [Ad, Bd, Cd, Dd]=c2cm (A, B, C, Ts, ‘foh’) [Ad, Bd, Cd, Dd]=c2cm (A, B, C, Ts, ‘tustin’) [Ad, Bd, Cd, Dd]=c2cm (A, B, C, Ts, ‘prewarp’) [Ad, Bd, Cd, Dd]=c2cm (A, B, C, Ts, ‘matched’) [numd, dend]=c2dm (num, den, Ts, ‘zoh’)	continuous to discrete conversion given a specific method

	[numd, dend]=c2dm (num, den, Ts, 'foh')	
	[numd, dend]=c2dm (num, den, Ts, 'tustin')	
	[numd, dend]=c2dm (num, den, Ts, 'prewarp')	
	[numd, dend]=c2dm (num, den, Ts, 'matched')	
impinvar	[numd, dend]=impinvar (num, den)	impulse invariant
	[numd, dend]=impinvar (num, den, Fs)	transformation

7.2.10 Linear System Representations

residue	[R, P, k]=residue (num,den)	continuous-time partial fraction expansion
	[num, den]=residue (R, P, k)	and inverse
residuez	[R, P, k]=residuez (num,den)	discrete-time partial fraction expansion
	[num, den]=residuez (R, P, k)	and inverse
sos2ss	[A, B, C, D]=sos2ss (sos)	second-order section to state-space
sos2tf	[num, den]=sos2tf (sos)	second-order section to transfer function
sos2zp	[Z, P, K]=sos2zp (sos)	second-order section to zero-pole
ss2sos	sos=ss2sos (A, B, C, D)	state-space to second-order section
	sos=ss2sos(A, B, C, D, iu)	
ss2tf	[Z, P, K]=ss2zp (A, B, C, D, iu)	state-space to transfer function
ss2zp	[Z, P, K]=ss2zp (A, B, C, D, iu)	state-space to zero-pole
tf2ss	[A, B, C, D] = tf2ss(num,den)	transfer function to state space
tf2zp	[Z, P, K] = tf2zp(num,den)	transfer function to zero-pole
zp2sos	sos = zp2sos(Z, P, K)	zero pole to second order section

8.0 DIGITAL FILTERS

Digital filters constitute one of the major applications of digital signal processing. Many operations such as bandlimiting of data signals, detection of specific frequencies and changing of sampling rate require filtering. Analog filter techniques have been used for many years, but digital filters have taken over many of these operations. We may begin by asking: "why use digital filters?" The answer is that they have a number of advantages:

1. Digital filters are much more stable over time, temperature and component variations than the analog types. Thus they are repeatable, stable and require no calibration. These factors are especially important for the more complex filters, say those above 6 or 8 poles, which are necessary for the extremely sharp rolloff rates required in some applications. Analog filters of that complexity tend to be very sensitive to component variations and thus are not very repeatable or stable.

2. The design process consists primarily of selecting and storing coefficient values to be used in later computations. Thus, modification of the filter characteristic is easy, and the filters can actually be adaptive - i.e. their characteristics can be modified "on the fly", in real time.

3. Efficient design processes have been developed, making the design of digital filters quite easy. Simulation of digital filters gives quite accurate results - significantly better than analog filters because of their sensitivity to component values.

4. Very efficient hardware has been developed specifically for digital filter design and other DSP operations. As we shall see, digital filters consist primarily of two operations: multiply and add. Computer architectures optimized for these two operations are readily available, at reasonable cost.

5. There are some filter characteristics, linear-phase for example, that can be achieved in digital filters, but not in analog filters.

To begin our study of digital filters, let us look at some basic filter concepts, which are applicable to both the analog and digital filter world. In a general sense, a filter is a system or circuit that modifies the amplitude or phase of a signal as a function of frequency. Although we normally think of a filter as affecting the amplitude only, there are cases in which the phase is also important. If the signal represents wide-band data, for example, it is likely that the phase of the signal is encoded with the information carried by the signal. Thus, if a filter distorts the phase, i.e. shifts the phase of some frequencies more than others, the encoded information will be degraded. The same is true for signals carrying audio frequencies, such as those from compact disks.

The concern about a filter's effect on phase is usually handled by deciding initially how important the phase is, and choosing a type of filter that gives the appropriate phase response. In many cases, phase is of no importance, so the filter is designed to give the best amplitude response, regardless of what happens to phase. In others, zero or uniform phase shift is required, and sometimes moderate phase distortion can be tolerated.

It is worth noting here that zero phase shift is, in general, not possible. However, phase shift that is ***linear with frequency*** is usually just as good, because it amounts to a ***constant time delay***. To see that we just need to note that:

$45°$ at 500 Hz = 1/8 cycle = 1/8 x 2 ms = 1/4 ms

$90°$ at 1000 Hz = 1/4 cycle = 1/4 x 1 ms = 1/4 ms

So the time delay is the same, and the signal will just be delayed, not distorted.

There are actually two types of phase response. One is the phase shift or phase delay, in degrees and radians, as discussed above. Ideally, this would be zero or linear with frequency. The second kind is group delay, which is the actual delay applied to the information content in a signal. It is defined as the derivative of the phase delay, so if the phase delay is a linear function of time, the group delay is a constant. Thus the information is all delayed by a constant amount of time, but with no delay distortion.

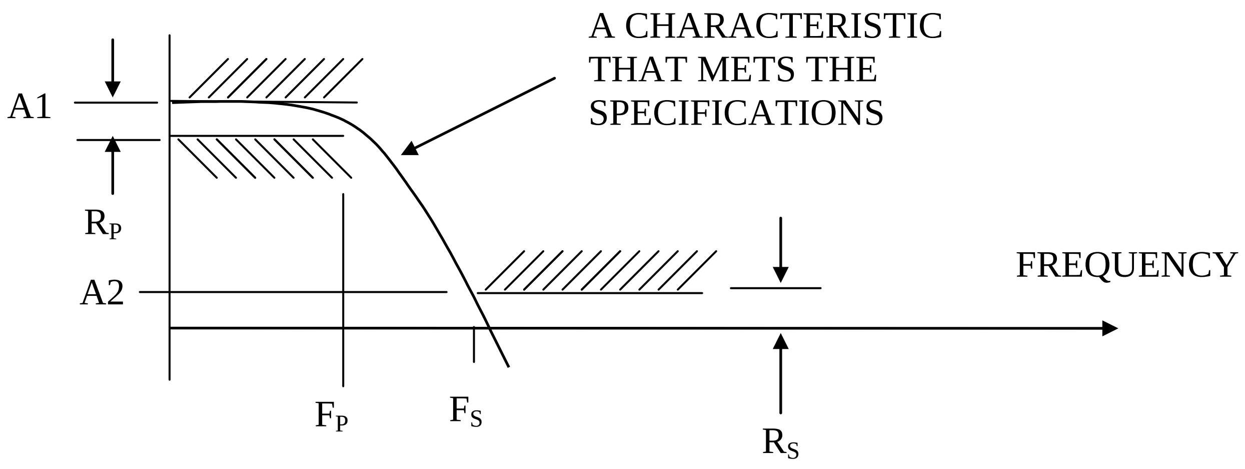
8.1 FILTER TYPES

Filters can be classified in a number of different ways. One is obviously the amplitude-vs-frequency characteristic, which leads to the general categories of low-pass, bandpass, high-pass and band-reject. In addition, there is the type of approximation used (realizing that all filters are only approximations to the ideal which is generally desired). These go by the names like Butterworth, Chebychev, Bessel, elliptic, etc. There are many of these, and they differ in their passband and stopband ripple characteristics, rate of attenuation between bands, etc. We will discuss them shortly. For analog filters, there is the additional characteristic of the circuit configuration, which doesn't apply in the same fashion to digital filters.

8.2 FILTER SPECIFICATIONS

In order to design a filter, we must first specify what we want it to do. The most common way to specify the filter requirements is to define one or more passbands and one or more stopbands. We then specify the nominal gain and tolerance in the passbands and the maximum gain in the stopbands. We can also have multiple passbands with different gain in each.

A typical specification for a low-pass filter's amplitude vs frequency characteristic would appear as follows:



Here the various parameters are identified as:

A1 is the maximum gain in the passband. It could also be specified as the nominal or average gain in that band. It is frequently normalized to one.

A2 is the maximum gain in the stopband. Ideally, it would be zero, but that isn't feasible, so a reasonable figure must be selected.

R_P is the allowable ripple in the passband. This may be specified initially as a percent or as an absolute value. There may not be any actual ripples, but the implication is that the response must be within the ripple band specified over the entire passband.

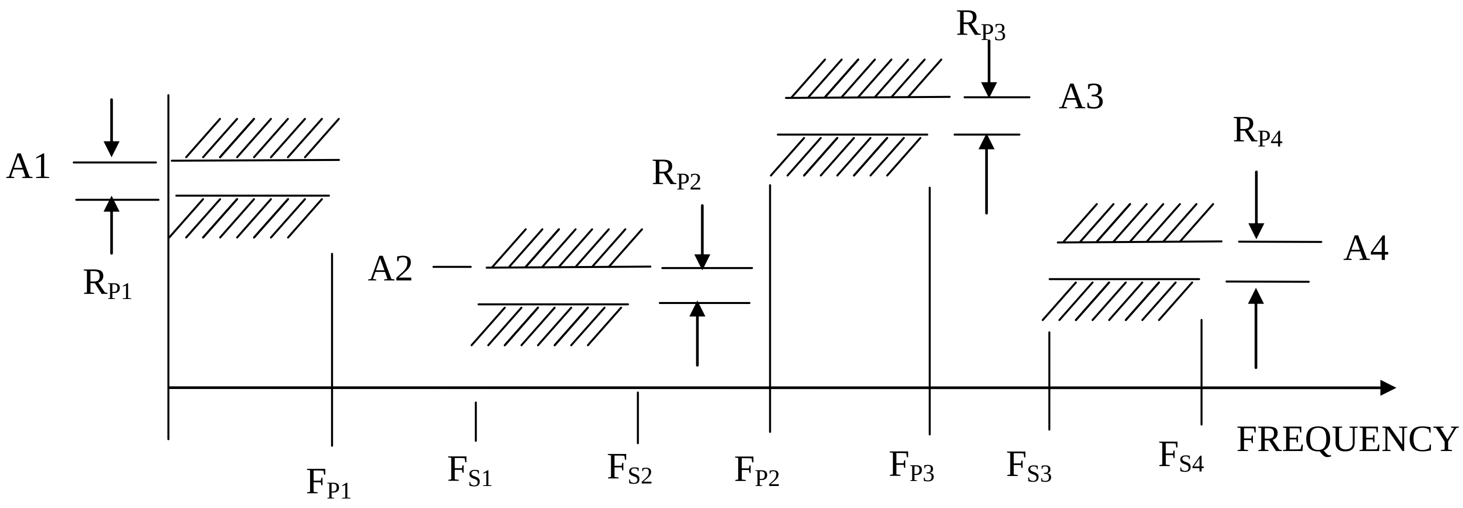
R_S is the allowable stopband response. It is typically specified as a maximum gain. The response may actually have ripples in it at frequencies above the edge of the stopband, as long as they do not rise above the limit imposed by R_S.

F_P is the edge of the passband – i.e. the frequency at which the passband amplitude specification is no longer met.

F_S is the edge of the stopband –i.e. the frequency at which the stopband specification must be met.

Note that the response in the transition regions outside the passbands and stopbands is not specified. This is not usually a problem, as the response usually varies smoothly from one band to another. However, it is possible for the out-of-band response to vary significantly from the specified bands, so checking it is worthwhile.

Other types of filter responses will require more complex specifications, such as that below:



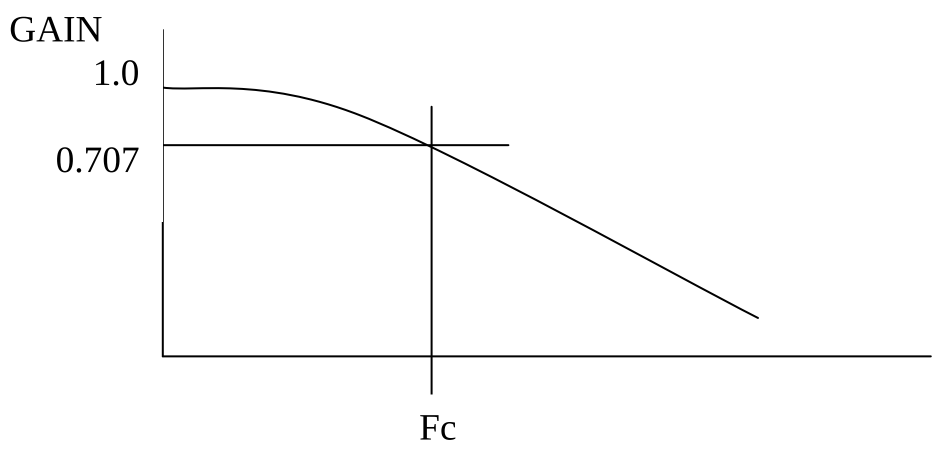
The general example shown has multiple passbands and stopbands. While not very common, it is certainly possible to design multiband digital filters. More likely, only certain of the bands would be used, resulting in lowpass, bandpass, highpass or bandstop filters. Note that each band has a nominal gain associated with it. In this example, the gain is shown as a maximum and the allowable ripple or variation in gain over the whole passband is shown as a peak-to-peak value. Sometimes, the gain is shown as an average or nominal gain, i.e. in the middle of the total ripple range, and the variation is shown as a plus/minus value above and below the nominal. Also, other notation is frequently used, such as δ for the ripple. Radian frequency is usually denoted as ω, but Matlab uses the letter W instead of the Greek ω as the dummy frequency variable.

<u>Filter implementations</u> Real filters are actually composed of a number of stages, either LC or op-amp sections in the analog world, and a sequence of computations in the digital. More stringent specifications require more stages. Since all stages are similar in complexity, the number of stages is a good measure of the complexity of the filter. Thus, it makes a good standard by which to compare filter designs.

8.3 FILTER APPROXIMATIONS

We mentioned earlier that there are a number of different filter approximations, and that they have different performance characteristics. It is worth reviewing them briefly, because some of the digital filter design techniques are based on transformations from the analog designs, and many of the characteristics of the analog designs are preserved in the transformation. Let us look at several of the popular filter approximations.

The ***Butterworth*** approximation is perhaps the most common. It has the significant feature that it is maximally flat in the passband. That means it stays closer to a normalized gain of 1 for a larger frequency range than any other. By definition, its response is down by 3 dB, to .707 at the edge of the passband. The characteristic is

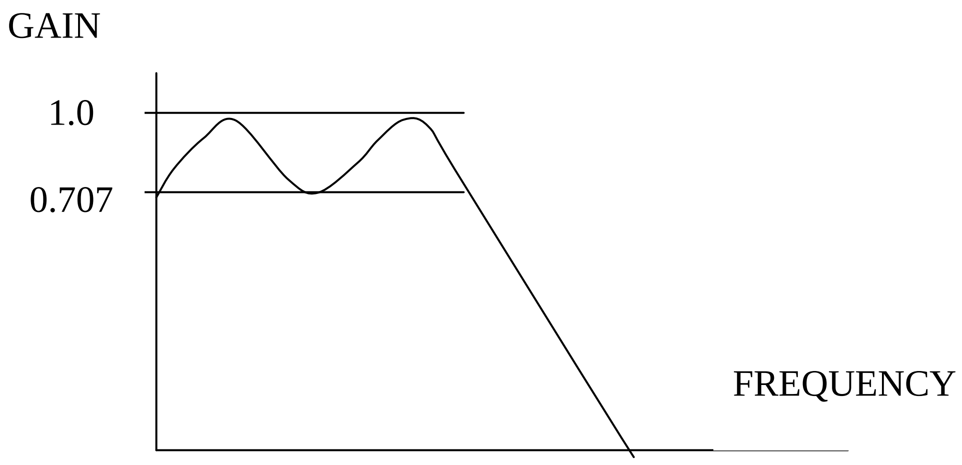


The relatively flat response over much of the passband and smooth rolloff are usually desirable features. The Butterworth filter has a moderate phase characteristic, but it is not good enough if zero or uniform phase response is required. It is a good general-purpose filter whose characteristics are frequently used.

The ***Chebyshev*** Approximation. There are really two of these. The Type I takes the approach of allowing ripples within the passband, instead of maximal flatness, and therefore needs fewer stages than Butterworth to get a given rolloff rate. The corner or cutoff frequency is usually specified in terms of the edge of the ripple band, i.e. the frequency at which the response falls outside the allowable ripple.

The Type II allows ripple in the stop band but not in the passband. It thus provides monotonic response in the passband, but the stopband response does not continue to drop with frequency as does the Type I.

The amount of ripple in the passband can be selected, and is usually in the range of 1 dB to 3 dB. The phase response of the Chebyshev varies significantly, and is worse than the Butterworth. A typical Type I , 3 dB Chebyshev response is



Elliptic Filters allow ripple in both the stopband and passband. They provide the steepest rolloff of all for a given order of filter. Their phase response is among the worst, however.

The ***Bessel-Thompson*** analog prototype filter has a substantially linear phase response, implying constant group delay, particularly for higher-order versions. It has a poor frequency response – worse even than the Butterworth - but may be the response of choice if linear phase response is necessary. The digital version, however, does not give the phase response of the analog prototype.

Fortunately, there are techniques for designing digital filters that do not depend on direct transformations from analog filters, and their responses can be better tailored to the requirements at hand.

8.4 DIGITAL FILTER DESIGN TECHNIQUES

We now take up the issue of actually designing digital filters. There are a number of different approaches. Some of them involve transformations from analog designs, while others apply only to digital designs. We will not cover them in enough detail to be able to do manual designs, but do need to cover some concepts and terminology used in computer-aided filter design. **NOTE** that if we know, through experience or intuition, that a certain filter will satisfy our requirements, we won't have to go through the step of specifying requirements and calculating the type and order of filter required. In fact, our first example will assume we know what filter we need, and we will then just calculate the coefficients needed to implement the chosen filter.

8.4.1 Transformation from analog systems

The first category of design techniques is based on transformations of continuous-time or analog systems into discrete-time designs. Since the continuous-time filters are generally **IIR** systems, the discrete-time systems based on them are also **IIR** systems.

The first **IIR** technique we will consider is ***impulse invariance*** design. This approach requires that a suitable analog design be completed first. The analog design would be based on the required filter response, including choice of approximation type to give the desired rolloff, phase response, etc. Then a transformation is made from the Laplace transform transfer function of the analog design to the z-transform system function for the digital version. With impulse invariant design, the transformation used is selected to preserve the impulse response of the analog design. Although we may not be interested specifically in the impulse response of the digital filter, it can be shown that its frequency response will closely match that of the underlying analog filter if the analog filter is sufficiently narrow that there is no aliasing in the digital version.

A similar technique can be used to duplicate the step response of the analog filter, where the time-domain response is important. This approach is called, logically enough, ***step-invariant*** design.

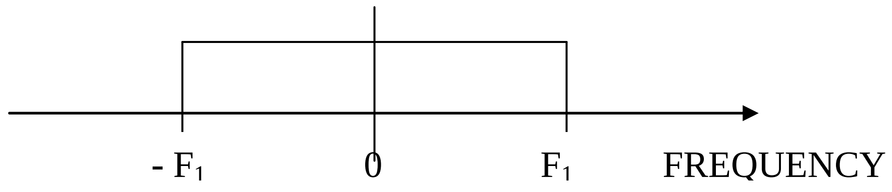
Another technique based on an analog filter prototype is the **bilinear transformation.** It is based on integrating the differential equations for the analog filter, then using a numerical approximation to design the digital version. The result provides a relatively simple transformation, accomplished by making the substitution

$$\mathbf{s = \frac{2\,(1 - z^{-1})}{T_d\,(1 + z^{-1})}}$$

where the T_d is a constant that cancels out in the transformation and its reverse, so is ignored where s is the Laplace transform variable in the transfer function for the analog filter. This transformation distorts the frequency axis substantially, so that the frequency response of the digital filter does not match that of the analog filter on which it is based. A pre-warping of the frequency axis can be done to make the digital filter response match that of the analog filter at selected frequencies, such as passband and/or stopband edges.

8.4.2 Windowing

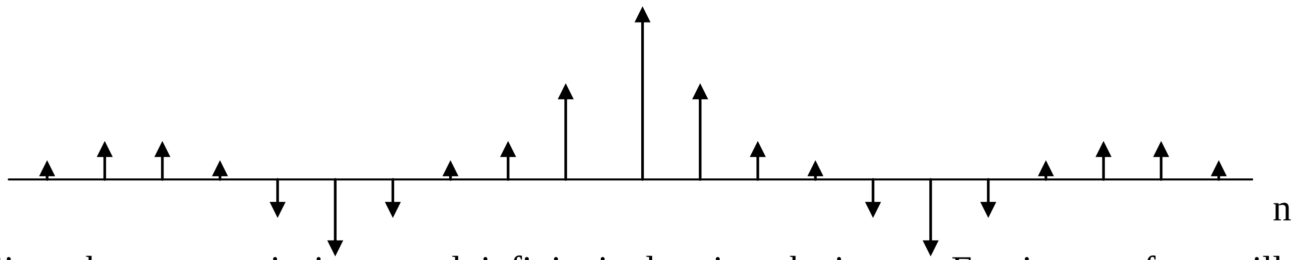
The second general category of digital filter design is not based on analog designs. Instead, it is based on direct approximations of the desired frequency response. The first of these is called the ***windowing*** approach. With this approach, a desired frequency response is first specified. For example, let us consider the ideal low-pass filter:



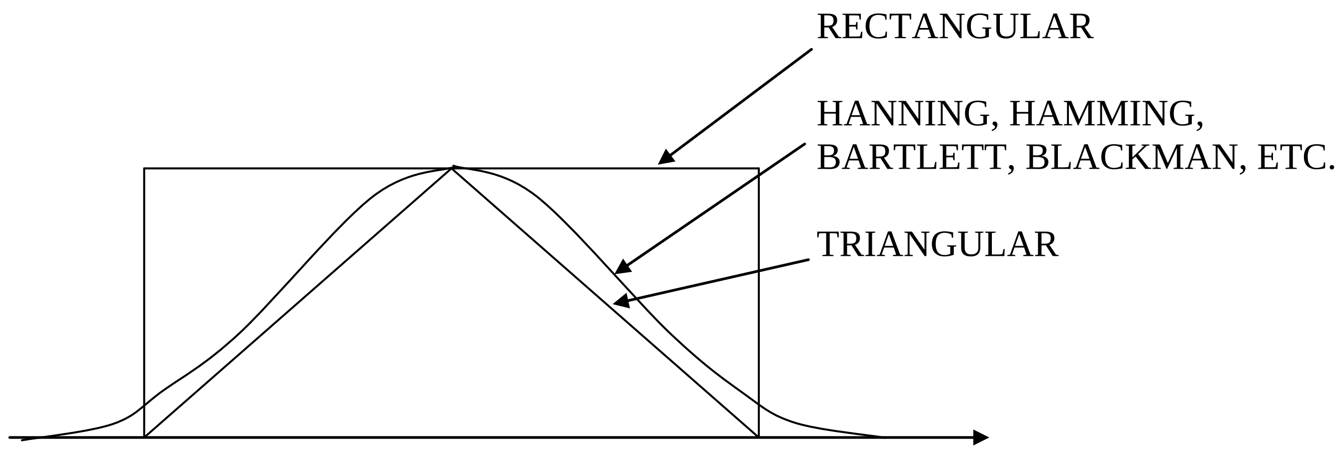
We saw earlier that the impulse response of a system is given by the inverse Fourier Transform of the frequency response. Consequently, we can use the inverse Discrete Fourier Transform, applied to the frequency response above, to find the impulse response of the system. For the rectangular frequency response shown above, the inverse Fourier transform would give the result that

$$h[n] = \frac{\sin(\pi f_1 n)}{\pi n} = h[0] + h[1] + h[2] + h[3] + \ldots + h[N-1]$$

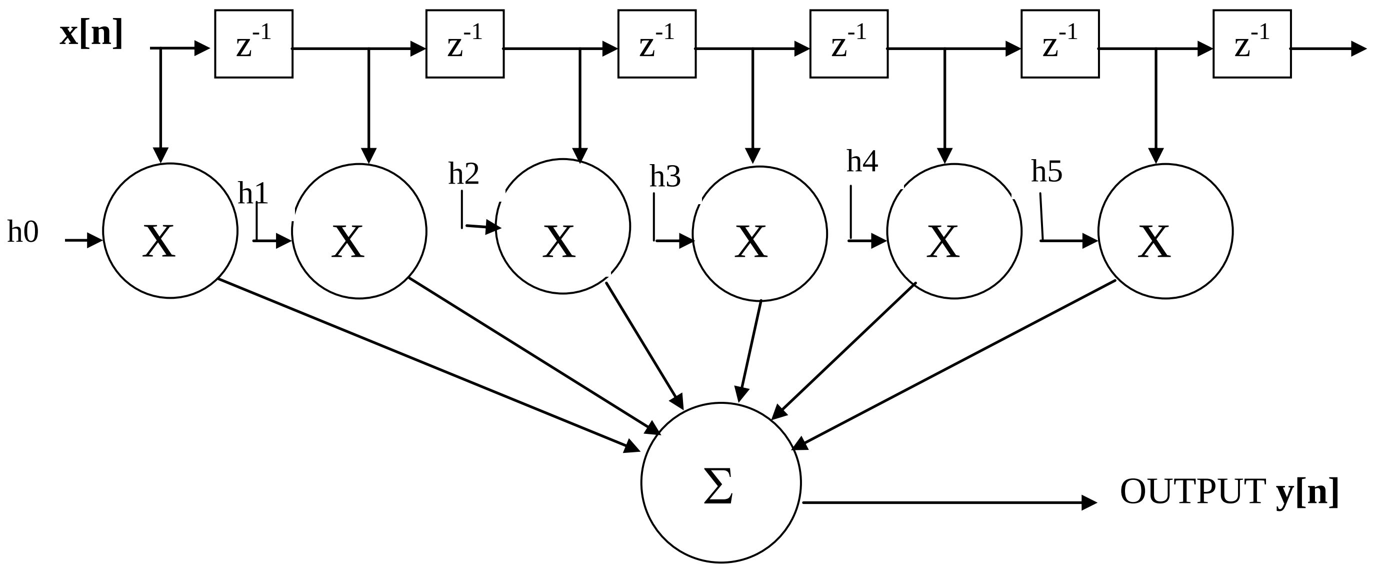
This is just a sequence of numbers which vary in polarity and magnitude. The sequence has infinite duration. A plot of the coefficient magnitudes has the appearance:



Since the response is, in general, infinite in duration, the inverse Fourier transform will not converge, i.e. it does not exist or cannot be evaluated. To get around this problem, the sequence of coefficients is multiplied by a window, or weighting function, which has magnitude of zero outside some selected range of n. Some typical windows are:



The process of design by windowing then, consists primarily of computing the sequence coefficients for the frequency response desired, computing the values or weights of the window function from the equation for the chosen window, then multiplying the two together to form a new set of coefficients. This set of coefficients then constitutes the sequence h[n] used in the implementation of the filter. A typical FIR filter architecture would be:

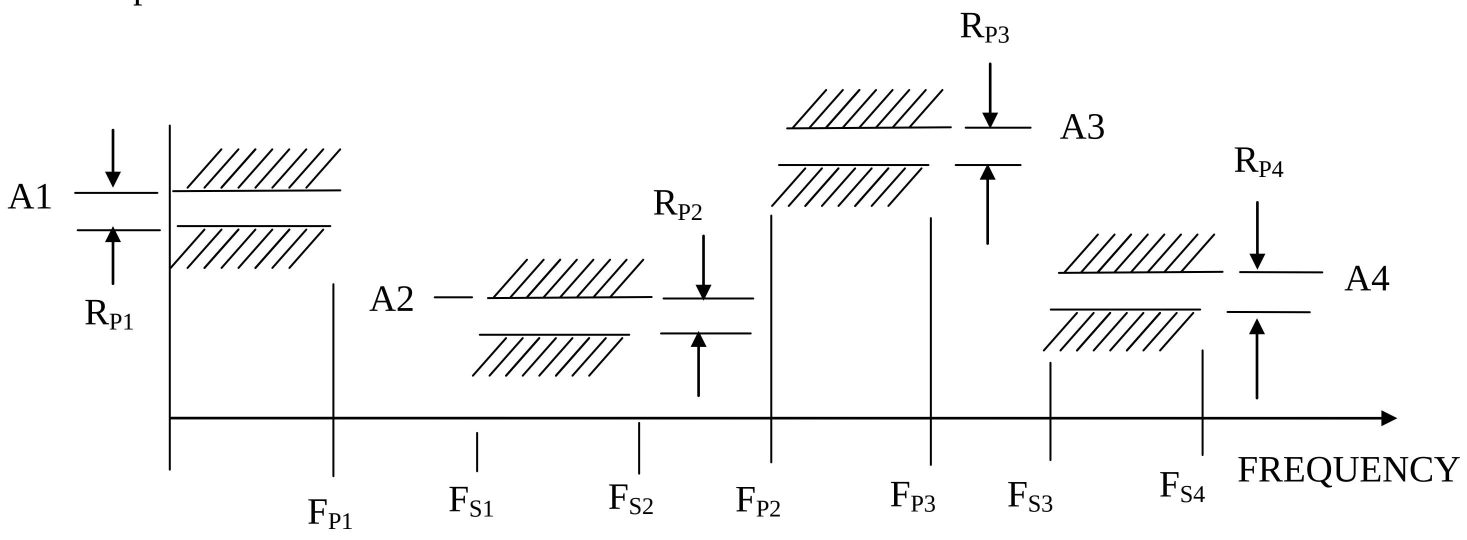


The coefficients h1, h2, etc. are the sequence values from the product of the impulse response and the window function. This sequence is the ultimate output of any FIR filter design - just a set of numbers which make up the weighting factors for the multipliers.

So, the implementation is quite straightforward, and consists primarily of delay elements and multipliers, with a final summation. The string of delay elements is sometimes compared to a tapped delay line.

A second approach to design of FIR filters is called the ***Parks-McClellan*** method, sometimes called the Remez Exchange algorithm, after the computational technique used. It is a technique that optimizes the filter parameters to minimize the maximum error between the actual filter and the desired response. It is an iterative method, in which a number of iterations are necessary to achieve the desired error.

The algorithm starts with a specification of the desired response. A typical desired response, although perhaps more general than we usually need, is represented by the example we used earlier:



The Parks-McClelland algorithm allows for several passbands, with different gains in each of them. It also allows different weighting factors for the error in each band.

The algorithm strives to minimize the maximum error within the bands, so it is said to use the ***minimax*** criterion.

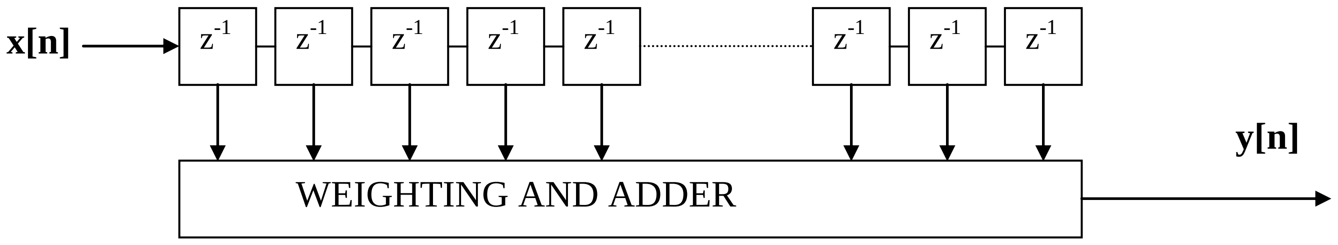
Filters designed by the window method or by the Remez Exchange method are FIR and therefore have linear phase response.

8.5 **<u>FIR vs IIR</u>** Previously we referred to the difference between Finite Impulse Response (**FIR**) and Infinite Impulse Response (**IIR**) systems. We noted that the basic difference is that the FIR response never actually dies out. For filters, the difference between the two types affects the filter characteristics.

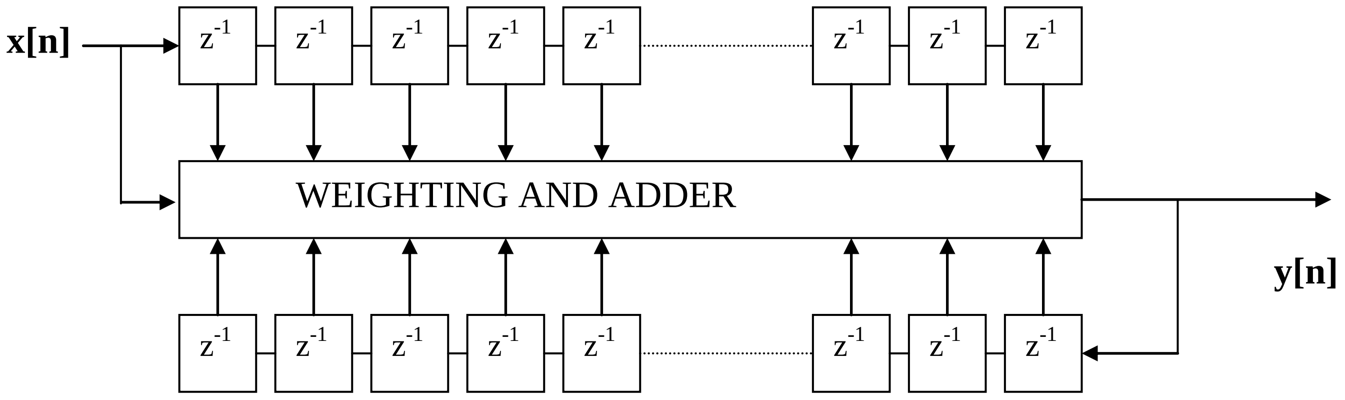
IIR filters are frequently derived from analog filter characteristics, which generally use feedback to achieve their performance. Thus**, IIR** digital filters also use feedback, with all the usual factors that feedback introduces. First, feedback introduces the possibility of oscillation, so stability is a concern. On the positive side, however, the feedback enhances the performance of the filter, so that a given filter response can be achieved with fewer stages (i.e. less computation) with an **IIR** filter than with a **FIR** one. Also, **IIR** filters cannot achieve linear or constant phase shift.

FIR filters, on the other hand, are inherently stable, as they have no feedback. They can provide linear phase shift, so there is no phase distortion on the signal. The biggest drawback to **FIR** filters is that they require substantially more stages than **IIR**, and thus more hardware and/or software, to satisfy a specific filter requirement. The tradeoff, then is simpler implementation for the **IIR** than for **FIR**, at the price of poor phase response and possible instability. **FIR** filters are necessary in many applications because their linear phase response is important.

Generalized flow diagrams for the two types also help to illustrate the difference. The general form for the **FIR** filter, which uses only delayed copies of the input is:



The **IIR** filter, on the other hand, uses delayed copies of the output as well as the input, and has the general appearance:



Note that the weighting for all the inputs to the adder has been omitted for clarity, but is included in the "weighting and adder" block.

9.0 DIGITAL FILTER DESIGN EXERCISES

Let us now go through a simple digital filter design. Instead of using the more general technique of defining the filter requirements, let us just specify a fourth-order Butterworth low-pass filter, with a “cutoff” or 3dB frequency of 1000 Hz and a 10kHz sampling rate. We will use the more general technique later. The specific function we need is:

```
[num,den]=butter(n,Wn)
```

If we don’t remember the format or syntax, we can just type: help butter

The result will be:

```
 BUTTER    Butterworth digital filter design.
        [B,A] = BUTTER(N,Wn) designs an N'th order lowpass digital
        Butterworth filter and returns the filter coefficients in length
        N+1 vectors B and A.  The cut-off frequency Wn must be
        0.0 < Wn < 1.0, with 1.0 corresponding to half the sample rate.

        If Wn is a two-element vector, Wn = [W1 W2], BUTTER returns an
        order 2N bandpass filter with passband  W1 < W < W2.
        [B,A] = BUTTER(N,Wn,'high') designs a highpass filter.
        [B,A] = BUTTER(N,Wn,'stop') is a bandstop filter if Wn = [W1 W2].

        When used with three left-hand arguments, as in
        [Z,P,K] = BUTTER(...), the zeros and poles are returned in
        length N column vectors Z and P, and the gain in scalar K.

        When used with four left-hand arguments, as in
        [A,B,C,D] = BUTTER(...), state-space matrices are returned.

        See also BUTTORD, CHEBY1, CHEBY2, FREQZ and FILTER.

»
```

From the help file we can see that the function returns the numerator coefficients and denominator coefficients in vectors B and A respectively. Note that Wn is the cutoff frequency, normalized to half the sampling frequency - in this example it is 1000 Hz normalized to 10000/2 Hz.

We set up the parameters as follows:
% set sampling frequency of 10 kHz

fs = 10000

% normalize the 3dB frequency to half of fs

wn = 1000/(fs/2)

Let us now execute the instruction

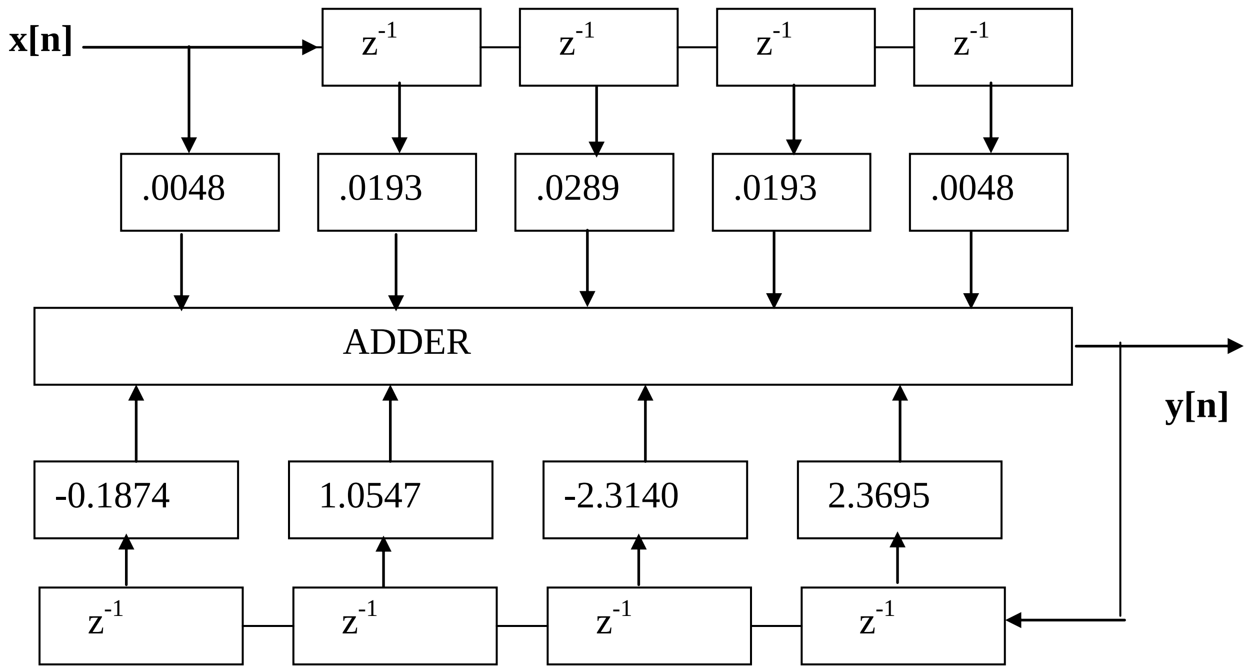
```
>>[b,a] = butter(4,wn)
```

Matlab responds with:

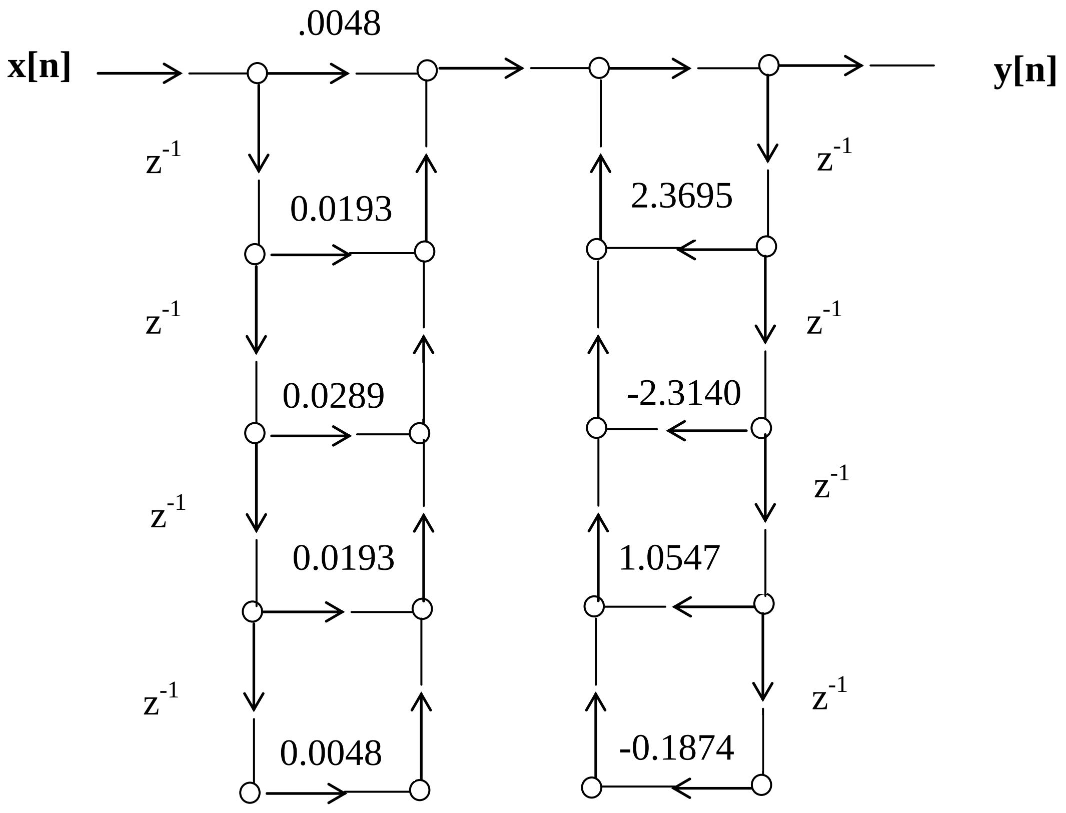
```
b =    0.0048    0.0193    0. 0289    0.0193    0.0048


a =    1.0000   -2.3695    2.3140   -1.0547    0.1874
```

Note that Matlab returns the contents of vectors b and a, which constitute the numerator and denominator coefficients of the transfer function for the digital filter. Recall that the numerator represents the forward path (the delayed and weighted values of x[n]), and the denominator represents 1-(reverse path), i.e. 1 - (the delayed and weighted values of y[n])

This means, then, that the block diagram for this filter is:

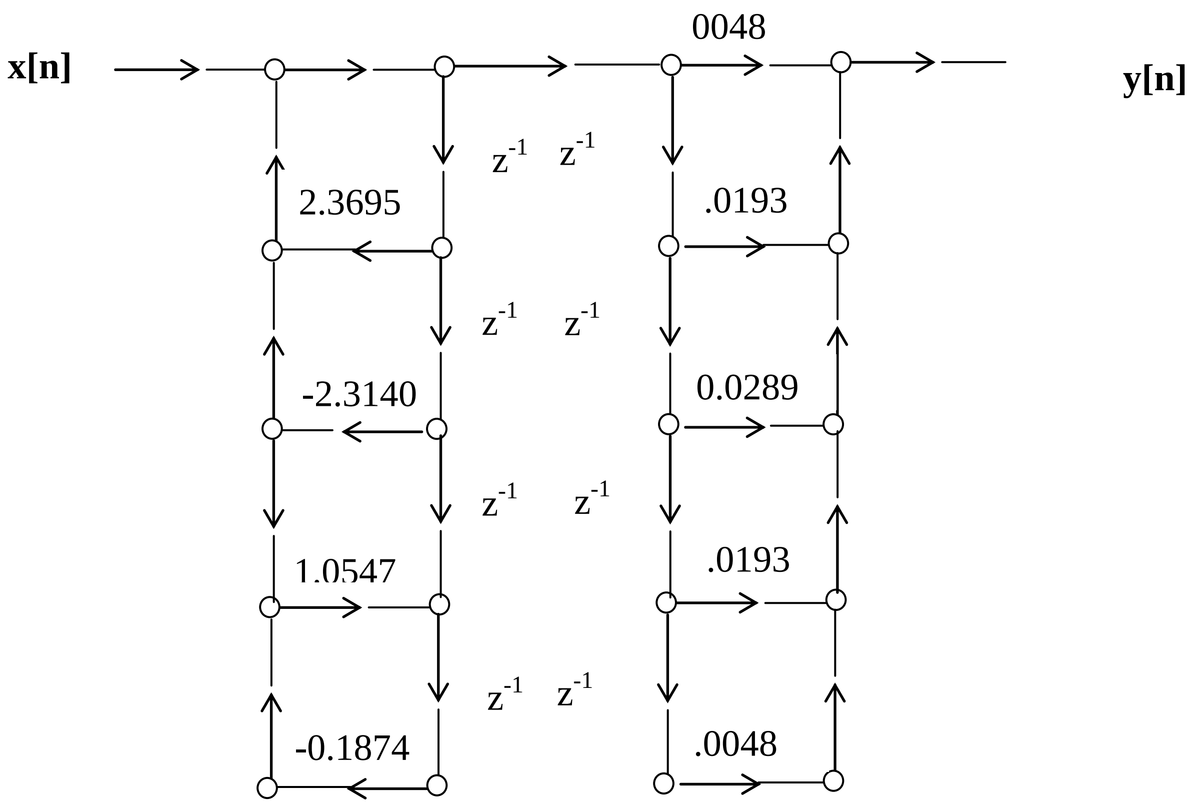


From this form we can easily see how to implement this filter: just delay inputs and outputs, multiply by the appropriate weighting factors, and sum the result.
In practice, the architecture above will probably not be used because it requires twice as many delay elements as is necessary. This is more evident if we represent the process by a signal flow graph, from section 5.5. The block diagram format we just developed can just as well be represented by a signal flow diagram, as below.

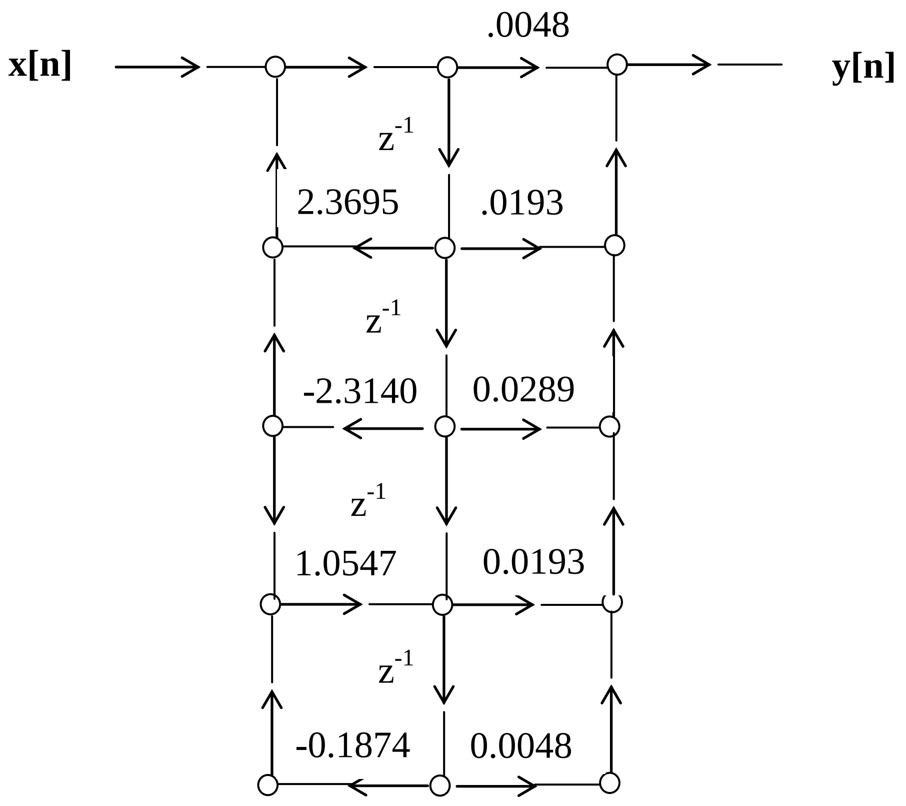


Note that all the same computations - multiplication and addition - are performed by both versions of the system representation. Recall that branches with no number on them represent a gain of 1.

It is apparent from the flow diagram that there are two sets of delay blocks, one for the inputs and one for the outputs. This is called a ***direct form I realization***. Since the system is assumed to be linear, we can reverse the two sections, getting a flow diagram like that shown below



The two sets of delays can be combined into a ***direct form II*** as follows:



It isn't quite as easy to verify the operation of the direct form II, but since it was developed from valid principles, it is a valid realization of the filter, and can be used to perform the computations. In fact, this is the form that Matlab uses.

To see the results of our design, let us use the Matlab frequency response function followed by the plot function. The frequency response function for digital filters is [H,W] = freqz(num,den). For the syntax, let us again call on the Matlab help facility, by typing

```
» help freqz

 FREQZ      Z-transform digital filter frequency response.
        When N is an integer, [H,W] = FREQZ(B,A,N) returns
        the N-point frequency vector W and the
        N-point complex frequency response vector G of the filter B/A:
                                      -1                  -nb
             jw  B(z)   b(1) + b(2)z  + .... + b(nb+1)z
          H(e) = ----  = -----------------------------------
                                    -1                 -na
               A(z)    1   + a(2)z + .... + a(na+1)z
        given numerator and denominator coefficients in vectors B and A. The
        frequency response is evaluated at N points equally spaced around the
        upper half of the unit circle. To plot magnitude and phase of a filter:
             [h,w] = freqz(b,a,n);
             mag = abs(h);  phase = angle(h);
             semilogy(w,mag), plot(w,phase)
        FREQZ(B,A,N,'whole') uses N points around the whole unit circle.
        FREQZ(B,A,W) returns the frequency response at frequencies designated
        in vector W, normally between 0 and pi. (See LOGSPACE to generate W).
        See also YULEWALK, FILTER, FFT, INVFREQZ, and FREQS.

»
```

From this, we can see that we want to use for form [H,W]=freqz(b,a,n).

Using the vectors a and b we developed above, and specifying a 128 point frequency vector, we can then compute the frequency response of the filter with:

```
[h,w]=freqz(b,a,128)
```

Matlab responds with a list of values. You can suppress the list by terminating the command with a semicolon (;):

```
h =
1.0000                    0.0041 + 0.0036i          0.0001 + 0.0000i
   0.9795 - 0.2013i       0.0037 + 0.0031i          0.0001 + 0.0000i
   0.9186 - 0.3953i       0.0034 + 0.0027i          0.0001 + 0.0000i
   0.8183 - 0.5748i       0.0031 + 0.0024i          0.0000 + 0.0000i
   0.6807 - 0.7325i       0.0028 + 0.0021i          0.0000 + 0.0000i
   0.5087 - 0.8607i       0.0026 + 0.0018i          0.0000 + 0.0000i
   0.3058 - 0.9509i       0.0023 + 0.0016i          0.0000 + 0.0000i
   0.0775 - 0.9932i       0.0021 + 0.0014i          0.0000 + 0.0000i
  -0.1670 - 0.9748i       0.0019 + 0.0013i          0.0000 + 0.0000i
  -0.4098 - 0.8817i       0.0018 + 0.0011i          0.0000 + 0.0000i
  -0.6190 - 0.7055i       0.0016 + 0.0010i          0.0000 + 0.0000i
  -0.7505 - 0.4589i       0.0015 + 0.0009i          0.0000 + 0.0000i
  -0.7697 - 0.1889i       0.0013 + 0.0008i          0.0000 + 0.0000i
  -0.6833 + 0.0398i       0.0012 + 0.0007i          0.0000 + 0.0000i
  -0.5389 + 0.1864i       0.0011 + 0.0006i          0.0000 + 0.0000i
  -0.3892 + 0.2525i       0.0010 + 0.0005i          0.0000 + 0.0000i
  -0.2639 + 0.2635i       0.0009 + 0.0005i          0.0000 + 0.0000i
  -0.1705 + 0.2456i       0.0008 + 0.0004i          0.0000 + 0.0000i
  -0.1052 + 0.2160i       0.0008 + 0.0004i          0.0000 + 0.0000i
  -0.0611 + 0.1842i       0.0007 + 0.0003i          0.0000 + 0.0000i
  -0.0320 + 0.1546i       0.0006 + 0.0003i          0.0000 + 0.0000i
  -0.0131 + 0.1286i       0.0006 + 0.0003i          0.0000 + 0.0000i
  -0.0012 + 0.1067i       0.0005 + 0.0002i          0.0000 + 0.0000i
   0.0062 + 0.0884i       0.0005 + 0.0002i          0.0000 + 0.0000i
   0.0106 + 0.0734i       0.0004 + 0.0002i          0.0000 + 0.0000i
   0.0130 + 0.0610i       0.0004 + 0.0002i          0.0000 + 0.0000i
   0.0141 + 0.0508i       0.0004 + 0.0001i          0.0000 + 0.0000i
   0.0144 + 0.0425i       0.0003 + 0.0001i          0.0000 + 0.0000i
   0.0142 + 0.0357i       0.0003 + 0.0001i          0.0000 + 0.0000i
   0.0136 + 0.0300i       0.0003 + 0.0001i          0.0000 + 0.0000i
   0.0129 + 0.0253i       0.0002 + 0.0001i          0.0000 + 0.0000i
   0.0120 + 0.0215i       0.0002 + 0.0001i          0.0000 + 0.0000i
   0.0112 + 0.0182i       0.0002 + 0.0001i          0.0000 + 0.0000i
   0.0103 + 0.0155i       0.0002 + 0.0001i          0.0000 + 0.0000i
   0.0095 + 0.0133i       0.0002 + 0.0001i          0.0000 + 0.0000i
   0.0087 + 0.0114i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0079 + 0.0098i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0072 + 0.0084i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0066 + 0.0073i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0060 + 0.0063i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0055 + 0.0054i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0050 + 0.0047i       0.0001 + 0.0000i          0.0000 + 0.0000i
   0.0045 + 0.0041i       0.0001 + 0.0000i
```

```
w =
         0        0.7854         1.5953         2.4053
    0.0245        0.8099         1.6199         2.4298
    0.0491        0.8345         1.6444         2.4544
    0.0736        0.8590         1.6690         2.4789
    0.0982        0.8836         1.6935         2.5035
    0.1227        0.9081         1.7181         2.5280
    0.1473        0.9327         1.7426         2.5525
    0.1718        0.9572         1.7671         2.5771
    0.1963        0.9817         1.7917         2.6016
    0.2209        1.0063         1.8162         2.6262
    0.2454        1.0308         1.8408         2.6507
    0.2700        1.0554         1.8653         2.6753
    0.2945        1.0799         1.8899         2.6998
    0.3191        1.1045         1.9144         2.7243
    0.3436        1.1290         1.9390         2.7489
    0.3682        1.1536         1.9635         2.7734
    0.3927        1.1781         1.9880         2.7980
    0.4172        1.2026         2.0126         2.8225
    0.4418        1.2272         2.0371         2.8471
    0.4663        1.2517         2.0617         2.8716
    0.4909        1.2763         2.0862         2.8962
    0.5154        1.3008         2.1108         2.9207
    0.5400        1.3254         2.1353         2.9452
    0.5645        1.3499         2.1598         2.9698
    0.5890        1.3744         2.1844         2.9943
    0.6136        1.3990         2.2089         3.0189
    0.6381        1.4235         2.2335         3.0434
    0.6627        1.4481         2.2580         3.0680
    0.6872        1.4726         2.2826         3.0925
    0.7118        1.4972         2.3071         3.1170
    0.7363        1.5217         2.3317
    0.7609        1.5463         2.3562
                  1.5708         2.3807
>>
```

To view the actual response, we can plot the data, h (the response) vs w (the frequency). Since the magnitude of the response is complex, we first have to find the magnitude of h. The function involved is abs(x), and we can get its syntax by using Matlab help on abs

```
ABS   Absolute value and string to numeric conversion.
      ABS(X) is the absolute value of the elements of X. When
      X is complex, ABS(X) is the complex modulus (magnitude) of
      the elements of X.

      See also ANGLE, UNWRAP.

      ABS(S), where S is a MATLAB string variable, returns the
      numeric values of the ASCII characters in the string.
      It does not change the internal representation, only the
      way it prints.
      See also SETSTR.:
```

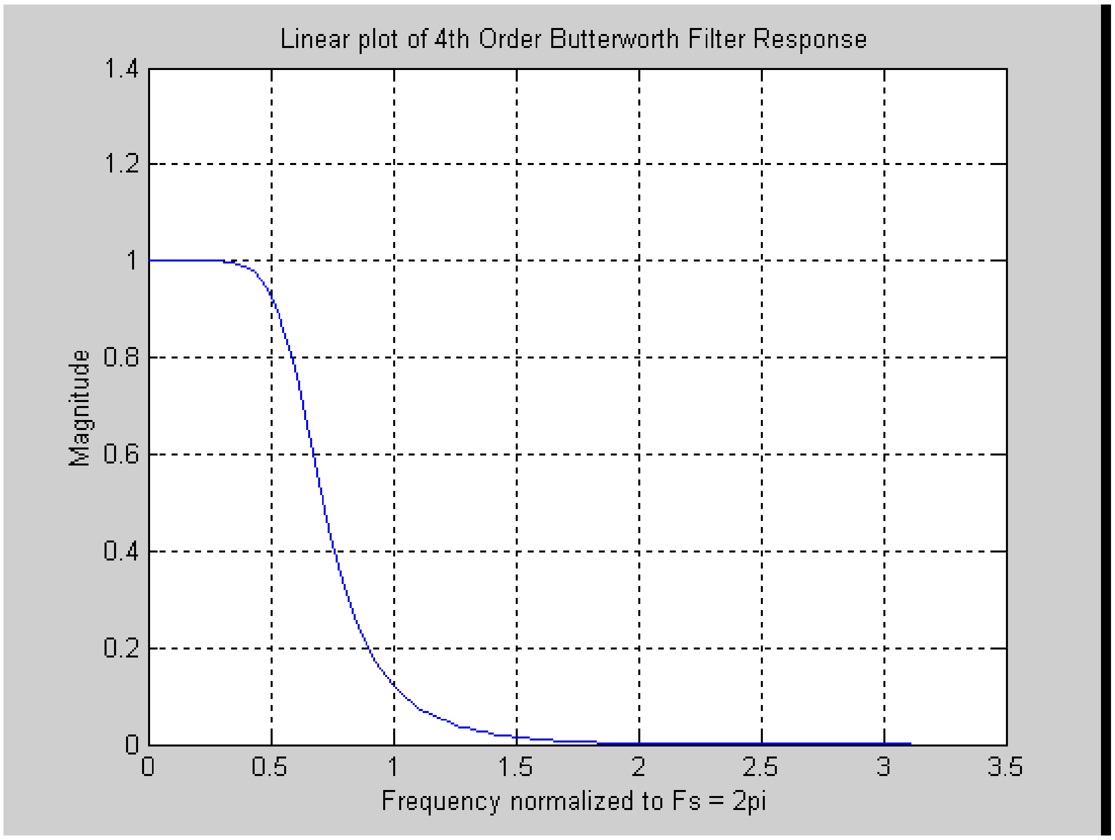
For our application, the form is abs(h), and executing that function, we get:
mag=abs(h)

```
mag =
1.0000              0.1067              0.0049
   1.0000           0.0886              0.0044
   1.0000           0.0741              0.0039
   1.0000           0.0624              0.0035
   1.0000           0.0528              0.0032
   0.9997           0.0449              0.0028
   0.9989           0.0384              0.0026
   0.9962           0.0329              0.0023
   0.9890           0.0284              0.0021
   0.9723           0.0246              0.0019
   0.9386           0.0214              0.0017
   0.8796           0.0186              0.0015
   0.7926           0.0163              0.0014
   0.6845           0.0143              0.0013
   0.5703           0.0126              0.0011
   0.4639           0.0111              0.0010
   0.3730           0.0098              0.0009
   0.2990           0.0087              0.0008
   0.2402           0.0077              0.0008
   0.1941           0.0069              0.0007
   0.1578           0.0061              0.0006
   0.1293           0.0055              0.0006
   0.0005                      0.0001                      0.0000
   0.0005                      0.0001                      0.0000
   0.0004                      0.0000                      0.0000
   0.0004                      0.0000                      0.0000
   0.0003                      0.0000                      0.0000
   0.0003                      0.0000                      0.0000
   0.0003                      0.0000                      0.0000
   0.0003                      0.0000                      0.0000
   0.0002                      0.0000                      0.0000
   0.0002                      0.0000                      0.0000
   0.0002                      0.0000                      0.0000
   0.0002                      0.0000                      0.0000
   0.0002                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000                      0.0000
   0.0001                      0.0000
```

We can find the phase angle of the filter by using the function phase=angle(h), with the result:

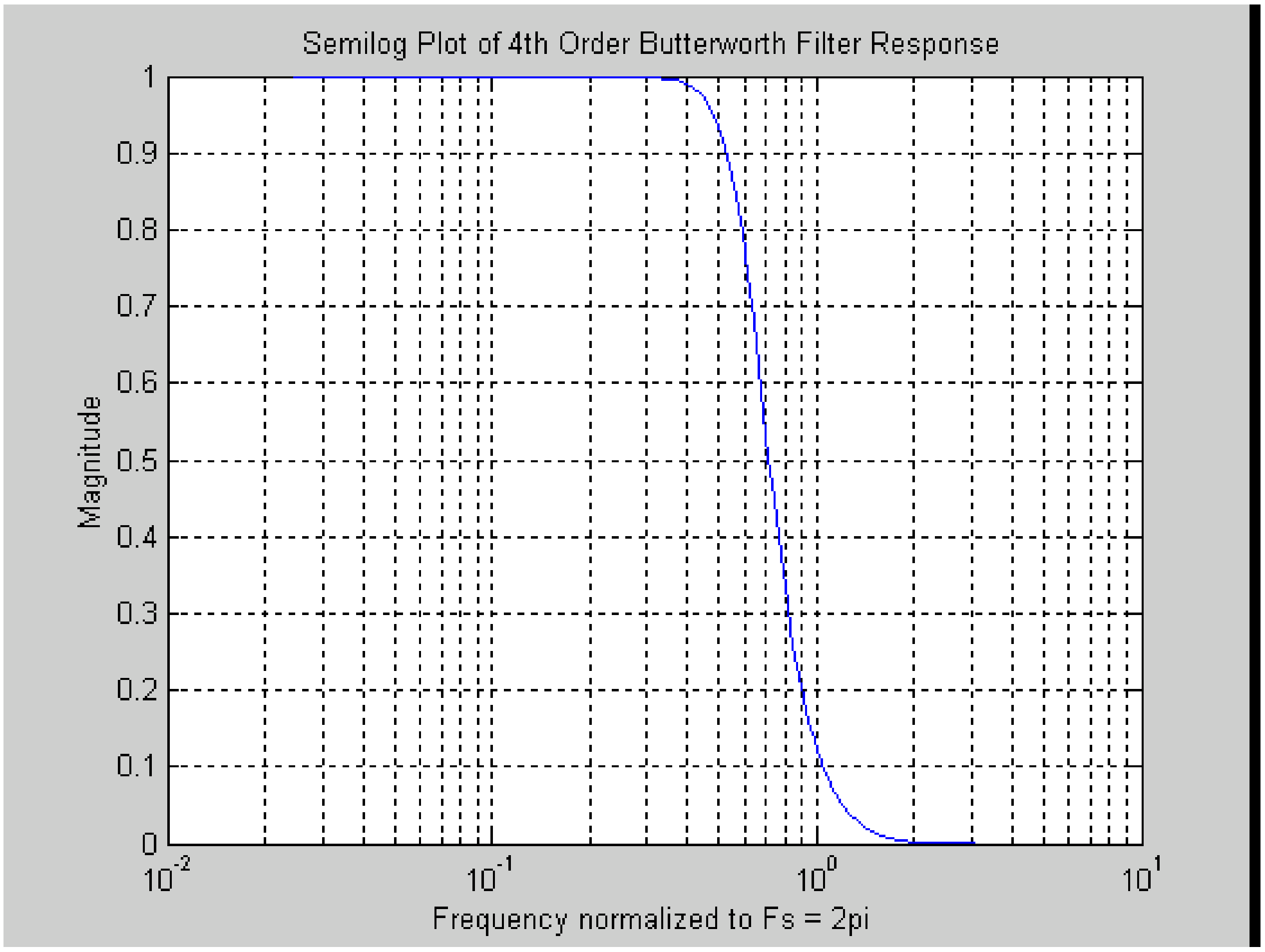
```
» phase=angle(h)

phase =             -1.5840        2.4188        1.3590
        0           -1.7090        2.3123        1.3200
  -0.0987           -1.8389        2.2139        1.2826
  -0.1976           -1.9742        2.1227        1.2469
  -0.2967           -2.1152        2.0383        1.2126
  -0.3964           -2.2618        1.9598        1.1796
  -0.4966           -2.4138        1.8868        1.1479
  -0.5976           -2.5702        1.8187        1.1174
  -0.6996           -2.7294        1.7549        1.0880
  -0.8028           -2.8895        1.6950        1.0596
  -0.9074           -3.0481        1.6388        1.0321
  -1.0137            3.0803        1.5857        1.0056
  -1.1221            2.9315        1.5355        0.9799
  -1.2329            2.7903        1.4880        0.9549
  -1.3464            2.6575        1.4429        0.9308
  -1.4633            2.5337        1.3999        0.9073
```

```
    0.8845          0.5711          0.3403          0.1474
    0.8623          0.5559          0.3282          0.1367
    0.8408          0.5409          0.3162          0.1260
    0.8198          0.5261          0.3044          0.1153
    0.7993          0.5116          0.2926          0.1047
    0.7793          0.4974          0.2809          0.0942
    0.7598          0.4833          0.2694          0.0836
    0.7408          0.4694          0.2579          0.0731
    0.7222          0.4558          0.2465          0.0626
    0.7040          0.4423          0.2352          0.0522
    0.6862          0.4290          0.2240          0.0417
    0.6687          0.4159          0.2129          0.0313
    0.6517          0.4029          0.2018          0.0208
    0.6349          0.3901          0.1908          0.0104
    0.6185          0.3775          0.1799
    0.6024          0.3649          0.1690
    0.5866          0.3526          0.1582
>>
```
Then we can plot the response vs frequency with the command plot(w,mag). The resultant plot is:



Note that you can add labels and titles with the commands xlabel, ylabel and title.

We can also plot with semilog or log-log axes. For example if we use the command semilogx(w,mag), xlabel('Frequency normalized to pi'), ylabel('Magnitude'), title('Semilog plot of 4th Order Butterworth Filter Response'),grid, the following plot results:



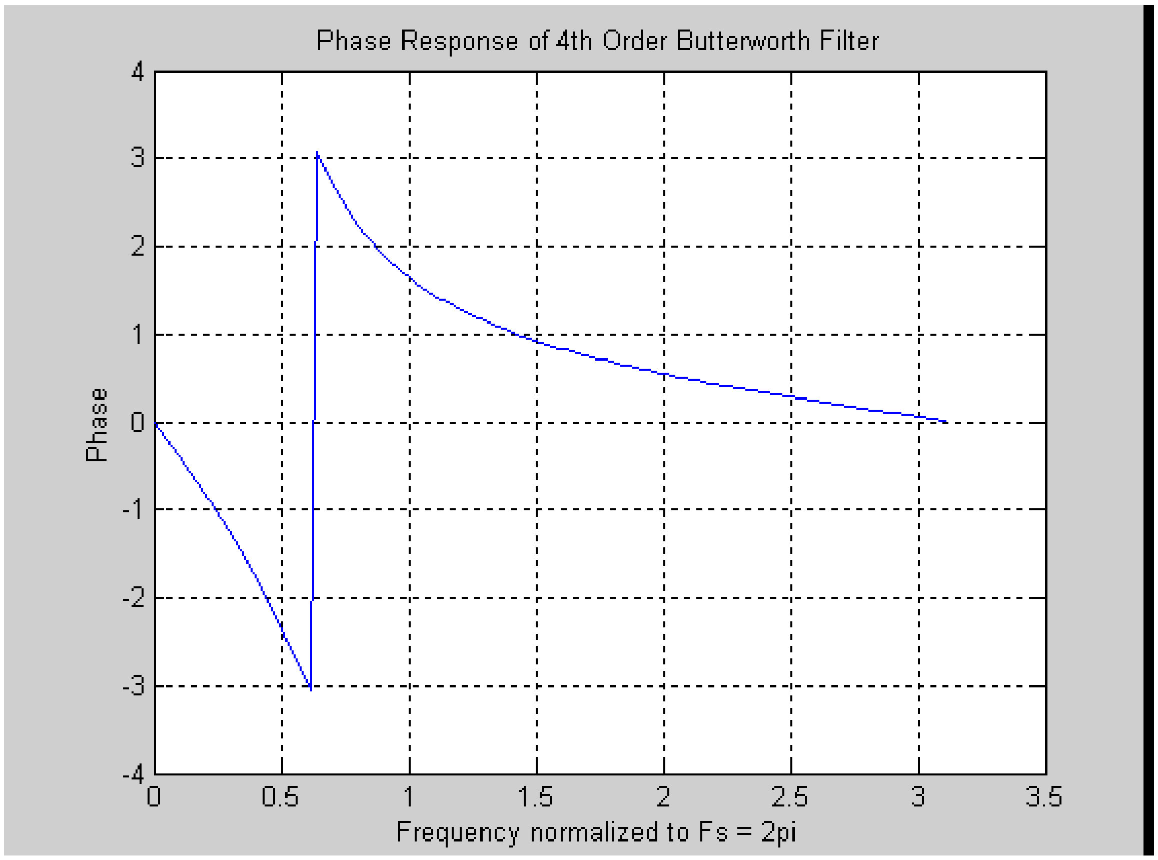
To plot the phase, we have to explicitly compute the phase by:

```
phase = angle(h)
```

And Matlab returns the phase figures:

```
Phase =
      0           -0.9074        -1.9742         2.9315
  -0.0987         -1.0137        -2.1152         2.7903
  -0.1976         -1.1221        -2.2618         2.6575
  -0.2967         -1.2329        -2.4138         2.5337
  -0.3964         -1.3464        -2.5702         2.4188
  -0.4966         -1.4633        -2.7294         2.3123
  -0.5976         -1.5840        -2.8895         2.2139
  -0.6996         -1.7090        -3.0481         2.1227
  -0.8028         -1.8389         3.0803         2.0383
```

```
1.9598          0.9799          0.5409          0.2465
1.8868          0.9549          0.5261          0.2352
1.8187          0.9308          0.5116          0.2240
1.7549          0.9073          0.4974          0.2129
1.6950          0.8845          0.4833          0.2018
1.6388          0.8623          0.4694          0.1908
1.5857          0.8408          0.4558          0.1799
1.5355          0.8198          0.4423          0.1690
1.4880          0.7993          0.4290          0.1582
1.4429          0.7793          0.4159          0.1474
1.3999          0.7598          0.4029          0.1367
1.3590          0.7408          0.3901          0.1260
1.3200          0.7222          0.3775          0.1153
1.2826          0.7040          0.3649          0.1047
1.2469          0.6862          0.3526          0.0942
1.2126          0.6687          0.3403          0.0836
1.1796          0.6517          0.3282          0.0731
1.1479          0.6349          0.3162          0.0626
1.1174          0.6185          0.3044          0.0522
1.0880          0.6024          0.2926          0.0417
1.0596          0.5866          0.2809          0.0313
1.0321          0.5711          0.2694          0.0208
1.0056          0.5559          0.2579          0.0104
```
: We can plot phase with the command: plot(w,phase), grid



The discontinuity at approximately 0.6 implies that the phase delay exceeded π, and the total shift is nearly 2π.

We have now gone through an entire design for a digital filter. Let us now use Matlab to see what the effect of this filter is - in effect we will generate a signal and pass it through the filter.

Let us create a signal that has three components: 500Hz, inside the passband of the filter, and 2000 and 3000 Hz, outside the passband. Recalling that the % indicates a comment in Matlab, we would set up the filter test as follows:

```
% Since we have only used normalized frequencies so far, we will have to declare the
sampling frequency explicitly to plot a waveform as a function of time.

fs = 10000  (actually, we already specified this earlier)

%  set up a time vector of 128 points at the sampling rate

t = (1:128)/fs
```

Matlab will print out a vector of 128 frequencies at which we will plot

```
t =

  Columns 1 through 7

    0.0001    0.0002    0.0003    0.0004    0.0005    0.0006    0.0007

  Columns 8 through 14

    0.0008    0.0009    0.0010    0.0011    0.0012    0.0013    0.0014

  Columns 15 through 21

    0.0015    0.0016    0.0017    0.0018    0.0019    0.0020    0.0021

  Columns 22 through 28

    0.0022    0.0023    0.0024    0.0025    0.0026    0.0027    0.0028
```

```
Columns 29 through 35

  0.0029    0.0030    0.0031    0.0032    0.0033    0.0034    0.0035

Columns 36 through 42

  0.0036    0.0037    0.0038    0.0039    0.0040    0.0041    0.0042

Columns 43 through 49

  0.0043    0.0044    0.0045    0.0046    0.0047    0.0048    0.0049

Columns 50 through 56

  0.0050    0.0051    0.0052    0.0053    0.0054    0.0055    0.0056

Columns 57 through 63

  0.0057    0.0058    0.0059    0.0060    0.0061    0.0062    0.0063

Columns 64 through 70

  0.0064    0.0065    0.0066    0.0067    0.0068    0.0069    0.0070

Columns 71 through 77

  0.0071    0.0072    0.0073    0.0074    0.0075    0.0076    0.0077

Columns 78 through 84

  0.0078    0.0079    0.0080    0.0081    0.0082    0.0083    0.0084

Columns 85 through 91

  0.0085    0.0086    0.0087    0.0088    0.0089    0.0090    0.0091

Columns 92 through 98

  0.0092    0.0093    0.0094    0.0095    0.0096    0.0097    0.0098

Columns 99 through 105
```

```
   0.0099    0.0100    0.0101    0.0102    0.0103    0.0104    0.0105

  Columns 106 through 112

   0.0106    0.0107    0.0108    0.0109    0.0110    0.0111    0.0112

  Columns 113 through 119

   0.0113    0.0114    0.0115    0.0116    0.0117    0.0118    0.0119

  Columns 120 through 126

   0.0120    0.0121    0.0122    0.0123    0.0124    0.0125    0.0126

  Columns 127 through 128

   0.0127    0.0128
```

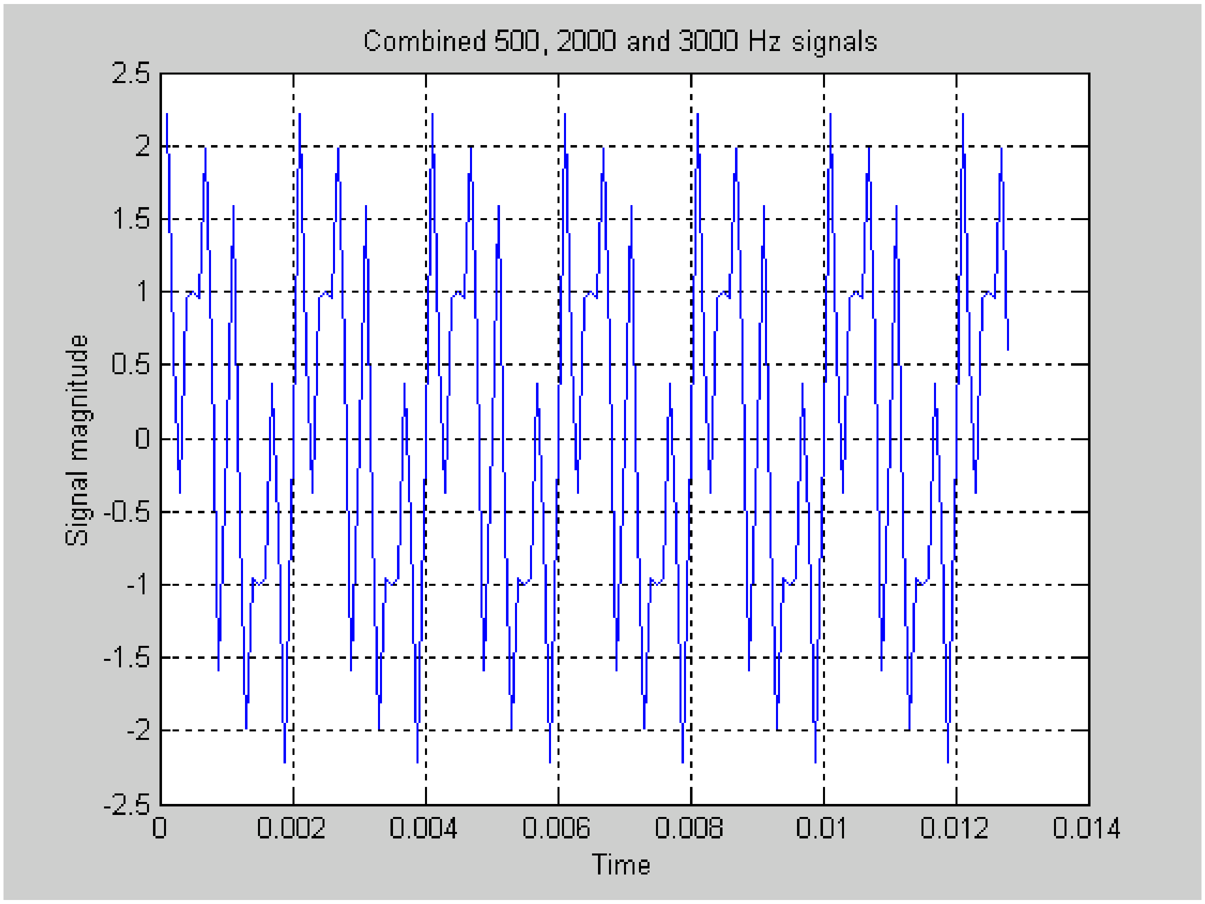
```
% Then the signal we generate will be:
```

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

We can plot the signal, and add a grid to it, with the command

```
plot(t,s),grid
```

The result is:



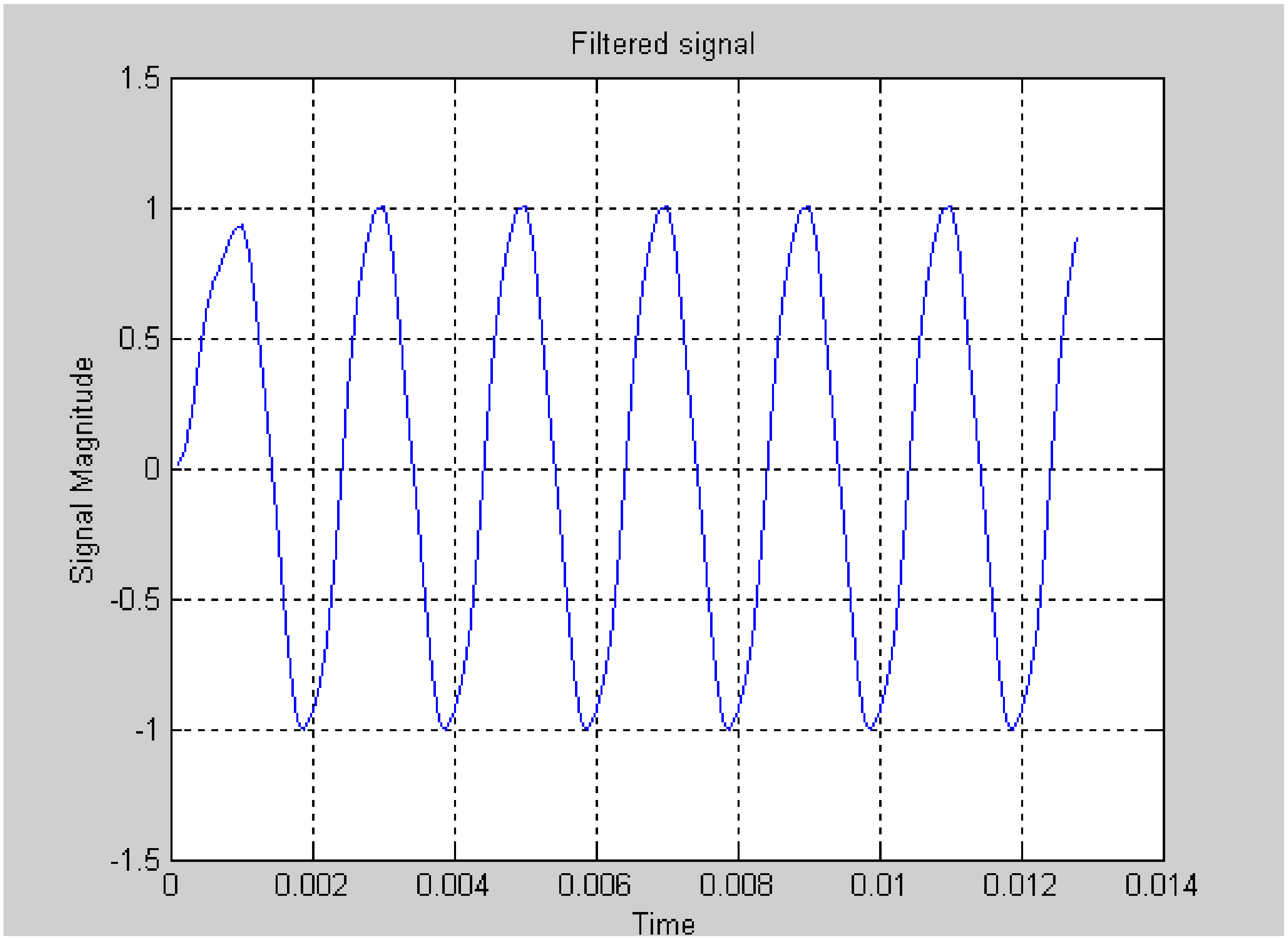
Then we filter the signal through our filter, as follows:

sf = filter(b,a,s)

Matlab prints out a list of magnitudes of the samples of the filtered signal.
Then we can plot the filtered signal by:

plot(t,sf),grid

The result is:



Note that it takes a millisecond or two for the initial transient to die out, which is typical of filter response. This one settles out rather quickly compared to some.

It is obvious that the higher frequency signals have been filtered out, and only the 500 Hz signal remains.

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:

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

where the coefficients a_k and b_k are determined by the equations

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

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

$$f(\omega_o) = \sum_{k=-\infty}^{\infty} c_k e^{jkwot}$$

where the coefficients $c_k = (1/T)\int^{T/2} f(\tau)e^{-jkwot}d\tau$

$-T/2$

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:

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

where

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

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

and

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

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:

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

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 x 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)
```

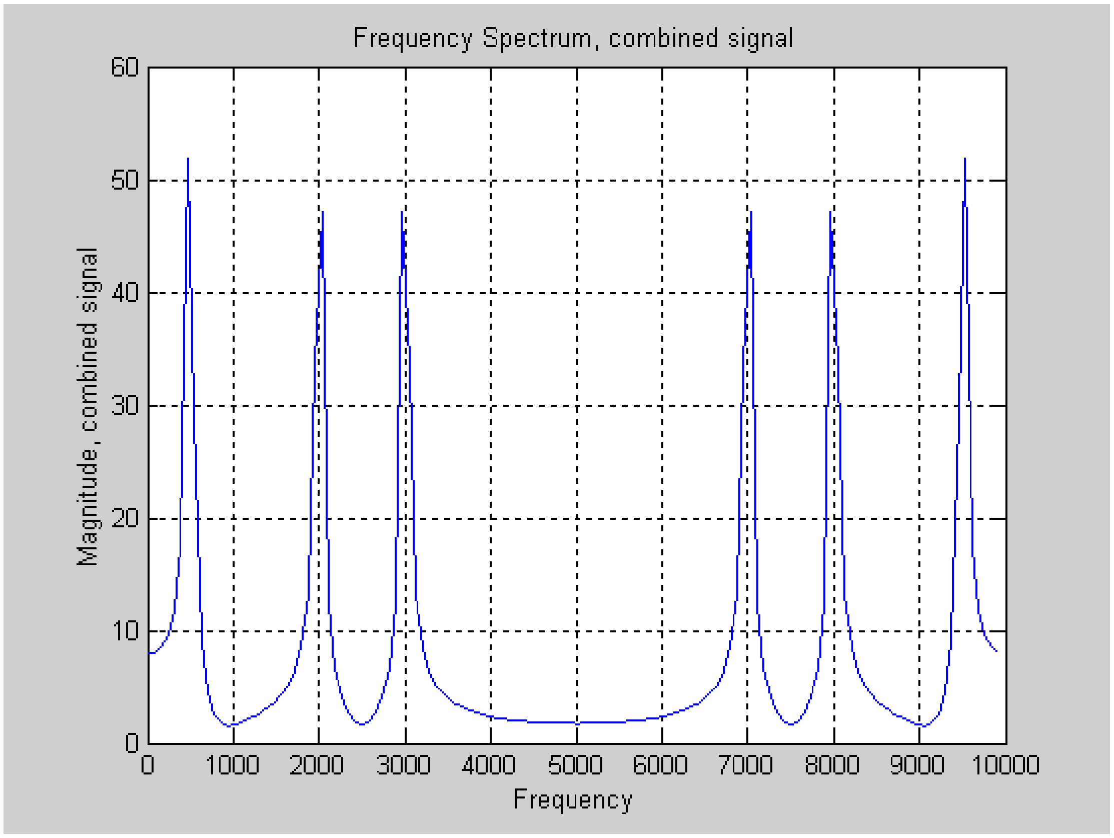
Matlab prints out two more vectors for the filtered signal.
% to plot, we need a frequency vector, for the scale in plotting
% so let's generate a 128-point vector, ranging from 0 to half of fs

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

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

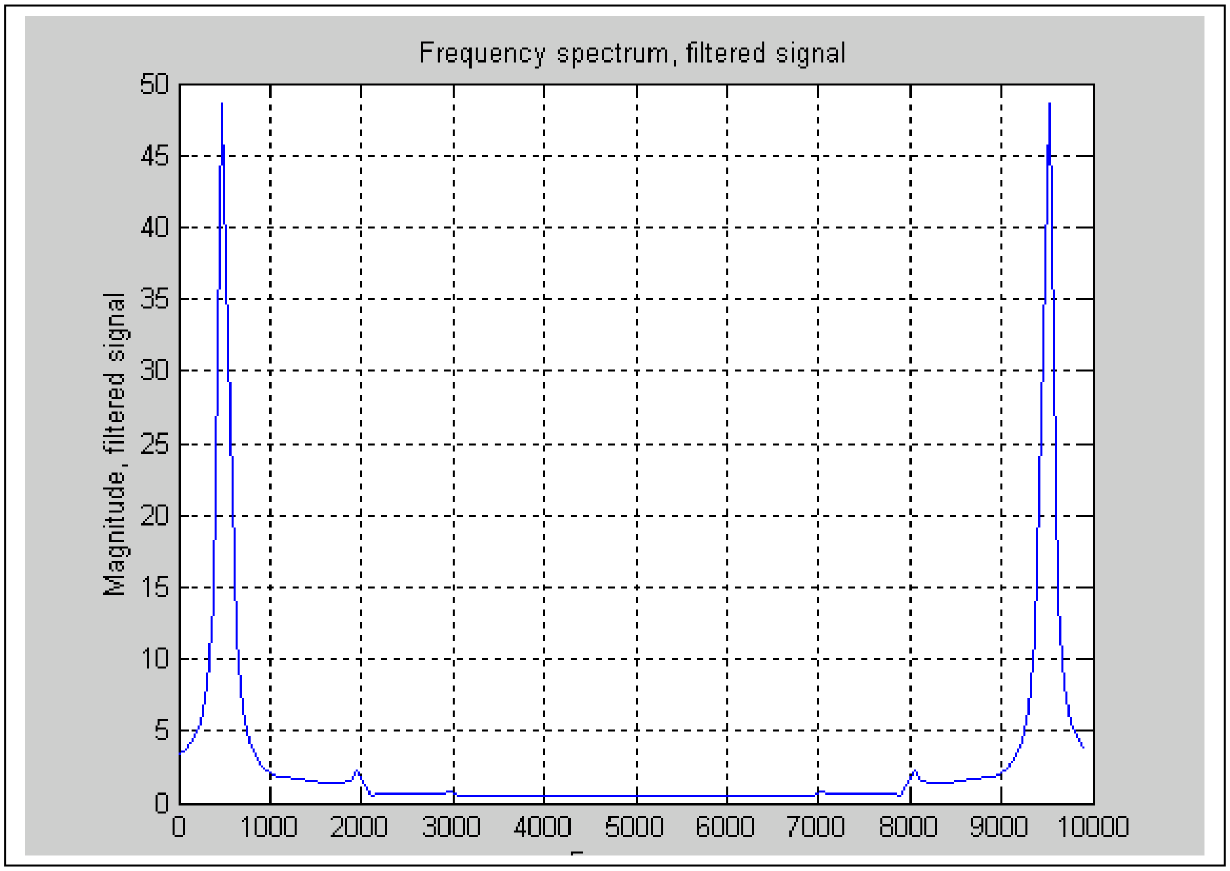
plot(f,absffts,f,absfftsf)

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.

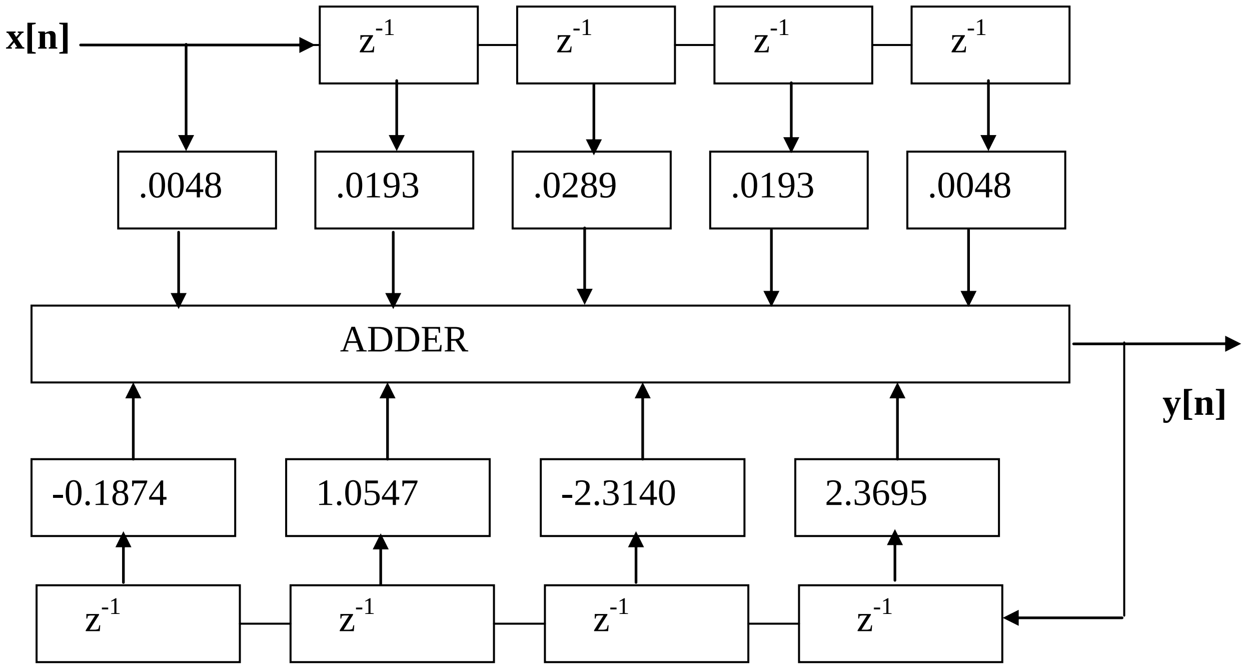
Or, if we just look at the filtered signal with
plot(f, absfftsf), we have



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.

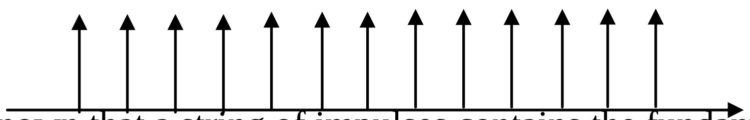
Digitize and store, usually using Pulse Code Modulation, or PCM. The most obvious way, and the simplest conceptually is simply to convert the speech to digital form and store the samples for later D/A conversion and output as analog speech. Data rates of 64 k bits/sec and more are usually used to ensure 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.

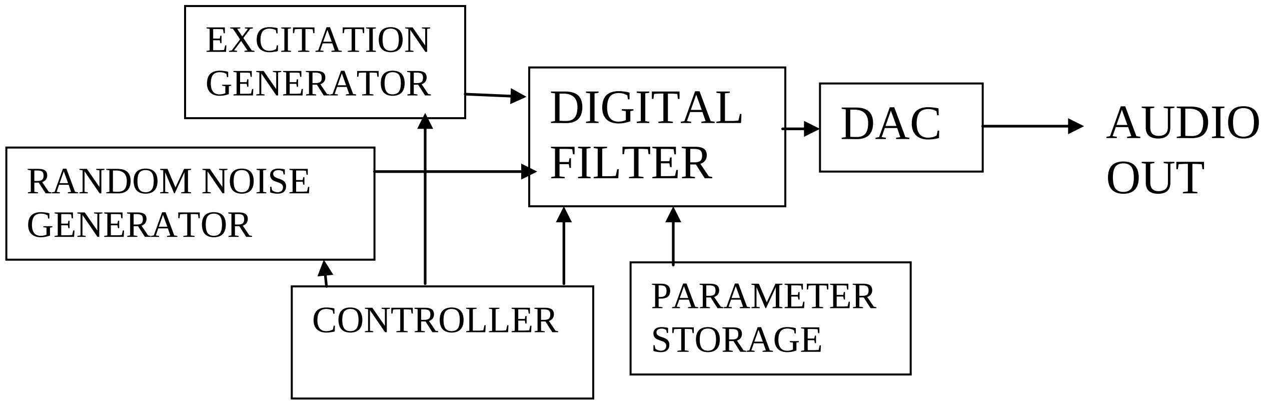
Humans produce only two distinct types of sound when speaking. One is the so-called “unvoiced” sounds, such as the “sssss…” or “th” sound. These are easily produced electronically by random noise. In the computer implementation, a random-number generator serves the purpose.

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 **δ[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')
```